SIAM J. Comput.
Vol. 4, No. 1, March 1975

ON SIMPLE GOEDEL NUMBERINGS AND TRANSLATIONS*

J. HARTMANIS and T. P. BAKER†

Abstract. In this paper we consider classes of Goedel numberings, viewed as simple models for programming languages, into which all other Goedel numberings can be translated by computationally simple mappings. Several such classes of Goedel numberings are defined and their properties are investigated. For example, one such class studied is the class of Goedel numberings into which all other Goedel numberings can be translated by finite automata mappings. We also compare these classes of Goedel numberings to the class of optimal Goedel numberings and show that translation into optimal Goedel numberings can be computationally arbitrarily complex, thus indicating that from a computer science point of view, optimal Goedel numberings have undesirable properties.

Key words. Goedel numberings, translations, complexity of translations, optimal Goedel numberings

1. Introduction. It is well known [1] that all (acceptable) Goedel numberings of the partial recursive functions are recursively isomorphic and thus, from an abstract recursive function theory point of view, they can all be considered equivalent. On the other hand, from a computational complexity point of view, this is definitely not the case, since translations between Goedel numberings can be computationally arbitrarily complex. In particular, if we view Goedel numberings as simple models for programming languages, we are interested in those Goedel numberings into which all other Goedel numberings can be translated easily.

In this paper we study the classification of Goedel numberings by the computational complexity of translating all other Goedel numberings into them. The central concept of this study is the "complexity class" of Goedel numberings, which is defined (for any computational complexity measure) by giving a recursive bound and then considering all Goedel numberings into which any other Goedel numbering can be translated by a mapping whose computational complexity does not exceed the given bound.

We show that there exist Goedel numberings into which all other numberings can be translated by finite automata mappings, and refer to these as *regular* Goedel numberings. It is easily shown that for regular Goedel numberings, the S_n^m-function and the function giving the fixed point, guaranteed by the recursion theorem [2], can also be chosen to be finite automata mappings. As a matter of fact, the computational complexity of translations into a Goedel numbering are directly related to the computational complexity of the S_1^1-function of the numbering.

We show that there exist infinitely many different complexity classes of Goedel numberings and investigate some properties of these classes. For example, using the operator gap theorem [3], we prove that there exist, in any complexity measure, infinitely many recursive bounds t such that all the Goedel numberings in the complexity class defined by the bound t are also isomorphic to each other under

* Received by the editors August 21, 1973.

† Department of Computer Science, Cornell University, Ithaca, New York 14850.

isomorphisms whose complexity is bounded by t. On the other hand, we have left as an open problem whether all the regular Goedel numberings are isomorphic under finite automata mappings.

We show that for every Goedel numbering ϕ (and any computational complexity measure [4]) we can effectively give a recursive bound $t(n)$ such that for any other Goedel numbering ψ, we can choose a translation of ψ into ϕ whose computational complexity is bounded by $t(n)$. We also consider the problem of recursively bounding the length of translated Goedel numbering indices to the indices which are translated. Here we again show that there exists, for every Goedel numbering ϕ, a recursive function f such that for any other Goedel numbering there exists a translation into ϕ which is bounded by f.

Finally, to relate the classification of Goedel numberings by the computational complexity of translations from all other Goedel numberings, we consider the previously studied optimal Goedel numberings [5]. A Goedel numbering ϕ is *optimal* if for any other Goedel numbering ψ there exists a constant c and a translation σ of ψ into ϕ, such that $\sigma(i) \leqq c \cdot i$. Though these optimal numberings have some nice mathematical properties, we show that, from a computer science point of view, they have undesirable properties, since the translations into or between optimal Goedel numberings can be computationally arbitrarily complex and that, similarly, their S_1^1-functions and the functions satisfying the recursion theorem must be computationally arbitrarily complex.

2. Complexity classes of Goedel numberings.

Let R_k and P_k denote the recursive and partial recursive functions of k variables, respectively. For all g in P_{k+1}, let $g_i = \lambda \bar{x}[g(i, \bar{x})]$.

A (an acceptable) *Goedel numbering*, GN, is a recursive enumeration of the partial recursive functions which satisfies the universal machine theorem and the S_n^m theorem [1], [2].

Thus a GN of P_k is a function ϕ^k in P_{k+1} such that for all g in P_{k+1} there exists a t in R_1 satisfying

$$g(i, x_1, x_2, \cdots, x_k) = g_i(x_1, x_2, \cdots, x_k) = \phi_{t(i)}(x_1, x_2, \cdots, x_k).$$

In this paper we are primarily concerned with GN's for P_1. Since we are interested in those GN's into which all others can be easily translated, we will define complexity classes of GN's in terms of the computational complexity of translations from other GN's.

Note that in actual translations from one programming language into another, the translations are mapping sequences over finite alphabets into other sequences over a finite alphabet. Thus we will view an index i in a Goedel numbering as the binary sequence representing i and express some of our results in terms of operations on sequences. It should be noted that the whole treatment can easily be transcribed to the convention that we are indexing algorithms by the set Σ^+, and thus avoid some of the technical difficulties of mixing integers and their binary representations.

DEFINITION. For any GN ϕ^1 of $P_1(\phi^1(i, x) = \phi_i(x))$ and every ψ in P_2, a *translation* of ψ into ϕ^1 is a recursive function σ such that

$$\psi(i, x) = \phi^1(\sigma(i), x) = \phi_{\sigma(i)}(x).$$

We will denote translations by writing $\sigma: \psi \to \phi^1$.

DEFINITION. Let C be any class of recursive functions. Then

$$\mathrm{GN}C = \{\phi^1 | \phi^1 \text{ is a GN and } (\forall \psi \text{ in } P_2)(\exists \sigma \text{ in } C)[\sigma : \psi \to \phi^1]\}.$$

Thus GNC consists of those GN's into which all other GN's can be translated by functions from C. Usually we will let C be some well-known class of functions of bounded computational complexity.

For example, let GNReg denote the class of all GN's into which all other GN's can be translated by finite automata mappings (i.e., deterministic gsm mappings [6]). We refer to these as *regular* GN's.

Similarly, let $C =$ Prfx and $C =$ Pstfx denote, respectively, the class of functions which prefix and postfix a fixed string to the representation of i. That is, σ, is in Prfx iff there exists w such that for all i, $\sigma(i) = wi$. We refer to GNPrfx and GNPstfx as *prefix* and *postfix* GN's, respectively.

The class GNLBA consists of all those GN's into which other GN's can be translated by deterministic linearly bounded automata mappings [6].

It should be observed that several programming languages and many natural GN's belong to GNReg and GNPrfx. Intuitively speaking, every GN or formal programming language in which we can "freely" program is in GNPrfx, since for any other GN ψ, we just have to use a prefix w with the meaning: "This is a description of GN ψ; what follows is a description of an index i; please compute ψ_i." Thus $\sigma(i) = wi$ will be the desired translation of ψ into ϕ.

Next we prove formally that postfix GN's exists. The postfix GN exhibited is the same as used by Schnorr [5] to show that there exist optimal GN's.

A GN ϕ is said to be *optimal* iff, for every GN ψ, there exists a positive constant c and a translation σ of ψ into ϕ such that for all i, $\sigma(i) \leqq c \cdot i$. We denote the class of optimal GN's by GNOpt.

THEOREM 1. *There exist postfix* GN'*s, and proper containment exists between the following classes of* GN'*s as indicated:*

$$\text{GNPstfx} \subset \text{GNReg} \subset \text{GNLBA} \subset \text{GNOpt}.$$

Proof. Let ϕ^2 be any GN of P_2 and let $g(i, n) = i2^{n+1} + 2^n - 1$. The pairing function g is a bijection, and if we interpret it as mapping sequences into sequences, where i is the binary representation of the ith integer and 1^n represents the sequence of n ones, we get $g(i, n) = i0(1^n)$. Thus $\lambda i[g(n, i)]$ is a postfix function for every n, and therefore $\phi_{g(n,i)}(x)$, given by

$$\phi^2[i, n, x] = \phi[g(n, i), x] = \phi_{g(n,i)}(x),$$

is a postfix GN.

It is seen from the definitions that

$$\text{GNPsfx} \subseteq \text{GNReg} \subseteq \text{GNLBA} \subseteq \text{GNOpt}.$$

To show that the containments are proper requires more work, but the proof that

$$\text{GNPsfx} \subset \text{GNReg} \subset \text{GNLBA}$$

follows by a reasonably straightforward construction, which we do not give here.

The proof that GNLBA $\subset$ GNOpt follows from two observations:

1. From the definition of GNLBA, we know that any other GN can be translated into any ϕ in GNLBA by a deterministic linearly bounded automaton [6]. Thus these translations are all in the complexity class of $L(n) = n$ tape-bounded Turing machine computations [4].

2. The proof of Theorem 10 (§3) shows that for every recursive $L(n)$ there exists a GN ϕ which cannot be translated into a ψ in GNOpt by any $L(n)$ tape-bounded Turing machine.

Thus GNLBA $\subset$ GNOpt, which completes the proof.

In [5] it was shown that ϕ is in GNOpt iff ϕ admitted an S^1_1-function which is *linearly size bounded* in the second variable, i.e., for every n there exists a c such that for all i, $\lambda i[S^1_1(n, i)] \leqq c \cdot i$. A similar proof shows our next result.

THEOREM 2. *A* GN ϕ *is in* GNPrfx, GNPstfx, GNReg *and* GNLBA *iff* ϕ *admits an* S^1_1*-function which is, in the second variable, a prefix, postfix,* gsm *or linearly bounded automaton mapping, respectively.*

Proof. We give the proof for a GN ϕ in GNReg.

Let ϕ be a GN of P_1, ϕ^2_n a GN of P_2 and S^1_1 such that for all n, i, x,

$$\phi^2_n(i, x) = \phi_{S^1_1(n,i)}(x),$$

and assume that for any fixed n, $S^1_1(n, i)$ is a gsm mapping. Then for any other GN ψ of P_1, there exists an n_0 such that the numbering ψ is given by $\psi_i(x) = \phi^2_{n_0}(i, x)$. But then

$$\phi_{S^1_1(n_0,i)}(x) = \phi^2_{n_0}(i, x),$$

and we see that $\sigma = \lambda i[S^1_1(n_0, i)]$ is the desired gsm translation.

Conversely, if GN ϕ is in GNReg and $\bar{\phi}$ is the GN of Theorem 1, then there exists a gsm mapping σ such that

$$\phi_{\sigma \circ g(n,i)}(x) = \bar{\phi}_{g(n,i)}(x) = \phi^2(i, n, x).$$

Since g (defined in Theorem 1) is a postfix translation for any fixed n, $\sigma \circ g(n, i)$ is a gsm mapping. Thus ϕ admits $S^1_1(n, i) = \sigma \circ g(n, i)$ as an S^1_1-function which is a gsm mapping in the second variable. The other cases follow by an identical argument.

Next we show that those GN's into which all others can be easily translated have easily computable recursion theorem fixed points.

We recall that by the recursion theorem [2], for every GN ϕ of P_1 there exists a recursive function n such that for all z,

$$\phi_{n(z)} = \phi_{\phi_z[n(z)]}.$$

THEOREM 3. *If* ϕ *is in* GNPrfx, GNPstfx, GNReg *or* GNLBA, *then there exists a prefix, postfix,* gsm *or* lba *mapping* n, *respectively, such that for every* z,

$$\phi_{n(z)} = \phi_{\phi_z[n(z)]}.$$

Proof. The proof follows the standard proof of the recursion theorem [2]. Define

$$\psi(u, x) = \phi_{\phi_u(u)}(x) \quad \textbf{if } \phi_u(u) \text{ converges } \textbf{else} \text{ divergent.}$$

Let g be a recursive function (translation) such that $\psi(u, x) = \phi_{g(u)}(x)$. Thus for ϕ_u total, we have

$$\phi_{\phi_u}(u) = \phi_{g(u)}.$$

Define

$$\mu(z, x) = \phi_z \circ g(x) \quad \textbf{if } \phi_z \circ g(x) \text{ converges } \textbf{else} \text{ divergent}$$

and let h be the recursive function (translation) such that $\mu(z, x) = \phi_{h(z)}(x)$. Thus for ϕ_z total, we have $\phi_{h(z)} = \phi_z \circ g$. By combining these equalities, we get for all z such that ϕ_z is total,

$$\phi_{g \circ h(z)} = \phi_{\phi_{h(z)}[h(z)]} = \phi_{\phi_z[g \circ h(z)]}.$$

Thus by setting $g \circ h(z) = n(z)$, we get that

$$\phi_{\phi_z[n(z)]} = \phi_{n(z)}.$$

Furthermore, since g and h are translations, we can choose them to be prefix, postfix, regular, or lba mappings, respectively, and therefore $n = g \circ h$ will be a mapping of the same type, as was to be shown.

Schnorr has shown that the optimal GN's are all isomorphic under linearly size bounded mappings. Thus, in a mathematical sense, they form a natural class of GN's.

For the prefix and postfix GN's, we know that they cannot be isomorphic under prefix and postfix mappings, since these mappings are (but for the trivial case) proper into mappings. On the other hand, our next result shows that they are isomorphic under gsm mappings. Thus they form classes of GN's which are very similar in a computational sense.

THEOREM 4. *Let ϕ and $\bar{\phi}$ be in* GNPrfx *or* GNPstfx. *Then there exists a permutation π such that π and π^{-1} are* gsm *mappings, and*

$$\phi_i = \bar{\phi}_{\pi(i)} \quad \textit{and} \quad \bar{\phi}_i = \phi_{\pi^{-1}(i)}.$$

Proof. We give the proof for GNPrfx. Let w and v be the prefix sequences which translate ϕ into $\bar{\phi}$ and $\bar{\phi}$ into ϕ, respectively. Then

$$\begin{aligned} 1(1 + 0)^* &= \{(wv)^k z \mid k = 0, 1, 2, \cdots \text{ and } z \notin w(0 + 1)^*\} \\ &\quad \cup \{w(vw)^k z \mid k = 0, 1, 2, \cdots \text{ and } z \notin v(0 + 1)^*\} \\ &= \{v(wv)^k z \mid k = 0, 1, 2, \cdots \text{ and } z \notin w(0 + 1)^*\} \\ &\quad \cup \{(vw)^k z \mid k = 0, 1, 2, \cdots \text{ and } z \notin v(0 + 1)^*\}. \end{aligned}$$

Define

$$\pi[(wv)^k z] = v(wv)^k z \quad \text{for } k = 0, 1, 2, \cdots \text{ and } z \notin w(0 + 1)^*,$$
$$\pi[w(vw)^k z] = (vw)^k z \quad \text{for } k = 0, 1, 2, \cdots \text{ and } z \notin v(0 + 1)^*.$$

We see that π is a permutation of the set $1(0 + 1)^*$, and since $\phi_i = \bar{\phi}_{wi}$ and $\bar{\phi}_j = \phi_{vj}$, we see that for $j = (wv)^k z$, $\bar{\phi}_j = \phi_{vj} = \phi_{\pi(j)}$ and for $j = w(vw)^k z$, $\phi_{(vw)^k z} = \bar{\phi}_{w(vw)^k z}$, and therefore $\bar{\phi}_j = \phi_{(vw)^k z} = \phi_{\pi(j)}$. Finally we note that a finite automaton can perform the permutation π. The arguments for π^{-1} can be carried out similarly, which completes the proof.

It should be noted that in the previous proof, the finite automaton computing π either prefixed the sequence v or removed the sequence w. A *restricted regular mapping* is a finite automaton mapping which can only prefix a fixed string or remove a fixed string from the input sequence. We call the GN's into which all other GN's can be mapped by such mappings *restricted regular* GN's. We conjecture that the restricted regular GN's are all isomorphic under restricted regular mappings. Unfortunately, so far we have not been able to prove this conjecture.

We also conjecture that regular GN's are not isomorphic under finite automata isomorphisms. Again, we have not been able to prove this simple sounding conjecture.

We can prove though that the complexity of the isomorphisms between GN's in GNReg or GNLBA, respectively, are of bounded computational complexity, and in the next sections we will see that this is not true for GNOpt.

To do this, we will prove a more general result which holds in all computational complexity measures. For this purpose we recall [4] that a *computational complexity measure* is given for a GN ϕ by assigning to every algorithm ϕ_i a step counting function Φ_i such that:

1. for all i and n $\phi_i(n)$ is defined iff $\Phi_i(n)$ is defined;
2. it is recursively decidable for all i, m, n whether $\Phi_i(n) = m$.

For a computational complexity measure, denoted by (ϕ, Φ) or just Φ, we define for every recursive function t a *complexity class*

$$C_t^\Phi = \{f \mid f \text{ is a recursive function, for some } i\ f = \phi_i \text{ and } \Phi_i(n) \leqq t(n) \text{ a.e.}\}.$$

Note that the complexity classes are not changed if we change the GN on which the computational complexity measure is defined. Thus quite often we will not explicitly mention on what GN the measure is defined.

Let GNC_t^Φ denote all the GN's into which all other GN's can be mapped by mappings in complexity class C_t^Φ.

THEOREM 5. *For every computational complexity measure* Φ, *there exist infinitely many different classes of* GN*'s* GNC_t^Φ.

Proof. By a straightforward diagonal argument, we can construct for every recursive t and GN ϕ another GN ψ such that ϕ cannot be translated into ψ by any function in C_t^Φ.

THEOREM 6. *For any computational complexity measure* Φ *and* t *in* R_1, *there exists a* t' *in* R_1 *such that the isomorphisms between* GN*'s in* GNC_t^Φ *can be chosen from* $C_{t'}^\Phi$.

Proof. We only outline the proof. Let ϕ and $\bar{\phi}$ be two GN's and $\phi_i = \sigma$ and $\phi_j = \rho$ be two translations of ϕ into $\bar{\phi}$ and $\bar{\phi}$ into ϕ, respectively. Then we know from the proof of the isomorphism theorem for Goedel numberings [1] that the index of the isomorphism δ yielded by the theorem (i.e., a 1–1 translation of ϕ onto $\bar{\phi}$) is given by a recursive function g of the indices of ϕ_i and ϕ_j, that is, $\delta = \phi_{g(i,j)}$. It is easily seen that there exists another recursive function h such that the computational complexity $\Phi_{g(i,j)}$ of the isomorphism $\delta = \phi_{g(i,j)}$ is bounded by $\phi_{h(i,j)}$:

$$\Phi_{g(i,j)}(x) \leqq \phi_{h(i,j)}(x) \quad \text{a.e.}$$

Furthermore, h can be so chosen that

$$\Phi_i(x) \text{ and } \Phi_j(x) \leqq \Phi_k(x) \quad \text{a.e.}$$

implies that

$$\Phi_{g(i,j)}(x) \leqq \phi_{h(k,k)}(x) \quad \text{a.e.}$$

From this it follows that for any two GN's ϕ and $\bar{\phi}$ in $\mathrm{GN}C^{\Phi}_{\phi_k}$, there exists an isomorphism δ between ϕ and $\bar{\phi}$ such that δ is in $C^{\Phi}_{\phi_{h(k,k)}}$, as was to be shown.

Recall that a *recursive operator* (say, in one variable) on the partial recursive functions is given by a recursive function $f: N \to N$ such that $\phi_i = \phi_j$ implies that $\phi_{f(i)} = \phi_{f(j)}$.

It is easily seen from the previous proof or from [1] that the isomorphism δ is yielded by a recursive operator from the two translations $\sigma = \phi_i$ and $\rho = \phi_j$ between the GN's ϕ and $\bar{\phi}$. The operator is defined by the recursive function g. Similarly, the recursive function h, yielding the complexity bound

$$\Phi_{g(i,j)}(x) \leqq \phi_{h(i,j)}(x) \leqq \phi_{h(k,k)}(x) \quad \text{a.e.},$$

can be viewed as a recursive operator. Thus we see that the computational complexity of the isomorphism δ is bounded by a recursive operator in the complexity of the two translations.

When we combine this observation with the operator gap theorem [3] we get a result obtained jointly with K. Mehlhorn.

COROLLARY 7. *For any computational complexity measure Φ, there exist arbitrarily large t in R_1 such that any two* GN'*s in* $\mathrm{GN}C^{\Phi}_t$ *are isomorphic under a permutation in C^{Φ}_t.*

Proof. Let $\mathscr{R}$ be the recursive operator yielded by the isomorphism theorem. Then there exists a recursive operator which bounds the complexity of the resulting isomorphism in terms of the complexity of the two into mappings; denote it by $\mathscr{R}'$. Then from the operator gap theorem [3], we know that there exists arbitrarily large recursive t such that

$$C^{\Phi}_t = C^{\Phi}_{\mathscr{R}'[t]}.$$

Thus all GN's in $\mathrm{GN}C^{\Phi}_t$ are isomorphic under mappings in C^{Φ}_t, as was to be shown.

It should be stated again that it would be very interesting to find some *natural* complexity classes of Goedel numberings, say GNC, such that all the GN's in GNC are isomorphic under permutations in C. For example, we conjecture that GNLBA is closed under lba isomorphisms.

We describe one reasonably natural class of Goedel numberings which is closed under isomorphisms of the same type.

Let GNPTIME denote the class of GN's into which all others can be translated by deterministic Turing machines whose computation times are bounded by a polynomial function of the input (index) (i.e., the index, not the length of the representation of the index!)

The following result is due to R. Constable.

THEOREM 8. *Any two* GN'*s in* GNPTIME *are isomorphic under a polynomial-time bounded Turing machine computable mapping.*

Proof. The proof is by a lengthy and careful estimation of run times in the proof of the isomorphism theorem [1].

A somewhat more natural (and better known) class of GN's would be the class into which all other GN's can be translated by Turing machines whose computation times are bounded by a polynomial in the length of the input (index). Unfortunately, the previous proof does not extend to this class of Goedel numberings. We conjecture that the answer is positive.

We conclude this section by showing that for any GN ϕ^1 we can give a recursive function which bounds the complexity of translations from all other GN's into ϕ^1. Furthermore, there exists for every fixed GN ϕ^1 a recursive function l and a translation from every other GN into ϕ^1 such that l bounds the length of the translated index to the length of the index.

THEOREM 9. *Let Φ be any computational complexity measure and ϕ^1 a fixed* GN. *Then we can recursively obtain from an index of ϕ^1 indices for two recursive functions s and l such that for any* GN *ϕ there exists a translation $\sigma:\phi \to \phi^1$ such that σ is in C_s^Φ and $|\sigma(i)| \leqq |l(i)|$ a.e.*

Proof. Let ψ be a prefix GN and let σ_0 be a translation mapping ψ into ϕ^1. Then for any GN ϕ there exists a sequence w such that $\sigma(i) = wi$ is a translation of ϕ into ψ. But then $\sigma_0 \circ \sigma(i) = \sigma_0(wi)$ is a translation of ψ into ϕ^1. To obtain the functions s and l, let $\Sigma_0(wi)$ be the step-counting function of the computation $\sigma_0(wi)$ in the complexity measure Φ and define

$$s(i) = \max_{|w| \leqq |i|} \{\Sigma_0(wi)\},$$

and similarly,

$$l(i) = \max_{|w| \leqq |i|} \{|\sigma_0(wi)|\}.$$

Clearly, for every ϕ there exists a translation $\sigma:\phi \to \phi^1$ such that σ is in C_s^Φ and $|\sigma(i)| \leqq |l(i)|$ a.e., as was to be shown.

3. Complexity of optimal Goedel numberings.

In this section we show that the computational complexity of translations into optimal Goedel numberings cannot be recursively bounded. Actually we will show that for any GN ϕ, there exist optimal GN's such that the translations from ϕ into these optimal GN's must be arbitrarily complex. Similarly, we will show that for optimal GN's the computational complexity of the S_1^1-function and the function n of the recursion theorem cannot be recursively bounded.

THEOREM 10. *For any computational complexity measure (ϕ, Φ) and recursive function t, there exists an optimal* GN *ψ into which ϕ cannot be translated by any σ in C_t^Φ.*

Proof. Let χ be an optimal GN. We will obtain ψ from χ by a recursive permutation which will not move any index upward by more than one place. Therefore ψ will also be an optimal GN. The construction of ψ is obtained by diagonalizing over all possible t-bounded σ.

We know that for sufficiently large recursive T, the complexity class C_T^Φ can be recursively enumerated [4]. Thus we can assume that $\phi_{k_1}, \phi_{k_2}, \phi_{k_3}, \cdots$ is an enumeration of the functions in C_T^Φ, or choose a larger complexity bound T with an enumerable complexity class.

Let $\phi_{j_1}, \phi_{j_2}, \phi_{j_3}, \cdots$ be a recursive sequence of constant functions such that $\phi_{j_k}(x) = k$.

We now define the stages in the computation of ψ.

Stage 1. Let $\psi_1 = \chi_1$.

Stage i. By the ith stage, let $\psi_1, \psi_2, \cdots, \psi_{N_i}$ be defined. We define ψ_j, $N_i < j \leqq N_{i+1}$, as follows: compute $\phi_{k_i}(j)$ dovetailed for $j = N_i + 1, N_i + 2, \cdots$, until

(a) $\phi_{k_i}(j) > N_i$ and $\phi_j(1)$ converges, or

(b) $\phi_{k_i}(j_l) \leqq N_i$ for more than N_i distinct values of l (where j_l is from the enumeration of $\phi_{j_1}, \phi_{j_2}, \cdots$).

This computation is eventually halted by (a) or (b). In case (a), let $n = \phi_{k_i}(j)$. Let q be the first p greater than n for which $\chi_p(1) \neq \phi_j(1)$ gotten by dovetailing the computation of $\chi_p(1)$ for $p = j + 1, j + 2, \cdots$. Let $N_{i+1} = q$. We want $\psi_n \neq \phi_j$, so we define $\psi_p = \chi_p$ for $p < n$, $\psi_n = \chi_q$, and $\psi_p = \chi_{p-1}$ for $n < p \leqq q$, (e.g., see Fig. 1.)

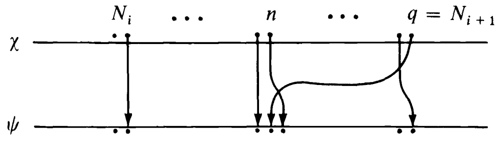


FIG. 1

In case (b), let $N_{i+1} = N_i + 1$ and $\psi_{N_i+1} = \chi_{N_i+1}$ and proceed to Stage $i + 1$.

If ϕ_{k_i} translates ϕ into ψ, then we would have

$$\phi_p = \psi_{\phi_{k_i}(p)}$$

for all p. This is not the case if the computation was halted by (a), because of the definition of ψ. In case (b), at least two different j_l, j_m exist such that $\phi_{k_i}(j_l) = \phi_{k_i}(j_m)$, but then

$$\psi_{\phi_{k_i}(j_l)} = \psi_{\phi_{k_i}(j_m)} \quad \text{and} \quad \phi_{j_l} \neq \phi_{j_m},$$

an inconsistency. Thus we see that no ϕ_k from C_t^Φ can translate ϕ into the optimal GN ψ, as was to be shown.

Next we show that optimal GN's require computationally arbitrarily complex S_1^1-functions.

THEOREM 11. *For every computational complexity measure Φ and recursive function t, there exists optimal* GN *ϕ such that for no S_1^1-function is $\lambda i[S_1^1(n, i)]$ in C_t^Φ for all n.*

Proof. Using the previous result we construct two optimal GN's ϕ and $\bar{\phi}$ such that $\bar{\phi}$ cannot be translated into ϕ by a mapping in C_t^Φ. Let S_1^1 be defined for ϕ. Then for some n_0, $\phi^2(n_0, i, x) = \bar{\phi}(i, x)$, and therefore

$$\phi^2(n_0, i, x) = \phi_{S_1^1(n_0,i)}(x) = \bar{\phi}_i(x).$$

But then $\sigma = \lambda i[S_1^1(n_0, i)]$ is a translation of $\bar{\phi}$ into ϕ, and therefore $\lambda i[S_1^1(n_0, i)]$ is not in C_t^Φ, as was to be shown.

Similarly, we show that optimal GN's require arbitrarily complex functions satisfying the recursion theorem.

THEOREM 12. *For any recursive function t and complexity measure (ϕ, Φ), there is an optimal Goedel numbering ψ such that every function ϕ_k satisfying the recursion theorem condition: for all j,*

$$\psi_{\phi_k(j)} = \psi_{\psi_j(\phi_k(j))},$$

is of complexity greater than t.

Proof. We define the desired ψ inductively, diagonalizing over all the possible t-bounded ϕ_k.

Without loss of generality, we may assume that ϕ is an optimal Goedel numbering. If it is not, we may construct another complexity measure (ϕ', Φ) with the same complexity classes such that ϕ' *is* an optimal Goedel numbering: start with an optimal Goedel numbering χ (we know there are infinitely many of these); let $\phi_j = \chi_{\sigma(j)}$ for every j, where σ is a recursive isomorphism from ϕ to χ.

Let $\phi_{j_1}, \phi_{j_2}, \cdots$ be a recursive subsequence of ϕ consisting of the constant functions $\phi_{j_i}(x) = \lambda(x)(i)$, $j_i > i + 3$. ϕ must have such a recursive subsequence by the S_n^m theorem.

Let

$$p(n) = \min \{j_{j_c} | c > n\} \quad \text{and} \quad q(n) = \min \{j_{j_d} | d > p(n)\}.$$

Note that these are both recursive functions and that they are indices in ϕ of constant functions which compute indices in ϕ of other constant functions. For any n, suppose $\phi_{p(n)} = \lambda(x)(a)$, $\phi_a = \lambda(x)(b)$, $\phi_{q(n)} = \lambda(x)(c)$, and $\phi_c = \lambda(x)(d)$; then

$$n < a < b < p(n) < c < d < g(n).$$

Since for sufficiently large t the functions of complexity t can be recursively enumerated, we assume without any loss of generality that $\phi_{k_1}, \phi_{k_2}, \cdots$ is a recursive enumeration of the functions of complexity t.

ψ will be defined as a recursive permutation of ϕ in which no index is increased by more than three, so there will be no question about ψ being an optimal Goedel numbering. We assume that $\psi_1, \psi_2, \cdots, \psi_{N_i}$ are defined by the ith stage and proceed to extend the definition to $\psi_{N_i+1}, \cdots, \psi_{N_i+1}$. Let $k = k_i$ and $N = N_i$. Compute $\phi_k(j_l)$ and $\phi_{j_l}(\phi_k(j))$ for $l = N + 1, N + 2, \cdots$, until one of the following cases holds. In each case, ψ is defined for certain critical indices so that for some j,

$$\psi_{\phi_k(j)} \neq \psi_{\psi_j(\phi_k(j))}.$$

Case 1. If $\phi_k(j_m) = \phi_k(j_l)$, $m > N$, and $l > j_m + 3$, let $\psi_{j_l} = \phi_{j_l}$, $\psi_l = \phi_{q(j_l)}$, $\psi_m = \phi_{p(j_m)}$, $\psi_{j_m} = \phi_{j_m}$, and $N_{i+1} = q(j_l)$. Then

$$\psi_{\psi_{j_l}(\phi_k(j_l))} = \psi_l = \phi_{q(j_l)} \neq \phi_{p(j_m)} = \psi_m = \psi_{\psi_{j_m}(\phi_k(j_m))}.$$

Case 2. If $\phi_k(j_l) = j_l$, let $\psi_{j_l} = \lambda(x)(a) = \phi_{p(j_l)}$, $\psi_a = \phi_a$ and $N_{i+1} = q(j_l)$. Then

$$\psi_{\psi_{j_l}(\phi_k(j_l))} = \psi_a = \phi_a \neq \phi_{p(j_l)} = \psi_{j_l} = \psi_{\phi_k(j_l)}.$$

Case 3. If $\phi_k(j_l) = l$, let $\psi_l = \phi_{q(j_l)}$, $\psi_{j_l} = \lambda(x)(a) = \phi_{p(j_l)}$, $\psi_a = \phi_a$, and $N_{i+1} = q(j_l)$. Then

$$\psi_{\psi_{jl}(\phi_k(j_l))} = \psi_a = \phi_a \neq \phi_{q(j_l)} = \psi_l = \psi_{\phi_k(j_l)}.$$

Case 4. If $\phi_k(j_l) \neq l$, $\neq j_l$, $> N$, and $m = \max(j_l, \phi_k(j_l))$, let $\psi_{j_l} = \phi_{j_l}$, $\psi_l = \phi_{p(m)}$, $\psi_{\phi_k(j)} = \phi_{q(m)}$, $N_{i+1} = q(m)$. Then

$$\psi_{\psi_{jl}(\phi_k(j_l))} = \psi_l = \phi_{p(m)} \neq \phi_{q(m)} = \psi_{\phi_k(j_l)}.$$

Furthermore, ψ_j is defined for all other j ($N_i < j \leqq N_{i+1}$) to be ϕ_j, ϕ_{j-1}, ϕ_{j-2} or ϕ_{j-3}, shifting the indices as little as possible; i.e.,

```
for $j := N_i + 1$ until $N_{i+1}$ do
    if ($\psi_j$ not yet defined)
    then $\psi_j := \Phi_{\min \{i | \phi_i \text{ not yet used to define any } \psi_k\}}$.
```

For example, in Case 1 we have Fig. 2.

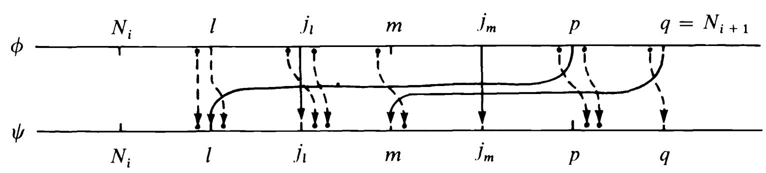


FIG. 2

To see that this computation must halt, suppose Cases 1, 2 and 3 fail for every $l > N_i$. It follows that Case 4 succeeds for large enough l, since:

$$\phi_k(j_l) \neq l \qquad \text{(by Case 3)}$$

$$\phi_k(j_l) \neq j_l \qquad \text{(by Case 2)}$$

$$\phi_k(j_l) > N_i \qquad \text{(because } (\phi_k(j_l) \leqq N \text{ for a.e. } l) \Rightarrow \text{Case 1)}.$$

Thus for no ϕ_k in C_t^Φ can we have that

$$\psi_{\phi_k(j)} = \psi_{\psi_j[\phi_k(j)]},$$

as was to be shown.

REFERENCES

[1] H. ROGERS, JR., *Goedel numberings of partial recursive functions*, J. Symbolic Logic, 18 (1971), pp. 444–475.

[2] ———, *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York, 1967.

[3] R. L. CONSTABLE, *The operator gap*, IEEE Conference Record of 1969 Tenth Annual Symposium on Switching and Automata Theory, 1969, pp. 20–26.

[4] J. HARTMANIS AND J. E. HOPCROFT, *An overview of the theory of computational complexity*, J. Assoc. Comput. Math., 18 (1971), pp. 444–475.

[5] C. P. SCHNORR, *Optimal enumerations and optimal Goedel numberings*, Math. Systems Theory, to appear.

[6] J. E. HOPCROFT AND J. D. ULLMAN, *Formal Languages and Their Relation to Automata*, Addison-Wesley, Reading, Mass., 1969.

SIAM J. COMPUT.
Vol. 4, No. 1, March 1975

REALIZATION WITH FEEDBACK ENCODING . I: ANALOGUES OF THE CLASSICAL THEORY*

DENNIS P. GELLER†

Abstract. For a finite state machine M to realize a machine M', we usually precede M by a memoryless input encoder to translate inputs intended for M' into the input alphabet of M. In this paper we introduce a modification to this paradigm by introducing feedback from the state of the realizing machine to the input encoder. The resulting form of realization depends in a very strong way on (graph) structural properties of the two machines. The characterization theorem, giving necessary and sufficient conditions for one machine to realize another in this way, involves a new class of mappings between digraphs. We also investigate a corresponding algebraic structure theory.

1. Introduction and definitions. A major concern of classical automata theory has been the realization of either the state behavior or the input-output behavior of machines. The main thrust of the research has always been to realize the given machine by a loop-free network; that is, by a network for which the digraph obtained by taking the modules as points and the flow lines between them as directed arcs has no directed cycles. In such a network there is no feedback, except possibly within modules.

As noted by Holland [8], "feedback is a prominent structural feature of most systems which exhibit complex behavior". Nevertheless many networks can be modeled by cascades, i.e., feedback-free networks. The modules of the new network are taken to be the strong components of the original network (see, for example, [13]). This construction, of course, greatly increases the complexity of the component modules.

There are advantages both to excluding and including feedback in networks. On the one hand, there is a large body of techniques available for analyzing feedback-free systems and also for deriving feedback-free realizations of a given behavior [6]. On the other hand, a feedback-free realization can be artifactual: it often consists of merely masking those parts of the network with feedback, the strong components, and considering them as "black boxes". We thus have a trade-off between simplicity of the components and simplicity of the interconnections. It is certainly to be expected that by restricting in some way the form that feedback is allowed to take, we can strike a suitable balance between the two extremes.

Thus one motivation of the study we are about to undertake is the desirability of defining a restricted form of feedback which will prove tractable in both theory and application. Another is the following. Suppose that we have some given state behavior which we want to realize by a sequential machine. If the behavior is so realizable, then, in fact, techniques exist for doing this in an optimal way. However, it is often the case that additional restrictions are placed on the realization. For

* Received by the editors January 18, 1973, and in final revised form March 7, 1974.

† Human Sciences and Technology Group, School of Advanced Technology, State University of New York at Binghamton, Binghamton, New York 13901. This research was supported in part by the National Institutes of Health under Grant GM-12236 and in part by the National Science Foundation under Grant GJ-519.

example, one might want the realizing machine to be expressible as a cascade of modules from a well-defined set. This, of course, has been studied by Zeiger [12], Krohn and Rhodes [9] and others. In general, it turns out that the behavior of the cascade which is derived properly includes the desired behavior. For another example, take the fault diagnosis problem. Here we might want a realizing machine, for example, which, if it fails in some way, allows us to determine the cause or location of the fault, or perhaps even to correct it. Even the simplest fault diagnosis properties cannot, in general, be incorporated into the reduced machine which realizes the given behavior, and it is usually necessary for the realizing machine to have more states or inputs (and thus, more state circuitry) than the reduced machine. A second motivation for the study to follow, therefore, is to develop a form of realization which will allow us to add additional constraints and at the same time will not cause the same state set growth as the classical theory.

A *machine*, or *finite-state automaton* M, consists of a set Q of *states*, a set I of *inputs*, where both Q and I are finite, and a map $\delta: Q \times I \rightarrow Q$. We will often omit explicit reference to δ, and write instead $\delta(q, i) = qi$. Of course, δ can be extended to have domain $Q \times I^+$, where I^+ is the set of all nonempty strings from I. We say that a machine M *realizes* a machine M' if there are maps $\phi: Q \xrightarrow{\text{onto}} Q'$ and $h: I' \xrightarrow{\text{onto}} I$ such that for all $q \in Q$, $x' \in I'$, $\phi(q)x' = \phi(qh(x'))$. The map h is called the *input encoder*, and ϕ is a *homomorphism* or SP-*homomorphism.* If ϕ is one-to-one, then it is an *isomorphism.* A machine M *simulates* a machine M' if there are maps $\phi: Q \rightarrow Q'$ and $h: I' \rightarrow I^+$ such that $\phi(q)x' = \phi(qh(x'))$.

A *Mealy machine* $M = \langle Q, I, \delta, \lambda, Y\rangle$ consists of a machine $\langle Q, I, \delta\rangle$ together with a finite set Y of *output symbols* and an *output function* $\lambda: Q \times I \rightarrow Y$. If for any states q_1 and q_2 and any inputs x_1 and x_2, $q_1x_1 = q_2x_2$ implies that $\lambda(q_1, x_1) = \lambda(q_2, x_2)$, then the output is a function of the next state, and we will write $\lambda(q, x) = \lambda(qx)$; such a machine is a *Moore machine.* We can extend λ to have domain $Q \times I^+$ in two ways. First, if $x = yx_1 \in I^+$, where $x_1 \in I$, then $\lambda(q, x) = \lambda(qy, x_1)$. On the other hand, $\beta: Q \times I^+ \rightarrow Y^+$ is the map $\beta(q, x_1 \cdots x_n) = \lambda(q, x_1)\lambda(qx_1, x_2) \cdots \lambda(qx_1 \cdots x_{n-1}, x_n)$. For a Moore machine, the last expression is $\beta(q, x_1 \cdots x_n) = \lambda(qx_1)\lambda(qx_1x_2) \cdots \lambda(qx_1x_2 \cdots x_n)$. It is common, instead, to write $\beta(q, x_1 \cdots x_n) = \lambda(q)\lambda(qx_1) \cdots \lambda(qx_1 \cdots x_n)$ (see [1, p. 79]). We will be ignoring the output $\lambda(q)$ of the starting state to facilitate later theoretical analyses, although one could certainly take advantage of this information in practice.

If M and M' are machines with output, then M *realizes* M' if there are maps $\phi: Q \xrightarrow{\text{onto}} Q'$, $h: I' \xrightarrow{\text{onto}} I$, $g: Y \rightarrow Y'$ such that $\phi(q)x' = \phi(qh(x'))$ and $\lambda'(\phi(q), x') = g(\lambda(q, h(x')))$.

Given machines M_1 and M_2, let Z be a map $Z: Q_1 \times I_1 \rightarrow I_2$. The *cascade* $M_1 \circ_Z M_2$ with connecting map Z is a machine M with $Q = Q_1 \times Q_2$, $I = I_1$, and $(q_1, q_2)x_1 = (q_1x_1, q_2Z(q_1, x_1))$; we say that M_1 is the *front component* and M_2 is the *tail component.*

The operation of cascading machines is, in general, not associative. On the other hand, if we write $M_1 \circ_{Z_1} M_2 \circ_{Z_2} M_3$, we can remove the ambiguity by completely specifying the maps Z_i. If we have a cascade M of n machines M_i, we can write $M = \prod M_i = N_1 \circ_Z N_2$, where, in general, both N_1 and N_2 can be specified as cascades of the M_i.

A *digraph* D consists of a set $V = V(D)$ of *points* together with a collection (repetitions permitted) $X = X(D)$ of ordered pairs, called *arcs*, from $V \times V$. If $uv = (u, v)$ is an arc, we write $uv \in D$ and say that u is *adjacent to* v, v is *adjacent from* u, uv is *incident from* u and *incident to* v. A *graph* G consists of a set V of points together with a set (repetitions not permitted) X of unordered pairs of points, called *lines*. If there is a line between u and v, we denote it by uv or vu. Two digraphs D and E are *isomorphic* ($D \cong E$) if there is a one-to-one correspondence between their point sets which preserves adjacency. Such a correspondence between $V(D)$ and $V(D)$ is an *automorphism* of D.

A *walk* in a digraph is a sequence $W = \langle x_1, \cdots, x_n \rangle$ of arcs, where $x_i = u_{i-1}u_i$. We may abbreviate this and write $W = u_0 u_1 \cdots u_n$. A digraph D is *strong*, or *strongly connected*, if there is a walk $W = v_1 \cdots v_n v_1$ such that each point of D appears at least once as one of the v_i.

In much of what follows, we will be dealing with maps between digraphs. Although it will be convenient to treat these as mappings between the point sets of the digraphs, we will provide a slightly unconventional definition (see [3]). The *reflexive closure* D^R of a digraph D is the smallest superdigraph of D which has a loop at each point. For a point $u \in D$ let $\vec{S}(u)$ be the set of arcs incident from u, and let $\overleftarrow{S}(u)$ be the set of arcs incident to u. Given digraphs D and E, by a *mapping from D onto E* we will mean a mapping $\phi : X(D) \to X(E^R)$ which is onto $X(E)$ and which satisfies: for each $u \in V(D)$ there is a $u' \in V(E)$ such that $\phi(\vec{S}(u)) \subset \vec{S}(u')$ and $\phi(\overleftarrow{S}(u)) \subset \overleftarrow{S}(u')$. A consequence of this definition is that ϕ can be considered as a map from $V(D)$ onto $V(E)$, and it will at times be convenient to do so. A mapping $\phi : D \to E$ is said to be *walkwise* (see [10]) if for every walk $\langle x'_1, x'_2, \cdots, x'_n \rangle$ in E there is a walk $\langle x_1, x_2, \cdots, x_n \rangle$ in D such that $\phi(x_i) = x'_i$.

A mapping $\phi : D \to D'$ is an *admissible homomorphism*, or *admissible*, if whenever $\phi(u) = \phi(v)$ and $uw \in D$ there is a point $\bar{w}$ such that $v\bar{w} \in D$ and $\phi(\bar{w}) = \phi(w)$.

THEOREM 1. *Every admissible homomorphism is walkwise.*

Proof. Let $\phi : V(D) \to V(D')$ be admissible, let $W' = w'_1 \cdots w'_n$ be a walk in D', and suppose that $w'_1 \cdots w'_{n-1}$ has a walk preimage $u_1 \cdots u_{n-1}$. Since $w'_{n-1} w'_n \in D'$, there are points $w_{n-1} \in \phi^{-1}(w'_{n-1})$ and $w_n \in \phi^{-1}(w'_n)$ such that $w_{n-1} w_n \in D$. But then, by admissibility, since $\phi(u_{n-1}) = \phi(w_{n-1})$, there is a point u_n in D such that $\phi(u_n) = \phi(w_n) = w'_n$ and $u_{n-1} u_n \in D$. Therefore $u_1 \cdots u_n$ is a walk in D, and the result follows by induction. □

If $M = \langle Q, I, \delta \rangle$ is a machine, the *digraph* $D(M)$ *of* M has for its points the set Q and for its arcs the collection of arcs uv such that for some $x \in I$, $ux = v$; if there are n inputs x_i such that $ux_i = v$, then there are n arcs from u to v. Note that this is not quite the same concept as the *state-transition graph* or *diagram* (see [2, p. 72]), in which the arcs are labeled with input symbols and, when the machine has outputs, the output symbols also appear as labels either on the lines (for a Mealy machine) or on the points (for a Moore machine).

2. Realization with feedback encoding. Let $M = \langle Q, I, \delta \rangle$ and $M' = \langle Q', I', \delta' \rangle$ be machines, and let $h : Q \times I' \to I$ be a map such that for each $q \in Q$ the map $h(q, \cdot) = h_q : I' \to I$ defined by $h_q(x') = h(q, x')$ is one-to-one and onto, and let $\phi : Q \xrightarrow{\text{onto}} Q'$. Then *M realizes M' with feedback encoding* if for all $q \in Q$ and all $x' \in I'$, $\phi(q)x' = \phi(qh(q, x'))$.

As in classical realization, the function h encodes inputs for M' into inputs for M. In this case, however, the machine exerts a measure of control over the encoding process by making the encoding dependent on the current state of the machine. For a simple example, let M and M' be the machines in Fig. 1. If $\phi(q_i) = r_i$ and h is defined by the table

h	a	b
q_1	1	0
q_2	0	1

,

then M realizes M' with feedback encoding.

We first show the necessity of the one-to-one restriction on the feedback encoder h.

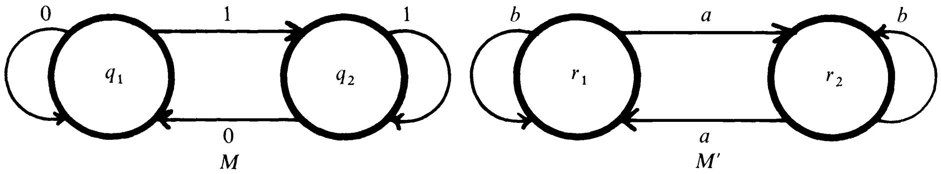


FIG. 1

THEOREM 2. *Let $M' = \langle Q', I', \delta' \rangle$ be any machine, and let $M = \langle Q, I, \delta \rangle$ be a machine such that*

(i) $|Q| \geqq |Q'|$,

(ii) *for each $q_i \in Q$ there is an input x_i such that for all $q_j \in Q$, $q_j x_i = q_i$.*

Then there are maps $\phi : Q \xrightarrow{\text{onto}} Q'$ and $h : Q \times I' \to I$ such that for all $q \in Q$ and $x' \in I'$,

$$\phi(q)x' = \phi(qh(q, x')).$$

Proof. Let ϕ be any map from Q onto Q'. Then if $\phi(q_i)y' = \phi(q_j)$, define $h(q_i, y')$ to be the reset input x_j; there may be many states q_j for which $\phi(q_j) = \phi(q_i)y'$, and any one of these may be chosen. Then for any $q \in Q$ and $x' \in I'$, $\phi(q)x' = \phi(qh(q, x'))$.

In other words, without the one-to-one and onto restriction, we could model any machine by a reset machine, whose transition function is independent of its present state, by, in effect, lumping all the behavior into the h map. While it is often desirable to transfer some of the complexities of a machine to a combinatorial circuit in this way, there is a (perhaps ill-defined) point at which it becomes fatuous: if our purpose is to study the complexities of sequential machines, i.e., machines with memory, it does us no good to define a canonical form in which all the complexity resides in a memoryless component.

Often we wish to realize a machine M' by a submachine of another machine M. In the present context this can be done by choosing a subset $\bar{I}$ of I and restricting h so that for each q, h_q is a one-to-one map onto $\bar{I}$.

THEOREM 3. *Let M_3 realize M_2 with feedback encoding, and let M_2 realize M_1 with feedback encoding. Then M_3 realizes M_1 with feedback encoding.*

Proof. Let the realization of M_2 by M_3 be defined by $\Psi : Q_3 \to Q_2$ and $g : Q_3 \times I_2 \to I_3$, and that of M_1 by M_2 be defined by $\phi : Q_2 \to Q_1$ and $h : Q_2 \times I_1 \to I_2$. For any $q_3 \in Q_3$, let q_2 be $\Psi(q_3)$ and let q_1 be $\phi(q_2)$. For any $x_1 \in I_1$,

$q_1x_1 = \phi(q_2h(q_2, x_1))$. But $h(q_2, x_1) \in I_2$, so $\phi(q_2h(q_2, x_1)) = \phi(\Psi(q_3g(q_3, h(q_2, x_1))))$. Let τ be the composition $\phi \circ \Psi$ and define $f: Q_3 \times I_1 \to I_3$ by $f(q_3, x_1) = g(q_3, h(\Psi(q_3), x_1))$. If we can show that for each $q_3 \in Q_3$ the map $f_q: I_1 \to I_3$ is one-to-one and onto, then τ and f will define a realization with feedback encoding of M_1 by M_3.

For each q_3 and each $x_3 \in I_3$, we know that there is an $x_2 \in I_2$ such that $g(q_3, x_2) = x_3$. Also, there is an $x_1 \in I_1$ such that $h(\Psi(q_3), x_1) = x_2$. Thus $f(q_3, x_1) = g(q_3, h(\Psi(q_3), x_1)) = x_3$, and so f_{q_3} is onto. Now, since $f_{q_3}(x_1) = g(q_3, h(\Psi(q_3), x_1))$ and since both g_{q_3} and $h\Psi_{(q_3)}$ are one-to-one, it follows immediately that f_{q_3} is one-to-one. □

The *outdegree* od(u) of a point u in a digraph D is the number of arcs uv in D; a digraph is *outregular* if all its points have the same outdegree. The following lemma is proved in [5].

LEMMA. *Let $\phi: D \to D'$ be admissible, and suppose that D and D' are both outregular of degree d. Then for any u and v, the number n of arcs from u to $\phi^{-1}(\phi(v))$ is the same as the number n' of arcs from $\phi(u)$ to $\phi(v)$.*

THEOREM 4. *If M realizes M' with feedback encoding, then there is an admissible homomorphism from $D(M)$ onto $D(M')$. Conversely, if there is an admissible homomorphism from $D(M)$ onto $D(M')$, and if $|I| = |I'|$, then M realizes M' with feedback encoding.*

Proof. Let the realization be defined by maps $\phi: Q \to Q'$ and $h: Q \times I' \to I$. Suppose that $\phi(u) = \phi(v)$ and that $uw \in D(M)$. We must show that there is a $\bar{w}$ such that $v\bar{w} \in D(M)$ and $\phi(\bar{w}) = \phi(w)$. Since $uw \in D(M)$, there is an input x such that $ux = w$. Since $h_u: I' \to I$ is onto, there is an $x' \in I'$ such that $h_u(x') = x$. Then $\phi(u)x' = \phi(w)$, so that $\phi(v)x' = \phi(w)$.

Let $\bar{w} = vh_v(x')$. Then $\phi(\bar{w}) = \phi(w)$, and, since $\bar{w} = vh_v(x')$, $v\bar{w}$ is certainly an arc of $D(M)$. Therefore, ϕ considered as a map from $D(M)$ onto $D(M')$ will be admissible once we can verify that it is onto $X(D(M))$. Since M realizes M' with feedback encoding, if there is a single arc $u'v' \in D(M')$, then this arc must have a preimage arc in $D(M)$. But if there is more than one arc between u' and v' in $D(M')$, then there are inputs, say $x'_1, \cdots, x'_r$, such that $u'x'_1 = u'x'_2 = \cdots = u'x'_r = v'$. Then for each $u \in \phi^{-1}(u')$, there are $v_1, \cdots, v_r$ such that $uh_u(x'_i) = v_i$ and $\phi(v_i) = v'$ for $i = 1, \cdots, r$. Then each arc $uv_i \in D(M)$ is a preimage for one of the arcs between u' and v'. While it may be the case that the v_i are not all distinct, since h_u is one-to-one we are assured that there will be r distinct arcs from u to points in $\phi^{-1}(v')$. Thus, $\phi: D(M) \to D(M')$ is admissible.

For the converse, suppose ϕ is an admissible map from $D(M)$ onto $D(M')$ and that $|I| = |I'|$. Let u be a state of M and suppose that $\phi(u)x' = w'$. Since ϕ is a mapping from $D(M)$ to $D(M')$, there must be states $\bar{u}$ and $\bar{w}$ in M such that $\phi(\bar{u}) = \phi(u)$, $\phi(\bar{w}) = w'$, and $\bar{u}\bar{w} \in D(M)$, But then, by admissibility, there is a $w \in M$ such that $\phi(w) = w'$ and $uw \in D(M)$. Thus we define $h_u(x')$ to be that input x which induces the arc uw. It is clear that once we verify that this assignment can be done so that $h_u: I' \to I$ is one-to-one and onto, the pair (ϕ, h) will define a realization with feedback encoding. But the one-to-one and onto properties will be satisfied just when the cardinality of the set of arcs from u' to w' is the same as the cardinality of the set of arcs from u to points in $\phi^{-1}(w')$, and this follows from the lemma. Thus, M realizes M' with feedback encoding. □

COROLLARY. *If M isomorphically realizes M' with feedback encoding, then $D(M)$ is isomorphic to $D(M')$.*

It is immediate that if M realizes M', then M realizes M' with feedback encoding. We thus have the following additional corollary.

COROLLARY. *If ϕ is an SP-homomorphism from M to M' and if $|I| = |I'|$, then ϕ is an admissible homomorphism from $D(M)$ to $D(M')$.*

Hedetniemi [7] asked whether the elementary homomorphisms of autonomous machines, as defined by Yoeli and Ginzburg [11], are always walkwise. That this is so follows from Theorem 1 and the preceding corollary.

COROLLARY. *Every SP-homomorphism is walkwise.*

A number of properties of admissible homomorphisms are presented in [5]. One which is particulary surprising involves the notion of an admissible partition; an *admissible partition* of a digraph D is a partition induced on $V(D)$ by some admissible homomorphism with domain D. Unlike the partitions with substitution property for automata, each of which is, of course, an admissible partition, the admissible partitions do not form a lattice. The join of two admissible partitions is admissible, but the meet may not be. Referring to Fig. 2, $\pi_1 = \{\overline{uv}; \overline{w_1w_3}; \overline{w_2w_4}\}$ and $\pi_2 = \{\overline{uv}; \overline{w_1w_4}; \overline{w_2w_3}\}$ are admissible, but their meet $\pi_1 \wedge \pi_2 = \{\overline{uv}; \overline{w}_1; \overline{w}_2; \overline{w}_3; \overline{w}_4\}$ is not.

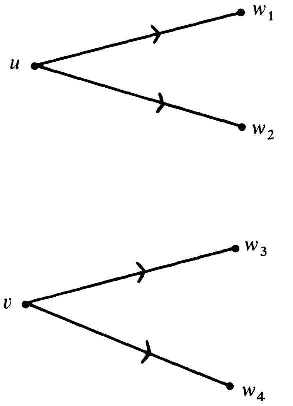


FIG. 2

As in the classical theory, we can study realization of machines by cascades of other machines, where, of course, the realization will involve feedback encoding. The results will be seen to be generalizations of the results for realization without feedback encoding.

Given an ordered n-tuple $x = \langle x_1, x_2, \cdots, x_n \rangle$, define $\text{proj}_j x$ to be the jth element, x_j.

THEOREM 5. *If a cascade $M_1 \circ_Z M_2$ realizes M isomorphically with feedback encoding, then there is an admissible homomorphism from $D(M)$ onto $D(M_1)$.*

Proof. Let the realization with feedback encoding be defined by maps $\phi: Q_1 \times Q_2 \to Q$ and $h: (Q_1 \times Q_2) \times I \to I_1$, so that for any input $x \in I$ and any state (q_1, q_2) in $Q_1 \times Q_2$, $\phi((q_1, q_2))x = \phi((q_1, q_2)h((q_1, q_2), x))$. Let $\rho = \phi^{-1}$, and for a state $q \in Q$, let $q(j)$ be the state $\text{proj}_j\, \rho(q)$. Now, define an equivalence relation on the states of Q by $q \equiv q'$ if and only if $q(1) = q'(1)$.

Suppose that $q_1 \equiv q_2$ and that for some $x \in I$, $q_1 x = q_3$. Let

$$S_1 = \{\text{proj}_1\, \rho(q_1 y) | y \in I\} = \{q_1(1) h(\rho(q_1), y) | y \in I\}$$

and

$$S_2 = \{\text{proj}_1\, \rho(q_2 y) | y \in I\} = \{q_2(1) h(\rho(q_2), y) | y \in I\}.$$

Since $q_1(1) = q_2(1)$ by hypothesis, and since $\{h(\rho(q_1), y) | y \in I\} = \{h(\rho(q_2), y) | y \in I\}$, it follows immediately that

$$S_1 = \{q_1(1) h(\rho(q_1), y)\} = \{q_1(1) h(\rho(q_2), y)\} = S_2.$$

Thus there is some state $q_4 \in Q$ and an input $\bar{y} \in I$ such that $q_2 \bar{y} = q_4$ and $q_4(1) = q_3(1)$; i.e., $q_4 \equiv q_3$. Thus $\text{proj}_1\, \rho : Q \to Q_1$ defines an admissible homomorphism from $D(M)$ onto $D(M_1)$. □

This also follows from the Hartmanis and Stearns results [6], for M_1 induces an SP-partition on the cascade, and this converts to an admissible partition on M.

COROLLARY. *If M' is a submachine of a cascade $M_1 \circ_Z M_2$ such that $\{\text{proj}_1 (q') | q' \in Q'\} = Q_1$ and if M' realizes a machine M isomorphically with feedback encoding, where $|I'| = |I_1|$, then there is an admissible homomorphism from $D(M)$ to $D(M_1)$.*

THEOREM 6. *Suppose there is an admissible homomorphism from $D(M)$ to $D(M_1)$, where $|I| = |I_1|$. Then there is a machine M_2 such that M can be isomorphically realized with feedback encoding by a submachine of a cascade $M_1 \circ_Z M_2$.*

Proof. Suppose that ϕ is an admissible homomorphism from $D(M)$ onto $D(M_1)$. We must exhibit a machine M_2 and a connecting map Z such that $M_1 \circ_Z M_2$ isomorphically realizes M with feedback encoding. Now, the map ϕ induces a partition π on the states of M; suppose that there are r partition classes P_i and that the largest of these contains s states. Number the states in each P_i as $1, 2, \cdots, n_i$, where $n_i \leqq s$, and form a new partition π' with s blocks B_i, where B_i consists exactly of those states, at most one from each P_j, which were numbered i. Then each state $q \in M$ can be associated with exactly one pair (P_i, B_j) of blocks such that $q \in P_i \cap B_j$. Clearly, the blocks P_i are identifiable with the states of M_1. We know from Theorem 4 that there is a map $h : Q \times I_1 \to I$ which is one-to-one and onto and which, taken together with ϕ, defines a realization with feedback encoding of M_1 by M. We can represent the state of M associated with blocks P_i and B_j by q_{ij}. Now suppose under some input x, $q_{ij} x = q_{km}$. Then $\phi(q_{ij}) h_{q_{ij}}^{-1}(x) = \phi(q_{km})$.

Machine M_2 will have the blocks B_j for its states. Now if $q_{ij} x = q_{km}$ and $x' = h_{q_{ij}}^{-1}(x)$, we will want $(P_i, B_j)x' = (P_i x', B_j Z(P_i, x')) = (P_k, B_j Z(P_i, x')) = (P_k, B_m)$. We already know that $P_i x' = P_k$, so we must define M_2 and Z in such a way that $B_j Z(P_i, x') = B_m$. Let M_2 have r inputs y_i. If $(P_i, B_j)x' = (P_k, B_m)$, then in M_2 we define $B_j y_k = B_m$; since not every pair (P_i, B_j) defines a state of M, this may leave us with don't-care conditions, which can be assigned arbitrarily. Now we define $Z(P_i, x') = y_k$, where $P_i x' = P_k$. With these definitions, it follows that if $q_{ij} x = q_{km}$, then $(P_i, B_j)x' = (P_i x', B_j Z(P_i, x')) = (P_k, B_j y_k) = (P_k, B_m)$. If ρ is the map which assigns to (P_i, B_j) the state q_{ij}, then for each $x \in I$, $\rho((P_i, B_j))x = \rho((P_i, B_j) h_{q_{ij}}^{-1}(x))$. Thus if $g : (\{P_i\} \times \{B_j\}) \times I \to I_1$ is defined by $g((P_i, B_j), x) = h_{q_{ij}}^{-1}(x)$, then ρ and g will define an isomorphic realization with feedback encoding of M by that submachine of $M_1 \circ_Z M_2$ consisting of the pairs of states (P_i, B_j) for which $P_i \cap B_j \neq \phi$. □

Combining these results, we have Theorem 7 as follows.

THEOREM 7. *A machine M can be isomorphically realized with feedback encoding by a submachine M' of a cascade $M_1 \circ_Z M_2$, where $|I'| = |I_1|$, if and only if there is an admissible homomorphism from $D(M)$ onto the subdigraph of $D(M_1)$ induced by the states in $\{\text{proj}_1 (q') | q' \in Q'\}$.*

COROLLARY. *A machine M can be isomorphically realized with feedback encoding by a submachine of a cascade $M_1 \circ_Z M_2$ if and only if there is an admissible homomorphism from $D(M)$ onto the digraph of a submachine of M_1.*

Having established Theorem 7, it is natural to investigate what happens if the feedback to the h map in a realization with feedback encoding by a cascade is from only one of the components of the cascade. Unfortunately, as we shall see, there does not seem to be too much to say about such situations. Following Fleck, et al. [2], if $D(M) \cong D(M')$ we say that M and M' are *graph-isomorphic.*

THEOREM 8. *Let M and M_1 be machines. Then M has an* SP-*homomorphic image which is graph-isomorphic to M_1 if and only if M can be isomorphically realized with feedback encoding by a cascade $M_1 \circ_Z M_2$ in such a way that the encoding map h has domain $Q_1 \times I$.*

Proof. If M_1 is graph-isomorphic to an SP-homomorphic image $\overline{M}_1$ of M, then in any isomorphic realization of M by a cascade $\overline{M}_1 \circ_Z M_2$, $\overline{M}_1$ can be replaced by M_1 together with the appropriate feedback encoder, which of course is independent of the state of M_2.

On the other hand, let the realization with feedback encoding be defined by the maps $\phi: Q_1 \times Q_2 \to Q$ and $h: Q_1 \times I \to I_1$. Let $\rho = \text{proj}_1 \phi^{-1}$ be the admissible homomorphism guaranteed by Theorem 5; $\rho: Q \to Q_1$. For each state q_i, write $\phi^{-1}(q_i) = (q_i(1), q_i(2))$. Now, suppose that $\rho(q_1) = \rho(q_2)$, $q_1 x = q_2$ and $q_3 x = q_4$. Then

$$\phi((q_1(1), q_1(2)))h(q_1(1), x) = \phi((q_2(1), q_2(2)))$$

and

$$\phi((q_3(1), q_3(2)))h(q_3(1), x) = \phi((q_4(1), q_4(2))),$$

where $q_2(1) = q_1(1)h(q_1(1), x)$ and $q_4(1) = q_3(1)h(q_3(1), x)$. But since $\rho(q_1) = \rho(q_3)$, it follows that $\rho(q_1 x) = \rho(q_3 x)$, so that ρ induces an SP-partition on M, and hence an SP-homomorphism $\overline{\phi}$ to a machine $\overline{M}_1$. When $\overline{\phi}$ is reinterpreted as a digraph mapping, it is identical to ρ; thus M_1 is graph isomorphic to $\overline{M}_1$. □

We will give two examples to illustrate feedback encoded realizations by cascades. In each example, there will be an advantage to using realization with feedback encoding. In one example, there is no cascade realization without feedback encoding, while in the other, we present a cascade of a two-state machine and a three-state machine; without feedback encoding it would be necessary to use a cascade of a two-state machine with a four-state machine.

Example 1. Let M be the machine

M	0	1
1	1	5
2	2	6
3	4	3
4	6	2
5	5	1
6	3	4

.

Notice that M has the SP-partition $\{\overline{15}; \overline{2346}\}$. Consider the following two machines and connecting map Z:

M_1	0	1
q_1	q_1	q_2
q_2	q_2	q_1

,

M_2	0	1
r_1	r_1	r_1
r_2	r_2	r_3
r_3	r_3	r_2

,

Z	0	1
q_1	0	1
q_2	1	0

.

Then the cascade $M_1 \circ_Z M_2$ is

$M_1 \circ_Z M_2$	0	1
$P_1 = q_1r_1$	$q_1r_1 = P_1$	$q_2r_1 = P_5$
$P_2 = q_1r_2$	$q_1r_2 = P_2$	$q_2r_3 = P_6$
$P_3 = q_1r_3$	$q_1r_3 = P_3$	$q_2r_2 = P_4$
$P_4 = q_2r_2$	$q_2r_3 = P_6$	$q_1r_2 = P_2$
$P_5 = q_2r_1$	$q_2r_1 = P_5$	$q_1r_1 = P_1$
$P_6 = q_2r_3$	$q_2r_2 = P_4$	$q_1r_3 = P_3$

.

Under the maps $\phi: P_1 \to i$ and h,

h	0	1
P_1	0	1
P_2	0	1
P_3	1	0
P_4	0	1
P_5	0	1
P_6	1	0

,

$M_1 \circ_Z M_2$ realizes M with feedback encoding.

Example 2. Let M_2 be as in the previous example, and let M_1 instead be isomorphic to M_2, with states $\{q_i | i = 1, 2, 3\}$. Under the connecting map

$$Z(q_i, x) = \begin{cases} x & \text{if } i \neq 3, \\ 1 - x & \text{if } i = 3, \end{cases}$$

the cascade $M_1 \circ_Z M_2$ is:

$M_1 \circ_Z M_2$	0	1
q_1r_1	q_1r_1	q_2r_2
q_1r_2	q_1r_2	q_2r_3
q_1r_3	q_1r_3	q_2r_1
q_2r_1	q_2r_1	q_3r_2
q_2r_2	q_2r_2	q_3r_3
q_2r_3	q_2r_3	q_3r_1
q_3r_1	q_3r_2	q_1r_1
q_3r_2	q_3r_3	q_1r_2
q_3r_3	q_3r_1	q_1r_3

.

Now, with the maps $\phi(q_ir_j) = 3(i - 1) + j$ and

$$h(q_ir_j, x) = \begin{cases} x & \text{if } j \neq 3, \\ 1 - x & \text{if } j = 3, \end{cases}$$

the following machine M is isomorphically realized with feedback encoding by $M_1 \circ_Z M_2$:

M	0	1
1	1	5
2	2	6
3	4	3
4	4	8
5	5	9
6	7	6
7	8	1
8	9	2
9	3	7

However, in this case, M has no nontrivial SP-partitions.

It should be clear by this point that whether or not a machine M has an isomorphic cascade realization with feedback encoding depends on the structure of $D(M)$. We will now strengthen this impression by giving a large class of families F of digraphs such that if some cascade has a submachine whose digraph is in F, then one component of the cascade also has a submachine whose digraph is in F.

Recall the following common notations. If integer n divides integer m, we write $n|m$, and if not we write $n\nmid m$. The greatest common divisor of a set $\{i_1, \cdots, i_n\}$ of integers is denoted $(i_1, \cdots, i_n)$.

The following simple result from number theory will be useful.

If $(i_1, \cdots, i_t, n) = g$, *then for any* r *there is a solution to* $\sum_{j=1}^{t} w_j i_j \equiv r \pmod{n}$ *if and only* if $g|r$.

Now let D be a digraph with n points in which each point has outdegree t, and suppose that there is an automorphism g of D which is an n-cycle; that is, for any point $v \in D$, the points $v, g(v), g^2(v), \cdots, g^{n-1}(v)$ are all distinct. Label the points by choosing v_o arbitrarily and then setting $v_j = g^j(v_o)$. Let $S = \{i_1, \cdots, i_t | i_k \leqq i_{k+1}\}$, where the arcs from v_o are $v_o v_{i_1}, v_o v_{i_2}, \cdots, v_o v_{i_t}$.

LEMMA. (i) *For any* $v_j \in D$, *the arcs from* v_j *are* $v_j v_{j+i_1}, v_j v_{j+i_2}, \cdots, v_j v_{j+i_t}$.

(ii) *If* $m|n$, *there is a diagraph* $D'_m(D)$ *with* m *points which is an admissible homomorphic image of* D *and whose automorphism group contains an* m*-cycle.*

Proof. (i) If $v_o v_{i_k} \in D$, then $g^j(v_o)g^j(v_{i_k}) \in D$; that is, $v_j v_{j+i_k} \in D$. But this accounts for t arcs from v_j, and there are only t.

(ii) Define $D'_m(D)$ to be a digraph whose points are the m sets $w_i = \{v_i, v_{i+m}, v_{i+2m}, \cdots\}$, where $w_i w_j \in D'_m(D)$ if and only if there are $v_\alpha \in w_i$, $v_\beta \in w_j$ such that $v_\alpha v_\beta \in D$. The w_i, as subsets of $V(D)$, are the orbits of the automorphism $g^{n/m}$. Consider the map $\phi: D \to D'_m(D)$, which takes each v_j to that set w_i of which it is an element. If $\phi(v_i) = \phi(v_j)$, then some power g^* of $g^{n/m}$ maps v_i to v_j. If $v_i u \in D$, then $g^*(v_i)g^*(u) = v_j g^*(u) \in D$, and, since u and $g^*(u)$ belong to the same orbit of $g^{n/m}$, $\phi(u) = \phi(g^*(u))$. Thus ϕ is admissible. □

By (i), we completely specify the digraph by giving the number of points, n, and the set S. Thus, we write $D = D(i_1, i_2, \cdots, i_t, n)$. Also, if M is a machine with $D(M) = D$, then we write $D'_m(M)$ for $D'_m(D)$.

THEOREM 9. *If* M *with digraph* $D(M) = D(i_1, \cdots, i_t, n)$ *is a submachine of a cascade* $M_1 \circ_Z M_2$ *and if the* g.c.d.$(i_1, \cdots, i_t, n) = 1$, *then there is a machine* M'_m *with digraph* $D'_m(M)$ *such that either* (i) $m|n$, $m > 1$, *and* M'_m *is a submachine of* M_1, *or*

(ii) *$m = n$ and M'_m is a submachine of M_2.*

Proof. Let M be a submachine of $M_1 \circ_Z M_2$. By Theorem 7, there is an admissible homomorphism ϕ from $D(M_1 \circ_Z M_2)$ onto $D(M_1)$— for the purposes of this proof we will be concerned only with the restriction of ϕ to M.

Assume first that $|\phi(M)| > 1$. Note that M has digraph $D(i_1, \cdots, i_t, n)$, and label its states cyclically as $q_0, \cdots, q_{n-1}$. We will first show that unless ϕ is an isomorphism from M, for some (minimal) $k > 0$, $\phi(q_0) = \phi(q_k)$ and, for all $1 \leqq j \leqq t$, $\phi(q_{i_j}) = \phi(q_{k+i_j})$.

Let k be the smallest integer for which there is a state q_i such that $\phi(q_i) = \phi(q_{i+k})$: by symmetry we can take $i = 0$, relabeling cyclically if necessary. Since q_0 is adjacent to q_s for each element s of the set $\Gamma = \{i_1, \cdots, i_t\}$, it follows from the admissibility of ϕ that for each $s \in \Gamma$, $\phi(q_s)$ is equal to the image of some point adjacent from q_k. Thus, by the lemma, for each $1 \leqq r \leqq t$ there is an $s_r \in \Gamma$ such that $\phi(q_{i_r}) = \phi(q_{k+s_r})$. If, for each such r, $s_r = i_r$, then the claim holds. Otherwise, let $1 \leqq R \leqq t$ be the smallest integer such that $s_R < i_R$. Then $\phi(q_{i_R}) = \phi(q_{k+s_R})$ and, since $s_R - i_R < 0$, $(k + s_R) - i_R < k$, a contradiction to the minimality of k: this verifies the claim.

Since $\phi(q_0) = \phi(q_k)$ and, for all $s \in \Gamma$, $\phi(q_s) = \phi(q_{s+k})$, it follows by a simple inductive argument that if $\alpha = w_1 i_1 + \cdots + w_t i_t$, then $\phi(q_\alpha) = \phi(q_{k+\alpha})$, for any nonnegative integers w_j. But $(i_1, \cdots, i_t, n) = 1$ by hypothesis, so that for any c there is a solution to $\sum w_j i_j \equiv c \pmod{n}$. It follows that for all c and d, $\phi(q_c) = \phi(q_{c+kd})$.

Finally, for $c > b$, $\phi(q_b) = \phi(q_c)$ must imply that $c - b$ is a multiple of k. For otherwise, we can write $c - b = dk + f$, $0 < f < k$, and conclude from the fact that $\phi(q_b) = \phi(q_{b+dk})$ that $\phi(q_{b+dk}) = \phi(q_c)$, a contradiction to the minimality of k since $c - (b + dk) = f < k$.

Clearly k must divide n, for otherwise it follows from the preceding that all points q_j would have the same image. Thus, $D(\phi(M)) \cong D_k(M)$.

We consider now the case in which all points q_j of M do have the same image under ϕ. We can write the states of M as $q_j = (q^*, q_j(2))$, where $q^* = \phi(q_0)$ and the $q_j(2)$ are states of M_2 which induce a submachine with digraph isomorphic to $D(M)$. This digraph is induced in the following way: if x is the input which takes q_c to q_d in M, then the input which takes $q_c(2)$ to $q_d(2)$ in M_2 is $Z(q^*, x)$. Therefore, in fact, the states $q_j(2)$ induce a submachine of M_2 which is isomorphic to M. □

COROLLARY. *If the machine M'_m has $D(M'_m) = D'_m(M) = D(j_1, \cdots, j_t, m)$, then $(j_1, \cdots, j_t, m) = 1$.*

Proof. The parameter j_k for M_m is the residue of i_k modulo m; that is, $0 \leqq j_k \leqq m - 1$ and $j_k \equiv i_k \pmod{m}$. Now suppose $g = (j_1, \cdots, j_t, m)$. Then $g|m$ and for each k, $g|j_k$. But since $j_k \equiv i_k \pmod{m}$, there is an $l_k \geqq 0$ such that $i_k = j_k + ml_k$, so that $g|i_k$. Also, since $g|m$ and $m|n$, $g|n$. Thus $g|(i_1, \cdots, i_t, n)$. Hence $g = 1$. □

We now proceed to consider the complexity of cascade realizations, with feedback encoding. These results are motivated by Zeigler's [13] generalization of the Burks–Wang conjecture. Up to now, we have been concerned only with cascades of two machines, but it is clear that we can generalize our results to more general iterated cascades.

Consider, for example, Theorem 9. Suppose that M, as defined in the theorem, is a submachine of a cascade $\prod N_i$. Generalized cascade is a binary procedure, so we can find the major connective and write $\prod N_i = L_1 \circ_{Z_1} L_2$. The theorem guarantees the existence of a machine M'_m in either L_1 or L_2. Since the parameters for M'_m satisfy $(j_1, \cdots, j_t, m) = 1$, if that component of the cascade $L_1 \circ_Z L_2$ which contains M'_m is itself a cascade, we can repeat the procedure. Eventually we will find a submachine M'_r with digraph $D'_r(M)$ of some component N_s of the cascade $\prod N_i$. This proves the following corollary.

COROLLARY. *If M with digraph $D(M) = D(i_1, \cdots, i_t, n)$ is a submachine of a generalized cascade $\prod N_j$, and if $(i_1, \cdots, i_t, n) = 1$, then there is a submachine M'_r with digraph $D'_r(M)$ of some component N_k of the cascade, where $r > 1$ and $r|n$.*

Suppose now that we have a cascade $N = \prod N_i$, and let machine N_i have p_i states. We define the *size* of the cascade, denoted size(N), to be the max $\{p_i\}$. Let S_σ be the collection of all generalized cascades N having size$(N) \leqq \sigma$. We will show that S_σ is not *universal* for isomorphic realization with feedback encoding, i.e., that (for any σ) there is a machine M which cannot be isomorphically realized with feedback encoding by any element of S_σ. In fact, we will show a slightly stronger result. First, however, we note that this statement would not be true without the restrictions we have placed on the h maps, as shown by the following corollary to Theorem 2.

COROLLARY. *Let $M = \langle Q_M, I_M, \delta_M \rangle$ be any machine. Then there is a cascade $N = \langle Q_N, I_N, \delta_N \rangle = \prod N_i$ of size 2 and maps $\rho: Q_M \to Q_N$, $h: Q_N \times I_M \to I_N$, where ρ is one-to-one, such that for each $q \in Q_M$ and $x \in I_M$, $qx = \rho^{-1}(\rho(q)h(\rho(q), x))$.*

The proof of this corollary is somewhat tedious (see [3]). It proceeds by letting N_r be the complete reset machine with 2^n states and 2^n inputs, and then showing that N_r is a cascade of size 2. Define N'_r from N_r by replacing each input by two inputs with the same action. Then for $n = \{\log_2 |Q_M|\}$, N_n is isomorphic to $N'_{n-1} \circ_Z N_1$ and for each r, N'_r is isomorphic to $N'_{r-1} \circ_Z N_1$.

We now return our attention to restricted feedback encodings.

THEOREM 10. *For any σ and any t, there is a t-input machine M which cannot be isomorphically realized with feedback encoding by any submachine of any element of S_σ.*

Proof. Let p be a prime larger than both σ and t and consider any p-state machine M with digraph $D(i_1, \cdots, i_t, p)$. Such a machine can, in fact, always be constructed such that $i_t \neq 0$. If M is isomorphically realized by a submachine N of an element $\prod N_i$ of S_σ, then N also has digraph $D(i_1, \cdots, i_t, p)$. Then since p is prime, Theorem 9 assures us that at least one component N_i of the cascade contains a submachine with digraph $D'_p(N) \cong D(N)$. But this submachine has $p > \sigma$ states so that N_i has more than σ states, and hence size$(\prod N_i) > \sigma$. □

COROLLARY. *For any σ and any t, there is a t-input machine M which cannot be isomorphically realized by a submachine of any element of S_σ.*

In the classical theory of realization, a similar result holds for both homomorphic realization and for simulation [13]. We will see in the next section that the corresponding result does not hold for simulation with feedback encoding. For homomorphic realizations, we can prove a result slightly more restricted than Theorem 10.

If a cascade $M_1 \circ_Z M_2$ contains an *n-cycle* C_n, i.e., a strong, autonomous, n-state machine, then by the corollary to Theorem 5, M_1 contains an admissible homomorphic image of C_n. It can be shown (see [5]) that any admissible homomorphic image of C_n is some C_m, where $m|n$. Thus, for some $m|n$, M_1 contains C_m. If C_n is defined by input x and has states $\langle q_0, \cdots, q_{n-1}\rangle$ consider the string $y = Z(q_0(1), x)Z(q_1(1), x) \cdots Z(q_{m-1}(1), x)$. It can be shown (next lemma) that the states, in M_2,

$$q_0(2), q_0(2)y, q_0(2)y^2, \cdots, q_0(2)y^{n/m-1}$$

are all distinct, and that $q_0(2)y^{n/m} = q_0(2)$. The string y thus induces a *string cycle* in M_2; we call n/m the *string period*. If p is a prime which divides n, then either $p|m$ or $p|(n/m)$, so one of M_1 or M_2 must have at least p states. Thus, size$(M_1 \circ_Z M_2) \geqq p$. We would like to be able to continue this process and show that if any cascade M contained C_n, and $p|n$, then size$(M) > p$. If $p|m$ and M were a cascade, we could indeed continue, since M_1 contains a cycle C_n. At some point in this "unfolding", however, we may come across the situation where the machine which is guaranteed to have at least p states is the tail component of a cascade, and then we have the problem of decomposing a string cycle of string period t, where $p|t$, into string cycles, one of whose string periods is a multiple of p. This can always be done.

LEMMA (Zeigler [13]). *Let M be a cascade $M_1 \circ_Z M_2$ which contains a string cycle of period n. Then there is some $k|n$ such that M_1 contains a string cycle of string period k and M_2 contains a string cycle of string period n/k.*

We are now ready to state a complexity result for homomorphic realization with feedback encoding.

THEOREM 11. *For any σ, there is a machine which cannot be homomorphically realized with feedback encoding by any submachine of any element of S_σ.*

Proof. Let M be C_p, where $p > \sigma$ is a prime. If a submachine N of $\prod N_i \in S_\sigma$ homomorphically realizes M, then, by the corollary to Theorem 4, $D(N)$ is an admissible homomorphic preimage of C_p. By the restrictions on the feedback encoder, for some n, N is C_{np}. Applying the lemma as outlined above, we can conclude that one of the N_i has at least p states, contradicting the hypothesis that $p > \sigma$. $\square$

We stated that this result is more restricted than Theorem 10. In fact, since for autonomous machines the notions of homomorphism and admissible homomorphism coincide, it really says nothing new. The distinction between Theorem 10 and 11 is that we have no detailed knowledge about admissible homomorphic preimages of complex structures, such as the digraphs $D(i_1, \cdots, i_t, n)$ of Theorem 10. Undoubtedly there is some relationship between these digraphs, their admissible homomorphic preimages, and the admissible homomorphic images of these, but what this might be is unclear at present.

3. Simulation with feedback encoding.

In this section we briefly examine the concept of simulation with feedback encoding. We also introduce two semigroups, one of which is the S^*-semigroup of Fleck et al. [2], and show their relationship to realization and simulation with feedback encoding. Many of the proofs in this section are quite similar, and only a few will be given; the rest of the proofs in this section can be found in [3].

If $M = \langle Q, I, \delta\rangle$ and $M' = \langle Q', I', \delta'\rangle$ are machines, then M *simulates* M' *with feedback encoding* if there are maps

$$\phi : Q \xrightarrow{\text{onto}} Q' \quad \text{and} \quad h : Q \times I' \longrightarrow I^+$$

such that for each $q \in Q$ and $x' \in I'$, $\phi(q)x' = \phi(qh(q, x'))$.

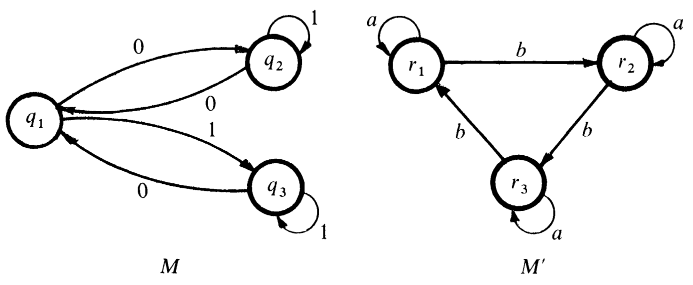


FIG. 3

If M and M' are the machines in Fig. 3, it is clear from Theorem 4 that M does not realize M' with feedback encoding. But the pair of functions (ϕ, h), where $\phi(q_i) = r_i$ and h has the following table, do define a simulation with feedback encoding

h	a	b
q_1	00	0
q_2	1	01
q_3	1	0

.

Let S and S' be semigroups with zero, where, without confusion, we will use the same symbol, 0, for both the zero of S and the zero of S'. A map $\Phi: S \to S'$ is a *zero-free homomorphism* if it satisfies:

1. $\ker(\Phi) = \{0\}$;
2. if $ab \neq 0$, then $\Phi(a)\Phi(b) = \Phi(ab)$;

3(i). if $\Phi(a)b' \neq 0$, then there is some $b \in \Phi^{-1}(b')$ such that $ab \neq 0$;
3(ii). if $a'\Phi(b) \neq 0$, then there is some $a \in \Phi^{-1}(a')$ such that $ab \neq 0$.

We will see examples of zero-free homomorphisms which are not homomorphisms later. For the present, we give some basic properties of zero-free homomorphisms.

THEOREM 12. (a) *Let S_1, S_2 and S_3 be semigroups, and let $\Phi: S_1 \to S_2$ and $\Psi: S_2 \to S_3$ be zero-free homomorphisms. Then*

(i) *if $\Phi(a)\Phi(b) = 0$, then $ab = 0$;*
(ii) *$\Psi\Phi: S_1 \to S_3$ is a zero-free homomorphism;*
(iii) *if Φ is one-to-one, then Φ is an isomorphism.*
(b) *If $\Phi: S_1 \to S_2$ is a semigroup homomorphism with*

$$\ker(\Phi) = \{0\},$$

then Φ is a zero-free homomorphism.

Let $M = \langle Q, I, \delta\rangle$ be a machine. An ordered three-tuple (s, x, t) is a *triple for M*, or simply a *triple*, if $s, t \in Q, x \in I^+$, and $sx = t$. A triple is *elementary* if the length $l(x) = 1$, i.e., if $x \in I$.

Let $\mathscr{S}(M)$ be the set of all triples of M together with a distinguished zero element 0. We introduce an operation on $\mathscr{S}(M)$ by the following rules:

$$(s, x, t_1)(t_2, y, r) = \begin{cases}(s, xy, r) & \text{if } t_1 = t_2,\\ 0 & \text{if } t_1 \neq t_2;\end{cases}$$

$$b0 = 0b = 0 \quad \text{for each } b \in \mathscr{S}(M).$$

THEOREM 13. *For any machine M, $\mathcal{S}(M)$ is a semigroup.*

Proof. Clearly, the operation is defined for each pair of elements in $\mathcal{S}(M)$. Thus, we need only show associativity. Let b_1, b_2, $b_3 \in \mathcal{S}(M)$. Since 0 commutes with every element of $\mathcal{S}(M)$, if any of the b_i are 0, then certainly $(b_1b_2)b_3 = b_1(b_2b_3) = 0$. Suppose that $b_1 = (q, x, r_1)$, $b_2 = (r_2, y, s_2)$, $b_3 = (s_3, z, t)$. If $r_1 = r_2$ and $s_2 = s_3$, then $b_1b_2 = (q, xy, s_2)$, $b_2b_3 = (r_2, yz, t)$, and $(b_1b_2)b_3 = (q_1, (xy)z, t) = (q, x(yz), t) = b_1(b_2b_3)$. If $r_1 \neq r_2$ and $s_1 \neq s_2$, then $b_1b_2 = b_2b_3 = 0$, so $(b_1b_2)b_3 = b_1(b_2b_3) = 0$. If $r_1 = r_2$ and $s_1 \neq s_2$, then $b_2b_3 = 0$ and $b_1(b_2b_3) = 0$. Now, $b_1b_2 = (q, xy, s_1)$, and so $(b_1b_2)b_3 = 0$. Similarly, if $r_1 \neq r_2$ and $s_1 = s_2$, then $(b_1b_2)b_3 = b_1(b_2b_3) = 0$. Thus $\mathcal{S}(M)$ is a semigroup. □

Suppose that M realizes M' with feedback encoding, where the realization is defined by maps ϕ and h. Although, for each $s \in Q$, h_s is a map from I' to I, h_s can be extended to a map $\bar{h}_s$ from I'^+ to I^+ in the following way. If $x' \in I'$, $\bar{h}_s(x') = h_s(x')$; if x', $y' \in I'^+$, $\bar{h}_s(x'y') = \bar{h}_s(x')\bar{h}_{sh_s(x')}(y')$. We must, of course, verify that if $w'z'$ is a different way of writing $x'y'$, then $\bar{h}_s(w'z') = \bar{h}_s(x'y')$.

LEMMA. (a) *For each $s \in Q$, $\bar{h}_s$ is a function.*

(b) *For each $s \in Q$, $x' \in I'^+$, $\phi(s)x' = \phi(s\bar{h}_s(x'))$.*

(c) *For each $s \in Q$, $x \in I^+$ there is a unique $x' \in I'^+$ such that $\bar{h}_s(x') = x$.*

Proof. (a) We proceed by induction. The result is true for strings of length 1 since h_s is a function, and true for strings of length 2 since, for such a string, there is only one decomposition into smaller strings. Suppose it to be true for strings of length $n - 1$, and let $x' = w'_1 \cdots w'_n$, where each $w'_i \in I'$. Choose $1 \leqq i \leqq j \leqq n - 1$. We must show that

$$\bar{h}_s(w'_1 \cdots w'_i)\bar{h}(s\bar{h}_s(w'_1 \cdots w'_i), w'_{i+1} \cdots w'_n)$$
$$= \bar{h}_s(w'_1 \cdots w'_j)\bar{h}(s\bar{h}_s(w'_1 \cdots w'_j), w'_{j+1} \cdots w'_n).$$

Let $s_1 = \bar{h}_s(w'_1 \cdots w'_i)$. By induction, we can write

$$\bar{h}_{s_1}(w'_{i+1} \cdots w'_n) = \bar{h}_{s_1}(w'_{i+1} \cdots w'_j)\bar{h}_{s_2}(w'_{j+1} \cdots w'_n),$$

where $s_2 = s_1\bar{h}_{s_1}(w'_{i+1} \cdots w'_j)$. Also by induction,

$$\bar{h}_s(w'_1 \cdots w'_i)\bar{h}_{s_1}(w'_{i+1} \cdots w'_j) = \bar{h}_s(w'_1 \cdots w'_j).$$

Thus

$$\bar{h}_s(w'_1 \cdots w'_i)\bar{h}_{s_1}(w'_{i+1} \cdots w'_n) = \bar{h}_s(w'_1 \cdots w'_j)\bar{h}_{s_2}(w'_{j+1} \cdots w'_n),$$

and the two expressions are equal.

(b) Let $s \in Q$ and $x' = w'_1 \cdots w'_n$, where each $w'_i \in I'$. If $l(x') = 1$, then the result holds since $\bar{h}_s(x') = h_s(x')$. Assume the result holds when $n \leqq m - 1$ and let $n = m$. Choose $1 \leqq i \leqq n - 1$. Then

$$\phi(s)x' = [\phi(s)w'_1 \cdots w'_i]w'_{i+1} \cdots w'_n = \phi(s\bar{h}_s(w'_1 \cdots w'_i))w'_{i+1} \cdots w'_n.$$

If $s_1 = s\bar{h}_s(w'_1 \cdots w'_i)$, then

$$\phi(s)x' = \phi(s_1)w'_{i+1} \cdots w'_n = \phi(s_1\bar{h}_{s_1}(w'_{i+1} \cdots w'_n)),$$

which, by part (a), is equal to $\phi(s\bar{h}_s(x'))$,

(c) This follows immediately from part (a) and the fact that it is true for h_s by the properties of a feedback encoder. □

From this point, we will use the symbol h_s for both h_s and $\bar{h}_s$. For a state $s \in Q$ and a string $x \in I^+$ we will write $h_s^{-1}(x)$ for the unique string $x' \in I'^+$ such that $h_s(x') = x$.

THEOREM 14. *If M realizes M' with feedback encoding, then there is a zero-free homomorphism from $\mathscr{S}(M)$ onto $\mathscr{S}(M')$.*

Proof. Let the realization be defined by maps ϕ and h. If $b = (s, x, t)$ is a triple of M, define $\Phi(b) = (\phi(s), h_s^{-1}(x), \phi(t))$. By the lemma, $\phi(s)h_s^{-1}(x) = \phi(sx) = \phi(t)$, so that $\Phi(b)$ is a triple for M'. Also, define $\Phi(0) = 0$. We will prove that Φ is a zero-free homomorphism from $\mathscr{S}(M)$ onto $\mathscr{S}(M')$. First, we show that Φ is onto. Let $b' = (s', x', t')$ be a triple of M'. Choose $s \in \phi^{-1}(s')$, and let $x = h_s^{-1}(x')$. If $t = sx$, then $\phi(t) = \phi(sx) = \phi(s)x' = t'$, so $\Phi((s, x, t)) = b'$.

Clearly $\ker(\Phi) = \{0\}$. Suppose that $b_1 = (s, x, t)$ and $b_2 = (t, y, r)$, so that $b_1 b_2 = (s, xy, r) \neq 0$. Now

$$\begin{aligned}\Phi(b_1)\Phi(b_2) &= (\phi(s), h_s^{-1}(x), \phi(t)) \cdot (\phi(t), h_t^{-1}(y), \phi(r))\\ &= (\phi(s), h_s^{-1}(x)h_t^{-1}(y), \phi(r)).\end{aligned}$$

But $h_s^{-1}(x)h_t^{-1}(y) = h_s^{-1}(xy)$, so $\Phi(b_1)\Phi(b_2) = (\phi(s), h_s^{-1}(xy), \phi(t)) = \Phi(b_1 b_2)$.

Suppose that $\Phi(b)d' \neq 0$, where $b = (s, x, t)$. Then $\Phi(b) = (\phi(s), x', \phi(t))$, so that for some $y' \in I'^+, r' \in Q', d' = (\phi(t), y', r')$. If $d = (t, h_t(y'), th_t(y'))$, then $\Phi(d) = d'$ and $bd \neq 0$. Similarly, if $d'\Phi(b) \neq 0$, then there is a d such that $\Phi(d) = d'$ and $db \neq 0$. □

COROLLARY. *If M isomorphically realizes M' with feedback encoding, then $\mathscr{S}(M) \cong \mathscr{S}(M')$.*

We can now give the examples mentioned above of zero-free homomorphisms which are not homomorphisms. Consider the map Φ guaranteed by Theorem 14. If s_1 and s_2 are two states of M for which $\phi(s_1) = \phi(s_2)$, then any triples of the form $b_1 = (r, x, s_1)$ and $b_2 = (s_2, y, t)$ have product $b_1 b_2 = 0$, while

$$\Phi(b_1)\Phi(b_2) = (\phi(r), xy, \phi(t)) \neq 0.$$

We will also prove a converse to this theorem. If $b = (s, x, t)$ is a triple of machine M, define $i(b) = s$ and $f(b) = t$. A state s of M is *reachable* if there is a triple b such that $f(b) = s$.

Let S be a semigroup with zero. For any $s \in S$, define $s^{\perp} = \{t | st \neq 0\}$ and $s^{\top} = \{t | ts \neq 0\}$. We can then define equivalence relations $\equiv_{\top}$ and $\equiv_{\perp}$ on S: $s \equiv_{\perp} t$ if $s^{\perp} = t^{\perp}$, and $s \equiv_{\top} t$ if $s^{\top} = t^{\top}$.

LEMMA. *For any machine M in which every state is reachable, the relation $\equiv_{\perp}$ on $\mathscr{S}(M)$ has finite index which is equal to $1 + |Q|$. In fact, $b_1 \equiv_{\perp} b_2$ if and only if $f(b_1) = f(b_2)$, and $[0]_{\perp} = \{0\}$.*

LEMMA. *For any machine M in which every state is reachable, the relation $\equiv_{\top}$ on $\mathscr{S}(M)$ has finite index equal to $1 + |Q|$; $[0]_{\top} = \{0\}$, and $b_1 \equiv_{\top} b_2$ if and only if $i(b_1) = i(b_2)$.*

THEOREM 15. *Let M and M' be machines in which each state is reachable. If $|I| = |I'|$ and there is a zero-free homomorphism $\Phi: \mathscr{S}(M) \xrightarrow{\text{onto}} \mathscr{S}(M')$, then M realizes M' with feedback encoding.*

Proof. We first show that $b_1 \equiv_{\perp} b_2$ implies that $\Phi(b_1) \equiv_{\perp} \Phi(b_2)$. Suppose that

$b_1 \equiv_\perp b_2$ and that $\Phi(b_1)d' \neq 0$, so that $d' \in \Phi(b_1)^\perp$. Then there is a d such that $\Phi(d) = d'$ and $b_1 d \neq 0$. Since $b_1 \equiv_\perp b_2$, $b_2 d \neq 0$. Then $\Phi(b_2)\Phi(d) = \Phi(b_2)d' = \Phi(b_2 d) \neq 0$ since $\ker(\Phi) = \{0\}$. Thus $d' \in \Phi(b_2)^\perp$. Similarly, $\Phi(b_2)^\perp \subseteq \Phi(b_1)$, so $\Phi(b_1)^\perp = \Phi(b_2)^\perp$, and hence $\Phi(b_1) \equiv_\perp \Phi(b_2)$. By a similar argument, $b_1 \equiv_\top b_2$ implies $\Phi(b_1) \equiv_\top \Phi(b_2)$.

It then follows from the lemmas that $f(b_1) = f(b_2)$ implies that $f(\Phi(b_1)) = f(\Phi(b_2))$ and $i(b_1) = i(b_2)$ implies that $i(\Phi(b_1)) = i(\Phi(b_2))$. Therefore Φ induces maps $\phi_i : Q \to Q'$ and $\phi_f : Q \to Q'$ such that $\Phi((s, x, t)) = (\phi_i(s), x', \phi_f(t))$. Now, suppose that $b_1 = (r, x, s)$ and $b_2 = (s, y, t)$, so that $b_1 b_2 \neq 0$. Then $\Phi(b_1) = (\phi_i(r), x', \phi_f(s))$, $\Phi(b_2) = (\phi_i(s), y', \phi_f(t))$. But we know that $\Phi(b_1)\Phi(b_2) \neq 0$; thus for each reachable state s, $\phi_i(s) = \phi_f(s)$. Since every state is reachable, $\phi_i = \phi_f = \phi$, a map from Q to Q'. Furthermore, ϕ is onto, since Φ is. Let $x \in I^+$, $x' \in I'$, be such that for some triple $b = (s, x, t)$, $\Phi(b) = (\phi(s), x', \phi(t))$. If we write $x = yz$, where neither y nor z is empty, then $b = (s, y, sy)(sy, z, t)$. Thus $\Phi((s, y, sy))\Phi((sy, z, t)) = \Phi(b) = (\phi(s), x', \phi(t))$. But this is an impossible situation: the second components of $\Phi((s, y, sy))$ and $\Phi((sy, z, t))$ are nonempty strings, say y' and z', so that $x' = y'z'$, a contradiction of the hypothesis that $x' \in I'$. Thus every preimage of an elementary triple of M' is an elementary triple of M.

We now show that if $b' = (s', x', t')$ and $\Phi(s) = s'$, then there is a triple b with $i(b) = s$ such that $\Phi(b) = b'$. Since s is reachable, there is some triple $d = (r, y, s)$. Now, $\Phi(d) = (\phi(r), y', \phi(s))$, so $\Phi(d)b' \neq 0$. Thus, since Φ is a zero-free homomorphism, there is a triple b such that $db \neq 0$ and $\Phi(b) = b'$. But $db = 0$ implies $i(b) = f(d) = s$.

Now let $|I| = |I'| = n$. Choose any $s' \in Q'$ and any s such that $\phi(s) = s'$. If the n elementary triples b'_j with $i(b'_j) = s'$ are $b'_j = (s', x'_j, t'_j)$, then for each b'_j there is a triple b_j with $\Phi(b_j) = b'_j$ and $i(b_j) = s$. Furthermore, as we have shown, each such b_j is an elementary triple. Also, since $|I| = |I'| = n$, there are exactly n elementary triples b with $i(b) = s$. Therefore, for each $s \in Q$ and each $x' \in I'$ there is a unique $x \in I$ such that $\Phi((s, x, sx)) = (\phi(s), x', \phi(sx))$; we can then define a map $h : Q \times I' \to I$ by setting $h(s, x') = x$, where $\Phi((s, x, sx)) = (\phi(s), x', \phi(sx'))$. The pair (ϕ, h) defines a realization with feedback encoding. □

COROLLARY. *If M and M' are machines in which each state is reachable with $|I| = |I'|$, and if $\mathcal{S}(M') \cong \mathcal{S}(M)$, then M isomorphically realizes M' with feedback encoding.*

We can get a result parallel to Theorem 15 for the case in which one machine simulates another. Let S and S' be semigroups with zero; we say that S' *zero-free divides* S if there is a subsemigroup S_1 of S which contains the zero of S such that there is a zero-free homomorphism from S_1 onto S'.

THEOREM 16. *Let M and M' be machines such that M simulates M' with feedback encoding, where the simulation is defined by maps ϕ and h. If the extended map $h : Q \times I'^+ \to I^+$ is one-to-one for each $s \in Q$, then $\mathcal{S}(M')$ zero-free divides $\mathcal{S}(M)$.*

Note. A sufficient condition for the extended map to be one-to-one will be given below.

Let M and M' be two machines, and $h : Q \times I' \to I^+$; we say that a subset $Q_1 \subseteq Q$ is *closed under* h if, for each $s \in Q_1$, $x' \in I'$, $sh_s(x') \in Q_1$. If there is also a map $\phi : Q_1 \xrightarrow{\text{onto}} Q'$ which satisfies $\phi(q)x' = \phi(qh(q, x'))$, then we will say that M' *divides M with feedback encoding*; we call Q_1 the *core* of the division.

COROLLARY. *If M' divides M with feedback encoding in such a way that the map $h_s : I'^+ \to I^+$ is one-to-one for each s, then $\mathscr{S}(M')$ zero-free divides $\mathscr{S}(M)$.*

It is the corollary to Theorem 16, rather than the theorem itself, which has a converse. We need one additional concept from the theory of semigroups. If S is a semigroup and $s, t, \in S$, then s *divides* t, written $s|t$, if there is an r such that either $sr = t$ or $rs = t$; t is *prime* if there is no s which divides it. Note that for any machine M, the only primes in $\mathscr{S}(M)$ are the elementary triples.

THEOREM 17. *Let M and M' be machines such that every state of M' is reachable and $\mathscr{S}(M')$ zero-free divides $\mathscr{S}(M)$. Then M' divides M with feedback encoding, and the extended map $h_s : I'^+ \to I^+$ is one-to-one for each s.*

We have thus shown that the semigroups $\mathscr{S}(M)$ reflect quite accurately the relationship of M to other machines as far as realization or simulation with feedback encoding. One important feature of the classical semigroup $S(M)$ of a machine is that for every finite semigroup S, there is a machine such that $S = S(M)$. A similar result cannot be possible for the semigroups $\mathscr{S}(M)$; for example, it is clearly impossible for $\mathscr{S}(M)$ to ever be a group. We can, however, characterize those infinite semgroups $\mathscr{S}$ such that, for some machine, $\mathscr{S} = \mathscr{S}(M)$. To do this, we need some preliminary definitions.

Let P be a partially ordered set under the relation $\leqq$. If $a, b \in P$, we say that b *covers* a if $a < b$ and there is no $c \in P$ such that $a < c < b$; the *cover* of a, cov (a), is the set of all b which cover a. We will call a partially ordered set P an *n-tree* if it has the following properties:

(i) P has a least element;
(ii) for each $a \in P$, $|\text{cov}(a)| = n$;
(iii) if $a \neq b$ then $\text{cov}(a) \cap \text{cov}(b) = \emptyset$.

For any semigroup S, we can define a relation $\leqq$ by defining $a < b$ if there is an element c such that $ac = b$, and $a \leqq b$ if either $a = b$ or $a < b$. This relation may, but need not, be a partial order, since it is not necessarily antisymmetric.

THEOREM 18. *Let $\mathscr{S}$ be an infinite semigroup. Then there is a machine M with $|I| = n$ such that $\mathscr{S} = \mathscr{S}(M)$ if and only if*

(i) *$\mathscr{S}$ has a zero,* 0;
(ii) *the relations $\equiv_\perp$ and $\equiv_\top$ have the same finite index, and* $[0]_\top = [0]_\perp = \{0\}$;
(iii) *if $st \neq 0$, then $st \in [s]_\top \cap [t]_\perp$.*
(iv) *each block of the equivalence relation $\equiv_\top$, except for $[0]_\top$, is a disjoint union of n n-trees under the relation $\leqq$.*

Before proceeding, we mention one interesting auxiliary property of the semigroups $\mathscr{S}(M)$. Suppose that the machine M is actually a finite-state acceptor; that is, there is a starting state q_0 and a set of final states $F \in Q$, so that we can associate with M the event $E(M) \subseteq I^+$ of strings x for which $q_0 x \in F$.

THEOREM 19. *If M is a finite state acceptor, then there is a right ideal R and a left ideal L of $\mathscr{S}(M)$ such that*

$$E(M) = \{\text{proj}_2(s) | s \in L \cap R - \{0\}\}.$$

Proof. Let $L = \{s | f(s) \in F\} \cup \{0\}$ and $R = \{s | i(s) = q_0\} \cup \{0\}$. Then L is a left ideal since, for any $t \in \mathscr{S}(M)$ and $s \in L$, if $ts \neq 0$, then $f(ts) = f(s) \in F$. Similarly, R is a right ideal. Then $L \cap R - \{0\}$ is the set $\{s | i(s) = q_0 \text{ and } f(s) \in F\}$, so that the second components of the strings of $L \cap R - \{0\}$ are all the walks from q_0 to F, i.e., $E(M)$. □

For any machine M, the *$\mathscr{S}^*$-semigroup* $\mathscr{S}^*(M)$ has for its elements all finite sets of triples (we will write 0 for the empty set $\varnothing$ of triples), where if $U = \{b_i | i = 1, \cdots, n\}$ and $V = \{d_j | j = 1, \cdots, m\}$ are elements of $\mathscr{S}^*(M)$, then $UV = \{b_i d_j | i = 1, \cdots, n; j = 1, \cdots, m\}$. We list some of the properties of $\mathscr{S}^*(M)$ in the next theorem [2]; a state s is *terminal* if for all $x \in I$, $sx = s$.

Assertion 1. If M isomorphically realizes M' with feedback encoding, then $\mathscr{S}^*(M) \cong \mathscr{S}^*(M')$.

Assertion 2. If M and M' are machines in which each state is reachable and there is at least one nonterminal state and if $\mathscr{S}^*(M) \cong \mathscr{S}^*(M')$, then M isomorphically realizes M' with feedback encoding.

These results make it seem reasonable to expect that we can find generalizations to homomorphic realization or simulation with feedback encoding. Suppose, for example, we had M realizing M' homomorphically with feedback encoding, the realization being defined by maps ϕ and h. It would be natural, as would in fact be done in proving Assertion 1 above to define, for each triple $b = (s, x, t)$ of M,

$$\phi^*(b) = (\phi(s), h_s^{-1}(x), \phi(t)).$$

(ϕ^* is just the map $\Phi: \mathscr{S}(M) \to \mathscr{S}(M')$ of Theorem 14.) We would then define $\Phi(\{b_i | i = 1, \cdots, n\})$ to be $\{\phi^*(b_i)\}$. Let $S = \{b_j\}$ and $T = \{d_k\}$, S, $T \in \mathscr{S}^*(M)$, and suppose $ST \neq 0$. For each pair b_j and d_k for which $b_j d_k \neq 0$, it will follow that $\phi^*(b_j d_k) = \phi^*(b_j)\phi^*(d_k)$, so that $\Phi(ST) \subseteq \Phi(S)\Phi(T)$. But should there be a pair b_j, d_k such that $b_j d_k = 0$ but $\phi^*(b_j)\phi^*(d_k) \neq 0$, then $\Phi(ST) \subsetneq \Phi(S)\Phi(T)$, so that Φ would not even be a zero-free homomorphism. Such a pair b_j, d_k would certainly exist unless $|Q| = |Q'|$.

The preceding discussion, of course, only shows that one particular approach to the problem is infeasible.

THEOREM 20. *Let M and M' be two machines, where M is strong. If there is a zero-free homomorphism $\Phi: \mathscr{S}^*(M) \to \mathscr{S}^*(M')$ and a map $\phi: Q \to Q'$ such that for each singleton $b = \{(s, x, t)\} \in \mathscr{S}^*(M)$, $\Phi(b) = \{(\phi(s), x', \phi(t))\}$, then $|Q| = |Q'|$.*

Proof. Suppose not, and let $\phi(q_1) = \phi(q_2) = q'$. Since q_1 is reachable, there is some $b_1 = \{(r, x, q_1)\} \in \mathscr{S}^*(M)$. For $y, z \in I^+$, let $b_2 = \{(q_1, y, q_1 y), (q_2, z, q_2 z)\}$. Since $b_1 b_2 \neq 0$, $\Phi(b_1)\Phi(b_2) = \{(\phi(r), x'y', \phi(q_1 y)), (\phi(r), x'z', \phi(q_2 z))\} = \Phi(b_1 b_2) = \{(\phi(r), (xy)', \phi(q_1 y))\}$. Among other things, this would imply that if q_3 is reachable from q_1 and q_4 is reachable from q_2, then $\phi(q_3) = \phi(q_4)$. Since M is strong, every state is reachable from every other, so that ϕ must be one-to-one. $\square$

The hypothesis that M be strong is probably more than is needed to reach the conclusion of the theorem, but it certainly does show that the $\mathscr{S}^*$-semigroups are not especially useful in a study of realization, with or without feedback encoding. On the other hand, Fleck, et al. made the following conjecture, which we can use the concept of simulation with feedback encoding to settle.

Conjecture. Let M and M' be any two strong machines with the same number of states. Then $\mathscr{S}^*(M)$ is isomorphic to a subsemigroup of $\mathscr{S}^*(M')$, and $\mathscr{S}^*(M')$ is isomorphic to a subsemigroup of $\mathscr{S}^*(M)$.

We first prove a crucial result, to which we alluded earlier. It gives both the sufficient condition which we mentioned in connection with Theorem 16 and also shows that simulation with feedback encoding is an essentially uninteresting concept.

THEOREM 21. *Let M be a strong machine with at least two inputs, and let M' be any machine with $|Q'| \leqq |Q|$. Then M' divides M with feedback encoding, and the feedback encoding can be chosen in such a way that the maps $h_s : I'^+ \to I^+$ are one-to-one.*

Proof. Since M is strong, for any $s, t \in Q$, there is a string w_{st} with $l(w_{st}) \geqq 1$ such that $sw_{st} = t$.

Let $\bar{Q}$ be any subset of Q with cardinality $|Q'|$, and $\phi : \bar{Q} \xrightarrow{\text{onto}} Q'$ be any map. Let $I' = \{x'_1, \cdots, x'_a\}$ and let $\eta \neq \bar{\eta}$ be two elements of I. For each $s \in \bar{Q}$, $x'_j \in I'$, let $t = \phi^{-1}(sx'_j)$ and define $h_s(x'_j) = \eta^j \bar{\eta} w_{qt}$, where $q = s\eta^j\bar{\eta}$. Then $\phi(sh_s(x'_j)) = \phi(s) x'_j$, so that M' divides M with feedback encoding; if $|Q| = |Q'|$, then M simulates M' with feedback encoding. Each map h_s is certainly one-to-one on strings. We show that the extended maps are also one-to-one.

Let $j_1 \cdots j_m$ and $k_1 \cdots k_{m'}$ be two strings from $(I')^+$, and let $h_s(j_1 \cdots j_m) = h_s(k_1 \cdots k_{m'}) = w$. Then, by definition, there are states $r, t \in \bar{Q}$ such that

$$w = h_s(j_1)h_r(j_2 \cdots j_m) = h_s(k_1)h_t(k_2 \cdots k_{m'}).$$

But there is a unique positive integer n such that the prefix of w having length $n + 1$ is the string $\eta^n\bar{\eta}$. This uniquely determines $j_1 = k_1 = x'_n$, so that $r = t = sx'_n$. Then $h_r(j_2 \cdots j_m) = h_r(k_2 \cdots k_{m'})$, and we can repeat the above process until we arrive at $m = m'$ and $j_p = k_p$, $p = 1, 2, \cdots, m$. $\square$

COROLLARY. *If M is strong with at least two inputs and M' has $|Q'| \leqq |Q|$, then $\mathscr{S}^*(M')$ zero-free divides $\mathscr{S}^*(M)$.*

COROLLARY. *If M is strong and M' has $|Q'| \leqq |Q|$, then M' divides M with feedback encoding.*

Proof. We need only cover the case in which $|I| = 1$. As in the theorem, we choose any $\bar{Q} \subset Q$ with $|\bar{Q}| = |Q'|$ and any map $\phi : \bar{Q} \xrightarrow{\text{onto}} Q'$. For $s \in \bar{Q}$ and $x'_j \in I'$, let $t = \phi^{-1}(\phi(s)x'_j)$, and define $h_s(x'_j) = w_{st}$. Then $\phi(s)x'_j = \phi(sh_s(x'_j))$, so M' divides M with feedback encoding. $\square$

THEOREM 22. *Let M be a strong automaton with n states and at least two inputs, and let M' be an automaton with $n' \leqq n$ states. Then $\mathscr{S}^*(M')$ is isomorphic to a subsemigroup of $\mathscr{S}^*(M)$.*

Proof. For the proof of the theorem, see [4].

COROLLARY. *Let M and M' be strong machines, with $|Q| = |Q'|$. Then, unless M is autonomous but M' is not, $\mathscr{S}^*(M')$ is isomorphic to a subsemigroup of $\mathscr{S}^*(M)$.*

The corollary settles the conjecture except for the case where M is autonomous but M' is not, and in that case the conjecture is false.

THEOREM 23. *If M is a strong machine with n states and at least 2 inputs, then $\mathscr{S}^*(M)$ cannot be isomorphically embedded in $\mathscr{S}^*(C_n)$.*

The proof, in [4], proceeds by showing that for the complete reset machine R_n, $\mathscr{S}^*(R_n)$ cannot be isomorphically embedded in $\mathscr{S}^*(C_n)$. But, by Theorem 22, $\mathscr{S}^*(R_n)$ is isomorphically embedded in $\mathscr{S}^*(M)$.

5. Summary. We have studied various properties of realization with feedback encoding, and, at this point, we should look back to see what we did and what we did not do.

Most of the study of the properties of these realizations was motivated by the classical theory of automata. Thus, for example, admissible homomorphisms play the same role for realizations with feedback encoding that SP-homomorphisms play for realizations. There are subtle differences between these two classes of mappings, however. On one hand, as we pointed out in §2 admissible homomorphisms do not exhibit all the lattice properties of SP-homomorphisms. Thus, the full power of the Hartmanis–Stearns techniques [6] cannot be applied to admissible homomorphisms. On the other hand, admissible homomorphisms, being defined on digraphs rather than on machines, are somewhat easier to manipulate and study. In this regard, an added advantage is that, as mappings between digraphs or graphs, they have interest independent of their applications to machines.

There appears to be little more that can be said about the basic properties of realizations with feedback encoding. One problem upon which we did not touch is the meaning of realization with feedback encoding when applied to logical nets. For example, Zeigler [13] shows that for any integer r, there is a machine M such that any logical net which isomorphically realizes M has a strong component S which contains a point whose indegree, in S, is greater than r. While we strongly suspect that a similar result would hold for isomorphic realization with feedback encoding, it is not clear how to attack the problem. Zeigler's proof techniques depend heavily on behavioral properties, which, of course, are blurred by realization with feedback encoding, and so resolution of the problem would probably depend on a better understanding of the relationship between net structure and transition graph structure.

The semigroup of a machine is quite important to the theory of realization. While we studied the $\mathscr{S}$-semigroups in great detail in §3, our motivation was to be able to develop decomposition properties, and we must conclude that this goal is very likely unattainable. For we showed that zero-free homomorphisms or divisions between $\mathscr{S}$-semigroups are both necessary and sufficient for realizations or divisions with feedback encoding, which makes it difficult to suppose that a finite algebraic structure with similar properties could be found. And certainly, even if decomposition properties could be related to the $\mathscr{S}$-semigroups, there is no real hope of usefully applying these infinite structures to the problems of finite automata theory.

In §2 we showed that cascade realizations with feedback encoding can be more economical, in terms of the sizes of the state sets of the component machines, than realizations without feedback encoding. Another application, to the problem of designing realizing machines with distinguishing sequences, is discussed in [3] and a forthcoming sequel to this paper. Perhaps other applications could have been developed, but these are sufficient to show the value, and limits, of realization with feedback encoding. For—and this cannot be stressed too strongly—there are no clear rules on when to use the techniques we have developed. Even when we can guarantee the applicability of the techniques to any machine, whether or not they are of any value will depend quite strongly on the specific problem. We have shown the potential power of realization with feedback encoding, but in any application it will be just one of a number of tools which can be tried, and will surely work better in some cases than in others.

REFERENCES

[1] T. Booth, *Sequential Machines and Automata Theory*, John Wiley, New York, 1967.

[2] A. Fleck, S. Hedetniemi and R. Oehmke, *$\mathscr{S}$-semigroups of automata*, J. Assoc. Comput. Mach., 19 (1972), pp. 3–10.

[3] D. GELLER, *Realization with feedback encoding*, Doctoral dissertation, Dept. of Computer and Communication Sci., Univ. of Michigan, Ann Arbor, 1972.

[4] ———, *Generalization of a theorem of Fleck, Hedetniemi and Oehmke on $\mathscr{S}^*$-semigroups of automata*, Discrete Math., 8 (1974), pp. 345–349.

[5] ———, *Walkwise and admissible mappings between digraphs*, Ibid., to appear.

[6] J. HARTMANIS AND R. E. STEARNS, *Algebraic Structure Theory of Sequential Machines*, Prentice-Hall, Englewood Cliffs, N.J., 1966.

[7] S. HEDETNIEMI, *Homomorphisms of graph and automata*, Tech. Rep. 03105-42-T, Dept. of Computer and Communication Sci., Univ. of Michigan, Ann Arbor, 1966.

[8] J. HOLLAND, *Cycles in logical nets*, Tech. Rep. 2722, 2794-4-7, Logic of Computers Group, Univ. of Mich., 1959; see also, *Cycles in logical nets*, J. Franklin Inst., 270 (1960), pp. 202–226.

[9] K. KROHN AND J. RHODES, *Algebraic theory of machines. I: Prime decomposition theorem for finite semigroups and machines*, Trans. Amer. Math. Soc., 116 (1965), pp. 450–464.

[10] R. MCNAUGHTON, *The loop complexity of pure-group events*, Information and Control, 11 (1967), pp. 167–176.

[11] M. YOELI AND A. GINZBURG, *On homomorphic images of transition graphs*, J. Franklin Inst., 278 (1964), pp. 291–296.

[12] H. P. ZEIGER, *Cascade synthesis of finite state machines*, Information and Control, 10 (1967), pp. 419–433.

[13] B. P. ZEIGLER, *On the feedback complexity of automata*, Tech. Rep. 08226-6-T, Dept. of Computer and Communication Sci., Univ. of Michigan, Ann Arbor, 1969; see also, *Proof of a conjecture by A. W. Davis and H. Wang: Some relations between net cycles and state cycles*, Information and Control, 21 (1972), pp. 185–195.

SIAM J. COMPUT.
Vol. 4, No. 1, March 1975

REALIZATION WITH FEEDBACK ENCODING. II: APPLICATIONS TO DISTINGUISHING SEQUENCES*

DENNIS P. GELLER†

Abstract. We continue the work of a previous paper by developing techniques for realizing (with feedback encoding) a given machine by one which admits a distinguishing sequence. We allow no expansion of state set or input set size, and attempt to minimize the number of additional outputs needed. With feedback encoding, this usually behavioral problem becomes one involving only (graph) structural properties of the given machine. In particular, we cast the problem of reducing the number of instances of sets of states merging under an input as one involving coloring a bipartite graph derived from the machine.

1. Introduction. In this paper, we apply the concepts of [2] to the problem of realizing a given machine by one with a distinguishing sequence. While the reader is assumed to be familiar with the results of [2], we will briefly review some of the more important definitions and notations.

A *machine*, or *finite-state automaton*, M, consists of a set Q of *states*, a set I of *inputs*, where both Q and I are finite, a *transition function* $\delta : Q \times I \to Q$, a set Y of *outputs*, and a function $\lambda : Q \times I \to Y$ called the *output function*; M will be represented by the quintuple $\langle Q, I, \delta, Y, \lambda\rangle$. We write qx for $\delta(q, x)$. If λ truly depends on both Q and I, we have a Mealy machine. By contrast, if we can associate output symbols with the states so that $\lambda(q, x)$ is the output associated with state qx, we have a Moore machine; in this case we sometimes express λ as a map from Q to Y.

If I^+ is the set of finite strings of input symbols, then we can express δ as a map from $Q \times I^+$ to Q by writing $\delta(q, xy) = \delta(\delta(q, x), y)$ for any $x \in I^+$ and $y \in I$. We can similarly extend λ in two different ways. For $x \in I^+$ and $y \in I$, $\lambda(q, xy) = \lambda(qx, y)$. Alternately, for $x_1, x_2, \cdots, x_n \in I$, $\beta(q, x_1x_2 \cdots x_n) = \lambda(q, x_1)\lambda(q, x_1x_2) \cdots \lambda(q, x_1x_2 \cdots x_n)$. Note that if $x = x_1x_2 \cdots x_n \in I^+$, then the *length* $\ell(x)$ of x is n.

Sometimes we are only concerned with the transition functions of machines, and then we omit reference to λ and Y. If $M = \langle Q, I, \delta\rangle$ and $M' = \langle Q', I', \delta'\rangle$ are machines, then M *realizes* M' *with feedback encoding* if there are maps $\phi : Q \xrightarrow{\text{onto}} Q'$ and $h : Q \times I' \to I$ such that for each $q \in Q$, the map $h(q, \cdot) = h_q : I' \to I$ defined by $h_q(x') = h(q, x')$ is one-to-one and onto, satisfying the condition for all $q \in Q$ and all $x' \in I'$,

$$\phi(q)x' = \phi(qh(q, x')).$$

If the map h, called the *input encoder*, did not actually depend on Q, then this would essentially be the same as the usual definition of realization.

A *digraph* D consists of a set $V = V(D)$ of points together with a collection

* Received by the editors January 18, 1973, and in revised form January 2, 1974.

† Human Sciences and Technology Group School of Advanced Technology, State University of New York at Binghamton, Binghamton, New York 13901. This research was supported in part by the National Institutes of Health under Grant GM-12236 and in part by the National Science Foundation under Grant No. GJ-519.

(repetitions permitted) $X = X(D)$ of ordered pairs, called *arcs* from $V \times V$. If $uv = (u, v)$ is an arc, we write $uv \in D$ and say that u is *adjacent to* v, v is *adjacent from* u, uv is *incident from* u and *incident to* v.

A *walk* in a digraph is a sequence $W = \langle x_1, \cdots, x_n \rangle$ of arcs, where $x_i = u_{i-1}u_i$; we may abbreviate this and write $w = u_0u_i \cdots u_n$. If $u_0 = u_n$ but the other points are distinct, the result is a *cycle* C_n. A digraph D is *strong* if there is a walk $u_0u_1 \cdots u_nu_0$ containing all the points of D.

In any digraph the number of arcs incident to a point u is its *indegree*, id(u) and the number of arcs incident from u is its *outdegree* od(u).

The *reflexive closure* D^R of a digraph D is the smallest superdigraph of D which has a loop at each point. For each point u of D let $\vec{S}(u)$ be the set of arcs incident from u and let $\overleftarrow{S}(u)$ be the set of arcs incident to u. Given digraphs D and E a *mapping from* $D \xrightarrow{\text{onto}} E$ is a mapping $\phi : X(D) \to X(E^R)$ which is onto $X(E)$ and which satisfies:

for each $u \in V(D)$ there is a $u' \in V(E)$ such that $\phi(\vec{S}(u)) \subset \vec{S}(u')$ and $\phi(\overleftarrow{S}(u)) \subset S(u')$.

A mapping $\phi : D \to D'$ is an *admissible homomorphism*, or simply *admissible*, if whenever $\phi(u) = \phi(v)$ and $uw \in D$, then there is a point $\bar{w}$ such that $v\bar{w} \in D$ and $\phi(\bar{w}) = \phi(w)$.

If $M = \langle Q, I, \delta \rangle$ is a machine, the *digraph* $D(M)$ *of* M has $V(D(M)) = Q$ and $X(D(M)) = \{uv | u, v \in Q$ and for some $x \in I$, $ux = v\}$; if there are n inputs x_i such that $ux_i = v$, then there are n arcs from u to v.

The following is Theorem 4 of [2].

THEOREM. *If M realizes M' with feedback encoding, then there is an admissible homomorphism from $D(M)$ onto $D(M')$. Conversely, if there is an admissible homomorphism from $D(M)$ onto $D(M')$ and if $|I| = |I'|$, then M realizes M' with feedback encoding.*

2. Distinguishing sequences.

Having presented some of the theoretical properties of realizations with feedback encoding, we now turn our attention to a specific applications area. Actually, we have already discussed one application in [2], where we showed that the use of feedback encoding can lead to more efficient cascade realizations.

Given a behavior which is to be realized, one often places additional requirements on the realizing machine; perhaps the simplest such requirement is that the machine be reduced. Another requirement is that the machine have a distinguishing sequence, an input string whose output sequence uniquely identifies the machine's starting state. In this chapter we will develop techniques for realizing some machine M', with feedback encoding, by a machine M having a distinguishing sequence. It is important to note that the machine which has the distinguishing sequence is M, and not M together with the feedback encoder.

In [2] we were concerned only with the realization of state-behaviors, but here we will instead study the realization of input-output behaviors. We will need to define the realization with feedback encoding of a machine having outputs in two ways, one for the Moore case and one for the Mealy.

If $M = \langle Q, I, \delta, Y, \lambda \rangle$ and $M' = \langle Q', I', \delta', Y', \lambda' \rangle$ are Moore machines, then *M realizes M' with feedback encoding* if there are maps $\phi : Q \xrightarrow{\text{onto}} Q'$, $h : Q \times I' \to I$, a feedback encoder, and $g: Y \to Y'$, such that

$$\phi(q)x' = \phi(qh_q(x')) \quad \text{and} \quad \lambda'(\phi(q)) = g(\lambda(q)).$$

Since in most of what follows the realizations will be isomorphic, the *output decoder*, g, may often be just a one-to-one correspondence, and can be omitted if we assume, without loss of generality that $\lambda'(\phi(q)) = \lambda(q)$, giving the alternate conditions

$$\phi(q)x' = \phi(qh_q(x')), \qquad \lambda'(\phi(q)) = \lambda(q)$$

Similarly, if M and M' are Mealy machines, then *M realizes M' with feedback encoding* if there are maps $\phi: Q \xrightarrow{\text{onto}} Q', h: Q \times I' \to I$, a feedback encoder, and $g: Y \to Y'$, such that

$$\phi(q)x' = \phi(qh_q(x')), \qquad \lambda'(\phi(q), x') = g(\lambda(q, h_q(x')))$$

or, if we take $Y = Y'$ and g to be the identity map,

$$\phi(q)x' = \phi(qh_q(x')), \qquad \lambda'(\phi(q), x') = \lambda(q, h_q(x')).$$

Example 1. Let M_1 and M_2 be the Mealy machines in Fig. 1. Then M_2 realizes M_1 with feedback encoding, if we define $\phi(r_i) = q_i$ and give h the function table

h	0	1
r_1	0	1
r_2	1	0
r_3	1	0

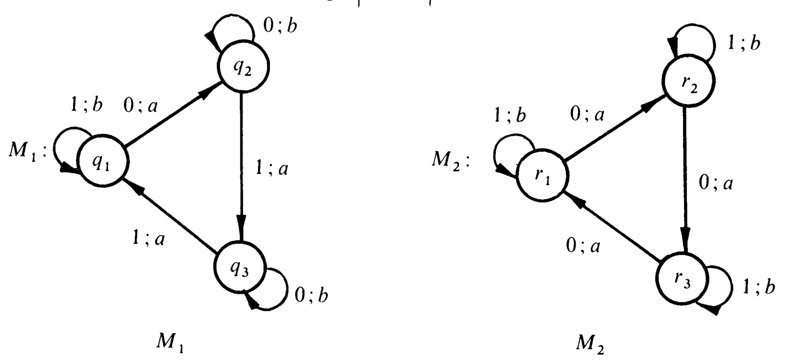


FIG. 1

From Example 1, we can see the relationship between the state transition graphs of two Mealy machines when one realizes the other isomorphically with feedback encoding. Not only are the digraphs isomorphic, but the isomorphism preserves the output labels, so that only the input labels are permuted.

As is well known, the concepts of Mealy and Moore machines are essentially interchangeable [5, p. 35]. This interchangeability carries over to realizing machines.

THEOREM 1. *Let M and M' be Mealy machines and $\tilde{M}$, $\tilde{M}'$ their Moore equivalents. If M realizes M' with feedback encoding, then $\tilde{M}$ realizes $\tilde{M}'$ with feedback encoding.*

Recall that for a state q and string $x_1 \cdots x_n$, $\beta_q(x_1 \cdots x_n) = \lambda(q, x_1)\lambda(qx_1, x_2) \cdots \lambda(qx_1 \cdots x_{n-1}, x_n)$; of course, in a Moore machine, this reduces to $\beta_q(x_1 \cdots x_n) = \lambda(qx_1)\lambda(qx_1x_2) \cdots \lambda(qx_1 \cdots x_n)$. Defining β in this manner, we are omitting the output associated with the current state, q, and therefore ignoring a potentially useful item of information. Since this information is usually available, especially in actual circuits, it is worthwhile to examine the effects of this omission on the work to follow. All the constructions which we give will be valid for either definition of β, but bounds involving the lengths of input sequences will be larger by 1 than they would be with the alternate definition of β. The advantage to the form taken here, as we shall see, is to make it possible to treat certain behavioral characteristics of a machine as though they were purely structural.

We say that a string x is a *distinguishing sequence* for a machine M if for any states $q \neq q'$, $\beta_q(x) \neq \beta_{q'}(x)$. Clearly [5], any machine which has a distinguishing sequence is reduced, but the converse does not hold. In the case that all the symbols in a distinguishing sequence x are identical, we say that x is a *repeated symbol distinguishing sequence* [8].

Before we begin our study in detail, a word is in order about just what it is that we wish to accomplish. The existence of a distinguishing sequence is properly a behavioral, rather than a structural, property. Nevertheless, feedback encoding techniques can, for some machines, produce realizing machines with distinguishing sequences and yet cause no increase in state size, input set size, or output set size. We will develop some results which will give techniques, in some cases, for finding realizing machines with distinguishing sequences. These techniques will not cover all the possibilities, however. Rather than simply giving techniques, we are more concerned with trying to demonstrate that feedback encoding techniques can be an effective tool. The usefulness of these techniques does not lie solely in the theorems which we are presenting.

In a machine M, two states q_1 and q_2 are said to *merge* if, for some input $x \in I$, $q_1x = q_2x$; states q_1 and q_2 *converge* if, under some input x, they merge and if, also, $\lambda(q_1, x) = \lambda(q_2, x)$. If q_1 and q_2 converge to q_3 under x, we write $(q_1, q_2)x = q_3$. Note that in a Moore machine, two states converge if and only if they merge.

LEMMA [5]. *If a reduced machine is convergence-free, then it has a distinguishing sequence.*

The converse is not true; a machine need not be convergence-free to have a distinguishing sequence. However, if a machine is reduced and k states converge to a single state, then by state-splitting techniques, and adding additional output symbols, the convergence can be eliminated; this requires adding on the order of $\{k/2\}$ new states and output symbols [8]. Of the two, addition of new states is the more costly, and the techniques we present will be geared towards producing no increase in state set size, although some increase in output set size will not always be avoidable. We will first concentrate on the problem of designing realizing machines which have repeated symbol distinguishing sequences.

Let $\langle s_i \rangle = \langle s_0, s_1, \cdots, s_{m-1} \rangle$ be a sequence of symbols, where it is understood that any reference to s_j is for j modulo m. For any integer n, we say that $\langle s_i \rangle$ has *property* P(n) if there is an integer i_n for which $s_{i_n} \neq s_{i_n+n}$.

Let $M = \langle Q, \{1\}, \delta, Y, \lambda \rangle$ be a strong autonomous machine with m states, where $q_i1 = q_{i+1}$, and let $\lambda(M)$ be $\langle \lambda(q_0, 1), \cdots, \lambda(q_{m-1}, 1) \rangle$.

THEOREM 2. *A strong autonomous machine M is reduced if and only if, for all $n = 1, 2, \cdots, [m/2]$, $\lambda(M)$ has* P(n).

Of course, Theorem 2 just says that $\lambda(M)$ has no proper subperiods. If $\langle s_i \rangle$ has P(n) for each $1 \leqq n \leqq [m/2]$, we say that it has *property* P; conversely, if $\langle s_i \rangle$ does not have P(n) for some n we say that P *fails* (*for* n).

LEMMA. *Let* $\langle s_i \rangle = \langle s_0, s_1, \cdots, s_{m-1} \rangle$.

(a) *If* P *fails for n, it fails for the greatest common divisor* (n, m).

(b) *If* P *fails for n_1 and n_2, it fails for* (n_1, n_2).

Proof. (a) Let $g = (n, m)$. It will be sufficient to show that $s_0 = s_g$. Since, by hypothesis, $s_0 = s_n = s_{2n} = \cdots$, we need only show that, for some k, $kn \equiv g \pmod m$. This congruence has a solution when $(n, m)|g$; but $g = (n, m)$, so that $s_0 = s_g$.

(b) Let $g = (n_1, n_2)$. In this case we look for k_1 and k_2 which satisfy $n_1k_1 + n_2k_2 \equiv g \pmod m$. Since $(n_1, n_2, m)|g$, there is a solution to the congruence, and again, $s_0 = s_g$.

THEOREM 3. *Let $\bar{s}_0 \neq s_0$. If $\langle s_0, \cdots, s_{m-1} \rangle$ does not have* P, *then $\langle \bar{s}_0, s_1, \cdots, s_{m-1} \rangle$ has* P.

Proof. Let $n^* = \min\{n | \text{P fails for } n\}$. We show that for $n \leqq n^*$, $\langle \bar{s}_0, s_1, \cdots, s_{m-1} \rangle$ has P(n).

If $n^* = 1$, then the s_i are identical, and hence the new sequence has P. If $n^* = 2$, then $m \geqq 4$. Thus $\bar{s}_0 \neq s_2$, so P does not fail for $n = 2$. But since s_0 must have been different from s_1 (as otherwise we would have had $n^* = 1$), we still have $s_2 = s_0 \neq s_1$, so the new sequence also has P(1). In general, we know $m \geqq 2n^*$. Now suppose for $k < n^*$, $s_{i_k} \neq s_{i_k+k}$. Then since P fails for n^*, $s_{i_k+n^*} = s_{i_k} \neq s_{i_k+k} = s_{i_k+k+n^*}$. Furthermore $i_k + n^* \not\equiv i_k \pmod m$ and $i_k + k + n^* \not\equiv i_k + k \pmod m$, since $m \geqq 2n^*$. Thus if we change s_0 to $\bar{s}_0$, we can not, for $n < n^*$, cause P(n) to fail for the sequence $\langle s_0, s_1, s_2, \cdots, s_{m-1} \rangle$. But, since $\bar{s}_0 \neq s_0$, $\bar{s}_0 \neq s_{n^*} = s_0$, so the new sequence has P(n^*).

Now, let $n_1^* = \min\{n | \text{P}(n) \text{ fails for } \langle s_i \rangle\}$ and $n_2^* = \min\{n | \text{P}(n) \text{ fails for } \langle \bar{s}_0, s_i, \cdots, s_{m-1} \rangle\}$. Then $n_2^* > n_1^*$. But we could have started with $\langle \bar{s}_0, s_i, \cdots, s_{m-1} \rangle$ and changed $\bar{s}_0$ to s_0, giving $\langle s_i \rangle$. Thus $n_2^* < n_1^*$. This is impossible, so $\langle \bar{s}_0, s_1, \cdots, s_{m-1} \rangle$ has P.

The next result holds for both Moore and Mealy machines; a subdigraph of a digraph is *spanning* if it contains all the points of the digraph.

LEMMA. *Let M be a machine and suppose that in the labeled digraph consisting of $D(M)$ together with state and output labels, there is a spanning cycle whose output sequence has property* P. *Then M can be isomorphically realized with feedback encoding by a machine M' with a repeated symbol distinguishing sequence.*

Proof. We choose M' to have the same labeled digraph as M. Let the states, in their order along the cycle, be $q_0, q_1, \cdots, q_{m-1}$. We will define an encoding map h such that at each state q_i, one input x_i for which $q_ix_i = q_{i+1}$ is coded as $h_{q_i}(x_i) = 1' \in I'$, and assign the other values of h arbitrarily, preserving set isomorphism. Then in M', the input symbol $1'$ will define a strong autonomous submachine M'' of M', which, by hypothesis will be reduced. But it is known [8] that a reduced autonomous machine has a (repeated symbol) distinguishing sequence, y. Since M'' has the same state set as M', y is a repeated symbol distinguishing sequence for M'.

A *2-factor* of a digraph is a spanning subdigraph in which each point u has $\mathrm{id}(u) = \mathrm{od}(u) = 1$; a 2-factor is a union of directed cycles. Suppose that a machine M has a 2-factor $Z = Z_1 \cup \cdots \cup Z_t$ consisting of t directed cycles. As in the preceding lemma, we can define a machine M' which realizes M isomorphically with feedback encoding, such that Z is the subdigraph induced by a single input symbol, x'. We now need only make sure that the autonomous submachine defined by x' is reduced. This can be done, in the worst case, by adding t new output symbols, $w_1, \cdots, w_t$. By Theorem 3, if we change one output label in Z_i to w_i, Z_i will be reduced. Furthermore, since $w_i \neq w_j$, no state in Z_i can be equivalent to a state in Z_j. Of course, in general we could expect to need fewer than t new output symbols. In the next lemma we list some of the known conditions for a digraph to have a 2-factor; if $D(M)$ satisfies any of these conditions, then M can be isomorphically realized with feedback encoding by a machine with a repeated symbol distinguishing sequence. By γS in part (a) we mean the set of points adjacent from S.

LEMMA. (a) *A digraph D has a 2-factor if and only if for each set S of points, $|S| \leqq |\gamma S|$* [1].

(b) *If a strong digraph D has p points and, for each point v, $\mathrm{id}(v) + \mathrm{od}(v) \geqq p$, then D has a spanning cycle* [3].

(c) *If a strong digraph D has p points and for every pair of points u and v such that $uv \notin D$, $\mathrm{od}(u) + \mathrm{id}(v) \geqq p$, then D has a spanning cycle* [11].

Of course, the fewer the number of cycles in a 2-factor, the fewer the number of output symbols which will have to be changed. On the other hand, the more cycles in the 2-factor, the shorter will be the length of the distinguishing sequence. This trade-off is expressed in the next theorem: by $[r > 0]$ in the theorem we mean the logical variable which takes the value 1 if $r > 0$ and 0 otherwise.

THEOREM 4. *Let M be a machine with p states and suppose that $D(M)$ has a 2-factor which contains t cycles, of which r are 1-cycles (loops). Then M can be isomorphically realized with feedback encoding by a machine which has a distinguishing sequence. Furthermore, the length L of the distinguishing sequence and the number N of output symbols which must be changed satisfy*

$$L \leqq p - 2t + r + 1, \quad N \leqq t - [r > 0], \quad L + N \leqq p.$$

Proof. We have already indicated how the existence of a 2-factor implies the existence of a realizing machine which has a distinguishing sequence. In the worst case, we need to change one output symbol on each of the $t - r$ cycles of length greater than one, both to reduce the cycle (see Theorem 3) and to make the cycles inequivalent. In the worst case, this requires adding $t - r$ new output symbols, although this can undoubtedly be reduced in practice. Each of the r loops is already reduced, so we need to add or change at most $r - 1$ output symbols to make them inequivalent. This gives $N \leqq (t - r) + (r - 1) = t - 1$ if $r > 0$ and $N \leqq t$ if $r = 0$; hence $N \leqq t - [r > 0]$. The length L of the distinguishing sequence is at most the length of a distinguishing sequence for the largest cycle, which is one less than the length of the cycle [5], [10]. For a given t and r, the longest cycle is attained when all but one of the $t - r$ cycles of length greater than 1 are 2-cycles, using $2(t - r - 1)$ of $p - r$ states. The remaining cycle then has length $p - (2(t - r) - 2) - r = p - 2t + r + 2$, so that $L \leqq p - 2t + r + 1$. Now, if $r > 1$, $L + N$

$\leqq t - 1 + p - 2t + r + 1 = p - t + r$, which takes its maximum when all cycles are 1-cycles, so that $t = r$. If $r = 0$, then $L + N \leqq t + p - 2t + 1 = p - t + 1$, which takes a maximum when $t = 1$, and the 2-factor is a spanning cycle. Thus $L + N \leqq p$.

The bound $L \leqq p - 2t + r + 1 \leqq p$ compares quite favorably with the bound $L \leqq (p - 1)p^p$ given in [4], although it is not known if the latter is a best possible upper bound.

Notice that while the theorem describes a sufficient procedure, it is certainly not necessary.

The simplest way for a machine to satisfy the conditions of the theorem is for each of its states to have the same indegree; such a machine is called *homogeneous* in [9], where it is shown that any machine whose reduced machine is strong is behaviorally equivalent to a homogeneous machine. Theorem 4 could then be applied to the homogeneous machine. Of course, the homogeneous machine has a much larger state set, but this disadvantage may be offset, as Miller and Winograd [9] note: "McNaughton and Booth [7], however, found that in the 2-input case a particularly uniform circuit structure (for the homogeneous machine) resulted for the state to state circuitry ... the uniform structure may be quite advantageous in practice, and ... can be readily seen to extend to the p-input case."

On the other hand, even if there is no 2-factor, we can use the techniques outlined above to produce a realizing machine with a distinguishing sequence.

Note first that any machine can be isomorphically realized by a machine with a repeated symbol distinguishing sequence, by choosing an autonomous submachine and adding output symbols so as to make it reduced [8]. With feedback encoding, a similar technique applies, but there is more freedom in choosing the substructure to reduce, and consequently less output augmentation may be necessary.

Any autonomous machine, whether or not it is a union of directed cycles, is a *functional digraph*: a digraph in which each point has outdegree exactly 1. Given any functional subdigraph $\bar{D}$ of $D(M)$, we can, with feedback encoding, make $\bar{D}$ the digraph of an autonomous submachine $\bar{M}$, and then add output symbols to make $\bar{M}$ reduced. Thus, for a machine M, we would choose a functional subdigraph $\bar{D}$ which was as "close to" being reduced as possible, and then add output symbols so as to make it reduced. The technique in doing this is first to make each cycle of $\bar{D}$ reduced, and to make the cycles inequivalent, as we would if $\bar{D}$ were a 2-factor. We then continue, making each component of $\bar{D}$ reduced. Two states can be equivalent only if they belong to the same component and there is a homomorphism which identifies them; homomorphisms of autonomous machines (as we noted in [2], the notions of SP and admissible homomorphisms coincide for autonomous machines) have been studied in detail in [12]. We can therefore state the following remark.

Remark. Any machine can be isomorphically realized with feedback encoding by a machine with a repeated symbol distinguishing sequence.

A reduced machine will fail to have a distinguishing sequence only if two states converge under some input. Using classical techniques, once a convergence is found to interfere with the existence of a distinguishing sequence, state-splitting and/or augmentation of outputs must be employed. With feedback encoding, however, it is often possible to eliminate the convergence without adding states or output symbols. We first state a theorem indicating when this can be done for a 2-input (Moore) machine.

THEOREM 5. *Let M be a Moore machine with two inputs x_1 and x_2 and exactly one convergence*: $(q_1^1, q_2^1)x_1 = q_1^2$. *Then, if there is no state q such that $qx_2 = q_1^2$, M can be isomorphically realized with feedback encoding by a convergence-free machine with the same output function.*

Proof. Note first that since M has only one convergence, $q_1^1x_2 \neq q_2^1x_2$. Let $q_2^1x_2 = q_2^2$ and $q_1^1x_2 = q_0^2$ (see Fig. 2).

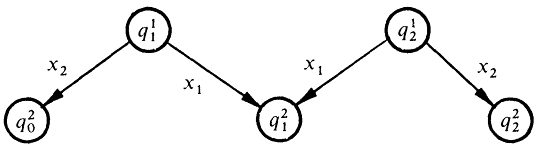


FIG. 2

We can recode inputs at q_1^1, eliminating the convergence, unless there is a state q_0^1 such that $q_0^1x_1 = q_0^2$; similarly, we can recode at q_2^1 unless there is a q_3^1 such that $q_3^1x_1 = q_2^2$. Suppose that we can recode neither at q_1^1 nor at q_2^1 (see Fig. 3).

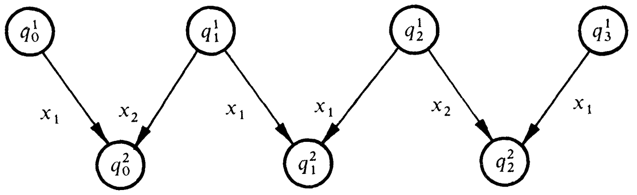


FIG. 3

If, for example, there is no state q_4^1 such that $q_4^1x_1 = q_3^1x_2$, then we can recode the inputs at q_3^1 without causing a new convergence; this, in turn, permits us to recode the inputs at q_2^1, and hence eliminate the convergence.

Continuing in this manner gives rise to sequences of states

$$\cdots, q_{-1}^1, q_0^1, q_1^1, q_2^1, q_3^1, q_4^1, \cdots$$

and

$$\cdots, q_{-1}^2, q_0^2, q_1^2, q_2^2, q_3^2, \cdots$$

such that if $j > 1$,

$$q_j^1x_1 = q_{j-1}^2, \qquad q_j^1x_2 = q_j^2,$$

and if $j \leqq 1$,

$$q_j^1x_1 = q_j^2, \qquad q_j^1x_2 = q_{j-1}^2.$$

Since M is finite, the process of extending these sequences must have repetitions. Suppose the first repetition is that two of the q_j^2 coincide. If we are at the stage of choosing q_j^2 and find that it is the same as some previously chosen q_k^2, then q_k^2 must be an x_2 image of two distinct states, by virtue of the way q_j^2 is chosen, unless $k = 1$. (As can be seen from Fig. 3, we choose q_j^2 as the x_2 image of the previously chosen state with superscript 1. Each point q_k^2 which has been chosen is an x_2-image, except for q_1^2, which was chosen as the x_1-image of two distinct states.) If $k \neq 1$, this implies that M has two convergences, which is a contradiction. If $k = 1$, then assume without loss of generality that $j > 0$ and continue the process at the "other end" of the sequence. It then becomes impossible for the process to fail for lack of a new q_{-n}^1, for this would have to imply that M has a second convergence.

The process therefore must stop because of an inability to choose a new q_{j+1}^1 (or q_{j-1}^1). But then we can progressively relabel inputs at $q_j^1, q_{j-1}^1, \cdots$ (or at $q_j^1, q_{j+1}^1, \cdots$) until we finally relabel at q_2^1 (or q_1^1), eliminating the convergence. At no time have we changed any output label; this proves the theorem.

The technique used in Theorem 5 is applicable to Moore or Mealy machines. More important, it may be used successfully with machines which have more convergences than specified in the theorem. One straightforward generalization is given by the following corollary.

COROLLARY. *If M has exactly n convergences $(q_1^i, q_2^i)x_1^i = q^i$, where the x_1^i, $i = 1, \cdots, n$ are distinct, and if there are n additional inputs x_2^i such that there is no state q for which $qx_2^i = q^i$, then the convergences can be eliminated.*

Proof. For each i, apply the theorem to that submachine of M defined by the two inputs x_1^i and x_2^i.

The corollary, of course, was phrased to insure that the n applications of the theorem would not conflict. Certainly, we can expect that the same techniques will apply to many machines which do not meet the strict condition of the corollary, by breaking up the convergences one at a time. Unfortunately, the extent to which the technique can be reapplied depends both on behavioral and structural properties of the given machine. While useful as a heuristic, this approach to eliminating convergences cannot easily be expressed in algorithmic form with clearcut rules for choosing—given at some stage a convergence $(q_1, q_2, \cdots, q_n)x_1 = q$—which subconvergence $(q_i, q_j)x_1 = q$ to break up and which input $x_2 \neq x_1$ to use.

In contrast to this essentially local attack on the problem of eliminating convergences, we will develop a global procedure for reducing, not the number of convergences, but rather the number of merges. With Moore machines as we have defined them, the concepts of "merge" and "converge" are, of course, identical, but this, as has been pointed out, is not always a useful identification to make. Thus the global technique we propose may often be inefficient, as it will reduce many merges which are not convergences. For machines which have a large number of convergences, however, we can expect the technique to substantially decrease the number of additional states or outputs which are required for a diagnosable realization. Unfortunately, since the technique deals with merges and not convergences, exact results on the number of additional states or output symbols which will be saved are not available. In fact, it is possible to devise examples where the procedure, while reducing the number of merges, increases the number of convergences. This will become clearer as we get into the actual mechanics of the procedure.

Let D be a digraph, and number the points $u_1, \cdots, u_p$ so that $\operatorname{id}(u_1) \leqq \operatorname{id}(u_2) \leqq \cdots \leqq \operatorname{id}(u_p)$. The *indegree sequence* of D is the sequence $\langle \operatorname{id}(u_1), \operatorname{id}(u_2), \cdots, \operatorname{id}(u_p)\rangle$. Suppose that D is the digraph of a Mealy machine, $D = D(M)$, and let M have t inputs. Let $\hat{\Xi}(D)$ be the quantity $\sum_{i=1}^{p} \max\{0, \operatorname{id}(u_i) - t\}$. For any arc uv, let $\xi(uv)$ be the input label on the arc. We will say that there are n merges at a state u if there are $n + 1$ arcs $v_1u, v_2u, \cdots, v_{n+1}u$ such that $\xi(v_1u) = \xi(v_2u) = \cdots = \xi(v_{n+1}u)$; note that if all these arcs had the same output label, it would be necessary to add n new output symbols to eliminate all the convergences. Let $\Xi(u)$ be the total number of merges at u and $\Xi(M)$ be the total number of merges in M.

LEMMA. *For any machine M with t inputs, $\Xi(M) \geqq \hat{\Xi}(D(M))$.*

Proof. We will show that for each state u, $\Xi(u) \geqq \max\{0, \operatorname{id}(u) - t\}$. Clearly, the equation holds for each state u with $\operatorname{id}(u) \leqq t$. If the number of different input labels on arcs leading to state u is $m \leqq t$, where $\operatorname{id}(u) > t$, then $\Xi(u) = \operatorname{id}(u) - m$. Thus if $\operatorname{id}(u) > t$, then $\Xi(u) = \operatorname{id}(u) - m \geqq \operatorname{id}(u) - t$.

THEOREM 6. *Any machine M can be isomorphically realized with feedback encoding by a machine M' with $\Xi(M') = \hat{\Xi}(D(M))$.*

To prove this theorem, we need to investigate those properties of the assignment function ξ which will guarantee that $\Xi(M) = \hat{\Xi}(D(M))$. To this end, we introduce some additional notions from graph theory.

A *graph* G consists of a set $V = V(G)$ of *points* together with a set $X = X(G)$ of unordered pairs of distinct elements from V, called *lines*; if X is instead a collection, with repetitions, then G is a *multigraph.* Concepts like incidence, adjacency and walks are the same as for digraphs, except that there is no longer any notion of direction. A *component* of a graph is a maximal subgraph any two of whose points lie of some walk. A set of lines is *independent* if no two are incident to the same point. A *bigraph* is a graph G whose point set V can be partitioned into two sets V_1 and V_2 such that all lines of G join points of V_1 with points of V_2. A *line-coloring* of a graph is an assignment of colors to the lines in such a way that any two lines which are incident with the same point receive different colors. The smallest n such that G can be line-colored with n colors is the *line-chromatic number* $\chi'(G)$. Clearly, the line-chromatic number of a graph G is not less than $\Delta(G)$, the maximum of the degrees of the points of G. For a bigraph, a stronger statement can be made ([6, p. 171]): for any bigraph G, $\chi'(G) = \Delta(G)$.

Now, let M be a machine with states $\{v_1, \cdots, v_p\}$, and form a bigraph G with points $\{v_{1i}, v_{2i} | i = 1, \cdots, p\}$, where v_{1i} is adjacent to v_{2j} if and only if $v_iv_j \in D(M)$; these are the only lines in G. Note how the degrees of the points of G are related to these of the points of $D(M)$: $\deg(v_{1i}) = \operatorname{od}(v_i)$ and $\deg(v_{2i}) = \operatorname{id}(v_i)$. If we "color" each line $v_{1i}v_{2j}$ of G with the corresponding input symbol $\xi(v_iv_j)$, then if two lines are incident with the same point v_{1i}, they are colored differently, since M is a deterministic machine. If lines x and y with $\xi(x) = \xi(y)$ are incident to point v_{2i}, there is a merge. As the preceding lemma shows, there must be merges at states v_i with $\operatorname{id}(v_i) > t$, the number of inputs. To prove Theorem 6 we will show that inputs may be assigned to the arcs of $D(M)$ in such a way that the resulting machine is complete and deterministic, hence realizing M with feedback encoding, and the only merges occur at v_i with $\mathrm{id}(v_i) > t$. If the line-chromatic number of the associated bigraph G is $\chi'(G) \geqq t$, we will color the lines of G from a set of colors $\{\beta_1, \cdots, \beta_{\Delta(G)}\}$ in such a way that only colors from $\{\beta_1, \cdots, \beta_t\}$ are used to color lines incident with points v_{2i} with $\deg(v_{2i}) \leqq t$. If we translate each color β_i in $\{\beta_1, \cdots, \beta_t\}$ to the input symbol x_i and assign input symbols from $\{x_1, \cdots, x_t\}$ to lines colored from $\{\beta_{t+1}, \cdots, \beta_\Delta\}$ in such a way as to give a complete deterministic machine M', then there will be no merges at states v_i with $\mathrm{id}(v_i) \leqq t$. Furthermore, if $\mathrm{id}(v_j) > t$, then since in the coloring of G each color $\beta_1, \cdots, \beta_t$ appears once on a line incident with v_{2j}, the number of merges at v_j will be exactly $\mathrm{id}(v_j) - t$. Thus we will have $\Xi(M') = \hat{\Xi}(D(M))$. To prove Theorem 6, it is then necessary only to demonstrate that a coloring of the prescribed type always exists.

THEOREM 7. *Let G be a bigraph in which* $\max\{\deg u | u \in V_1\} = n = d_0$, *and suppose that the degrees greater than n which are realized in V_2 are $n < d_1 < d_2 < \cdots < d_r = \Delta$. Then there is a line-coloring of G from $\{\beta_1, \cdots, \beta_\Delta\}$ such that all lines colored $\beta_{d_i+1}, \cdots, \beta_{d_{i+1}}$ are incident to points of degree greater than d_i, for $i = 0, \cdots, r - 1$.*

To prove the theorem, we first develop a sequence of lemmas.

LEMMA. *Let G be a bigraph such that all points of maximum degree are in V_2. Then G can be line-colored from $\{\beta_1, \cdots, \beta_\Delta\}$ such that the only lines colored β_Δ are incident with points of maximum degree.*

Proof. We suppose the result to be true for bigraphs with $q - 1$ lines. Let G have q lines and let all points of maximum degree be in V_2; let $x = uv$ be incident with one such point v. If $\Delta(G - x) < \Delta(G) = \Delta$, then v was the only point of maximum degree in G. So any line-coloring of $G - x$ from $\{\beta_1, \cdots, \beta_{\Delta-1}\}$ extends to a line-coloring of G in which only x is colored β_Δ.

If $\Delta(G - x) = \Delta(G) = \Delta$, then we can color $G - x$ with Δ colors so that all lines colored β_Δ are incident with points of maximum degree; in particular, no line colored β_Δ is incident with v. If there is no β_Δ-line at u, then x can be colored β_Δ in G. Otherwise, there is a β_Δ-line uv_1 (where $\deg v_1 = \Delta$). Since $\deg u < \Delta$, there is some color α which does not appear at u. Clearly, however, there is a line v_1u_1 colored α. Thus we get a sequence $\langle u = u_0, v_1, u_1, v_2, \cdots, \rangle$ such that each v_i has maximum degree, each u_jv_{j+1} is colored β_Δ, and each v_ju_j is colored α. At each step of this process we are choosing a new point. If we have just chosen v_j, then u_j cannot be a previously chosen $u_k : u_j \neq u_0$ as α does not appear at u_0 and u_j cannot be some other previously chosen u_k, for otherwise there would be two lines colored α incident to u_k. Similarly, if u_j has just been chosen, v_{j+1} cannot be a previously chosen v_k, or otherwise there would be two lines colored β_Δ at v_k. Of course, since $G - x$ is a bigraph, no v_j can be equal to any u_k. Then since $G - x$ is finite, this process must terminate when we are unable to choose a new u_j or a new v_{j+1}. Since $\deg v_j = \Delta$, the process cannot stop with a v_j, so it must stop at some u_j at which there is no β_Δ-line. We have thus defined a component of the subgraph $G - x|_{\alpha,\beta_\Delta}$, and can interchange the colors α and β_Δ in this subgraph, preserving the validity of the coloring. But now β_Δ does not appear at u, so x can be colored with β_Δ.

LEMMA. *In a bigraph G, suppose* $\max\{\deg u | u \in V_1\} = n$ *and that there are at least two degrees greater than or equal to n realized by points of V_2, the two* *largest being $n \leqq \Delta' < \Delta$. Then there is a line-coloring of G from $\{\beta_1, \cdots, \beta_\Delta\}$ such that all lines colored $\beta_{\Delta'+1}, \cdots, \beta_\Delta$ are incident with points of maximum degree.*

Proof. Clearly the result is true whenever $n = 1$. Suppose it to be true for $n - 1$, and suppose that in G, $\max\{\deg u | u \in V_1\} = n$. Note that if $\Delta - \Delta' = 1$, then the result holds by the previous lemma.

Suppose now that the result is true for $\Delta - \Delta' = t - 1$, and that in G, $\Delta - \Delta' = t$, where $v_1, \cdots, v_r$ are the points of degree Δ. We remove an independent set X of r lines, one adjacent to each of $v_1, \cdots, v_r$, to get $G' : \Delta(G') = \Delta(G) - 1$. If, in G', $\max\{\deg u | u \in V_1\} = n$, then G' can be colored with $\Delta(G) - 1$ colors in the prescribed manner: the only lines colored $\beta_{\Delta'+1}, \cdots, \beta_{\Delta-1}$ are incident with the v_r. Now, the lines of X can be colored β_Δ.

Otherwise, $\max\{\deg u | u \in V_1\} = n - 1$. By induction on n, a line-coloring of G' with the desired properties can be achieved, and this coloring uses only $\Delta - 1$ colors, as above. Again, the lines of X can be colored β_Δ.

LEMMA. *Let G be a bigraph in which $\max\{\deg u | u \in V_1\} = n$, $\Delta(G) = \Delta > n$, and suppose there are no points of degree $n + 1, \cdots, \Delta - 1$. Then G has a line-coloring from $\{\beta_1, \cdots, \beta_\Delta\}$ such that all lines colored $\beta_{n+1}, \cdots, \beta_\Delta$ are incident with points of maximum degree.*

Proof. We know the result is true, for any n, if $\Delta = n + 1$. Also, the result is trivially true whenever $n = 1$. Suppose that the result holds when $\max\{\deg u | u \in V_1\} = n - 1$, and let G have $\max\{\deg u | u \in V_1\} = n$. Since we know that the result holds for $\Delta = n + 1$, suppose that it holds for $\Delta = n + k - 1$, and let G have $\Delta(G) = \Delta = n + k$. Suppose the points of degree Δ are $v_1, \cdots, v_r$. Remove an independent set X of lines which covers $\{v_1, \cdots, v_r\}$, and let $G - X = G'$. If, in G', $\max\{\deg u | u \in V_1\} = n$, then the resulting graph satisfies the conditions of the theorem with $\Delta = n + k - 1$, so there is a line-coloring where all lines colored $\beta_{n+1}, \cdots, \beta_{n+k-1}$ are incident with the v_i. Then the lines in X can be colored β_{n+k} and the result holds.

Otherwise, in G', $\max\{\deg u | u \in V_1\} = n - 1$. Then the result holds for G' by induction unless there were points of degree n in V_2. If so, G' satisfies the conditions of the previous lemma with $\Delta'(G') = n$, $\Delta(G') = \Delta - 1$, and so there is a line-coloring of G' from $\{\beta_0, \cdots, \beta_{\Delta-1}\}$ such that all lines colored $\beta_{\Delta'+1} = \beta_{n+1}, \cdots, \beta_{\Delta-1}$ are incident with the v_i. By coloring the lines of X with β_Δ the result holds. If V_2 had no points of degree n, then by the inductive hypothesis on n, in the line-coloring of G' all lines colored $\beta_n, \beta_{n+1}, \cdots, \beta_{\Delta-1}$ are incident with the v_i. We can again color the lines of X with β_Δ, proving the theorem.

Proof of Theorem 7. The result is trivial for $n = 1$. Also, by the last lemma, it holds whenever $r = 1$, so we can assume it true for bigraphs in which $\max\{\deg u | u \in V_1\} \leqq n - 1$ and also for bigraphs in which $\max\{\deg u | u \in V_1\} = n$ and $r - 1$ degrees greater than n are realized in V_2. Let G have $\max\{\deg u | u \in V_1\} = n$, and let degrees $n < d_1 < \cdots < d_r = \Delta$ be realized in V_2. We first prove the result in the case $d_r - d_{r-1} = 1$. Suppose we remove an independent set X covering the points of degree d_r to get a graph G'. If the maximum degree of the points in V_1 is reduced to $n - 1$, then G' has a line-coloring of the desired type; in particular, all lines colored $\beta_{d_{r-2}}, \cdots, \beta_{d_{r-1}}$ are incident with points of degree d_{r-1} in G', those being the points of degree d_{r-1} or d_r in G. Then by coloring the lines of X with β_Δ, the desired coloring results. If in G' the maximum degree of the points in V_1 is n, then the inductive hypothesis on r guarantees the desired coloring for G', and again we can color the lines of X with β_Δ.

Now suppose the result holds for $d_r - d_{r-1} = t - 1$ and suppose that in G, $d_r - d_{r-1} = t$. Let $v_1, \cdots, v_s$ be the points of degree $d_r = \Delta$. We can remove an independent set of lines X which covers $\{v_1, \cdots, v_s\}$, giving a graph G' with maximum degree $\Delta - 1$ and next largest degree d_{r-1}; note that $(\Delta - 1) - d_{r-1} = t - 1$. If, in G', $\max\{\deg u | u \in V_1\} = n$, we get a line-coloring of G' from $\{\beta_1, \cdots, \beta_{\Delta-1}\}$ with the desired properties by the inductive hypothesis on t. If not, we get a line-coloring by the inductive hypothesis on n. Either way, we can color the lines of X with β_Δ.

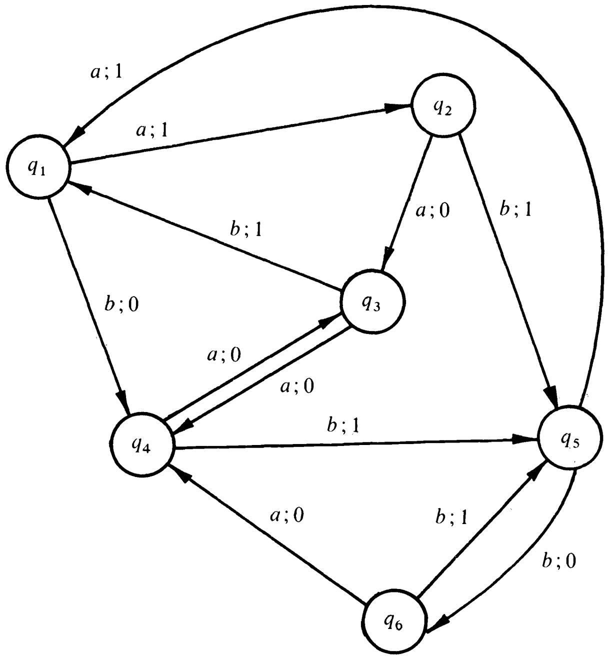


FIG. 4.

Example 2. Consider the machine M in Fig. 4. There are convergences at states q_3, q_4 and q_5, and M certainly does not have a distinguishing sequence. The associated bigraph G has $\Delta(G) = 3$, and so can be line-colored with three colors. After coloring G in accordance with the theorem, we get the machine in Fig. 5, which has only two convergences.

As we have noted, the procedure which we have outlined reduces merges rather than convergences, and, for machines M in which $\hat{\Xi}(D(M))$ is small but nonzero, may actually increase the number of convergences. Nevertheless, there are two cases in which the procedure can be shown to be of definite advantage. We state these as corollaries.

COROLLARY. *If M has t inputs, then M can be isomorphically realized with feedback encoding by a convergence-free machine if and only if every state q has* id $(q) = t$.

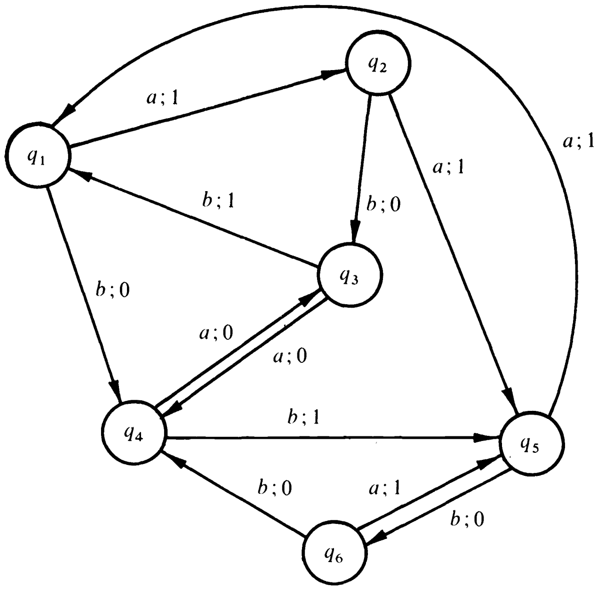


FIG. 5.

Such machines, of course, also satisfy Theorem 4; this corollary, therefore, provides an alternate method of attack.

Any machine M can be isomorphically realized with feedback encoding by a machine which has at most $\hat{\Xi}(D(M))$ convergences, since every convergence is a merge. Of special interest is the case when M has more than $\hat{\Xi}(D(M))$ convergences.

COROLLARY. *If M has more than $\hat{\Xi}(D(M))$ convergences, then M can be isomorphically realized with feedback encoding by a machine with fewer convergences.*

3. Conclusion. We have covered both special-case and general applications of the concept of realization with feedback encoding to the distinguishing sequence problem. In some cases, such as Theorem 4, the results should have real value in the cases to which they apply. In the more general settings, such as Theorem 6, the practicality may be less obvious: a more effective algorithm than the inductive proof of Theorem 7 would be desirable, and the theorem focuses on merges rather than convergences.

In this paper, and in [2], we have tried to show the potential of the notion of feedback encoding, much of which rises from its transformation of "behavioral" properties to "structural" ones. At this stage, the concept is probably best looked upon as a heuristic which might be tried in a particular problem. Work still needs to be done on estimating the costs incurred by the techniques proposed and on developing reliable algorithms which will indicate whether a particular problem is amenable to them.

REFERENCES

[1] C. BERGE, *The Theory of Graphs*, Methuen, London, 1964.

[2] D. GELLER, *Realization with feedback encoding. I: Analogues of the classical theory*, this Journal, 4 (1975), pp. 12–33.

[3] A. Ghouila-Houri, *Une condition suffisante d'existence d'un circuit Hamiltonien*, C. R. Acad. Sci. Paris, 251 (1960), pp. 495–497.

[4] A. Gill, *Introduction to the Theory of Finite-State Machines*, McGraw-Hill, New York, 1962.

[5] F. Hennie, *Finite State Models for Logical Machines*, John Wiley, New York, 1968.

[6] D. König, *Theorie der Endlichen und Unendlichen Graphen*, Chelsea, New York, 1950.

[7] R. McNaughton and T. Booth, *General switching theory*, Tech. Documentary Rep. ASD-TDR-62-599, Moore School of Electrical Engrg., Univ. of Pa., Philadelphia, 1962.

[8a] J. Meyer, *Theory and design of reliable spacecraft data systems*, Systems Engrg. Lab., Univ. of Michigan, Ann Arbor, September 1970.

[8b] J. Meyer and K. Yeh, *Diagnosable machine realizations of sequential behavior*, Digest of the 1971 International Symposium on Fault-Tolerant Computing, IEEE Computer Society Publications, 1971.

[9] R. Miller and S. Winograd, *On the number of transitions entering the states of a finite automaton*, IBM Res. Note NC-320, 1963; IRE Trans. Electronic Computers, EC-13 (1964), pp. 463–464.

[10] E. Moore, *Gedanken experiments on sequential machines*, Automata Studies, C. E. Shannon and J. McCarthy, eds., Princeton University Press, Princeton, N.J., 1956.

[11] D. R. Woodall, *Sufficient conditions for circuits in graphs*, Proc. London. Math. Soc. (Sec. 3), 24 (1972), pp. 739–755.

[12] M. Yoeli and A. Ginzburg, *On homomorphic images of transition graphs*, J. Franklin Inst., 278 (1964), pp. 291–296.

SIAM J. COMPUT.
Vol. 4, No. 1, March 1975

COMPUTATIONAL COMPLEXITY OF INNER PRODUCTS OF VECTORS (AND THAT OF OTHER BILINEAR FORMS) OVER A NONCOMMUTATIVE RING (AUXILIARY FUNCTIONS ALLOWED)*

ROBERT MANDL† AND THOMAS VARI‡

Abstract. We prove that the minimum number of ring multiplications necessary to compute the inner product of two n-vectors over a noncommutative ring is n, even if any number of auxiliary functions (each being a polynomial in the elements of exactly one of the vectors) are allowed "for free". This is the "noncommutative" analogue of a result of Winograd's stating that the minimum number of multiplications needed to compute the inner product over a commutative ring is n, if auxiliary functions are not allowed, and, respectively, $\approx (n/2)$, if auxiliary functions are allowed. More generally, given a bilinear form whose matrix is S, the minimum number of multiplications necessary to compute it over a noncommutative ring is rank(S), whether or not auxiliaries are allowed. The method used is a modification of Floyd's linear algebra approach: the inner product $\mathbf{x} \cdot \mathbf{y}$ (or any other bilinear form) is regarded as a quadratic form in the $2n$ indeterminates $x_1, \cdots, x_n, y_1, \cdots, y_n$ (rather than as a bilinear form in the two separate sets of n indeterminates).

The same method can also be applied to bilinear forms over commutative rings; when the form is the inner product, the method yields an improved lower bound, thereby closing the difference between Winograd's achievable upper bound $\lceil n/2 \rceil$ and his proven lower bound $\lfloor n/2 \rfloor$.

Another result of Winograd's, that the minimum number of binary operations necessary for computing the inner product is $2n - 1$ (even if auxiliary functions are allowed), can also be extended to noncommutative rings.

Key words. fast matrix multiplication, computational complexity, inner product, auxiliary functions

1. Introduction.

The interest in the computational complexity of inner products of vectors over noncommutative rings arose from work on fast matrix multiplication. It is known that two $n \times n$ matrices can be multiplied with only $O(n^{\log_2 7})$ multiplications of matrix elements (exactly $n^{\log_2 7}$ if n is a power of 2) [4]; if the matrix elements are from a commutative ring, the matrix product can be obtained in $\sim\frac{1}{2}n^3$ ring multiplications ([7]; see also [5]). Although Strassen's method, for sufficiently large n, requires fewer multiplications than any other known method, Winograd's is better for small values of n ($\frac{1}{2}n^3 < n^{\log_2 7}$ if $\log_2 n < 1/(3 - \log_2 7) \approx 5.19$, i.e., for $n \leqq \lfloor 2^{5.19} \rfloor = 36$; moreover, it has been determined from practical tests [1] that Winograd's method is faster than Strassen's for $n \lesssim 250$), and if for even one such value of n some procedure, similar to Winograd's but *not* using commutativity, yielded the matrix product with fewer multiplications than Strassen's, then that procedure could be extended to a general method (*faster* than Strassen's) in the same way in which Strassen parlayed his "2 × 2-using-7" procedure into a general method. For some general

* Received by the editors December 6, 1972. This work was supported in part by Project MAC, an MIT research program sponsored by the Advanced Research Projects Agency, Department of Defense, under ONR Contract N00014-70-A-0362-001.

† Project MAC, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139.

‡ York University, Downsview, Ontario, Canada.

remarks and results on the influence of commutativity (or lack of it) on the complexity of algorithms, the reader is referred to Hopcroft and Kerr [3] (especially p. 30 and p. 35).

It is thus necessary to determine whether the inner product of n-vectors can be computed, without using commutativity, through fewer than n multiplications. This question was asked by A. R. Meyer in September 1970, and the present paper (also written in September 1970, except for the Introduction) answers it in the negative. Of course, we have to allow auxiliary functions, since otherwise the minimum number of multiplications is n even if the ring is commutative [6]; for these rings it is known that the presence of auxiliary functions reduces the minimum to $n/2$[9].

The method used in our proof follows Floyd's linear algebra approach [2], but with the following difference: the inner product $\mathbf{x} \cdot \mathbf{y}$ is regarded not as a bilinear form in the two sets of indeterminates $x_1, x_2, \cdots, x_n$ and $y_1, y_2, \cdots, y_n$, but rather as a quadratic form in the $2n$ indeterminates $x_1, x_2, \cdots, x_n, y_1, y_2, \cdots, y_n$. The algorithms considered involve additions, subtractions, multiplications by constants, and general multiplications (not by constants); we are counting only the latter (multiplications by nonconstants). Each operand is an indeterminate or a constant or an allowable auxiliary function (a polynomial in the components of $\mathbf{x}$ or in the components of $\mathbf{y}$; its value is available "for free") or the result of a previous operation.

2. Preliminary reductions. The following three theorems effect preliminary reductions of the form of the algorithm; they correspond to Floyd's Theorems 1–3 [2], but in the present formulations *products are not assumed to be commutative* and *auxiliary functions are allowed free.* We shall assume that the algorithm steps situated between each general multiplication and the next following general multiplication have been consolidated into a "macro-step". Since polynomials in a vector are not defined, we shall sometimes shorten "polynomial in $x_1, \cdots, x_n$" to "polynomial in $\mathbf{x}$".

THEOREM 1. *Without loss of generality, we may assume that the expressions being multiplied have no constant terms.*

THEOREM 2. *Without loss of generality, we may assume that the expressions being multiplied have no nonlinear terms.*

THEOREM 3. *Without loss of generality, we may assume that the general products are of the form*

$$\left(\sum_j a_{ij}x_j + \sum_j b_{ij}y_j\right)\left(\sum_j c_{ij}x_j + \sum_j d_{ij}y_j\right). \tag{1}$$

The proofs are the same as Floyd's and will not be repeated here.

The inner product is therefore computed according to a formula of the form

$$\mathbf{x} \cdot \mathbf{y} = \sum_i \left[\left(\sum + \sum\right)\left(\sum + \sum\right)\right] + P(\mathbf{x}) + Q(\mathbf{y}), \tag{2}$$

where P and Q are polynomials.

Remark. Without loss of generality, we might assume that for each i,

$$\sum_j (a_{ij}^2 + c_{ij}^2) \cdot \sum_j (b_{ij}^2 + d_{ij}^2) \neq 0$$

(because otherwise the resulting quadratic forms would depend on $\mathbf{x}$ alone or on $\mathbf{y}$ alone, and could be incorporated in the "free polynomials"). This is not essential to the argument.

Both "free polynomials" are homogenous of the second degree (since so are $\mathbf{x} \cdot \mathbf{y}$ and $\sum(\cdot)(\cdot)$), and therefore they may be written in the form

$$P(\mathbf{x}) = \sum_i \sum_j p_{ij} x_i x_j = (x_1 \quad x_2 \cdots x_n)\mathbf{P} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \mathbf{x}'\mathbf{P}\mathbf{x},$$

where $\mathbf{P} = \operatorname{mat}_{ij}(p_{ij}) = \sum_i \sum_j p_{ij} E_{ij}$ and prime denotes transposition, and

$$Q(y) = \sum_i \sum_j q_{ij} y_i y_j = \cdots = \mathbf{y}'\mathbf{Q}\mathbf{y}.$$

3. The main theorem.

THEOREM 4. *The minimum number of multiplications required to compute the inner product of two vectors over a noncommutative ring is n, the dimensionality of the vectors, even when auxiliary functions (polynomials depending only on* $\mathbf{x}$, *or only on* $\mathbf{y}$) *are allowed "for free".*

Proof. Let us define

$$\mathbf{M}_0 = \mathbf{P} \oplus \mathbf{Q} = \left[\begin{array}{ccc|ccc} p_{11} & \cdots & p_{1n} & & & \\ p_{21} & \cdots & p_{2n} & & 0 & \\ \vdots & & & & & \\ p_{n1} & \cdots & p_{nn} & & & \\ \hline & & & q_{11} & \cdots & q_{1n} \\ & 0 & & q_{21} & \cdots & q_{2n} \\ & & & \vdots & & \vdots \\ & & & q_{n1} & \cdots & q_{nn} \end{array}\right],$$

$$\mathbf{z} = (x_1 \quad x_2 \quad \cdots \quad x_n \quad y_1 \quad y_2 \quad \cdots \quad y_n)',$$

$$\mathbf{v}_i = (a_{i1} \quad a_{i2} \quad \cdots \quad a_{in} \quad b_{i1} \quad b_{i2} \quad \cdots \quad b_{in})',$$

$$\mathbf{w}_i = (c_{i1} \quad c_{i2} \quad \cdots \quad c_{in} \quad d_{i1} \quad d_{i2} \quad \cdots \quad d_{in})',$$

$$\mathbf{U}^n = \left[\begin{array}{c|c} 0 & \begin{matrix} 1 & & & & & \\ & 1 & & & 0 & \\ & & 1 & & & \\ & & & \ddots & & \\ & 0 & & & 1 & \\ & & & & & 1 \end{matrix} \\ \hline 0 & 0 \end{array}\right].$$

Then the ith product can be written as $(\mathbf{z}'\ \mathbf{v}_i)(\mathbf{w}_i'\ \mathbf{z})$, or, using the associativity of matrix multiplication, $\mathbf{z}'\ \mathbf{M}_i\ \mathbf{z}$, where $\mathbf{M}_i$ is the ("external") product $\mathbf{v}_i\mathbf{w}_i'$, a $(2n) \times (2n)$ matrix. The identity (2) becomes

$$\mathbf{z}'\ \mathbf{U}^n\ \mathbf{z} = \sum_i (\mathbf{z}'\ \mathbf{M}_i\ \mathbf{z}) + \mathbf{z}'\ \mathbf{M}_0\ \mathbf{z}, \tag{3}$$

or

$$\mathbf{M}_0 + \mathbf{M}_1 + \mathbf{M}_2 + \cdots = \mathbf{U}^n. \tag{4}$$

Thus we see that the inner product of two n-vectors can be computed by at most $n - 1$ multiplications if and only if one can find "suitably formed" matrices $\mathbf{M}_0, \mathbf{M}_1, \cdots, \mathbf{M}_{n-2}, \mathbf{M}_{n-1}$ such that (4) is satisfied:

$$\mathbf{M}_0 + \mathbf{M}_1 + \cdots + \mathbf{M}_{n-1} = \mathbf{U}^n, \tag{4'}$$

or, more exactly, iff there exist two $n \times n$ matrices $\mathbf{P}$, $\mathbf{Q}$ and $n - 1$ pairs of $2n$-vectors $\mathbf{v}_i, \mathbf{w}_i$ such that

$$(\mathbf{P} \oplus \mathbf{Q}) + \sum_1^{n-1} \mathbf{v}_i\mathbf{w}_i' = \mathbf{U}^n. \tag{5}$$

Since for each i, rank $(\mathbf{v}_i\mathbf{w}_i') = 1$ (except in the trivial case $|\mathbf{v}_i| = |\mathbf{w}_i| = 0$), we necessarily have rank $(\sum_1^{n-1} \mathbf{v}_i\mathbf{w}_i') \leqq n - 1$. All the $n \times n$ submatrices of $\sum_1^{n-1} \mathbf{M}_i$ have therefore null determinants, and in particular this is true of the one in the upper right corner. Since the addition of $\mathbf{M}_0$ does not affect that corner ($\mathbf{M}_0 = \mathbf{P} \oplus \mathbf{Q}$), the upper right $n \times n$ submatrix of $\sum_0^{n-1} M_i$ is *singular*, while for $\mathbf{I}_n$, the similarly placed submatrix of $\mathbf{U}^n$, we have $\det(\mathbf{I}_n) = 1 \neq 0$. Thus the assumption that $n - 1$ multiplications are sufficient has led to a contradiction, and the necessity of at least n multiplications is proved.

4. Generalizations.

THEOREM 5. *The minimum number of multiplications required to evaluate a bilinear form $\sum_i \sum_j s_{ij}x_iy_j$ of full rank over a noncommutative ring is n, the dimensionality of the vectors, even when auxiliary functions (polynomials depending only on $\mathbf{x}$, or only on $\mathbf{y}$) are allowed "for free".* (A bilinear form is *of full rank* if its nullity is 0, i.e., if its rank equals the dimensionality of the vectors upon which it operates.)

Proof. The proof is similar to the proof of Theorem 4.

THEOREM 6. *The minimum number of multiplications required to evaluate a bilinear form $\sum_i \sum_j s_{ij}x_iy_j$ over a noncommutative ring is equal to the rank of the associated matrix $\mathbf{S}$, even when auxiliary functions (polynomials depending only on $\mathbf{x}$, or only on $\mathbf{y}$) are allowed "for free".*

Proof. The proof is similar to the proof of Theorem 4.

5. Bilinear forms over a commutative ring.

For the case when the evaluation of bilinear forms is effected without the benefit of auxiliary functions, Floyd's Theorem 5 [2] yields the following corollary.

COROLLARY. *The minimum number of multiplications required to evaluate a bilinear form $\sum_i \sum_j s_{ij}x_iy_j$ over a commutative ring (no auxiliary functions allowed) is $\max(\pi(S)), \nu(S))$, where $\pi(S)$ and $\nu(S)$ are, respectively, the positive index of inertia and the negative index of inertia of S.*

Floyd proved this for quadratic forms; if a bilinear form could be evaluated with fewer multiplications, then so could a quadratic form (by identification), thus yielding a contradiction and proving the corollary.

We remark that, by our Theorem 6, if the ring is not commutative, the minimum number of multiplications is not the larger of the two indices of inertia, but their sum ($=\operatorname{rank}(\mathbf{S})$).

Turning our attention now to the case when the evaluation of the bilinear forms is effected with the benefit of auxiliary functions, we denote by $\mathbf{R}$ the upper right $n \times n$ submatrix of the $2n \times 2n$ matrix $\mathbf{M}_0 + \mathbf{M}_1 + \mathbf{M}_2 + \cdots$, and by $\mathbf{L}$ its lower left $n \times n$ submatrix. When the ring was not commutative (see Theorem 4) the problem reduced to the existence of vectors $\mathbf{v}_i$, $\mathbf{w}_i$ such that $\mathbf{R} = \mathbf{S}$ and $\mathbf{L} = \mathbf{0}$. In our case, however, a term αx_iy_j could be represented as $\beta x_iy_j + (\alpha - \beta)y_jx_i$, and therefore we are content if we achieve $\mathbf{R} + \mathbf{L}' = \mathbf{S}$. For a given $\mathbf{S}$, the minimum number of multiplications necessary to compute the associated form is the minimum, over *all* $n \times n$ matrices $\mathbf{X}$, of the numbers of the form $\max(\operatorname{rank}(\mathbf{X}'), \operatorname{rank}(\mathbf{S} - \mathbf{X}))$, i.e., it is

$$\min_{\mathbf{X}} \max(\operatorname{rank}(\mathbf{X}), \operatorname{rank}(\mathbf{S} - \mathbf{X}))$$

(noticing that $\operatorname{rank}(\mathbf{X}') = \operatorname{rank}(\mathbf{X})$). This is certainly a lower bound for the number of multiplications, but it seems to be quite intractable in its general form. We shall therefore derive another bound, possibly worse but a little more tractable. Let us restrict ourselves to matrices $\mathbf{X}$ of the form $\operatorname{mat}_{ij}(\varepsilon_{ij}s_{ij})$ where the ε's are 0 or 1; this corresponds to the following: when a term αx_iy_j is represented as

$\beta x_i y_j + (\alpha - \beta) y_j x_i$, β is restricted to the two values α and 0. (We are trying to maximize the number of null entries in the matrices **R** and **L**). Thus the new bound is

$$\min_{(\forall i,j)\varepsilon_{ij}B\in\{0,1\}} \max\left(\operatorname{rank}\left(\operatorname{mat}_{ij}(\varepsilon_{ij}s_{ij})\right), \operatorname{rank}\left(\operatorname{mat}_{ij}(\bar{\varepsilon}_{ij}s_{ij})\right)\right)$$

($\bar{a}$ means $1 - a$), and we conjecture that this new bound is no worse than the previous one.

Particular case. The form to be evaluated is the inner product; $\mathbf{S} = \mathbf{I}_n$. In this case, we have

$$\begin{aligned}\operatorname{rank} \mathbf{X} = \operatorname{rank}\left(\operatorname{mat}_{ij}(\varepsilon_{ij}s_{ij})\right) &= \operatorname{trace}\left(\operatorname{mat}_{ij}(\varepsilon_{ij}s_{ij})\right) = \operatorname{trace}\left(\operatorname{mat}_{ij}(\varepsilon_{ij})\right) \\ &= \sum_i \varepsilon_{ii} = \text{the number of 1's in } \mathbf{X},\end{aligned}$$

and similarly

$$\operatorname{rank}(\mathbf{S} - \mathbf{X}) = \cdots = \text{the number of 1's in } \mathbf{S} - \mathbf{X}.$$

Noticing that $\operatorname{rank}(\mathbf{X}) + \operatorname{rank}(\mathbf{S} - \mathbf{X}) = \text{const.} = n$, we shall write the minimax in the form

$$\min_{\substack{k \leq n \\ k \text{ integer}}} \max(n - k, k),$$

which readily evaluates to $\lceil n/2 \rceil$. We have thus rederived Winograd's result [9] and at the same time improved it from $\lfloor n/2 \rfloor$ to $\lceil n/2 \rceil$. Since the achievable bounds [7] for n even and for n odd both coincide with the new bound, the new bound is exact, and the abovementioned conjecture passes the test in this particular case.

6. Minimal number of binary operations. We conclude this paper by making a few remarks on the total number of binary operations (of all kinds) required to compute the inner product of n-vectors over noncommutative rings. First, Winograd's result for commutative rings [8], stating that $2n - 1$ operations are necessary, holds also for noncommutative rings since its proof did not use commutativity (except in examples). Second, this lower bound can be improved, in both cases (commutative/noncommutative rings), to

$$(2n - 1) + (\text{number of auxiliary functions actually used}).$$

In particular, this implies that for the case of commutative rings, where

$$(\min \# \text{ multipl.}) + (\min. \# \text{ add./s.}) = (n - 1) + \left\lceil \frac{n}{2} \right\rceil = \left\lceil \frac{3}{2} n \right\rceil - 1 < 2n - 1$$

and the two minima are not achieved simultaneously, any attempt at reducing the number of multiplications will cause an increase in the number of additions, which not only offsets the saving in multiplications but actually *exceeds* that saving, the excess being equal to the number of auxiliary functions actually used. This is exemplified by the fact that when we use Winograd's procedure for reducing the number of multiplications from n to $n/2$, the number of additions increases from $n - 1$ to $\frac{3}{2}n + 1$, i.e., by 2 units more than the saving in multiplications, and exactly 2 auxiliary functions are used by the algorithm.

REFERENCES

[1] R. P. Brent, *Error analysis of algorithms for matrix multiplication and triangular decomposition using Winograd's identity*, Numer. Math., 16 (1970), pp. 145–156.

[2] R. W. Floyd, *Notes on computational complexity of inner products, matrix-vector products, matrix products, and sets of quadratic forms*, to appear.

[3] J. E. Hopcroft and L. R. Kerr, *On minimizing the number of multiplications necessary for matrix multiplication*, SIAM J. Appl. Math., 20 (1971), pp. 30–35.

[4] V. Strassen, *Gaussian elimination is not optimal*, Numer. Math., 13 (1969), pp. 354–356.

[5] A. Waksman, *On Winograd's algorithm for inner products*, IEEE Trans. Computers, 19 (1970), pp. 360–361.

[6] S. Winograd, *On the number of multiplications required to compute certain functions*, Proc. Nat. Acad. Sci. USA, 58 (1967), pp. 1840–1842.

[7] ———, *A new algorithm for inner product*, IEEE Trans. Computers, 17 (1968), pp. 693–694.

[8] ———, *On the algebraic complexity of inner product*, IBM Res. Rep. RC-2729, 1969.

[9] ———, *On the number of multiplications necessary to compute certain functions*, Comm. Pure Appl. Math., 23 (1970), pp. 165–179.

SIAM J. COMPUT.
Vol. 4, No. 1, March 1975

FAITHFUL REPRESENTATION OF A FAMILY OF SETS BY A SET OF INTERVALS*

KAPALI P. ESWARAN†

Abstract. Let $Q = \{q_1, q_2, \cdots, q_m\}$ be a family of finite, nonempty sets, and $S = \bigcup_{q_i \in Q} \{q_i\}$. Suppose there exists a one-to-one function f that maps elements of S into points in the real line $\mathbb{R}$ such that for each $q_i \in Q$ there is an interval I_i containing images of all elements of q_i but not images of any elements not in q_i. Then the function f and the set of intervals $\{I_1, I_2, \cdots, I_m\}$ are said to faithfully represent Q. Necessary and sufficient conditions and an algorithm for faithful representation of Q are developed. An important kind of file organization, called the consecutive retrieval file organization, is shown to be a direct application of the property of faithful representation.

Key words. faithful representation, linear (total) ordering, consecutive retrieval file organization, intersection graph, complete graph, covering of a graph, directed semantic graph, Hamiltonian path, acyclic graph, query inverted file organization

1. Introduction. Consider a family of sets $Q = \{q_1, q_2, \cdots, q_m\}$, where q_i, for $1 \leqq i \leqq m$, is finite and nonempty. Let $S = \bigcup_{q_i \in O} \{q_i\} = \{a_1, a_2, \cdots, a_n\}$ denote the set of elements belonging to the sets in the family Q. Elements belonging to the set $S - q_i$ are called *foreign* with respect to q_i. Suppose there exists a one-to-one function f that maps the elements of S into (points in) the real line $\mathbb{R}$ such that for each $q_i \in Q$, there exists an interval I_i containing images of all elements $\in q_i$ but not images of any foreign elements with respect to q_i. Then we say that the family Q is *linearly orderable* (L.O.) and the function f and the set of intervals $\{I_1, I_2, \cdots, I_m\}$ *faithfully represent* the sets in Q. Hereafter, Q shall denote a family of sets $\{q_1, q_2, \cdots, q_m\}$ and S the set $\bigcup_{q_i \in Q} \{q_i\} = \{a_1, a_2, \cdots, a_n\}$. The statement that $(f;\ I_1, I_2, \cdots, I_m)$ faithfully represents Q means that the function f and the intervals $I_1, I_2, \cdots, I_m$ faithfully represent Q.

Example 1. Let $Q = \{q_1, q_2, q_3\}$, $q_1 = \{a_1, a_2, a_3\}$, $q_2 = \{a_2, a_3, a_5\}$ and $q_3 = \{a_4, a_5\}$. Let $f(a_1) = 1$, $f(a_3) = 2$, $f(a_2) = 3$, $f(a_5) = 4$ and $f(a_4) = 5$. Let $I_1 = [1, 3]$, $I_2 = [2, 4]$ and $I_3 = [4, 5]$ correspond to q_1, q_2 and q_3, respectively. Then $(f; I_1, I_2, I_3)$ faithfully represents Q. This is shown in Fig. 1 □

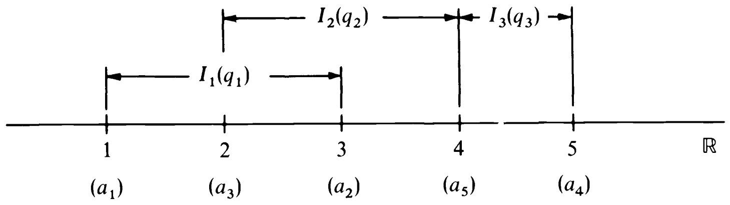


FIG. 1. *The preimages of i = 1, 2, 3, 4 and 5 are given below for each i in parenthesis. Intervals with the corresponding sets are also shown*

* Received by the editors September 14, 1973, and in revised form May 5, 1974.

† Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, California. Now at International Business Machines Corporation, San Jose, California 95114. This work was supported by the Naval Electronic Systems Command under Contract N00039-71-C-0255 while the author was at University of California at Berkeley.

One of the applications of the above model is in the area of information retrieval. Let us assume that we know the family of queries Q regarding a file F. We need to arrange the file on a *linear storage medium.* A storage medium S is called *linear* if the storage locations of S can be arranged linearly and the access time between any two storage locations is an increasing function of the distance between them. Tapes, tracks of a disk, books in a library shelf and shops in a street are examples of a linear storage medium. We shall assume that the storage devices are one-dimensional, i.e., the shops in the street have just the ground floor, etc.

Suppose that the query family Q is such that there exists a 1–1 function f which maps the records belonging to the file F into storage locations of a linear storage medium satisfying: (i) for each query $q_i \in Q$, there exists a sequence S_i of consecutive storage locations containing all records pertinent to q_i, and (ii) S_i does not contain any record not pertinent to q_i. We then say that the family of queries Q has the *consecutive retrieval property* (CR property) [1]. A file organization having this property is called a CR organization. Note that each record is stored only once. By knowing the first and the last pertinent records of a query in a CR organization, all relevant records of all queries can be retrieved. If the queries in Q are equally likely, then a CR organization for Q guarantees minimum overall retrieval time and minimum storage space. We can see that Q has the CR property if Q can be faithfully represented when we consider each record in the file as an element a_i and each query q_j as a set of elements.

The paper is in four sections. The results in § 2 concern a family of sets where every pair of sets in the family has at least one common element. Section 3 deals with the conditions under which the family $Q_1 \cup Q_2$ is faithfully representable, where Q_1 and Q_2 are faithfully representable families having pairwise nondisjoint sets. Section 4 extends the conditions of § 3 to a family of sets which is a union of more than two such families. Given a family of sets Q, we proceed as follows. (i) We shall express Q as a union of subfamilies such that the sets in each of the subfamilies are pairwise nondisjoint. (ii) We will check if each one of these subfamilies are faithfully representable. (iii) We will verify if the union of such faithfully representable subfamilies is faithfully representable.

The idea fits well under a graph theoretic background. There is a correspondence between a family Q of pairwise nondisjoint sets and a graph representation of Q called the *intersection graph* of Q.

2. Intersection graphs and faithfully representable families.

LEMMA 1. *If Q is linearly orderable, then $Q' \subseteq Q$ is linearly orderable.*

LEMMA 2. *Let $Q = \{q_1, q_2, \cdots, q_m\}$, $S = \bigcup_{q_i \in Q} \{q_i\} = \{a_1, a_2, \cdots, a_n\}$ and $\bar{q}_j = \{a_j\}$ for $1 \leqq j \leqq n$. Then Q is L.O. iff $\bar{Q}$ is L.O., where $\bar{Q} = Q \cup \{\bar{q}_i\}$, $i \in \{1, 2, \cdots, n\}$.*

By Lemmas 1 and 2, we can assume that as far as faithful representation (linear ordering) is concerned, no set in Q is a singleton.

The *intersection graph* of Q is denoted by $\Omega(Q)$ and is defined as follows: for each set $q_i \in Q$, there exists a corresponding node $\bar{q}_i$ in $\Omega(Q)$ and vice versa and for $i \neq j$, $\bar{q}_i$ is connected with $\bar{q}_j$ iff $q_i \cap q_j \neq \phi$. A graph G is called *complete* iff every pair of distinct nodes of G is joined by an edge in G.

Let $\{G_1, G_2, \cdots, G_p\}$ be a set of subgraphs of a graph G such that every node and edge of G is in at least one of the subgraphs $G_1, G_2, \cdots, G_p$. Then $\{G_1, G_2, \cdots, G_p\}$ is said to *cover* G. All these definitions and the ones that follow are more or less standard in graph theory (see [2], [3]).

Example 2. *Let* $Q = \{q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8\}$. *Let* $q_1 = \{a_1, a_2, a_3\}$, $q_2 = \{a_2, a_3, a_4, a_5\}$, $q_3 = \{a_2, a_3, a_4\}$, $q_4 = \{a_4, a_5, a_6\}$, $q_5 = \{a_4, a_5, a_6, b_1\}$, $q_6 = \{b_1, b_2\}$, $q_7 = \{a_7, a_8\}$ and $q_8 = \{a_7, a_9\}$.

We have $S = \bigcup_{q_i \in Q} \{q_i\} = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, b_1, b_2\}$. $\Omega(Q)$ is given in Fig. 2. Let R be the connectivity relation of $\Omega(Q)$, i.e., $\bar{q}_i\ R\ \bar{q}_j$ iff there is an edge between $\bar{q}_i$ and $\bar{q}_j$ in $\Omega(Q)$. Then

$$G_1 = [\overbrace{\{\bar{q}_1, \bar{q}_2, \bar{q}_3\}}^{\text{Set of nodes of } G_1}, R],$$

$$G_2 = [\{\bar{q}_2, \bar{q}_3, \bar{q}_4, \bar{q}_5\}, R],$$

$$G_3 = [\{\bar{q}_5, \bar{q}_6\}, R]$$

and

$$G_4 = [\{\bar{q}_7, \bar{q}_8\}, R]$$

are some of the complete subgraph of $\Omega(Q)$. We also see that G_1, G_2, G_3 and G_4 cover $\Omega(Q)$. □

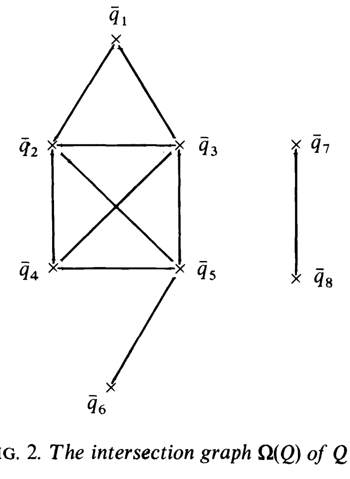


FIG. 2. *The intersection graph* $\Omega(Q)$ *of* Q

LEMMA 3. *If* $\Omega(Q)$ *is complete and* Q *is faithfully representable, then*

$$I = \bigcap_{q_i \in Q} q_i \neq \phi.$$

Proof. Let $Q = \{q_1, q_2, \cdots, q_m\}$, and let $\{f; I_1, I_2, \cdots, I_m\}$ faithfully represent Q. The proof is by induction on m. The basis of the induction is obvious.

Assume that the lemma is true for $m = k - 1$. Consider $m = k$. (i) Intervals $I_1, I_2, \cdots, I_{k-1}$ overlap (i.e., there exists an element in S whose image is in $I_1, I_2, \cdots, I_{k-1}$) since $q_1 \cap q_2 \cap q_3 \cdots q_{k-1} \neq \emptyset$ by induction. (ii) The interval I_k must overlap with each one of the intervals $I_1, I_2, \cdots, I_{k-1}$ since $q_k \cap q_i \neq \emptyset$ for $i = 1, 2, \cdots, k - 1$. (i) and (ii) imply the existence of an element $a_l \in S$ such that a_l is in $I_1, I_2, \cdots, I_k$. Since $(f; I_1, I_2, \cdots, I_k)$ faithfully represents Q, a_l is not foreign to any set in Q, i.e., $\bigcap_{q_i \in Q} q_i \neq \emptyset$. □

Lemma 3 and the following lemmas lead us to Theorem 1, which gives necessary and sufficient conditions for a family Q having pairwise nondisjoint sets to be faithfully representable. We shall draw a correspondence between the ordering of the elements of the sets of such a family and a Hamiltonian path in a graph defined on the elements. Let us define that graph.

Define a *directed semantic graph* $\bar{G} = [V, R, I]$. V is a finite nonempty set of nodes. R is an irreflexive relation on V such that for all $a_i, a_j \in V$, $i \neq j$, $a_i R a_j$ iff there is an edge from a_i to a_j in $\bar{G}$. R is the connectivity relation of $\bar{G}$. I is a subset of V. Nodes in I are called *direction-changer nodes* and are denoted by an * in $\bar{G}$. Nodes in $V - I$ are non-direction-changer nodes. $\{a_i, a_j\}$ denotes the edge between a_i and a_j, ignoring the direction on the edge. (a_i, a_j) denotes the directed edge from node a_i to node a_j. A *path* in a directed semantic graph (DSG) $\bar{G}$ is a sequence of distinct nodes $a_0, a_1, \cdots, a_i, a_{i+1}, \cdots, a_k$ of $\bar{G}$ such that for $0 \leqq i < k$, (a_i, a_{i+1}) is an edge of $\bar{G}$ when in direct mode and (a_{i+1}, a_i) is an edge of $\bar{G}$ when in reverse mode, where the modes are defined as follows. If a path starts with a non-direction-changer node, then the mode is direct. If it starts with a direction-changer node, the mode is reverse. Whenever a direction-changer node is reached from a non-direction-changer node, the mode is switched. (If a direction-changer node is reached from a direction-changer node, no change of mode occurs.) If $P = a_0, a_1, \cdots, a_k$ is a path of $\bar{G}$, then a_0 is called the *starting* node of P, a_k the *end* node of P and $a_1, a_2, \cdots, a_{k-1}$ the *intermediate* nodes of P. A *Hamiltonian path* in a DSG $\bar{G}$ is a path that passes through all the nodes of $\bar{G}$.

Example 3. Consider the DSG $\bar{G}$ in Fig. 3. $\bar{a}_2$ and $\bar{a}_3$ are direction-changer nodes. $\langle \bar{a}_1, \bar{a}_2, \bar{a}_3, \bar{a}_4 \rangle$, $\langle \bar{a}_2, \bar{a}_3, \bar{a}_1 \rangle$ are examples of paths in $\bar{G}$. $\langle \bar{a}_1, \bar{a}_2, \bar{a}_3, \bar{a}_4, \bar{a}_5 \rangle$, $\langle \bar{a}_5, \bar{a}_4, \bar{a}_3, \bar{a}_2, \bar{a}_1 \rangle$ are some of the Hamiltonian paths in $\bar{G}$. □

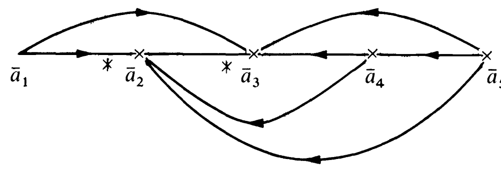


FIG. 3. *The* DSG $\bar{G}(Q)$ *of* Q

We now define the DSG *of a family of sets* Q. Let $I = \bigcap_{q_i \in Q} q_i$. Let $\bar{R}$ be an irreflexive relation defined on S as follows: $a_i \bar{R} a_j$ iff $i \neq j$ and for all $q_k \in Q$, $a_i \in q_k$ implies $a_j \in q_k$. Note that $\bar{R}$ is transitive. The DSG of Q is denoted by $\bar{G}(Q)$ and is $[S', \bar{R}, I']$. S' is the set of nodes of $\bar{G}(Q)$ and is $\{\bar{a}_1, \bar{a}_2, \cdots, \bar{a}_i, \cdots, \bar{a}_n\}$, where node $\bar{a}_i$ corresponds to element $a_i \in S$ and vice versa. $\bar{a}_i \bar{R} \bar{a}_j$ iff $a_i \bar{R} a_j$. We can use the same symbol $\bar{R}$ for a relation between two elements of S and for the connectivity relation of $\bar{G}(Q)$ since there is no confusion. I' is the set of direction-changer nodes of $\bar{G}(Q)$, with $\bar{a}_p \in I'$ iff $a_p \in I$.

Example 4. Let $Q = \{q_1 q_2, q_3\}$, $q_1 = \{a_1, a_2, a_3\}$, $q_2 = \{a_2, a_3, a_4, a_5\}$ and $q_3 = \{a_2, a_3, a_4\}$. Then $S = \{a_1, a_2, a_3, a_4, a_5\}$. For instance, $a_1 \bar{R} a_2$, $a_1 \bar{R} a_3$. Note that a_3 is not $\bar{R}$-related to a_1, since q_3 contains a_3 and not a_1. $\bar{G}(Q)$ is shown in Fig. 3.

LEMMA 4. *Let $\bar{G}(Q)$ be the* DSG *of Q and h be a Hamiltonian path in $\bar{G}(Q)$. Then there does not exist a subpath h' of h such that the starting and end nodes of h' are direction-changer nodes and the intermediate nodes are non-direction-changer nodes.*

Proof. Let $I = \bigcap_{q_i \in Q} q_i$. Assume, to the contrary, that there exists a subpath h' of $h = \langle \bar{a}_i, \bar{a}_{i+1}, \cdots, \bar{a}_{j-1}, \bar{a}_j \rangle$, where $\bar{a}_i$ and $\bar{a}_j$ are direction-changer nodes and $\bar{a}_{i+1}, \bar{a}_{i+2}, \cdots, \bar{a}_{j-1}$ are not. Since a_i, $a_j \in I$ and a_{i+1}, $a_{j-1} \in (S - I)$, we have that $(\bar{a}_{i+1}, \bar{a}_i)$ and $(\bar{a}_{j-1}, \bar{a}_j)$ are edges of $\bar{G}(Q)$ and $(\bar{a}_i, \bar{a}_{i+1})$ and $(\bar{a}_j, \bar{a}_{j-1})$ are not edges of $\bar{G}(Q)$ (see Fig. 4).

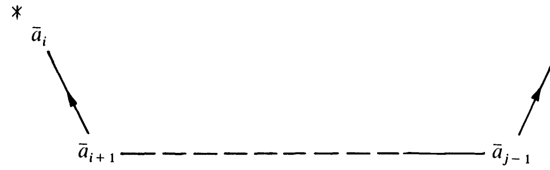


FIG. 4

In the subpath h', since there are no direction-changer nodes between $\bar{a}_i$ and $\bar{a}_j$, we should have either (i) edges $(\bar{a}_i, \bar{a}_{i+1})$ and $(\bar{a}_{j-1}, \bar{a}_j)$ or (ii) edges $(\bar{a}_{i+1}, \bar{a}_i)$ and $(\bar{a}_j, \bar{a}_{j-1})$. In either case, we have a situation that contradicts the earlier statement that $(\bar{a}_i, \bar{a}_{i+1})$ and $(\bar{a}_j, \bar{a}_{j-1})$ are not edges of $\bar{G}(Q)$. □

LEMMA 5. *Let $h = \langle \bar{a}_1, \cdots, \bar{a}_i, \bar{a}_{i+1}, \cdots, \bar{a}_{j-1}, \bar{a}_j, \cdots, \bar{a}_n \rangle$ be a Hamiltonian path* of $\bar{G}(Q)$. *If $\{a_i, a_j\} \subseteq q_p \in Q$, then $\{a_i, a_{i+1}, a_{i+2}, \cdots a_{j-1}, a_j\} \subseteq q_p$.*

Proof. Let $I = \bigcap_{q_i \in Q} q_i$. We have three situations 1–3 below.

1. Both $\bar{a}_i$ and $\bar{a}_j$ are direction-changer nodes. By Lemma 4, all the nodes between $\bar{a}_i$ and $\bar{a}_j$ are also direction-changer nodes. Then the elements corresponding to $\bar{a}_{i+1}, \bar{a}_{i+2}, \cdots, \bar{a}_{j-1}$ belong to I. Hence the lemma.

2. Both $\bar{a}_i$ and $\bar{a}_j$ are not direction-changer nodes. For this situation, we have the following possible cases.

Case (i). Let h visit all the direction-changer nodes after leading $\bar{a}_j$. Then $(\bar{a}_i, \bar{a}_{i+1})$, $(\bar{a}_{i+1}, \bar{a}_{i+2})$, $\cdots$, $(\bar{a}_{j-1}, \bar{a}_j)$ are edges of $\bar{G}(Q)$. Since $(\bar{a}_l, \bar{a}_m) \Rightarrow$ if $a_l \in q_k$ then $a_m \in q_k$, for all q_k in Q, we have $a_i, a_{i+1}, \cdots, a_{j-1}, a_j \in q_p$.

Case (ii). Let h visit all the direction-changer nodes before reaching $\bar{a}_i$. An argument similar to that in Case (i) leads us to the conclusion that $a_j, a_{j-1}, \cdots, a_{i-1}, a_i \in q_p$ when $a_i, a_j \in q_p$.

Case (iii). The direction-changer nodes are between $\bar{a}_i$ and $\bar{a}_j$ in h. Let $\bar{a}_l$ and $\bar{a}_{l+k}$ be the starting and end nodes of the subpath of h that consists only of the direction-changer nodes (by Lemma 4). Then the path h is

$$\langle \bar{a}_1, \cdots, \bar{a}_i, \cdots, \underbrace{\bar{a}_l, \cdots, \bar{a}_{l+k},}_{\text{direction changers}} \cdots, \bar{a}_j, \cdots, \bar{a}_n \rangle.$$

By Case (i), all the elements that correspond to nodes between $\bar{a}_i$ and $\bar{a}_{l-1}$ in h belong to q_p, and by Case (ii), all the elements corresponding to nodes between $\bar{a}_{l+k+1}$ and $\bar{a}_j$ in h belong to q_p. The direction-changer nodes correspond to the elements of I which are subsets of all sets in Q. Hence we have the lemma.

3. Either $\bar{a}_i$ or $\bar{a}_j$ is a direction-changer node. Without loss in generality, suppose $\bar{a}_i$ is. Let $\bar{a}_{i+k}$ be the end node of the subpath of h that consists only of the direction-changer nodes. By Case (ii) above, $\{a_j, a_{j-1}, \cdots, a_{i+k+1}\} \subseteq q_p$. We know that $\{a_i, a_{i+1}, \cdots, a_{i+k}\} \subseteq I \subseteq q_p$. Hence $\{a_i, a_{i+1}, \cdots, a_{j-1}, a_j\} \subseteq q_p$. □

The following lemma is the counterpart of Lemma 4 and is easily proved by contradiction [6].

LEMMA 6. *Let $\Omega(Q)$ be complete and suppose Q is faithfully representable. Let $I = \bigcap_{q_i \in Q} q_i$ and let $(f; I_1, I_2, \cdots, I_m)$ faithfully represent Q. Then there does not exist a_b, a_c, a_d such that $a_b, a_d \in I$ and $a_c \in (S - I)$ and $f(a_c)$ is between $f(a_b)$ and $f(a_d)$.*

The theorem that follows gives the necessary and sufficient conditions for a family Q whose intersection graph is complete to be faithfully representable.

THEOREM 1. *Let $\Omega(Q)$ be complete. Q is faithfully representable iff there exists a Hamiltonian path in $\bar{G}(Q)$.*

Proof. The sufficiency part of the theorem is easy to prove as follows. Let $h = \langle \bar{a}_1, \bar{a}_2, \cdots, \bar{a}_n \rangle$ be a Hamiltonian path in $\bar{G}(Q)$. Consider h as an n-tuple. Define a function f_h corresponding to h so that f_h maps elements of $S = \{a_1, a_2, \cdots, a_n\}$ onto integers $1, 2, \cdots, n$ as follows: $f_h(a_i) = j$ iff the jth member of h is $\bar{a}_i$. For each $q_i \in Q$, define an interval

$$I_i = [\min_{a_p \in q_i} \{f_h(a_p)\}, \max_{a_p \in q_i} \{f_h(a_p)\}].$$

It can be oberved that I_i contains the images of all elements in q_i. To see that I_i does not contain images of any element not in q_i, suppose, to the contrary, that it does. Then there exist $a_b, a_{c_1}, a_{c_2}, \cdots, a_{c_k}, \cdots, a_{c_j}, a_d$ belonging to S with $a_{c_k} \notin q_i$ and $f_h(a_{c_k})$ is between $f_h(a_b)$ and $f_h(a_d)$ and $\{a_b, a_d\} \subseteq q_i$. By the definition of f_h, this implies that the node $\bar{a}_{c_k}$ is between $\bar{a}_b$ and $\bar{a}_d$ in h, which contradicts Lemma 5. Hence $(f; I_i, I_2, \cdots, I_m)$ faithfully represents Q.

Now we proceed to the necessity part of theorem 1. By Lemma 3, $I = \bigcap_{q_i \in 0} q_i \neq \varnothing$. Let $I = \{a_p, a_{p+1}, \cdots, a_{p+l}\}$. Let $(f_h; I_1, I_2, \cdots, I_m)$ faithfully represent Q. We can define a total ordering on the elements of S such that a_i precedes a_j in the total ordering iff $f_h(a_i) < f_h(a_j)$. Without loss of generality, we can assume that f_h is such that

$$f_h(a_1) < f_h(a_2) < \cdots < \underbrace{f_h(a_p) < f_h(a_{p+1}) < \cdots < f_h(a_{p+l})}_{\text{images of elements} \in I} < \cdots < f_h(a_n)$$

(by Lemma 6).

All the intervals contain the images of elements belonging to I. Hence, whenever an interval I_i contains $f_h(a_1)$, it has to contain $f_h(a_2), f_h(a_3), \cdots, f_h(a_{p+l})$. Then, $a_1 \in q_i$ implies $\{a_2, a_3, \cdots a_p, \cdots a_{p+l}\} \subseteq q_i$. For if any of the elements of $\{a_2, a_3, \cdots, a_{p-1}\}$ is foreign to q_i, we will have a contradiction, i.e., that $(f_h; I_1, I_2, \cdots, I_m)$ does not faithfully represent Q.

Thus we have that $(\bar{a}_1, \bar{a}_2)$ is an edge of $\bar{G}(Q)$. By considering intervals that contain $f_h(a_2), f_h(a_3), \cdots, f_h(a_{p-1})$ and repeating the same argument as above, we see that $(\bar{a}_2, \bar{a}_3), (\bar{a}_3, \bar{a}_4), \cdots, (\bar{a}_{p-1}, \bar{a}_p)$ are among the edges of $\bar{G}(Q)$.

A similar argument as above shows that $(\bar{a}_n, \bar{a}_{n-1}), (\bar{a}_{n-1}, \bar{a}_{n-2}), \cdots, (\bar{a}_{p+l+1}, \bar{a}_{p+l})$ are also edges of $\bar{G}(Q)$. Since the relation $\bar{R}$ is symmetric for I, every pair of nodes belonging to $I' = \{\bar{a}_p, \bar{a}_{p+1}, \cdots, \bar{a}_{p+l}\}$ is connected and directed both ways. But the nodes $\in I'$ are precisely the direction-changer nodes of $\bar{G}(Q)$. Hence

$$\langle \bar{a}_1, \bar{a}_2, \cdots, \bar{a}_{p-1}, \bar{a}_p, \cdots, \bar{a}_{p+l}, \bar{a}_{p+l+1}, \cdots, \bar{a}_n \rangle$$

is a path of $\bar{G}(Q)$ which is Hamiltonian. □

Example 5. Consider the family Q in Example 4. The DSG of Q is given in Fig. 3. $h = \langle \bar{a}_1, \bar{a}_2, \bar{a}_3, \bar{a}_4, \bar{a}_5 \rangle$ is a Hamiltonian path in $\bar{G}(Q)$. Hence Q is faithfully representable.

Define $f_h : f_h(a_1) = 1$ as $\bar{a}_1$ is the first member of h. Similarly, $f_h(a_2) = 2$, $f_h(a_3) = 3, f_h(a_4) = 4$ and $f_5(a_5) = 5$. Let

$$I_1 = [\min_{a_i \in q_1}(f_h(a_i)), \max_{a_i \in q_1}(f_h(a_i))] = [f_h(a_1), f_h(a_3)] = [1, 3].$$

Similarly, $I_2 = [2, 5]$ and $I_3 = [2, 4]$. Thus $(f_h; I_1, I_2, I_3)$ faithfully represents Q. See Fig. 5.

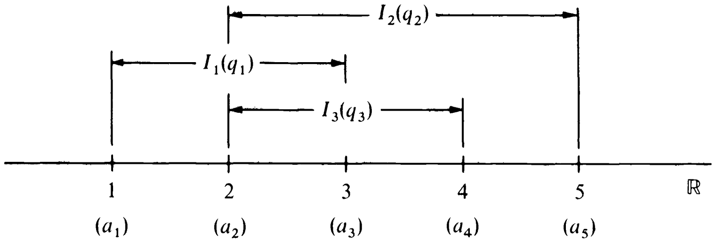


FIG. 5. *A faithful representation of Q*

3. Union of two linearly orderable families whose intersection graphs are complete. In the remainder of the paper, let Q_1 and Q_2 denote two families of sets. $Q_1 \cap Q_2$ need not be empty. $\bar{G}(Q_1)$ and $\bar{G}(Q_2)$ represent the DSG of Q_1 and Q_2, respectively. S_1 denotes the set $\bigcup_{q_i \in Q_1} \{q_i\}$ and S_2 the set $\bigcup_{q_i \in Q_2} \{q_i\}$. S indicates $(S_1 \cup S_2)$. We shall first prove some results regarding the elements of $S_1 \cap S_2$ in an ordering that implies a faithful representation of $Q_1 \cup Q_2$. We shall then introduce some definitions regarding the orderings of elements in S_1 and S_2 in graph theoretic terms. Finally, we shall prove a theorem to check if $Q_1 \cup Q_2$ is faithfully representable when Q_1, Q_2 are faithfully representable and $\Omega(Q_1), \Omega(Q_2)$ are complete.

LEMMA 7. *Let $Q_1 \cup Q_2$ be faithfully representable and $\Omega(Q_1), \Omega(Q_2)$ be complete. Let $I = S_1 \cap S_2$ and f be a function that defines a linear ordering of the elements of S such that $(Q_1 \cup Q_2)$ is faithfully represented. Then there does not exist a_p, a_i, a_j such that $a_p \in (S - I)$ and $\{a_i, a_j\} \subseteq I$ and $f(a_p)$ is between $f(a_i)$ and $f(a_j)$.*

Proof. The proof is by contradiction. Suppose that there exist such a_p, a_i and a_j. Without loss of generality, we shall assume that $f(a_i) < f(a_p) < f(a_j)$.

By Lemma 1, Q_1 and Q_2 are linearly orderable. Then by Lemma 3,

$$I_1 = \bigcap_{q_i \in Q_1} q_i \neq \emptyset \quad \text{and} \quad I_2 = \bigcap_{q_i \in Q_2} q_i \neq \emptyset \cdot$$

As $a_i, a_j \in S_1$, there exist sets $q_c, q_d \in Q_1$ such that $a_i \in q_c$ and $a_j \in q_d$. $I_1 \subseteq q_c$ and $I_1 \subseteq q_d$.

Case (i). $a_i \in I_1$. Then the interval I_d^I corresponding to q_d contains $f(a_i), f(a_j)$ and hence $f(a_p)$. Since the linear ordering defined by f faithfully represents $Q_1 \cup Q_2$, a_p is not foreign to q_d.

Case (ii). $a_j \in I_1$. By the same arguments as in Case (i), we have $a_p \in q_c$.

Case (iii). If $a_i, a_j \notin I_1$, then there exists an element $a_l \in I_1$ such that either $f(a_l) < f(a_i)$ or $f(a_i) < f(a_l)$. In either case, $a_p \in q_c$ or $a_p \in q_d$ since f faithfully represents $(Q_1 \cup Q_2)$.

Thus there exists a $q_i \in Q_1$ such that $a_p \in q_i$. Hence, $a_p \in S_1$. By similar arguments, $a_p \in S_2$. This means that $a_p \in I$, a contradiction. □

LEMMA 8. *Let $Q_1 \cup Q_2$ be faithfully representable and $\Omega(Q_1)$, $\Omega(Q_2)$ be complete. Let $I = S_1 \cap S_2 \neq \emptyset$, S_1 or S_2. Let f define a linear ordering of the elements of S faithfully representing $(Q_1 \cup Q_2)$. Let a_c and a_k be such that $f(a_c) = \min_{a_i \in I}\{f(a_i)\}$ and $f(a_k) = \max_{a_i \in I}\{f(a_i)\}$. Then*

(i) *for all $a_i \in (S_1 - I)$, either $f(a_i) < f(a_c)$ or $f(a_i) > f(a_k)$, i.e., there does not exist $a_p, a_q \in (S_1 - I)$ such that $f(a_p) < f(a_c)$ and $f(a_q) > f(a_k)$;*

(ii) *for all $a_i \in (S_1 - I), f(a_i) < f(a_c) \Leftrightarrow$ for all $a_j \in (S_2 - I), f(a_j) > f(a_k)$.*

Proof. (i) This is proved by contradiction (see [4]) by observing the fact that Q_1 and Q_2 are faithfully representable and $I_1 = \bigcap_{q_i \in Q_1} q_i \neq \emptyset$ and $I_2 = \bigcap_{q_i \in Q_2} q_i \neq \emptyset$.

(ii) $\Rightarrow$. We have $f(a_i) < f(a_c)$ for all $a_i \in (S_1 - I)$. Assume, to the contrary, that there exists an $a_j \in (S_2 - I)$ such that $f(a_j) < f(a_k)$. By Lemma 7, $f(a_j) < f(a_c)$. Since a_j is foreign to all sets containing elements of $S_1 - I$, we have

$$f(a_j) < \min_{a_r \in S_1} f(a_r).$$

There exists a set $\bar{q}_i \in Q_2$ such that $\bar{q}_i \supseteq \{a_j\} \cup I_2$. As $\bar{q}_i \cap (S_1 - I) = \emptyset$, $f(a_l) < \min_{a_r \in S_1} f(a_r)$ for all $a_l \in I_2$. Now consider the set $\bar{q}_p \in Q_2$ such that $\bar{q}_p = \{a_k\} \cup I_2$. The interval corresponding to $\bar{q}_p$ contains the images of elements of $S_1 - I$ which are foreign to all sets in Q_2. This leads to a contradiction.

$\Leftarrow$. The same arguments as above direct us to the conclusion that if for all $a_j \in (S_2 - I)$, $f(a_j) > f(a_k)$, then for all $a_j \in (S_1 - I)$, $f(a_j) < f(a_c)$. □

Let $P = \langle \bar{a}_1, \bar{a}_2, \cdots, \bar{a}_i, \bar{a}_{i+1}, \cdots, \bar{a}_k \rangle$ be a path in $\bar{G}(Q)$. We say that each $\bar{a}_i$ in P, for $1 < i < k$, has both *left* and *right neighbors.* The left and right neighbor of $\bar{a}_i$ are $\bar{a}_{i-1}$ and $\bar{a}_{i+1}$, respectively. $\bar{a}_1$ has only a right neighbor, namely, $\bar{a}_2$ and $\bar{a}_k$ has only a left neighbor, which is $\bar{a}_{k-1}$. The left neighbor of $\bar{a}_1$ is said to be *empty* and so is the right neighbor of $\bar{a}_k$.

Two paths P_1 and P_2 are *equal* (or *nondistinct*) iff the starting and end nodes of P_1 are the starting and end nodes of P_2, and for all $\bar{a}_i \in P_1$ such that $\bar{a}_i$ is not the starting node of P_1, the left neighbor of $\bar{a}_i$ in P_1 = the left neighbor of $\bar{a}_i$ in P_2 and for all $\bar{a}_i \in P_1$ such that $\bar{a}_i$ is not the end node of P_1, the right neighbor of $\bar{a}_i$ in P_1 = the right neighbor of $\bar{a}_i$ in P_2.

Let h_1 and h_2 be Hamiltonian paths in $\bar{G}(Q_1)$ and $\bar{G}(Q_2)$, respectively. Let $I = S_1 \cap S_2$. We see that h_1 induces a subpath in the set of nodes that correspond to I in $\bar{G}(Q_1)$. The starting and end nodes of this subpath are nodes that correspond to some elements in I, and the subpath contains all the nodes which correspond to the elements of I. Let h_1^I denote this subpath. Similarly h_2^I is the subpath induced by h_2 in the set of nodes that correspond to I in $\bar{G}(Q_2)$. We say that the Hamiltonian paths h_1 and h_2 are *consistent* (written $h_1 \sim h_2$) iff exactly one of the following holds:

(i) $S_1 \cap S_2 = I = \varnothing$, or (ii) $h_1^I = h_2^I$ and the left neighbor of the starting node of h_1^I is empty in h_1 or h_2 and the right neighbor of the end node of h_1^I is empty in h_1 or h_2.

Example 6. In Example 5, we saw that $h_1 = \langle \bar{a}_1, \bar{a}_2, \bar{a}_3, \bar{a}_4, \bar{a}_5 \rangle$ is a Hamiltonian path in $\bar{G}(Q_1)$ where Q_1 is given in Example 4.

Let $Q_2 = \{q_2, q_3, q_4, q_5\}$ where q_2, q_3, q_4 and q_5 are given in Example 2. One can draw $\bar{G}(Q_2)$ and observe that $h_2 = \langle \bar{a}_2, \bar{a}_3, \bar{a}_4, \bar{a}_5, \bar{a}_6, \bar{b}_1 \rangle$ is a Hamiltonian path in $\bar{G}(Q_2)$. Let

$$I = S_1 \cap S_2 = \{a_1, a_2, a_3, a_4, a_5\} \cap \{a_2, a_3, a_4, a_5, a_6, b_1\}$$
$$= \{a_2, a_3, a_4, a_5\}.$$

Then $h_1^I = \langle \bar{a}_2, \bar{a}_3, \bar{a}_4, \bar{a}_5 \rangle = h_2^I$. The starting node of $h_1^I = \bar{a}_2$, and the end node of $h_1^I = \bar{a}_5$. The left neighbor of $\bar{a}_2$ is empty in h_2, and the right neighbor of $\bar{a}_5$ is empty in h_1. Hence $h_1 \sim h_2$. □

THEOREM 2. *Let $\Omega(Q_1)$ and $\Omega(Q_2)$ be complete. If $Q_1 \cup Q_2$ is faithfully representable, then there exist Hamiltonian paths h_1 in $\bar{G}(Q_1)$ and h_2 in $\bar{G}(Q_2)$ such that h_1 and h_2 are consistent.*

Proof. By Lemma 1 and Theorem 1, there exist Hamiltonian paths in $\bar{G}(Q_1)$ and in $\bar{G}(Q_2)$.

Case (i). $I = S_1 \cap S_2 = \varnothing$. The theorem is true.

Case (ii). $I \neq \varnothing$. Let f be a function that defines a linear ordering of the elements $\in S$ faithfully representing $Q_1 \cup Q_2$. We have the following situations 1–3.

1. $I = S_2$, i.e., $S_2 \subseteq S_1$ and $S = S_1$. Define $f_1(a_i) = f(a_i)$ for all $a_i \in S_1$ and $f_2(a_i) = f(a_i)$ for all $a_i \in S_2$. Clearly, f_1 (f_2) defines a linear ordering, say $\mathcal{O}_1$ ($\mathcal{O}_2$), of the elements S_1 (S_2) faithfully representing Q_1 (Q_2). Since $f_2(a_i) = f_1(a_i)$ for all $a_i \in S_2$, we have that for all $a_l, a_k \in S_2$, a_l precedes a_k in $\mathcal{O}_2$ iff a_l precedes a_k in $\mathcal{O}_1$. By Lemma 7, there does not exist an $a_p \in (S_1 - S_2)$ and $a_c, a_d \in S_2$ such that $f_1(a_p)$ is in between $f_1(a_c)$ and $f_1(a_d)$. Hence, if h_1 and h_2 are the Hamiltonian paths in $\bar{G}(Q_1)$ and $\bar{G}(Q_2)$ corresponding to $\mathcal{O}_1$ and $\mathcal{O}_2$, respectively (see proof of Theorem 1), then $h_2 = h_2^I = h_1^I$. The left (right) neighbor of the starting (end) node of h_2 is empty. Thus $h_1 \sim h_2$.

2. $I = S_1$, i.e., $S_1 \subseteq S_2$. The proof is similar to that of part 1 above.

3. $I \neq \varnothing$ or S_1 or S_2. Let $I = \{a_1, a_2, \cdots, a_k\}$. Without loss of generality, we can let f be such that $f(a_1) < \cdots < f(a_k)$. By Lemma 8, we can assume without any loss in generality that for all $a_i \in (S_1 - I)$, $f(a_i) < f(a_1)$ and for all $a_i \in (S_2 - I)$, $f(a_i) > f(a_k)$.

Now, define $f_1, f_2 : f_1(a_i) = f(a_i)$ for all $a_i \in S_1$ and $f_2(a_i) = f(a_i)$ for all $a_i \in S_2$. Clearly, f_1 and f_2 define linear orderings, say $\mathcal{O}_1$ and $\mathcal{O}_2$, of the elements belonging to S_1 and S_2, respectively, such that Q_1 and Q_2 are faithfully represented. Let h_1 and h_2 be the Hamiltonian paths corresponding to $\mathcal{O}_1$ and $\mathcal{O}_2$ in $\bar{G}(Q_1)$ and $\bar{G}(Q_2)$ (see the proof of Theorem 1). Then $h_1^I = \langle \bar{a}_1, \bar{a}_2, \cdots, \bar{a}_k \rangle = h_2^I$. $\bar{a}_1$ is the starting node of h_1^I and its left neighbor is empty in h_2. The right neighbor of $\bar{a}_k$, the end node of h_1^I, is empty in h_1. We thus have $h_1 \sim h_2$. □

4. Union of linearly orderable families whose intersection graphs are complete. In this section, we consider the union of linearly orderable families having pairwise nondisjoint sets. Necessary and sufficient conditions for the union to be linearly orderable is given in terms of a new graph called the partial order graph. An algorithm for faithful representation is also given.

Let $Q = \{Q_1, Q_2, \cdots, Q_m\}$ be a set of families of sets with $Q_i \cap Q_j$ not necessarily empty. For $1 \leqq i \leqq m$, let $\Omega(Q_i)$ be complete and let S_i denote the set $\bigcup_{q_j \in Q_i} \{q_j\}$. $\tilde{S}$ indicates $\bigcup_{i=1,\cdots,m} S_i = \{a_1, a_2, \cdots, a_n\}$. Let $h_1, h_2, \cdots, h_m$ be pairwise consistent Hamiltonian paths in $\bar{G}(Q_1), \bar{G}(Q_2), \cdots, \bar{G}(Q_m)$, respectively, where, for $1 \leqq i \leqq m$, $\bar{G}(Q_i)$ is the DSG of Q_i. Define a directed graph $\tilde{G}(Q) = [\tilde{S}', \tilde{R}]$. $\tilde{S}'$ is the set of nodes of $\tilde{G}(Q)$ and is $\{\bar{a}_1, \bar{a}_2, \cdots, \bar{a}_i, \cdots, \bar{a}_n\}$, where node $\bar{a}_i$ corresponds to element $a_i \in \tilde{S}$ and vice versa. $\bar{a}_i \tilde{R} \bar{a}_j$ iff there exists a Hamiltonian path h_k, $1 \leqq k \leqq m$, in which $\bar{a}_i$ precedes $\bar{a}_j \cdot \bar{a}_i \tilde{R} \bar{a}_j$ iff $(\bar{a}_i, \bar{a}_j)$ is an edge of $\tilde{G}(Q)$. $\tilde{G}(Q)$ is called a *partial order* (P.O.) graph of Q corresponding to $Q_1, Q_2, \cdots, Q_m$.

An *undirected path* or simply a *path* in a directed graph G is a sequence of distinct nodes $\bar{a}_1, \bar{a}_2, \cdots, \bar{a}_k$ such that for $i = 1, 2, \cdots, (k-1)$, $\{\bar{a}_i, \bar{a}_{i+1}\}$ are edges of G. Note that we ignore the direction of the edges in G. A connected-directed graph is a directed graph in which there is a path between every pair of distinct nodes. A *component* G' of a directed graph G is a subgraph of G such that G' is a connected-directed graph and is not properly contained in any other connected-directed subgraph of G. A *directed path* in a directed graph G is a sequence of distinct nodes $\bar{a}_1, \bar{a}_2, \cdots, \bar{a}_k$ such that for $1 \leqq i \leqq k-1$, $(\bar{a}_i, \bar{a}_{i+1})$ are edges of G. A *Hamiltonian path* in a directed graph G is a directed path that passes through all the nodes of G. If $P = \langle \bar{a}_1, \bar{a}_2, \cdots, \bar{a}_k \rangle$ is a path in G and $(\bar{a}_k, \bar{a}_1)$ is also an edge of G, then P is also a *directed cycle*. We can distinguish by context whether we mean by P a directed path or a directed cycle. If G does not have any directed cycles, then G is called *acyclic*. The *length* of a cycle is the number of nodes in the cycle.

THEOREM 3. *Let $G_1, G_2, \cdots, G_m$ be a set of complete subgraphs of $\Omega(Q)$ that cover $\Omega(Q)$. Let $Q_i \subseteq Q$ be such that $G_i = \Omega(Q_i)$ for $1 \leqq i \leqq m$. Q is faithfully representable iff there exists a P.O. graph $\tilde{G}(Q)$ corresponding to $Q_1, Q_2, \cdots, Q_m$ and any $\tilde{G}(Q)$ acyclic.*

Proof. The "only if" part of the theorem: since every subfamily of Q is faithfully representable (Lemma 1), there exists a P.O. graph $\tilde{G}(Q)$ of Q.

Suppose, to the contrary, that there exists a $\tilde{G}(Q)$ containing directed cycles. Let $C = \langle \bar{a}_1, \cdots, \bar{a}_i, \bar{a}_{i+1}, \cdots, \bar{a}_k \rangle$ be a cycle of minimum length in $\tilde{G}(Q)$. $(\bar{a}_i, \bar{a}_j)$ or $(\bar{a}_j, \bar{a}_i)$ is an edge of $\tilde{G}(Q)$ iff there exists a S_p, $1 \leqq p \leqq m$, such that $S_p \supseteq \{a_i, a_j\}$. Since $h_1, h_2, \cdots, h_m$ are pairwise consistent, the length of C must be at least 3. Furthermore, since C is of minimum length, no set S_p contains more than two elements corresponding to nodes in C. Hence, without loss in generality, we can assume that $a_i, a_{i+1} \in S_i$ for $1 \leqq i \leqq (k-1)$ and $a_k, a_1 \in S_k$.

Since Q is faithfully representable, there exists a function f defining a linear ordering of the elements of S faithfully representing Q. Considering Q_i, we observe that $f(a_l)$ is not between $f(a_i)$ and $f(a_{i+1})$ for $l = 1, 2, \cdots, i-1, i+2, \cdots, k$ since $\{a_i, a_{i+1}\} \subseteq S_i$ and a_l is foreign to all sets in Q_i. Considering $Q_1, Q_2, \cdots, Q_{k-1}$ and applying the above argument, we get a contradiction that $f(a_l)$ is between and is not between $f(a_1)$ and $f(a_k)$ for $l = 2, 3, \cdots, k-1$. Thus, Q is faithfully representable implies $\tilde{G}(Q)$ exists and any $\tilde{G}(Q)$ is acyclic.

The "if" part of the theorem: to prove the sufficiency of the conditions, we shall show how to construct, for any family Q satisfying the conditions, a function f and a set of intervals faithfully representing Q.

Let $\tilde{G}_1 \tilde{G}_2, \cdots, \tilde{G}_p$ be the components of $\tilde{G}(Q)$. Define function f as follows: (i) for all $\bar{a}_i, \bar{a}_j \in \tilde{G}_k$, $1 \leqq k \leqq p$, $f(a_i) < f(a_j)$ iff there exists a directed path from $\bar{a}_i$ to $\bar{a}_j$, (ii) for all $a_i \in \tilde{G}_k$ and all $a_j \in \tilde{G}_l$ and $k < l$, $f(a_i) < f(a_j)$. Since $\tilde{G}(Q)$ is acyclic, such a function exists. For each $q_i \in Q$, define $I_i = [\min_{a_p \in q_i} f(a_p), \max_{a_p \in q_i} f(a_p)]$. Interval I_i contains the images of all elements of q_i.

To see that I_i does not contain images of any foreign elements with respect to q_i, suppose, to the contrary, that it does. Then there exist $a_b, a_d \in q_i$; $a_{c_2}, \cdots, a_{c_j} \in (\tilde{S} - q_i)$ such that for $1 \leqq k \leqq j$, $f(a_{c_k})$ is between $f(a_b)$ and $f(a_d)$. Without loss in generality, we can assume that $f(a_b) < f(a_{c_1}) < \cdots < f(a_{c_j}) < f(a_d)$. Then $(\bar{a}_b, \bar{a}_{c_1})$ is an edge of $\tilde{G}(Q)$ and $\bar{a}_{c_1}$ is the right neighbor of $\bar{a}_b$ in some Hamiltonian path h_t used to define $\tilde{G}(Q)$.

Our q_i belongs to at least one complete subgraph, say G_l, that was chosen to cover $\Omega(Q)$. Let Q_l be such that $\Omega(Q_l) = G_l$, and let h_l be the Hamiltonian path in $\bar{G}(Q_l)$ that was used in the definition of $\tilde{G}(Q)$. If $\bar{a}_d$ precedes $\bar{a}_b$ in h_l, then $(\bar{a}_d, \bar{a}_b)$ will be an edge of $\tilde{G}(Q)$ and hence $\bar{a}_b, \bar{a}_{c_1}, \cdots, \bar{a}_{c_k}, \cdots, \bar{a}_{c_j}, \bar{a}_d$ will be a directed cycle of $\tilde{G}(Q)$, which is not possible. Hence let $\bar{a}_b$ precede $\bar{a}_d$ in h_l. Since $\bar{a}_{c_1}$ is foreign to q_i, by Lemma 6, $\bar{a}_{c_1}$ is not between $\bar{a}_b$ and $\bar{a}_d$ in h_l. Thus $h_l \neq h_t$.

The right neighbor of $\bar{a}_b$ is not empty in both h_l and h_t. The right neighbor of $\bar{a}_b$ in $h_t = \bar{a}_{c_1}$, which is not the right neighbor of $\bar{a}_b$ in h_l. This leads us to the contradiction that h_l and h_t are not consistent. □

Example 7. Consider the family of sets Q in Example 2. We found that G_1, G_2, G_3 and G_4 are complete subgraphs that cover $\Omega(Q)$. Thus $Q_1 = \{q_1, q_2, q_3\}$, $Q_2 = \{q_2, q_3, q_4, q_5\}$, $Q_3 = \{q_5, q_6\}$ and $Q_4 = \{q_7, q_8\}$. $\bar{G}(Q_1)$ is given in Fig. 3 and discussed in Examples 4 and 5. $h_1 = \langle \bar{a}_1, \bar{a}_3, \bar{a}_3, \bar{a}_4, \bar{a}_5 \rangle$ was found to be a Hamiltonian path in $\bar{G}(Q_1)$. Similarly,

$$h_2 = \langle \bar{a}_2, \bar{a}_3, \bar{a}_4, \bar{a}_5, \bar{a}_6, \bar{b}_1 \rangle,$$

$$h_3 = \langle \bar{a}_4, \bar{a}_5, \bar{a}_6, \bar{b}_1, \bar{b}_2 \rangle,$$

and

$$h_4 = \langle \bar{a}_8, \bar{a}_7, \bar{a}_9 \rangle$$

are Hamiltonian paths in $\bar{G}(Q_2)$, $\bar{G}(Q_3)$ and $\bar{G}(Q_4)$, respectively. h_1, h_2, h_3, h_4 are pairwise consistent. We can thus define a $\tilde{G}(Q)$. $\tilde{G}(Q)$ is given in Fig. 6. For sake of

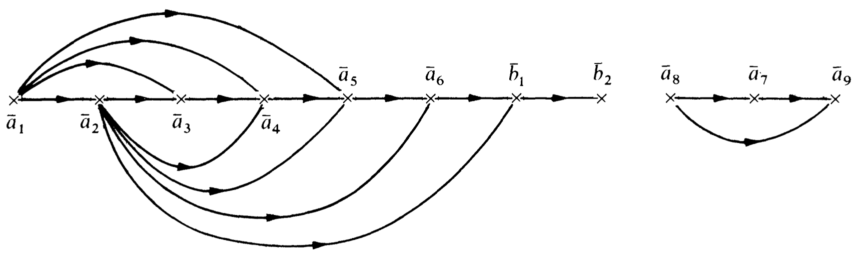


FIG. 6. *$\tilde G(Q)$ of the family of sets Q in Example 2. Note that not all the edges of $\tilde G(0)$ are shown*

clarity, we have not shown all the edges of the graph. We find $\tilde G(Q)$ to be acyclic and thus conclude that Q is faithfully representable. □

THEOREM 4. *If G_1 and G_2 are complete subgraphs of $\Omega(Q)$ such that G_1 and G_2 cover $\Omega(Q)$, then Q is faithfully representable iff there exist consistent Hamiltonian paths in $\bar G(Q_1)$ and $\bar G(Q_2)$, where $Q_1 \subseteq Q$, $Q_2 \subseteq Q$ and $\Omega(Q_1) = G_1$, $\Omega(Q_2) = G_2$.*

Proof. The "only if" part of the theorem is Theorem 2 (see § 3). If there exist consistent Hamiltonian paths in $\bar G(Q_1)$ and $\bar G(Q_2)$, then there exists a $\tilde G(Q)$. By arguments similar to the ones in the proof of Theorem 3, one can show that $\tilde G(Q)$ is always acyclic. Then, by Theorem 3, Q is faithfully representable. □

Based on the fact that every pair of Hamiltonian paths used in the definition of a P.O. graph $\tilde G(Q)$ of Q is consistent, we have the following lemma.

LEMMA 9. *There exists a directed path between every pair of nodes in each component of $\tilde G(Q)$ (i.e., if $\bar a_i$ and $\bar a_j$ are two nodes belonging to the same component in $\tilde G(Q)$ then there is a path from $\bar a_i$ to $\bar a_j$ or from $\bar a_j$ to $\bar a_i$).* □

LEMMA 10. *Let G be a directed graph such that there exists a directed path between every pair of nodes of G. If G is acyclic, then there exists one and only one Hamiltonian path in G.*

Proof. The proof is by induction on the number of nodes n of G. Assume that the lemma is true for $n = k - 1$. Let G be a graph satisfying the hypothesis of the lemma and $n = k$. Then there exists one and only one source, s, in G (a *source* is a node with at least one outgoing edge and zero incoming edges). Delete from G the node s and all the edges incident on s, and obtain the graph $\bar G$. In G, any path between nodes i and j $(i, j \neq s)$ does not pass through s. Hence $\bar G$ satisfies the hypothesis of the lemma. The number of nodes of $\bar G$ is $k - 1$. By induction, there exists one and only one Hamiltonian path in $\bar G$. Let it be $s_1, \cdots, s_m$. (s, s_1) is the one and only edge between s and s_1 in G. Then $s, s_1, \cdots, s_m$ is a Hamiltonian path in G and is the only one. □

From Lemmas 9 and 10, the following theorem is immediate.

THEOREM 5. *There exists one and only one Hamiltonian path in every component of $\tilde G(Q)$ if $\tilde G(Q)$ is acylcic.* □

We now present an algorithm to obtain a function f and a set of intervals $\{I_1, I_2, \cdots, I_m\}$ such that $(f; I_1, I_2, \cdots, I_m)$ faithfully represents Q.

(i) Obtain a P.O. graph $\tilde G(Q)$ of Q. If there does not exist a $\tilde G(Q)$ or if $\tilde G(Q)$ is not acyclic, Q is not faithfully representable.

(ii) Let $\tilde G_1, \tilde G_2, \cdots, \tilde G_p$ be the components of $\tilde G(Q)$. Get the Hamiltonian paths $H_1, H_2, \cdots, H_p$ in $\tilde G_1, \tilde G_2, \cdots, \tilde G_p$.

(iii) Define function f as follows: (a) for all $\bar{a}_i \in \tilde{G}_k, \bar{a}_j \in \tilde{G}_r, k < r, f(a_i) < f(a_j)$, and (b) for all $\bar{a}_i, \bar{a}_j \in \tilde{G}_k, i \neq j, 1 \leqq k \leqq p, f(a_i) < f(a_j)$ iff $\bar{a}_i$ precedes $\bar{a}_j$ in H_k.

(iv) for all $q_i \in Q$, define $I_i = [\min_{a_j \in q_i} f(a_j). \max_{a_j \in q_i} f(a_j)]$.

THEOREM 6. *The function and the intervals defined by the above algorithm imply that the family Q is faithfully representable.*

Proof. By theorem 5, $H_1, H_2, \cdots, H_p$ exist. The rest of the proof is the same as the proof of the "only if" part of Theorem 3. □

5. Conclusions. We have given necessary and sufficient conditions and an algorithm for faithful representation of a family of sets. The complexity of the algorithm is discussed in [6].

We mentioned earlier that the faithful representation corresponds to the consecutive retreval (CR) file organization. If any record in a file is stored more than once in a file organization, then that record is called *redundant*. When a file is inverted on queries, we get a *query inverted file organization* (see [5] and [6]). A CR file organization is then a query inverted file organization with zero redundancy. When a family of queries does not possess the CR property, we may be interested in finding a query inverted file organization with as small a redundancy as possible. This is discussed in [5] and [6].

If Q does not have the CR property, we may like to find a file organization with minimum overall retrieval time possible with the constraint that each record be stored only once. This is discussed in [6].

Acknowledgments. The author would like to express his sincere thanks to Professor Lofti A. Zadeh and Dr. Sakti P. Ghosh for their suggestions and help.

REFERENCES

[1] S. P. GHOSH, *File organization: The consecutive retrieval property*, Comm. ACM, 15 (1972), pp. 802–808.
[2] F. HARARY, R. NORMAN AND D. CARTWRIGHT, *Structural Models: An Introduction to Theory of Directed Graphs*, John Wiley, New York, 1965.
[3] C. BERGE, *Theory of Graphs and its Applications*, John Wiley, New York, 1962.
[4] K. ESWARAN, *A graph theoretic approach to linearly orderable sets*, Memo, M369, Electronics Res. Lab., Univ. of Calif., Berkeley.
[5] S. P. GHOSH, *Consecutive storage of relevant records with redundancy*, IBM Res. Rep. 933, 1971.
[6] K. ESWARAN, *Consecutive retrieval information systems*, Ph.D. thesis, Memo. M384, Electronics Res. Lab., Univ. of Calif., Berkeley, 1973.

SIAM J. COMPUT.
Vol. 4, No. 1, March 1975

A DECISION PROCEDURE FOR THE FIRST ORDER THEORY OF REAL ADDITION WITH ORDER*

JEANNE FERRANTE† AND CHARLES RACKOFF‡

Abstract. Consider the first order theory of the real numbers with the function $+$ (plus) and the predicate $<$ (less than). Let S be the set of true sentences of this theory. We first present an elimination of quantifiers decision procedure for S, and then analyze it to show that it takes at most time $2^{2^{cn}}$, where c is a constant, to decide sentences of length n.

We next show that a given sentence does not change in truth value when each of the quantifiers is limited to range over an appropriately chosen finite set of rationals. This fact leads to a new decision procedure for S which uses at most space 2^{cn}. We also remark that our methods lead to a decision procedure for Presburger arithmetic which operates within space $2^{2^{cn}}$. These upper bounds should be compared with the results of Fischer and Rabin [2] that for some constant c, real addition requires time 2^{cn} and Presburger arithmetic requires time $2^{2^{cn}}$.

Key words. real addition, decision procedures, quantifier-bounding, elimination of quantifiers, Presburger arithmetic.

1. Introduction. In this paper we present an efficient decision procedure for the first order theory of the real numbers with the function $+$ (plus) and the predicate $<$ (less than). Of course, the decidability of the theory in question is a consequence of Tarski's theorem that the real numbers under $+, \cdot$ (times), and $<$ is decidable [5]; however, Tarski's procedure is far from efficient for the restricted theory we are interested in. We propose to exhibit a procedure which is nearly optimal in its computational efficiency. Fischer and Rabin [2] show that there is a constant $c > 0$ such that any nondeterministic Turing machine which decides real addition (even without order) requires, for almost every n, time 2^{cn} to decide some sentences of length n. We will present a deterministic procedure for the theory of addition on the ordered set of real numbers which uses at most space 2^{dn} and time $2^{2^{gn}}$ (where d and g are constants) to decide sentences of length n. Thus there appears to be a gap of approximately one exponential between upper and lower time bounds. But since the upper bound is deterministic and the lower bound is nondeterministic, this gap should be viewed in the light of a long-standing, unproved conjecture of automata theory which states that nondeterministic time t is equal in power to deterministic time 2^t.

The procedure we give replaces unbounded quantifiers by quantifiers ranging over a finite set of rationals; truth of a sentence about the real numbers will thus be determined by checking finitely many instances of a matrix. In order to prove the correctness of our procedure, we first present an elimination of quantifiers procedure with the important feature that it does not require the sentence to be put in disjunctive normal form at each quantifier elimination.

In § 2 we define the language under consideration. In § 3 we give our elimination of quantifiers procedure. Our method utilizes an idea used by Cooper [1] in deciding

* Received by the editors May 15, 1973, and in revised form April 24, 1974. This research was supported by the National Science Foundation under Grant GJ-34671.

† Project MAC, Massachusetts Institute of Technology, Cambridge, Massachusetts. Now at Tufts University, Medford, Massachusetts 02155.

‡ Project MAC, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139.

integral addition. In § 4 we show—via an analysis of § 3—that each quantifier in a formula can be replaced by a suitably bounded quantifier, and then show that the desired space bound can be achieved. In § 5 we remark on further applications of our methods.

2. Notation.

2. Notation. We now define a language $\mathscr{L}$ of the first order predicate calculus:

$\mathscr{L}$ has *variables* $x_0, x_1, x_{10}, \cdots$ (i.e., the subscripts are written in binary);

$\mathscr{L}$ has a *constant symbol i* (written in binary) for every integer i;

$\mathscr{L}$ has *rational constant symbols* composed of integer constant symbols, that is, if a and b are nonzero integers, then (a/b) is a rational constant symbol of $\mathscr{L}$;

$\mathscr{L}$ has *terms* of the form $(a_1/b_1)y_1 + (a_2/b_2)y_2 + \cdots + (a_n/b_n)y_n$ (abbreviated $\sum_{i=1}^{n} (a_i/b_i)y_i$), where (a_i/b_i) is a rational constant for $1 \leqq i \leqq n$ and where $y_1, \cdots, y_n$ represent distinct variables of $\mathscr{L}$. The constant symbol 0 will also be considered a term of $\mathscr{L}$.

An *atomic formula* of $\mathscr{L}$ is either the string TRUE, the string FALSE, or a formula of the form $t_1 = t_2$ or of the form $t_1 < t_2$, where t_1 and t_2 are terms; the formulas and sentences of $\mathscr{L}$ are built up from the atomic formulas in the usual way using the symbols $\vee, \exists, \forall, \sim, (,)$.

Let R be the set of real numbers. We interpret the formulas of $\mathscr{L}$ as follows: if (a/b) is a rational constant symbol and x is interpreted as having the value $r \in R$, then we give $(a/b)x$ the value $(a/b) \cdot r$. We interpret $=$ as equality, $+$ as the usual operation of addition on R, and $<$ as the usual ordering on R. The atom TRUE is always taken to be true, and FALSE is always taken to be false.

Let S be the set of sentences of $\mathscr{L}$ true under this interpretation. We will exhibit a decision procedure for S (that is, an algorithmic procedure for deciding whether an arbitrary sentence of $\mathscr{L}$ is in S or not) such that if B is a sentence of length n, the algorithm determines whether or not $B \in S$ within space 2^{dn}, where d is a constant.

A remark should be made here as to why we have defined the terms of $\mathscr{L}$ as we have; if we had only allowed *integer* coefficients in our terms, then the resulting language would have been no less powerful, yet it would have been more difficult to arrive at a decision procedure. The reason is that our definition of a term reflects the fact that R is not only an ordered group (under addition), but it is also divisible and torsion-free. That R is divisible means that for every $r \in R$ and every positive integer k, there is an $s \in R$ such that $k \cdot s = r$, i.e.,

$$\underbrace{s + s + \cdots + s}_{k \text{ times}} = r.$$

That R is torsion-free means that for every $r \in R$ and every positive integer k, there exists at most one $s \in R$ such that $k \cdot s = r$. It is because R is divisible and torsion-free that it makes sense to talk about division by positive integers, and hence multiplication by rational constants.

In fact, a close examination of our decision procedure will reveal that the only fact we use about R is that it is an ordered, divisible, torsion-free Abelian group. Hence our procedure will work as well on Q, the set of rationals (under the usual addition and order).

3. Elimination of quantifiers.

DEFINITION. Let $F_1(x_1, \cdots, x_n)$ and $F_2(x_1, \cdots, x_n)$ be formulas of $\mathscr{L}$. Then F_1 and F_2 are *equivalent* if for every $r_1, \cdots, r_n \in R, F_1(r_1, \cdots, r_n)$ is true $\Leftrightarrow F_2(r_1, \cdots, r_n)$ is true.

The goal of this section is to prove the following theorem.

THEOREM 1. *For every formula $F(x_1, \cdots, x_n)$, there exists an equivalent quantifier-free formula $F'(x_1, \cdots, x_n)$. In fact, there is an effective procedure for going from F to F'.*

It is clear how Theorem 1 leads to a decision procedure for S. To decide if a sentence F is true, one need merely find an equivalent quantifier-free sentence F'; F' will be a Boolean combination of the atoms TRUE and FALSE, which we know how to decide.

The proof of Theorem 1 is by induction on the complexity of $F(x_1, \cdots, x_n)$. If F is an atomic formula, then we can take F' to be F. If F is $F_1 \vee F_2$, then we can take F' to be $F'_1 \vee F'_2$, where F'_1 and F'_2 are quantifier-free formulas equivalent, respectively, to F_1 and F_2. If F is $\sim F_1$, then we can take F' to be $\sim F'_1$. The remaining two cases, $\forall x F_1$ and $\exists x F_1$, are handled by the following lemma, since the quantifier $\forall x$ is equivalent to $\sim \exists x \sim$.

LEMMA 1. *Let $B(x, x_1, \cdots, x_n)$ be a quantifier-free formula. Then there exists an effective procedure for obtaining another quantifier-free formula, $B'(x_1, \cdots, x_n)$, such that $B'(x_1, \cdots, x_n)$ is equivalent to $\exists x B(x, x_1, \cdots, x_n)$.*

Proof. Let $B(x, x_1, \cdots, x_n)$ be a quantifier-free formula.

Step 1. "Solve for x" in each atomic formula of B to obtain a quantifier-free formula, $D(x, x_1, \cdots, x_n)$, such that every atomic formula of D either does not involve x or is of the form (i) $x < t$, (ii) $t < x$, or (iii) $x = t$, where t is a term not involving x.

Step 2. We now make the following definitions:

Given $D(x, x_1, \cdots, x_n)$, to get $D_{-\infty}(x_1, \cdots, x_n)$ ($D_{\infty}(x_1, \cdots, x_n)$), replace

$$\begin{gathered} x < t \text{ in } D \text{ by TRUE (FALSE)}, \\ t < x \text{ in } D \text{ by FALSE (TRUE)}, \\ x = t \text{ in } D \text{ by FALSE (FALSE)}. \end{gathered}$$

Clearly, for any real numbers $r_1, \cdots, r_n$, if r is a sufficiently small real number, then $D(r, r_1, \cdots, r_n)$ and $D_{-\infty}(r_1, \cdots, r_n)$ are equivalent. A similar statement can be made for D_{∞} for r sufficiently large.

Step 3. We will now eliminate the quantifier from $\exists x D(x, x_1, \cdots, x_n)$ using a method very similar to that used by Cooper in his decision procedure for Presburger arithmetic [1]. Let U be the set of all terms t (not involving x) such that $t < x$, $x < t$, or $x = t$ is an atomic formula of D.

LEMMA 1.1. *$\exists x D(x, x_1, \cdots, x_n)$ is equivalent to the quantifier-free formula $B'(x_1, \cdots, x_n)$ defined to be*

$$D_{-\infty} \vee D_{\infty} \vee \bigvee_{t,v \in U} D((t + v)/2, x_1, \cdots, x_n).$$

Proof. Suppose we are given real numbers $r_1, \cdots, r_n$.

$(B' \to \exists xD)$: Suppose

$$D_{-\infty} \vee D_{\infty} \vee \bigvee_{t,v \in U} D((t+v)/2, r_1, \cdots, r_n)$$

is true. If one of the disjuncts $D((t+v)/2, r_1, \cdots, r_n)$ is true, so is $\exists xD(x, r_1, \cdots, r_n)$. So suppose one of the first two disjuncts is true, say $D_{-\infty}$. (The proof for D_∞ is similar.) Then since we can pick r sufficiently small so that $D(r, r_1, \cdots, r_n)$ is equivalent to $D_{-\infty}$, $\exists xD(x, r_1, \cdots, r_n)$ is true.

$(\exists xD \to B')$: Suppose $\exists xD(x, r_1, \cdots, r_n)$ is true. Let $t_1, \cdots, t_m$ be the distinct real numbers, in increasing order, obtained by substituting $r_1, \cdots, r_n$ for $x_1, \cdots, x_n$ in the terms in U. Since $\exists xD(x, r_1, \cdots, r_n)$ is true, there is some real number r such that $D(r, r_1, \cdots, r_n)$ is true. Now r must satisfy a specific order relation with respect to the numbers $t_1, \cdots, t_m$. That is, exactly one of the following must hold:

(a) $r < t_1$,

(b) $t_m < r$,

(c) $r = t_i$ for some i, $1 \leqq i \leqq m$,

(d) $t_i < r < t_{i+1}$ for some i, $1 \leqq i \leqq m - 1$.

If any other real number r' satisfies the same order relations with respect to $t_1, \cdots, t_m$ as r, then $D(r', r_1, \cdots, r_n)$ is true. So if (a) holds, $D_{-\infty}$ must be true; if (b) holds, D_∞ must be true; if (c) holds, $D((t_i + t_i)/2, r_1, \cdots, r_n)$ must be true; if (d) holds, $D((t_i + t_{i+1})/2, r_1, \cdots, r_n)$ must be true.

So Lemma 1.1, Lemma 1 and Theorem 1 are proven. The key point of the proof was in Step 3, where (following Cooper) instead of putting the formula D in disjunctive normal form as is usually done, we replaced $\exists xD(x, x_1, \cdots, x_n)$ by (essentially) a disjunct of formulas of the form $D(t, x_1, \cdots, x_n)$ for t a term in our language.

4. Bounds on the procedure.

The purpose of this section is to show that the desired space bound can be attained. In order to do this, we want to compute a space bound on the elimination of quantifiers procedure given in § 3.

It should be noted that we are using as our model of computation the deterministic, one-tape Turing machine; space bounds, or the number of tape squares used by the Turing machine, are given as a function of n, the length of the sentence the machine is deciding. As is widely known, this model is not restrictive for bounds as large as exponential, since it can simulate a multitape or nondeterministic machine in space at most the square of the space required by the more powerful model [4]. Of course, we describe our procedure informally, leaving it to the reader to convince himself or herself that straightforward implementation of our procedure on a Turing machine would achieve the claimed bounds on time and space.

Notation. If F is a formula, let $l(F)$ be the length of F and let $s(F)$ be the largest absolute value of any integral constant appearing in any rational constant in F. (We assume, for ease of computing the complexity of our procedure, that $l(F) \geqq 2$ and $s(F) \geqq 2$.) By the "length" of an integer, we merely mean its length when written out in binary.

DEFINITION. Let r be a real number and let k be a positive integer. Then r is *limited by k*, written $r \preceq k$, if r is rational and if there exist integers a, b such that $r = a/b$ and $|a| \leqq k$ and $|b| \leqq k$.

Remark. Let $r_1, r_2, \cdots, r_k$ be real numbers limited by the positive integers $w_1, w_2, \cdots, w_k$ respectively. Then $r_1 + r_2 + \cdots + r_k \preceq k \cdot w_1 \cdot w_2 . \cdots . w_k$ and $r_1 \cdot r_2 . \cdots . r_k \preceq w_1 \cdot w_2 . \cdots . w_k$. Now let $B(x, x_1, \cdots, x_k)$ be a quantifier-free formula and let $B'(x_1, \cdots, x_k)$ be the formula obtained by applying the elimination of quantifiers procedure of § 3 to $\exists xB$. Let $s_0 = s(\exists xB)$ and let $l_0 = l(\exists xB)$. We compute an upper bound on $s(B')$ in terms of s_0 and an upper bound on $l(B')$ in terms of l_0.

Step 1 of the procedure, "Solve for x," first involves putting each atomic formula of B which contains x in the form $ax = t$, or $t < ax$ or $ax < t$, where t is a term not containing x. Call the resulting formula $C(x, x_1, \cdots, x_k)$. Obtaining C involves, for each variable in each atomic formula, subtracting one rational coefficient from another. Hence by the remark above, $s(C) \leqq 2(s_0)^2$. Step 1 then entails dividing through in each atomic formula of C by the coefficient of x (if it is nonzero) to obtain the formula $D(x, x_1, \cdots, x_k)$. Clearly, $s(D) \leqq (s(C))^2 \leqq 4(s_0)^4$.

No new integer constant is created by writing down D_∞ and $D_{-\infty}$.

Step 3 of the procedure involves writing $D((t + v)/2, x_1, \cdots, x_k)$ for every pair of terms t, v in D which don't contain x. Now $s(t + v) \leqq 2 \cdot (s(D))^2$, so we have

$$(1) \qquad s((t + v)/2) \leqq 4 \cdot (s(D))^2 \leqq (s_0)^{14}.$$

So $s(B') \leqq (s_0)^{14}$.

To calculate $l(B')$, note that $l(D_\infty)$ and $l(D_{-\infty})$ are both $\leqq l_0$. D looks exactly like B except that the atomic formulas have been changed, so D has no more than l_0 terms. Therefore we have to write down no more than l_0^2 formulas of the form $D((t + v)/2, x_1, \cdots, x_k)$. To determine the length of each $D((t + v)/2, x_1, \cdots, x_k)$, note that in each of the at most l_0 atomic formulas, we may have to write two terms, each term containing k rational coefficients, each numerator and denominator of each coefficient bounded in size by $(s_0)^{14}$ and in length by $14 \cdot \text{length}\,(s_0)$. So the length of each formula $D((t + v)/2, x_1, \cdots, x_k) \leqq l_0 \cdot 2 \cdot k \cdot 2 \cdot (14 \cdot \text{length}\,(s_0)) \leqq 56(l_0)^3$. So $l(B') \leqq 2l_0 + l_0^2(56(l_0)^3) \leqq (l_0)^{14}$.

We now compute the amount of space it would take to eliminate quantifiers in a formula E where $l(E) = l_0$, $s(E) = s_0$, and the number of quantifiers in E is n_0. Our analysis is similar to that given by Oppen [3] for Cooper's procedure for integral addition. We first put E in prenex normal form, using the standard algorithm but always choosing variables with the shortest subscripts possible, obtaining E'. Note that E' is of length $\leqq l_0 \log (l_0)$; this is because there are at most l_0 occurrences of variables, and thus any subscript of a variable in E will be increased in length by a factor of at most $\log (l_0)$. Note that the prenex normal form procedure does not change the number of quantifiers or the size of constants, so E' has n_0 quantifiers and $s(E') = s(E)$.

Clearly, the largest formula obtained in the course of eliminating quantifiers from E' is of length at most

$$(l_0 \log l_0)^{14^{n_0}} \leqq (l_0 \log l_0)^{14^{l_0}} \leqq 2^{2^{c_0 l_0}}$$

for some constant c_0. Also, the largest integer constant (in absolute value) encountered is at most

$$(s_0)^{14^{n_0}}.$$

Notice that if E is a sentence, then the total space used in eliminating quantifiers from E need be no more than

$$2^{2^{c_0 \cdot l(E)}}$$

Since the number of steps involved in each quantifier elimination (and also in the final step of evaluating a Boolean combination of TRUE and FALSE) is only a fixed polynomial in the total space used, we see that our procedure operates within time

$$2^{2^{c_1 \cdot l(E)}}$$

for some constant c_1.

Our next goal is to derive a new decision procedure for S which will be approximately as efficient as the previous one with respect to time but more efficient with respect to space.

DEFINITION. A quantifier Qx, where Q is $\forall$ or $\exists$, is *limited by the positive integer* k (written $Qx \preceq k$) if, instead of ranging over all real numbers, it ranges over the numbers limited by k.

LEMMA 2. *There exists a constant c such that the following is true. Let $F(x, x_1, \cdots, x_k)$ be a formula containing n quantifiers; let $s_0 = s(F)$ and let $r_1, \cdots, r_k$ be any real numbers limited by the positive integers $w_1, \cdots, w_k$, respectively; let Q be either a universal or existential quantifier. Then $QxF(x, r_1, \cdots, r_k)$ is true if and only if*

$$[Qx \preceq (s_0)^{2^{c(n+k)}}(w_1 \cdot \cdots \cdot w_k)]F(x, r_1, \cdots, r_k)$$

is true. (*If $k = 0$, then we take $w_1 \cdot \cdots \cdot w_k$ to equal* 1).

Proof. Since $\forall x$ is equivalent to $\sim \exists x \sim$, we may assume without loss of generality that Q is existential. Let $F'(x, x_1, \cdots, x_k)$ be the quantifier-free formula equivalent to F obtained by our quantifier elimination procedure. If we solve for x in F' and take the average of any two terms that appear, (1) tells us that every rational coefficient will be limited by $(s(F'))^{14}$.

Assume now that some value of x satisfies $F'(x, r_1, \cdots, r_k)$, where $r_i \preceq w_i$ for $1 \leqq i \leqq k$. Then some value of x satisfying $F'(x, r_1, \cdots, r_k)$ is either equal to the average of two terms obtained by solving for x in $F'(x, r_1, \cdots, r_k)$ or is 1 bigger than or 1 smaller than all such averages. It is sufficient, therefore, to show that any average is limited by

$$(s_0)^{2^{c(n+k)}}(w_1 \cdot \cdots \cdot w_k).$$

But by the above paragraph, any such average is equal to $\sum_{i=1}^k a_i r_i$ for some $a_1, \cdots, a_k$ limited by $(s(F'))^{14}$. Since $a_i r_i \preceq (s(F'))^{14} \cdot w_i$, for $1 \leqq i \leqq k$, we have $\sum_{i=1}^k a_i r_i \preceq k \cdot \prod_{i=1}^k [(s(F'))^{14} w_i]$. Since $s(F') \leqq (s_0)^{14^n}$, one can easily calculate that

$$\sum_{i=1}^{k} a_i r_i \preceq (s_0)^{2^{c(n+k)}}(w_1 \cdot \cdots \cdot w_k).$$

for some constant c.

LEMMA 3. *Let c be the constant of Lemma 2, let $Q_1 x_1 Q_2 x_2 \cdots Q_n x_n F(x_1, \cdots, x_n)$ be a sentence, where F is quantifier-free and where Q_i is $\forall$ or $\exists$ for each i, $1 \leqq i$*

$\leqq n$, and let $s_0 = s(F)$. Let $w_1 = (s_0)^{2^{cn}}$ and let $w_{k+1} = (s_0)^{2^{cn}}(w_1 \cdot \cdots \cdot w_k)$ for $1 \leqq k < n$. Then $Q_1x_1 \cdots Q_nx_nF(x_1, \cdots, x_n)$ is true if and only if $(Q_1x_1 \preceq w_1) \cdot (Q_2x_2 \preceq w_2) \cdots (Q_nx_n \preceq w_n)F(x_1, \cdots, x_n)$ is true.

Proof. The proof is immediate from Lemma 2.

THEOREM 2. *There is a constant d, and a decision procedure for S, such that to decide a sentence B of length n takes at most space 2^{dn}.* (Note that the procedure must therefore take time $\leqq 2^{2^{d'n}}$, for some constant d', because of a well-known theorem of automata theory relating time and space.)

Proof. Let B be a sentence of length n, and let $s_0 = s(B)$. Put B in prenex normal form to obtain a sentence B'. Now $l(B') \leqq n \log(n)$, $s(B') = s_0$, and B' has no more than n quantifiers, so we can assume B' looks like $Q_1x_1 \cdots Q_nx_nF(x_1, \cdots, x_n)$, where F is quantifier-free and Q_i is $\forall$ or $\exists$ for $1 \leqq i \leqq n$.

Define $w_1 = (s_0)^{2^{cn}}$ and $w_{k+1} = (s_0)^{2^{cn}}(w_1 \cdot \cdots \cdot w_k)$ for $1 \leqq k \leqq n$. Then by Lemma 3, B' is equivalent to $(Q_1x_1 \preceq w_1) \cdots (Q_nx_n \preceq w_n)F(x_1, \cdots, x_n)$. It is easy to calculate that $w_k = ((s_0)^{2^{cn}})^{2^{k-1}}$ for $1 \leqq k \leqq n$, so $w_n \leqq (s_0)^{2^{(c+1)n}}$. Since $s_0 \leqq 2^n$, we have $w_n \leqq 2^{2^{c'n}}$ for some constant c'. Note that every rational constant limited by $2^{2^{c'n}}$ can be written in space proportional to $2^{c'n}$ (since integer constants are written in binary). So B' can be decided by cycling through the set of rationals associated with each quantifier appropriately, all the time testing the truth of F on different n-tuples of rational constants. We let the reader convince himself or herself that a Turing machine implementing this outlined procedure need use only 2^{dn} tape squares for some constant d.

5. Applications.

The idea of deciding truth in a particular theory as outlined above can be applied to many other theories, thereby obtaining procedures of considerable computational efficiency. That is, given a particular theory, one gives an elimination of quantifiers procedure, analyzes it to see how "large" constants can grow, and then uses this analysis and the original procedure (in a manner similar to that given above) to limit quantifiers to range over finite sets instead of an infinite domain.

In particular, we consider the efficient quantifier elimination procedure given by Cooper [1] for deciding truth in the first order theory of integer addition. Define the first order language $\mathscr{L}'$ as follows:

$\mathscr{L}'$ has variables $x_0, x_1, x_{10}, \cdots$ (i.e., the subscripts are written in binary); for each integer i, $\mathscr{L}'$ has a *constant symbol i* (written in binary);

$\mathscr{L}'$ has *terms* of the form $a_1y_1 + \cdots + a_ky_k$, where a_i is an integer constant for $1 \leqq i \leqq k$ and where $y_1, y_2, \cdots, y_k$ are distinct formal variables;

$\mathscr{L}'$ has *atomic formulas* of the form $t_1 \leqq t_2$ (read "t_1 is less than or equal to t_2") or $a|t_1$ (read "a divides t_1"), where t_1 and t_2 are terms and a is a positive integer constant, or TRUE, or FALSE.

Sentences and formulas are built up in the usual way.

Let S' be the set of sentences of $\mathscr{L}'$ which are true of Z, the set of integers, when the symbols of $\mathscr{L}'$ are interpreted in the obvious way. Cooper decides S' by elimination quantifiers, and Oppen [3] has determined bounds for this procedure.

DEFINITION. An integer n is *limited by the positive integer k*, written $n \preceq k$, if $|n| \leqq k$.

DEFINITION. If F is a formula of $\mathscr{L}'$, then $s(F)$ is the smallest integer $\geqq 2$ such that every integer constant of F is limited by $s(F)$.

THEOREM 3 (Oppen). *There exists a constant e such that the following is true. If F is a formula of $\mathscr{L}'$ with n quantifiers, then when Cooper's procedure is applied to F, every integer constant encountered is limited by*

$$(s(F))^{2^{2^{en}}}.$$

We can now state a lemma.

LEMMA 4. *There exists a constant f such that the following is true. Let $F(x, x_1, \cdots, x_k)$ be a formula of $\mathscr{L}'$ containing n quantifiers; let $s_0 = s(F)$ and let $n_1, \cdots, n_k$ be integers limited by the positive integer w. Then $\exists x F(x, n_1, \cdots, n_k)$ is true of Z if and only if*

$$[\exists x \precsim (s_0)^{2^{2^{f(n+k)}}} \cdot w] F(x, n_1, \cdots, n_k)$$

is true of Z.

Proof. Use Theorem 3, Cooper's procedure, and an analysis similar to that given for real addition.

LEMMA 5. *There exists a constant g such that the following is true. Let B be the formula $Q_1x_1 \cdots Q_nx_nF(x_1, \cdots, x_n)$, where F is quantifier-free and Q_i is $\forall$ or $\exists$ for each i, $1 \leqq i \leqq n$; let $s_0 = s(F)$. Then B is true of Z if and only if*

$$(Q_1x_1 \precsim (s_0)^{2^{2^{gn+1}}})(Q_2x_2 \precsim (s_0)^{2^{2^{gn+2}}}) \cdots (Q_nx_n \precsim (s_0)^{2^{2^{gn+n}}})F(x_1, \cdots, x_n)$$

is true of Z.

Proof. Apply the previous lemma.

We can now state the following theorem.

THEOREM 4. *There exists a constant h and a decision procedure for S' such that to decide a sentence of length n takes at most $2^{2^{hn}}$ space.*

Remark. Theorem 4 should be compared to the following result of Fischer and Rabin [2].

THEOREM (Fischer and Rabin). *There exists a constant $j > 0$ such that any nondeterministic Turing machine which decides S' requires for almost every n time $2^{2^{jn}}$ to decide some sentences of length n.*

Acknowledgments. We would like to acknowledge some helpful suggestions by Henry Baker in the early stages of our work and the assistance of Albert Meyer in the conception and preparation of this paper.

REFERENCES

[1] C. D. COOPER, *Theorem-proving in arithmetic without multiplication*, Machine Intelligence 7, Meltzer and Michie, ed., John Wiley, 1972, pp. 91–99.

[2] M. FISCHER AND M. O. RABIN, *Super exponential complexity of Presburger arithmetic*, Project MAC Tech. Mem. 43, Mass. Inst. of Tech., Cambridge, 1974.

[3] D. C. OPPEN, *Elementary bounds for Presburger arithmetic*, 5th ACM Sympos. on the Theory of Computing, 1973, pp. 34–37.

[4] W. J. SAVITCH, *Relationships between nondeterministic and deterministic tape complexities*, J. Comput. System Sci., 4 (1970), pp. 177–191.

[5] A. TARSKI, *A Decision Method for Elementary Algebra and Geometry*, 2nd ed., University of California Press, Berkeley, 1951.

SIAM J. COMPUT.
Vol. 4, No. 1, March 1975

FINDING ALL THE ELEMENTARY CIRCUITS OF A DIRECTED GRAPH*

DONALD B. JOHNSON†

Abstract. An algorithm is presented which finds all the elementary circuits of a directed graph in time bounded by $O((n + e)(c + 1))$ and space bounded by $O(n + e)$, where there are n vertices, e edges and c elementary circuits in the graph. The algorithm resembles algorithms by Tiernan and Tarjan, but is faster because it considers each edge at most twice between any one circuit and the next in the output sequence.

Key words. algorithm, circuit, cycle, enumeration, digraph, graph

1. Introduction. Broadly speaking, there are two enumeration problems on sets of objects. The one, which we call *counting*, is determining how many objects there are in the set. The other, which we call *finding*, is the construction of every object in the set exactly once. Indeed, objects may always be counted by finding them if a method to do so is at hand. But knowing the count is usually of little aid in finding the objects.

We give an algorithm for finding the elementary circuits of a directed graph which is faster in the worst case than algorithms previously known. As far as we know, it is also the fastest method known for the general enumeration problem as well (see [1, p. 226]). Specific counting problems are, of course, solved. For example, there are exactly

$$\sum_{i=1}^{n-1} \binom{n}{n-i+1} (n-i)!$$

elementary circuits in a complete directed graph with n vertices. Thus the number of elementary circuits in a directed graph can grow faster with n than the exponential 2^n. So it is clear that our algorithm, which has a time bound of $O((n + e)(c + 1))$ on any graph with n vertices, e edges and c elementary circuits, is feasible for a substantially larger class of problems than the best algorithms previously known [2], [3], which realize a time bound of $O(n \cdot e(c + 1))$.

A *directed graph* $G = (V, E)$ consists of a nonempty and finite set of vertices V and a set E of ordered pairs of distinct vertices called *edges*. There are n vertices and e edges in G. A *path* in G is a sequence of vertices $p_{vu} = (v = v_1, v_2, \cdots, v_k = u)$ such that $(v_i, v_{i+1}) \in E$ for $1 \leqq i < k$. A *circuit* is a path in which the first and last vertices are identical. A path is *elementary* if no vertex appears twice. A circuit is elementary if no vertex but the first and last appears twice. Two elementary circuits are distinct if one is not a cyclic permutation of the other. There are c distinct elementary circuits in G. Our definitions exclude graphs with loops (edges of the form (v, v)) and multiple edges between the same vertices. It is obvious that for any circuit q_{vv}, there exists a vertex u such that q_{vv} is composed of a path p_{vu} followed by edge (u, v). If q_{vv} is elementary, then p_{vu} is also elementary.

* Received by the editors December 10, 1973, and in final revised form June 10, 1974.

† Computer Science Department, Pennsylvania State University, University Park, Pennsylvania 16802.

F is a *subgraph of* G *induced by* W if $W \subseteq V$ and $F = (W, \{(u, v)|u, v \in W$ and $(u, v) \in E\})$. An induced subgraph F is a (maximal) *strong component of* G if for all $u, v \in W$ there exist paths p_{uv} and p_{vu} and this property holds for no subgraph of G induced by a vertex set $\overline{W}$ such that $W \subset \overline{W} \subseteq V$.

The literature contains several algorithms which find the elementary circuits of any direct graph. In the algorithms of Tiernan [4] and of Weinblatt [5], time exponential in the size of the graph may elapse between the output of one circuit and the next [2]. Tarjan [2] presents a variation of Tiernan's algorithm in which at most $O(n \cdot e)$ time elapses between the output of any two circuits in sequence, giving a bound of $O(n \cdot e(c + 1))$ for the running time of the algorithm on an entire graph in the worst case. Ehrenfeucht, Fosdick, and Osterweil [3] give a similar algorithm which realizes the same bound.

In the case of Tarjan's algorithm, the worst-case time bound is realized, for instance, on the graph shown in Fig. 1. We assume that the algorithm begins with vertex 1 and, in any search from vertices 1 through $k + 1$, it visits vertices $k + 2$ through $2k + 1$ before a first visit to vertex $2k + 2$. In the course of finding each of the k elementary circuits which contain vertex 1, the subgraph on vertices $2k + 2$ through $3k + 3$ will be explored k times, once for each of the vertices $k + 2$ through $2k + 1$. Thus exploration from vertex 1 alone consumes $O(k^3)$ time. Since there are exactly $3k$ elementary circuits in the entire graph, the running time is at least $O(n \cdot e(c + 1))$.

The worst-case time bound for Tarjan's algorithm is also realized on the graph in Fig. 2. Assuming a start at vertex 1, the algorithm takes $O(k)$ time to find the one elementary circuit of the graph. Then the fruitless searches from vertices 2 through k take $O(k^2)$ time, which is $O(n \cdot e(c + 1))$. So we see that there are two ways in which the time bound is realized in Tarjan's algorithm. One is through repeated fruitless searching of a subgraph while seeking circuits with a certain least vertex; the other is through fruitless searches from many vertices which are least vertices in no elementary circuit.

The graphs of Figs. 1 and 2 are strongly connected, and their undirected versions are biconnected. On such graphs, obvious preprocessing techniques, such as reducing the graph to its strong components (as Weinblatt does [5]) or to components which in addition to strong connectivity have biconnected undirected versions, do not improve the performance of Tarjan's algorithm. Stronger techniques are needed to get a better asymptotic running time in the worst case.

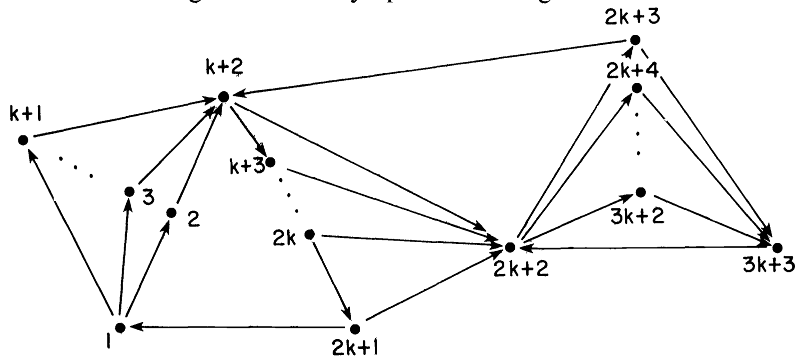


FIG. 1. *A worst-case example for Tarjan's algorithm*

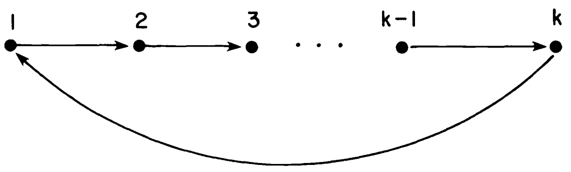


FIG. 2. *A second worst-case example for Tarjan's algorithm*

2. The algorithm. In our algorithm, the time consumed between the output of two consecutive circuits as well as before the first and after the last circuits never exceeds the size of the graph, $O(n + e)$. We employ the basic notion of Tiernan's algorithm. Elementary circuits are constructed from a root vertex s in the subgraph induced by s and vertices "larger than s" in some ordering of the vertices. Thus the output is grouped according to least vertices of the circuits.

To avoid duplicating circuits, a vertex v is *blocked* when it is added to some elementary path beginning in s. It stays blocked as long as every path from v to s intersects the current elementary path at a vertex other than s. Furthermore, a vertex does not become a root vertex for constructing elementary paths unless it is the least vertex in at least one elementary circuit. These two features avoid much of the fruitless searching of Tiernan's, Weinblatt's and Tarjan's algorithms and of the algorithm of Ehrenfeucht, Fosdick, and Osterweil.

The algorithm accepts a graph G represented by an adjacency structure A_G composed of an adjacency list $A_G(v)$ for each $v \in V$. The list $A_G(v)$ contains u if and only if edge $(v, u) \in E$. The algorithm assumes that vertices are represented by integers from 1 to n.

The algorithm proceeds by building elementary paths from s. The vertices of the current elementary path are kept on a stack. A vertex is appended to an elementary path by a call to the procedure CIRCUIT and is deleted upon return from this call. When a vertex v is appended to a path it is blocked by setting blocked (v) = true, so that v cannot be used twice on the same path. Upon return from the call which blocks v, however, v is not necessarily unblocked. Unblocking is always delayed sufficiently so that any two unblockings of v are separated by either an output of a new circuit or a return to the main procedure.

CIRCUIT-FINDING ALGORITHM

```
begin
    integer list array $A_K(n)$, $B(n)$; logical array blocked $(n)$; integer $s$;
    logical procedure CIRCUIT (integer value $v$);
        begin logical $f$;
            procedure UNBLOCK (integer value $u$);
                begin
                    blocked $(u)$ := false;
                    for $w \in B(u)$ do
                        begin
                            delete $w$ from $B(u)$;
                            if blocked$(w)$ then UNBLOCK$(w)$;
                        end
                end UNBLOCK;
            $f$ := false;
            stack $v$;
            blocked($v$) := true;
    L1:     for $w \in A_K(v)$ do
                if $w = s$ then
                    begin
                        output circuit composed of stack followed by s;
                        $f$ := true;
                    end
                else if $\neg$blocked($w$) then
                        if CIRCUIT($w$) then $f$ := true;
    L2:     if $f$ then UNBLOCK($v$)
            else for $w \in A_K(v)$ do
                    if $v \notin B(w)$ then put $v$ on $B(w)$;
            unstack $v$;
            CIRCUIT := $f$;
        end CIRCUIT;
    empty stack;
    $s$ := 1;
    while $s < n$ do
        begin
            $A_K$ := adjacency structure of strong component $K$ with least
                    vertex in subgraph of $G$ induced by $\{s, s+1, \cdots, n\}$;
            if $A_K \neq \emptyset$ then
                begin
                    $s$ := least vertex in $V_K$;
                    for $i \in V_K$ do
                        begin
                            blocked($i$) := false;
                            $B(i)$ := $\emptyset$;
                        end;
    L3:             dummy := CIRCUIT(s);
                    $s$ := $s+1$;
                end
            else $s$ := $n$;
        end
end;
```

The correctness of the algorithm depends on no vertex remaining blocked when there is a path from the vertex to s which intersects the stack only at s. On the other hand, the bound on running time depends on all vertices remaining blocked as long as possible, consistent with the requirements of correctness. The following lemma establishes these properties. It will be seen that the B-lists are used to remember information obtained from searches of portions of the graph which do not yield an elementary circuit. The procedure UNBLOCK has the property that if there is a call UNBLOCK(x) and vertex y is on list $B(x)$, then there will be a call UNBLOCK(y) following which blocked(y) will be false.

LEMMA 1. *At* L2, *for any vertex* $x \neq s$, *there is a call* UNBLOCK(v) *which sets* blocked(x) = false *if and only if*

(i) *there is a path, containing v, from x to s on which only v and s are on the stack, and*

(ii) *there is no path from x to s on which only s is on the stack.*

Proof. Assume, to the contrary, that there is an execution of L2 at which the lemma first fails and that the lemma fails for no vertex before it fails for vertex y. Two cases are possible under this assumption.

Case 1. Suppose that the path conditions, (i) and (ii), hold for y at L2, but blocked(y) is not set false. Because there is an edge (v, z) on the path from y to s, f is true at L2. This fact is immediate if $z = s$. If $z \neq s$, it follows from our assumption that the lemma holds for z before the return from the call CIRCUIT(z). Thus there is a call UNBLOCK(v) and, without loss of generality, a path $(y = v_1, v_2, \cdots, v_k = v)$ on which only v is on the stack and only y is not unblocked as a result of the call UNBLOCK(v) at L2. But when y was last blocked, y was on the stack. Since y remained blocked when y was removed from the stack, y was put on list $B(v_2)$. So there was a call UNBLOCK(y), a contradiction.

Case 2. Suppose that there is a call UNBLOCK(v) at L2 and that blocked(y) is set false, but that either (i) or (ii) is not satisfied. It cannot be that $v = s$ because it is clear that the lemma holds when only $v = s$ is on the stack, and s cannot be stacked more than once. Since f is true there is an edge (v, z) such that either $z = s$ or f was set true when the call CIRCUIT(z) returned. It follows from our assumption that, when f was set true, there was a path from z to s on which only s was on the stack. It may be that several calls to CIRCUIT occur after f is set true and before the current call UNBLOCK(v). In any event, the current stack (when the call UNBLOCK(v) occurs) is identical to the stack when f was set true, so there is a path $(v, z, \cdots, s)$ on which only v and s are on the stack. Since v is on the stack, (i) and (ii) would be satisfied if $y = v$.

So $y \neq v$, and there is some vertex t which is unblocked before y is unblocked such that y is on $B(t)$. By assumption, there is a path from t to s on which only v and s are on the stack. Furthermore, when y was last put on $B(t)$, blocked(t) was true and y was on the stack. But blocked(t) has to remain true until the current call UNBLOCK(v). Otherwise y would have been removed from $B(t)$. Since y must have been unstacked after y was put on $B(t)$, there was some execution of L2 where the stack was a prefix of the current stack, (i) and (ii) held for t, and blocked(t) was not set false. But by assumption, the lemma did not fail then for t. From this contradiction, we find that Case 2 is also impossible. □

COROLLARY 1. *The algorithm outputs only elementary circuits.*

Proof. Certainly only circuits are output. By Lemma 1, a vertex is only unblocked if it will be off the stack before any call to CIRCUIT can occur. Thus no vertex can be repeated on the stack. □

LEMMA 2. *The algorithm outputs every elementary circuit exactly once.*

Proof. No circuit is output more than once since, for any stack $(s = v_1, v_2, \cdots, v_k)$ with v_k on top, once v_k is removed the same stack cannot reoccur.

Let $(v_1, v_2, \cdots, v_l, v_1)$ be an elementary circuit such that $v_1 \leqq v_i$, $1 \leqq i \leqq l$. A first call CIRCUIT(v_1) will eventually occur at L3 since there is a strong component with least vertex v_1. Since no vertex is blocked when this first call occurs, it follows by induction using Lemma 1 that whenever the stack is $(s = v_1, v_2, \cdots, v_i)$ for $i < l$, the stack will later be $(s = v_1, v_2, \cdots, v_{i+1})$. Thus every elementary circuit is output. □

The foregoing results show that the algorithm does indeed find all the elementary circuits of a directed graph. The bound on running time follows from the next lemma.

LEMMA 3. *At most $O(n + e)$ time can elapse in a call at* L3 *to* CIRCUIT *before either the call returns or a circuit is output.*

Proof. First we show that no vertex can be unblocked twice in succession unless a circuit is output. Then we show that no more than $O(n + e)$ time can elapse before some vertex is unblocked a second time.

Suppose a circuit is output and then some vertex y is unblocked. By Lemma 1, as soon as v is unstacked there is a path $(y = v_1, v_2, \cdots, v_k = s)$ on which only s is on the stack. Let some vertex v_i, $1 \leqq i < k$, be the first vertex on this path to be put on the stack again. We see by induction on the execution of the algorithm that eventually the stack will be $(s, \cdots, v_i, v_{i+1}, \cdots, v_{k-1})$ and a new circuit output. Until the new circuit is output, no vertex on the path will be unstacked. Thus no vertex can be unblocked more than once before a circuit is output.

Charge a unit of cost to a vertex if it is an argument to a procedure call and a unit of cost to an edge if consideration of this edge by the **for** loop at L1 in CIRCUIT does not result in a procedure call. The cost of all work in the procedure CIRCUIT will be bounded by a constant times the number of units charged. For any vertex x, calls to CIRCUIT and UNBLOCK must alternate. Consequently, no more than three units can be charged to each vertex before some vertex is unblocked twice. As to edge charges, let some edge originate in vertex x. A unit may be charged to this edge only when blocked(x) is true and, once a unit is charged, x must be unblocked and blocked again before a second unit can be charged to the same edge. It follows that at most two units can be charged to any edge before some vertex is unblocked twice. □

COROLLARY 2. *The algorithm runs in $O((n + e)(c + 1))$ time and uses $O(n + e)$ storage space plus the space used for output.*

Proof. The time bound follows directly from Lemma 3 and a known algorithm [6] for finding strong components in $O(n + e)$ time. The space bound is immediate from the observation that no vertex appears more than once on any B-list. □

3. Discussion of running time. We have shown that in the worst case, our algorithm is asymptotically faster than algorithms previously known. With respect to Tarjan's algorithm, a stronger statement can be made. There is a constant factor which bounds how much slower our algorithm can be compared to his on any graph, provided the same adjacency structure is used as input to both algorithms. Such a constant, of course, is implementation dependent. Its existence follows from two facts. First, the time spent by our algorithm in finding the strong component with least vertex $s \geqq k$ is of no greater order than the search in Tarjan's algorithm for circuits with least vertex k, for $1 \leqq k \leqq n$. Second, if the calls to UNBLOCK are ignored, on identical adjacency structures the sequence of edge explorations generated by our algorithm is embedded in the sequence generated by Tarjan's. But for every edge in the sequence, for our algorithm there can occur at most one call to UNBLOCK. Therefore, since the search time in each algorithm is related by constant factors to the number of edge explorations, the effort spent by our algorithm in finding circuits from a given base vertex, s, is bounded by a constant factor times the effort expended by Tarjan's algorithm for the corresponding search on the same adjacency structure for any graph.

Experimental results are shown in Tables 1 and 2. The algorithms were implemented in ALGOL W [7] and were run on an IBM 370/168 with virtual address hardware inoperative. The benefit predicted for our algorithm on worst cases is apparent in the results in Table 1. Table 2 shows superior performance by our algorithm on complete graphs as well. Although only a constant factor is involved, this second result is somewhat surprising since on complete graphs, both algorithms make the same number of edge explorations. The result, however, appears to be explained by two features of Tarjan's algorithm. He maintains two vertex stacks and tests in the innermost loop for elimination of vertices less than s. If his algorithm were redesigned to correct these problems, ^he analysis of the

TABLE 1
Running times on the family of graphs of Fig. 1

Number of vertices	Number of circuits	Running time on IBM 370/168, seconds[1]		T_T/T_J
		T_J (Johnson's algorithm)	T_T (Tarjan's algorithm)	
5	15	.03	.06	2
10	30	.11	.27	2.5
20	60	.32	1.67	5.2
40	120	1.17	11.51	9.8
60	180	2.61	36.89	14.1
80	240	4.46	86.66	19.4

[1] Timer resolution 1/60 second. Because running times fluctuate with system load, all data shown were taken from one computer run. Results are averages of two times rounded to the second decimal place.

TABLE 2
Running times on complete directed graphs

Number of vertices	Number of circuits	Running time on IBM 370/168, seconds[2]		T_T/T_J
		T_J (Johnson's algorithm)	T_T (Tarjan's algorithm)	
2	1	0	0	—
3	5	0	0	—
4	20	.02	0	—
5	84	.02	.02	1
6	409	.07	.08	1.1
7	2365	.35	.51	1.5
8	16064	2.43	3.63	1.5
9	125664	20.17	30.13	1.5

[2] Timer resolution 1/60 second. Because running times fluctuate with system load, all data shown were taken from one computer run. Results are averages of two times rounded to the second decimal place.

preceding paragraph would still hold. In both tests, the space bound of $O(n + e)$ was confirmed.

4. Conclusions. The algorithm we have shown is faster asymptotically in the worst case than algorithms previously known. The algorithm appears particularly suited for general use because of the stronger property, which we have shown in relation to Tarjan's algorithm, of being never slower on any graph by more than a constant factor. In fact, in the tests run, our algorithm was always faster except on trivially small graphs.

Acknowledgment. Credit is due to the referees for suggesting that, for comparative tests, a family of graphs be found in which the members are strongly connected and whose undirected versions are biconnected, and for a suggestion that led to an improvement, by a constant factor, in running time.

REFERENCES

[1] F. HARARY AND E. PALMER, *Graphical Enumeration*, Academic Press, New York, 1973.
[2] R. TARJAN, *Enumeration of the elementary circuits of a directed graph*, this Journal, 2 (1973), pp. 211–216.
[3] A. EHRENFEUCHT, L. FOSDICK AND L. OSTERWEIL, *An algorithm for finding the elementary circuits of a directed graph*, Tech. Rep. CU-CS-024-23, Dept. of Computer Sci., Univ. of Colorado, Boulder, 1973.
[4] J. C. TIERNAN, *An efficient search algorithm to find the elementary circuits of a graph*, Comm. ACM, 13 (1970), pp. 722–726.
[5] H. WEINBLATT, *A new search algorithm for finding the simple cycles of a finite directed graph*, J. Assoc. Comput. Mach., 19 (1972), pp. 43–56.
[6] R. TARJAN, *Depth-first search and linear graph algorithms*, this Journal, 1 (1972), pp. 146–160.
[7] R. L. SITES, *Algol W reference manual*, Tech. Rep. STAN-CS-71-230, Computer Sci. Dept., Stanford Univ., Stanford, Calif., 1971.

SIAM J. COMPUT.
Vol. 4, No. 1, March 1975

STATE-SPLITTING FOR STOCHASTIC MACHINES*

EUGENE S. SANTOS†

Abstract. In this paper, a systematic theory of the "state-splitting" technique for the decomposition of stochastic machines is presented, which makes use of a stochastic generalization of the conventional concept of covers or set systems.

1. Introduction. The "state-splitting" technique, which makes use of the concept of covers or set systems, is one of the most powerful techniques in the study of the decomposition of deterministic machines [4]. Although the possibility of "state-splitting" had been considered in [2] and an example can be found in [5], the present paper, to the best of our knowledge, is the first attempt to provide a systematic theory to this aspect of the decomposition theory of stochastic machines.

The main contents of the present paper are contained in §§2–5. In §2, the basic concepts and notations which are needed in subsequent discussions are introduced. Following [8], quasi-stochastic systems [7] are used. However, for simplicity, only quasi-stochastic state-machines (QSSM) are considered. Most of the concepts and results presented in §2 are the state-machines analogue of that of [8].

In §3, the properties of regular assignments are further investigated and two characterizations are derived. In §4, connections of concurrently operating QSSM's are examined. However, instead of quasi-series connection, which was introduced in [1] and adopted in [8], the cascade connection, which was introduced in [3], is considered. A necessary and sufficient condition for a QSSM to admit a cascade decomposition of its state-behavior is derived which makes use of the concept of regular SP-partitions introduced in [8].

In §5, the concept of covers is introduced. Using the results established in §§3 and 4, it is shown that if a QSSM has a regular SP-cover, then it admits a cascade decomposition.

2. Basic concepts and notations. In this section, we shall introduce the basic concepts and notations which are needed in subsequent discussions. Most of these concepts are similar to those introduced in [8], and the readers are referred there for motivations and further details.

Notation. E is a column matrix of appropriate order whose entries are all 1.

DEFINITION. Let K be a real matrix. (i) K is a *1-matrix* iff $KE = E$. (ii) K is a *2-matrix* iff $KE = E$ and all entries of K are nonnegative, i.e., K is a stochastic matrix. (iii) K is a *3-matrix* iff $KE = E$ and all entries of K are either 0 or 1.

DEFINITION. Let $k = 1, 2,$ or 3. A k-QSSM (short for *quasi-stochastic state-machines*) is a system $M = (U^M, \{P^M(u): u \in U^M\})$ where U^M is a nonempty finite set, and for each $u \in U^M$, $P^M(u)$ is an $n \times n$ k-matrix, where $n = |M|$, the order of M.

* Received by the editors August 28, 1973, and in revised form April 22, 1974.

† Department of Mathematics, Youngstown State University, Youngstown, Ohio 44503. This work was supported in part by the University Research Council of Youngstown State University.

In the above definition, U^M is the input set, $|M|$ is the number of states of M and $P^M(u)$ is the transition matrix when input u is applied.

In what follows, the symbol M, with or without subscripts, will always represent a 1-QSSM, and k will stand for 1, 2 or 3, unless otherwise stated.

Notation. (i) $\bar{U}^M$ is the free monoid generated by U^M. (ii) $\lg(\bar{u})$ is the length of $\bar{u} \in \bar{U}^M$. (iii) $P^M(u_1 u_2 \cdots u_n) = P^M(u_1)P^M(u_2) \cdots P^M(u_n)$ for all $u_1, u_2, \cdots, u_n \in U^M$.

DEFINITION. Let K be a k-matrix such that $P^{M_1}(u)K = KP^{M_2}(u)$ for all $u \in U^{M_1} \cap U^{M_2}$. (i) If $U^{M_1} \subseteq U^{M_2}$ and the rows of K are linearly independent, then M_1 is a *k-submachine* of M_2. (ii) If $U^{M_1} \supseteq U^{M_2}$ and the columns of K are linearly independent, then M_1 is a *k-split* of M_2. The matrix K will be referred to as the *associated* matrix.

Notation. $H_k(M)$ is the collection of all $1 \times |M|$ k-matrices. Moreover, $H(M) = H_1(M)$, and $H^*(M)$ is the collection of all nonempty subsets of $H(M)$.

Notation. Let W be a real vector space, and for $i = 1, 2, \cdots, n$, $W_i \subseteq W$ and c_i real numbers, then the set

$$\sum_{i=1}^{n} c_i W_i = \left\{ \sum_{i=1}^{n} c_i w_i : w_i \in W_i,\ i = 1, 2, \cdots, n \right\}.$$

DEFINITION. Let β be a function from $H(M_1)$ into $H^*(M_2)$. β is *regular* iff $h_1, h_2 \in H(M_1)$ and c_1, c_2 are real numbers such that $c_1 + c_2 = 1$ implies $\beta(c_1 h_1 + c_2 h_2) = c_1\beta(h_1) + c_2\beta(h_2)$.

Notation. $H_0(M)$ is the $|M|$-dimensional Euclidean space and

$$\bar{H}_0(M) = \{h \in H_0(M) : hE = 0\}.$$

THEOREM 2.1. *Let β be a function from $H(M_1)$ into $H^*(M_2)$. The following statements are equivalent*:

(a) *β is regular*;

(b) *there exist a subspace W of $\bar{H}_0(M_2)$ and a 1-matrix K_1 such that $\beta(h_1) = h_1 K_1 + W$ for every $h_1 \in H(M_1)$; and*

(c) *there exist 1-matrices K_1 and K_2 such that $\beta(h_1) = \{h_2 \in H(M_2) : h_2 K_2 = h_1 K_1 K_2\}$ for every $h_1 \in H(M_1)$.*

Proof. Let β be regular and $W = \{h_2 - h_2' : h_2, h_2' \in \beta(h_1)\}$. By [8, Thm. 3.4], W does not depend on the choice of $h_1 \in H(M_1)$. Let K_1 be a matrix whose ith row is an element of $\beta(e_i)$, where $\{e_i\}$ is the standard basis of $H_0(M_1)$. It is clear that $\beta(h_1) = h_1 K_1 + W$ for all $h_1 \in H(M_1)$. Thus (a) implies (b). Since $W \subseteq \bar{H}_0(M_2)$, there exists a 1-matrix K_2 such that $g \in W$ iff $gK_2 = 0$. Thus (b) implies (c). Now suppose (c) holds. Let $h_2 \in \beta(h_1)$, $h_2' \in \beta(h_1')$ and a, b are real numbers such that $a + b = 1$. Then $(ah_2 + bh_2')K_2 = (ah_1 + bh_1')K_1K_2$. This implies that $a\beta(h_1) + b\beta(h_1') \subseteq \beta(ah_1 + bh_1')$. By [8, Thm. 3.2], β is regular. Thus (c) implies (a). Q.E.D.

DEFINITION. A (regular) *k-assignment* of M_1 into M_2 is an ordered pair (α, β), where α is a function from U^{M_1} into U^{M_2} and β is a (regular) function from $H(M_1)$ into $H^*(M_2)$, such that for every $u \in U^{M_1}$, $h_1 \in H(M_1)$ and $h_2 \in \beta(h_1)$, (i) $h_2 P^{M_2}(\alpha(u)) \in \beta[h_1 P^{M_1}(u)]$, (ii) $h_1 \in H_k(M_1)$ implies $\beta(h_1) \cap H_k(M_2) \neq \varnothing$, and (iii) $\beta(H(M_1))$ is the affine span of $\beta(H_3(M_1)) \cap H_k(M_2)$, i.e., for each $h_2 \in \beta(H(M_1))$, $h_2 = \sum_{i=1}^{n} c_i h_2^i$ where $h_2^i \in \beta(H_3(M_1)) \cap H_k(M_2)$ and c_i are real, for all $i = 1, 2, \cdots, n$, and $\sum_{i=1}^{n} c_i = 1$. If, in addition, (iv) $\beta(h_1) \cap \beta(h'_1) = \varnothing$ for all $h_1 \neq h'_1$, then (α, β) is a (regular) *strong k-assignment* of M_1 into M_2.

Remark. Condition (iv) is necessary since we are dealing with state-machines. Since β is regular, it is clear that $\beta(H(M_1))$ is the affine span of $\beta(H_3(M_1))$. Condition (iii) states that it suffices to consider only $\beta(H_3(M_1)) \cap H_k(M_2)$.

DEFINITION. Let (α, β) be a regular (strong) k-assignment of M_1 into M_2. If for every $h_1 \in H(M_1)$, $\beta(h_1)$ contains exactly one element, then (α, β) is a *reduced* (strong) *k-assignment* of M_1 into M_2.

DEFINITION. M_2 is a (reduced, regular) (strong) *k-realization* of M_1 iff there exists a (reduced, regular) (strong) k-assignment of M_1 into M_2.

For simplicity, 1-QSSM, 1-assignment and 1-realization will also be called QSSM, assignment and realization, respectively. Moreover, if (α, β) is a reduced k-assignment of M_1 into M_2, then we shall identify β with the function from $H(M_1)$ into $H(M_2)$ which maps every $h \in H(M_1)$ ino the unique element of $\beta(h)$.

DEFINITION. Let (α, β) be a reduced k-assignment of M_1 into M_2. (i) (α, β) is a *k-homomorphism* of M_1 onto M_2 iff both α and β are onto. In this case, we say that M_1 is k-homomorphic to M_2. (ii) (α, β) is a *state-behavior* k-assignment of M_1 into M_2 iff both α and β are one-to-one. In this case, we say that M_2 is a state-behavior k-realization of M_1. (iii) (α, β) is a k-isomorphism of M_1 onto M_2 iff both α and β are one-to-one and onto. In this case, we say that M_1 is k-isomorphic to M_2.

THEOREM 2.2. *M_2 is a reduced k-realization of M_1 iff there exists a function α from U^{M_1} into U^{M_2} and a k-matrix K such that for every $u \in U^{M_1}$, $P^{M_1}(u)K = KP^{M_2}(\alpha(u))$. Moreover,*

(a) *M_1 is k-homomorphic to M_2 iff α is onto and the columns of K are linearly independent;*

(b) *M_2 is a state-behavior k-realization of M_1 iff α is one-to-one and the rows of K are linearly independent; and*

(c) *M_1 is k-isomorphic to M_2 iff both α^{-1} and K^{-1} exist.*

Proof. The proof follows from Theorem 2.1 above or [8, Thm. 3.6].

The matrix K in the above theorem will be referred to as an associated matrix of β.

It follows from the above theorem that if α is the identity function, then k-submachine and k-split coincide with state-behavior k-realization and k-homomorphic, respectively.

Although the above definition of k-assignment differs from that given in [8], they coincide in the case of reduced k-assignment, which is the central issue of [8].

3. Properties of regular realizations.

Some basic properties of regular realization were derived in [8]. In this section, we shall present two additional characterizations of regular realization.

Notation. Let K be a real matrix. Then K^+ will denote a pseudoinverse [6] of K, i.e., $KK^+K = K$.

THEOREM 3.1. *M_2 is a regular (strong) realization of M_1 iff there exists a* QSSM *M_3 such that M_2 is a 1-split of M_3 and M_3 is a reduced (strong) realization of M_1.*

Proof. Let (α, β) be a regular (strong) assignment of M_1 into M_2. By Theorem 2.1, there exist 1-matrices K_1 and K_2 such that $\beta(h_1) = \{h_2 \in H(M_2): h_2K_2$

$= h_1K_1K_2\}$ for all $h_1 \in H(M_1)$. Without loss of generality, we may assume that the columns of K_2 are linearly independent. Let $u \in U^{M_1}$. Since $h_1K_1 \in \beta(h_1)$ for all $h_1 \in H(M_1)$, therefore $h_1K_1P^{M_2}(\alpha(u)) \in \beta[h_1P^{M_1}(u)]$ for all $h_1 \in H(M_1)$. This implies that $K_1P^{M_2}(\alpha(u))K_2 = P^{M_1}(u)K_1K_2$. On the other hand, if $gK_2 = 0$, then $gP^{M_2}(\alpha(u))K_2 = 0$. Therefore there exists a 1-matrix $P(\alpha(u))$ such that $P^{M_2}(\alpha(u))K_2 = K_2P(\alpha(u))$. Moreover, $P^{M_1}(u)K_1K_2 = K_1P^{M_2}(\alpha(u))K_2 = K_1K_2P(\alpha(u))$. Let M_3 be the QSSM where $U^{M_3} = \alpha(U^{M_1})$ and $P^{M_3}(\alpha(u)) = P(\alpha(u))$ for all $u \in U^{M_1}$. It follows from the foregoing discussions and Theorem 2.2 that M_2 is a 1-split of M_3, and M_3 is a reduced realization of M_1. If M_2 is a regular strong realization of M_1, then the rows of K_1K_2 are linearly independent. Therefore M_3 is a reduced strong realization of M_1.

Conversely, suppose there exists a QSSM M_3 such that M_2 is a 1-split of M_3 with associated matrix K_2, and M_3 is a reduced (strong) realization of M_1 with assignment (α, β_0) and associated matrix K_3. For each $h_1 \in H(M_1)$, define $\beta(h_1) = \{h_2 \in H(M_2){:}h_2K_2 = h_1K_3\}$. Since the columns of K_2 are linearly independent, $h_1K_3H_2^+ \in \beta(h_1)$ for all $h_1 \in H(M_1)$. Thus, β is a function from $H(M_1)$ into $H^*(M_2)$. With the aid of [8, Thm. 3.2], it is easy to verify that β is regular. Let $u \in U^{M_1}$, $h_1 \in H(M_1)$ and $h_2 \in \beta(h_1)$. Then $h_2K_2 = h_1K_3$. Therefore

$$h_2P^{M_2}(\alpha(u))K_2 = h_2K_2P^{M_3}(\alpha(u)) = h_1K_3P^{M_3}(\alpha(u)) = h_1P^{M_1}(u)K_3.$$

This implies that $h_2P^{M_2}(\alpha(u)) \in \beta[h_1P^{M_1}(u)]$. Hence (α, β) is a regular assignment of M_1 into M_2. If M_3 is a reduced strong realization of M_1, then the rows of K_3 are linearly independent. Therefore $\beta(h_1) \cap \beta(h'_1) = \varnothing$ for all $h_1 \neq h'_1$. Thus, M_2 is a regular strong realization of M_1. Q.E.D.

THEOREM 3.2. *M_2 is a regular strong k-realization of M_1 iff there exists a* QSSM *M_3 such that M_3 is a k-submachine of M_2 and M_3 is* 3-*homomorphic to M_1.*

Proof. Let (α, β) be a regular strong k-assignment of M_1 into M_2. Let K_3 be a k-matrix whose rows form a basis of the smallest subspace containing $\beta(H_3(M_1)) \cap H_k(M_2)$. Then $\beta(H(M_1))$ is the collection of all h_3K_3. Let u be an element of U^{M_1}. For every h_3, $h_3K_3 \in \beta(h_1)$ for some $h_1 \in H(M_1)$. Therefore, $h_3K_3P^{M_2}(\alpha(u)) \in \beta(h_1P^{M_1}(u))$. This implies that $h_3K_3P^{M_2}(\alpha(u))$ is of the form h'_3K_3. Thus there exists a 1-matrix $P(\alpha(u))$ such that $K_3P^{M_2}(\alpha(u)) = P(\alpha(u))K_3$. Let h_1^i be an element of $H(M_1)$ such that the ith row of K_3 belongs to $\beta(h_1^i)$, and let K_4 be a matrix whose ith row is h_1^i. By the construction of K_3, it follows that K_4 is a 3-matrix. Moreover, for every h_3, $h_3K_3 \in \beta(h_3K_4)$. Since for every $h_1 \in H(M_1)$, $\beta(h_1) \neq \varnothing$, therefore h_1 is of the form h_3K_4. This implies that the columns of K_4 are linearly independent. Furthermore, for each h_3, $h_3P(\alpha(u))K_3 \in \beta(h_3P(\alpha(u))K_4)$. On the other hand, $h_3P(\alpha(u))K_3 = h_3K_3P^{M_2}(\alpha(u)) \in \beta(h_3K_4P^{M_1}(u))$. Since (α, β) is a strong assignment, therefore $P(\alpha(u))K_4 = K_4P^{M_1}(u)$. Let M_3 be the QSSM where $U^{M_3} = \alpha(U^{M_1})$ and $P^{M_3}(\alpha(u)) = P(\alpha(u))$ for all $u \in U^{M_1}$. It follows from the foregoing discussions and Theorem 2.2 that M_3 is a k-submachine of M_2, and M_3 is 3-homomorphic to M_1.

Conversely, suppose there exists QSSM M_3 such that M_3 is a k-submachine of M_2 with associated matrix K_3, and M_3 is 3-homomorphic to M_1 with assignment (α, β_0) and associated matrix K_4. For each $h_1 \in H(M_1)$, define $\beta(h_1) = \{h_2 \in H(M_2){:}h_2 = h_3K_3,\ h_1 = h_3K_4$ for some $h_3 \in H(M_3)\}$. Since the columns of K_4 are linearly independent, $h_1K_4^+K_3 \in \beta(h_1)$ for all $h_1 \in H(M_1)$.

Thus, β is a function from $H(M_1)$ into $H^*(M_2)$. With the aid of [8, Thm. 3.2], it is easy to verify that β is regular. Assume $u \in U^{M_1}$, $h_1 \in H(M_1)$ and $h_2 \in \beta(h_1)$. Then $h_2 = h_3K_3$ and $h_1 = h_3K_4$. Therefore

$$h_2P^{M_2}(\alpha(u)) = h_3K_3P^{M_2}(\alpha(u)) = h_3P^{M_3}(\alpha(u))K_3$$

and

$$h_3P^{M_3}(\alpha(u))K_4 = h_3K_4P^{M_1}(u) = h_1P^{M_1}(u).$$

Thus $h_2P^{M_2}(\alpha(u)) \in \beta[h_1P^{M_1}(u)]$. This shows that (α, β) is a regular assignment of M_1 into M_2. Let h_1 be an element of $H_k(M_1)$. Since K_4 is a 3-matrix, therefore there exists a k-matrix h_3 such that $h_3K_4 = h_1$. But K_3 is also a k-matrix; therefore h_3K_3 is a k-matrix. This shows that $\beta(h_1) \cap H_k(M_2) \neq \varnothing$ provided $h_1 \in H_k(M_1)$. Since K_4 is a 3-matrix, each row of K_3 belongs to $\beta(H_3(M_1))$. Moreover, since each element of $\beta(H(M_1))$ is of the form h_3K_3 for $h_3 \in H(M_3)$, and since K_3 is a k-matrix, therefore $\beta(H(M_1))$ is the affine span of $\beta(H_3(M_1)) \cap H_k(M_2)$. Lastly, it is easy to verify that $\beta(h_1) \cap \beta(h_1') = \varnothing$ for all $h_1 \neq h_1'$, since the rows of K_3 are linearly independent. This shows that (α, β) is a regular strong k-assignment of M_1 into M_2. Q.E.D.

Remark. If M_1 and M_2 are 3-QSSM's, i.e., deterministic state-machines, and $k = 3$, then the M_3 in the above theorem can always be chosen to be a 3-QSSM. Thus, in this case, the above theorem reduces to [4, Thm. 1.6].

Observe that Theorem 3.1 holds only for $k = 1$, while Theorem 3.2 holds for $k = 1, 2$ *and* 3.

COROLLARY 3.3. *M_2 is a regular strong k-realization of M_1 with assignment (α, β), where α is the identity function, iff there exists a* QSMM *M_3 such that M_3 is a k-submachine of M_2 and M_3 is a* 3*-split of M_1.*

COROLLARY 3.4. *M_2 is a regular strong realization of M_1 iff there exists a* QSSM *M_3 such that M_3 is a* 1*-submachine of M_1 and M_3 is* 1*-homomorphic to M_1.*

4. Cascade decomposition.

Two types of decompositions appeared in the literature, namely, quasi-series decomposition [1] and strong decomposition [3]. The former type of decomposition was considered in [8]. In the present paper, we shall consider the latter type of decomposition, which we shall rename cascade decomposition. Although most of the results given in [8] can be strengthened by adopting cascade decomposition, only those which have direct bearing to the present work will be examined below.

Notation. Let $P_1 = (a_{ij})$ be an $m \times n$ real matrix and $P_2 = (A_{ij})$ an $m \times n$ matrix whose entries A_{ij} are real matrices of order $r \times s$. Then $P_1 \otimes P_2$ is an $mr \times ns$ matrix whose $((i, k), (j, l))$th entry is $a_{ij}A_{ij}(k, l)$, where $A_{ij}(k, l)$ is the (k, l)-th entry of the matrix A_{ij}.

DEFINITION. The *cascade connection* of two QSSM's M_1 and M_2, for which $U^{M_2} = S \times S \times U^{M_1}$, where $S = \{1, 2, \cdots, |M_1|\}$, is the QSSM M_3, where $U^{M_3} = U^{M_1}$ and for every $u \in U^{M_3}$, $P^{M_3}(u) = P^{M_1}(u) \otimes P(u)$, where $P(u)$ is a matrix whose (i,j)th entry is $P^{M_2}(i, j, u)$. If $P^{M_2}(i, j, u)$ is independent of j, then M_3 is a *quasi-series connection* of M_1 and M_2. If $P^{M_2}(i, j, u)$ is independent of both i and j, then M_3 is a *parallel connection* of M_1 and M_2.

DEFINITION. A *partition* of M is a family π of mutually disjoint nonempty subsets of $H(M)$ whose union is equal to $H(M)$. These subsets will be called *blocks* of π. Each partition π of M defines an equivalence relation mod π, where $h_1 \equiv h_2 \pmod{\pi}$ iff h_1 and h_2 belong to the same block of π.

DEFINITION. Let π be a partition of M. (i) π is a *SP-partition* of M iff $h_1 \equiv h_2 \pmod{\pi}$ implies $h_1 P^M(u) \equiv h_2 P^M(u) \pmod{\pi}$ for all $u \in U^M$. (ii) π is *regular* iff $h_i \in B_i \in \pi$, $i = 0, 1, 2$, and $h_0 = c_1 h_1 + c_2 h_2$ implies $B_0 = c_1 B_1 + c_2 B_2$.

THEOREM 4.1. *Let π be a partition of M. The following statements are equivalent:*

(a) *π is regular;*

(b) *there exists a subspace $W \subseteq \bar{H}_0(M)$ such that $h_1 \equiv h_2 \pmod{\pi}$ iff $h_1 - h_2 \in W$; and*

(c) *there exists a 1-matrix K such that $h_1 \equiv h_2$ (mod π) iff $h_1 K = h_2 K$.*

Proof. The proof is similar to [8, Thm, 4.2].

The subspace W and the matrix K in the above theorem will be referred to as the associated subspace and associated matrix, respectively.

DEFINITION. A *regular k-partition* of M is a regular partition of M associated with a k-matrix.

THEOREM 4.2. *There exists a regular k-SP-partition of M iff there exists a k-matrix K and for every $u \in U^M$, there exists a 1-matrix $P(u)$ such that $P^M(u)K = KP(u)$.*

Proof. The proof is similar to [8, Thm. 4.5].

In the rest of the section, we shall present a necessary and sufficient condition for a QSSM to admit a cascade decomposition of its state-behavior. In order to establish the condition, we have to show that every k-matrix with linearly independent columns can be extended in a certain way to a k-matrix with linearly independent rows.

Notation. E_{rs} is the $rs \times s$ 3-matrix whose (i,j)th entry is 1 iff $i = r(j-1) + t$, where $1 \leqq t < r$. Thus, for example, E_{23} is the 6×3 matrix

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

LEMMA 4.3. *Let K_0 be an $m \times n$ k-matrix whose columns are linearly independent. There exists an integer $t \leqq m - n + 1$ and an $m \times nt$ k-matrix K such that $KE_{tn} = K_0$ and the rows of K are linearly independent.*

Proof. If $m = n$, then there is nothing to prove. Therefore, we shall assume that $m > n$. Let E^m be the m-dimensional Euclidean space whose vectors are represented by column matrices, and let $\Gamma = \{e_1, e_2, \cdots, e_m\}$ be the standard basis of E^m. For every vector $g \in E^m$, we say that g is *commensurate* with $e_i \in \Gamma$ iff the ith row of g is not zero. Since $m > n$, there exists an $n \times (m - n)$ matrix L such that each column of L is a member of Γ, and the columns of K_0 and L together span E^m. Let Φ be a "partition" of the columns of L into n disjoint blocks (some blocks may be empty) such that a column e of L is contained in block j implies that the jth column of K_0 is commensurate with e. Since K_0 is a 1-matrix, each column of L must be contained in at least one block of Φ. Let t_j be the number of elements in the jth block of Φ, and let $t - 1$ be the largest of all such t_j. Clearly, $t \leqq m - n + 1$. For each $j = 1, 2, \cdots, n$, let K_j be an $m \times t$ 3-matrix whose first t_j columns are the elements of the jth block of Φ and all remaining columns, except the last column, are 0. Let $K = K_0 \otimes K'$ where $K' = (A_{ij})$ and A_{ij} is the ith row of K_j. Since K_j is a 3-matrix, it follows that K is a k-matrix and $KE_{tn} = K_0$. It remains to show that the rows of K are linearly independent. It follows from the construction of K that each column of L is a scalar multiple of some column of K. Moreover, since $KE_{tn} = K_0$, therefore each column of K_0 is a linear combination of columns of K. Thus, the columns of K span E^m. This shows that the rows of K are linearly independent. Q.E.D.

The matrix K will be called an *extension* of K_0.

Remark. If K_0 is a 3-matrix, then K_0 is a matrix associated with a regular 3-partition π of some M. But π, being a regular 3-partition, corresponds to a partition π' of the states of M. In this case, the t obtained in the above theorem reduces to the number of elements of the largest block of π'.

THEOREM 4.4. *There exist M_1 and M_2 such that the cascade connection of M_1 and M_2 is a state-behavior k-realization of M iff M has a regular k-SP-partition.*

Proof. Suppose the cascade connection of M_1 and M_2 is a state-behavior k-realization of M with assignment (α, β) and associated matrix K. Then for every $u \in U^M$, $P^M(u)K = K(P^{M_1}(\alpha(u)) \otimes P(\alpha(u)))$, where $P(\alpha(u)) = (P^{M_2}(i, j, \alpha(u)))$. Thus $P^M(u)KE_{mn} = KE_{mn}P^{M_1}(\alpha(u))$, where $m = |M_2|$ and $n = |M_1|$. Since K is a k-matrix, so is KE_{mn}. By Theorem 4.2, M has a regular k-SP-partition.

Conversely, suppose M has a regular k-SP-partition, and let K_0 be the associated matrix. Let u be an element of U^M. By Theorem 3.2, there exists a 1-matrix $P_1(u)$ such that $P^M(u)K_0 = K_0P_1(u)$. Let K be an $m \times nt$ matrix which is an extension of K_0. Let

$$P_3(u) = K^+P^M(u)K + E_{tn}P_1(u)E_{tn}^+ - K^+P^M(u)K_0E_{tn}^+.$$

Since the rows of K are linearly independent, therefore KK^+ is the identity matrix, and

$$KP_3(u) = P^M(u)K + K_0P_1(u)E_{tn}^+ - P^M(u)K_0E_{tn}^+ = P^M(u)K.$$

Moreover, since the columns of E_{tn} are linearly independent, therefore $E_{tn}^+E_{tn}$ is the identity matrix, and

$$P_3(u)E_{tn} = K^+P^M(u)K_0 + E_{tn}P_1(u) - K^+P^M(u)K_0 = E_{tn}P_1(u).$$

This implies that $P_3(u) = P_1(u) \otimes P_2(u)$ for some $P_2(u)$. Let $P_2(u) = (A_{ij}(u))$, where $A_{ij}(u)$ are $t \times t$ 1-matrices. Define M_1 and M_2 where $U^{M_1} = U^M$, $U^{M_2} = S \times S \times U^M$, $S = \{1, 2, \cdots, n\}$, and for every $u \in U^M$, $i, j \in S$, $P^{M_1}(u) = P_1(u)$ and $P^{M_2}(i, j, u) = A_{ij}(u)$. It follows from the foregoing discussions that the cascade connection of M_1 and M_2 is a state-behavior k-realization of M.

It follows from the above proof that M_1 and M_2 can be chosen to have fewer states than M iff M has a nontrivial k-SP-partition. The trivial partitions of M are: the partition which has only one block and the partition where each block contains exactly one element.

COROLLARY 4.5. *Let M be a k-QSSM. There exist k-QSSM's M_1 and M_2 such that the cascade connection of M_1 and M_2 is a state-behavior 3-realization of M iff M has a regular 3-SP-partition.*

For $k = 2$, a proof for the sufficiency part of the above corollary can be found in [3].

Remark. The above corollary does not hold, in general, if 3 is replaced by k. This is due to the fact that the $P_3(u)$ given in the proof of Theorem 4.4 is not necessarily a k-matrix.

We shall conclude this section with an example illustrating the procedures given in the proofs of Lemma 4.3 and Theorem 4.4.

Let $U = \{u_1, u_2\}$,

$$P^M(u_1) = \begin{pmatrix} \frac{1}{6} & \frac{3}{4} & 0 & \frac{1}{12} \\ 0 & 1 & 0 & 0 \\ 0 & \frac{7}{9} & 0 & \frac{2}{9} \\ 0 & \frac{7}{12} & \frac{1}{4} & \frac{1}{6} \end{pmatrix}$$

and

$$P^M(u_2) = \begin{pmatrix} 0 & \frac{1}{4} & 0 & \frac{3}{4} \\ 1 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Let

$$K_0 = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ 0 & 1 \\ \frac{2}{3} & \frac{1}{3} \\ 1 & 0 \end{pmatrix},$$

and let π be the regular partition of M with associated matrix K_0. Since

$$P^M(u_1)K_0 = K_0\begin{pmatrix} \frac{1}{3} & \frac{2}{3} \\ 0 & 1 \end{pmatrix}$$

and

$$P^M(u_2)K_0 = K_0\begin{pmatrix} 1 & 0 \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix},$$

therefore, it follows from Theorem 3.2 that π is an SP-partition of M. We shall now construct an extension K of K_0 following the procedure given in the proof of Lemma 4.3.

Let

$$L = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}.$$

Clearly, the columns of K_0 and L together span E^4. Since the first and second columns of L are commensurate with the first and second columns of K_0, respectively, therefore, a possible choice of Φ is to take the first and second blocks of Φ to contain the first and second column of L, respectively. In this case, $t_1 = t_2 = 1$, $t = 2$,

$$K_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix} \quad \text{and} \quad K_2 = \begin{pmatrix} 0 & 1 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

The last column of K_1 and K_2 are uniquely determined since both matrices are 1-matrices. Thus

$$K' = (K_1|K_2) = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}.$$

Therefore

$$K = K_0 \otimes K' = \begin{pmatrix} \frac{1}{2}(1\ 0) & \frac{1}{2}(0\ 1) \\ 0(0\ 1) & 1(0\ 1) \\ \frac{2}{3}(0\ 1) & \frac{1}{3}(1\ 0) \\ 1(0\ 1) & 0(0\ 1) \end{pmatrix} = \begin{pmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 1 \\ 0 & \frac{2}{3} & \frac{1}{3} & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}.$$

It is easy to verify that K is an extension of K_0.

Next, we shall construct M_1 and M_2 following the procedure given in the proof of Theorem 4.4. Clearly,

$$K^+ = \begin{pmatrix} 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 3 & -2 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

is a pseudoinverse of K, and

$$E_{22}^{+} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

is a pseudoinverse of E_{22}. Moreover, both K^{+} and E_{22}^{+} are 1-matrices. Thus,

$$P_3(u_1) = \begin{pmatrix} \frac{1}{6} & \frac{1}{6} & 0 & \frac{2}{3} \\ 0 & \frac{1}{3} & \frac{1}{12} & \frac{7}{12} \\ 0 & 0 & -\frac{1}{6} & \frac{7}{6} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

and

$$P_3(u_2) = \begin{pmatrix} -\frac{1}{2} & \frac{3}{2} & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} \end{pmatrix}.$$

The matrices $P_3(u_1)$ and $P_3(u_2)$ are obtained by using the equation $P_3(u) = K^{+}P^{M}(u)K + E_{22}P_1(u)E_{22}^{+} - K^{+}P^{M}(u)K_0E_{22}^{+}$ with

$$P_1(u_1) = \begin{pmatrix} \frac{1}{3} & \frac{2}{3} \\ 0 & 1 \end{pmatrix} \quad \text{and} \quad P_1(u_2) = \begin{pmatrix} 1 & 0 \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix}.$$

Hence the QSSM M_1 is defined by $P^{M_1}(u_1) = P_1(u_1)$ and $P^{M_2}(u_2) = P_1(u_2)$. Moreover, the QSSM M_2 is defined by:

$$P^{M_2}(1, 1, u_1) = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ 0 & 1 \end{pmatrix}, \qquad P^{M_2}(1, 2, u_1) = \begin{pmatrix} 0 & 1 \\ \frac{1}{8} & \frac{7}{8} \end{pmatrix},$$

$$P^{M_2}(2, 1, u_1) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \qquad P^{M_2}(2, 2, u_1) = \begin{pmatrix} -\frac{1}{6} & \frac{7}{6} \\ 0 & 1 \end{pmatrix},$$

$$P^{M_2}(1, 1, u_2) = \begin{pmatrix} -\frac{1}{2} & \frac{3}{2} \\ 0 & 1 \end{pmatrix}, \qquad P^{M_2}(1, 2, u_2) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},$$

$$P^{M_2}(2, 1, u_2) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad \text{and} \quad P^{M_2}(2, 2, u_2) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

5. Covers. The concept of covers or set systems is one of the most powerful tools in the study of decomposition of deterministic machines [4]. In this section, we shall generalize this concept and show how it could be used in the decomposition of stochastic machines.

DEFINITION. A *cover* μ of M is a collection of subsets of $H(M)$. The subsets in μ will also be called *blocks* of μ. If there exists a finite collection $\mathscr{C}_\mu$ of subspaces of $\bar{H}_0(M)$ such that each block of μ is of the form $h + W$, where $h \in H(M)$ and $W \in \mathscr{C}_\mu$, then μ is a *regular* cover of M. If, in addition, every W in $\mathscr{C}_\mu$ is the subspace associated with some regular k-partition of M, then μ is a *regular k-cover* of M.

DEFINITION. Let μ be a cover of M. μ is an SP-*cover* of M iff for every $B \in \mu$, there exists a $B' \in \mu$ such that $u \in U^M$ and $h \in B$ implies $hP^M(u) \in B'$.

Clearly, every regular k-SP-partition of M is a regular k-SP-cover of M. Moreover, we have the following theorem

THEOREM 5.1. *If μ is a regular* SP-*cover of M and $\mathscr{C}_\mu = \{W_1, W_2, \cdots, W_n\}$, then the regular partition π of M with associated subspace $W_0 = W_1 + W_2 + \cdots + W_n$ is a regular* SP-*partition of M.*

Proof. Since μ is a regular SP-partition of M, therefore, for every $W \in \mathscr{C}_\mu$, there exists a $W' \in \mathscr{C}_\mu$ such that $u \in U^M$ and $g \in W$ implies $gP^M(u) \in W'$. This implies that $g_0 P^M(u) \in W_0$ for all $g_0 \in W_0$ and $u \in U^M$. Thus π is a regular SP-partition of M.

THEOREM 5.2. *If μ is a regular k-*SP-*cover of M, then there exist* QSSM's *M_0, M_1 and M_2 such that M_0 is a 3-split of M, M_0 is a k-submachine of the cascade connection of M_1 and M_2, and $|M_2| \leqq 1 + \dim \mathscr{C}_\mu \leqq |M|$, where* dim $\mathscr{C}_\mu$ *is the dimension of the subspace in $\mathscr{C}_\mu$ with the largest dimension.*

Proof. Let $\mathscr{C}_\mu = \{W_1, W_2, \cdots, W_n\}$ and for $i = 1, 2, \cdots, n$, let L_i be a matrix whose rows constitute a basis for W_i. Since μ is a regular SP-cover of M, therefore, for each $W_i \in \mu$, there exists $W_{j_i} \in \mu$ such that for every $u \in U^M$ and $g_i \in W_i$, $g_i P^M(u) \in W_{j_i}$. This implies that for every $i = 1, 2, \cdots, n$, and $u \in U^M$, there exists $Q_i(u)$ such that $L_i P^M(u) = Q_i(u) L_{j_i}$. For each $u \in U^M$, let $Q(u) = (Q_{ij}(u))$, where $Q_{ij}(u) = Q_i(u)$ if $j = j_i$, and $Q_{ij}(u) = 0$ otherwise. Let $K_1 = (K'_{ij})$, where for $i = 1, 2, \cdots, n$, and $j = 1$, K'_{ij} is the identity matrix of order $|M|$. For each $u \in U^M$, let $P_0(u) = (P^0_{ij}(u))$, where for $i, j = 1, 2, \cdots, n$, $P^0_{ij}(u) = P^M(u)$ if $j = j_i$, and $P^0_{ij}(u) = 0$ otherwise. Let M_0 be the QSSM where $U^{M_0} = U^M$ and for each $u \in U^M$, $P^{M_0}(u) = P_0(u)$. Clearly, $P^{M_0}(u)K_1 = K_1 P^M(u)$ for all $u \in U^M$. Thus M_0 is a 3-split of M since K_1 is a 3-matrix with linearly independent columns. Let $L = (L_{ij})$, where for $i, j = 1, 2, \cdots, n$, $L_{ij} = L_i$ if $i = j$, and $L_{ij} = 0$ otherwise. It is easy to verify that $LP^{M_0}(u) = Q(u)L$ for all $u \in U^M$. Let W_0 be the subspace spanned by the rows of L, and let π be the regular partition of M_0 with associated subspace W_0. Then π is a regular SP-partition of M_0. Moreover, for $i = 1, 2, \cdots, n$, let K^i be the k-matrix associated with π_i, the regular k-partition of M with associated subspace W_i. Let $K_2 = (K^2_{ij})$, where for $i, j = 1, 2, \cdots, n$, $K^2_{ij} = K^i$ if $i = j$, and $K^2_{ij} = 0$ otherwise. Clearly, K_2 is a k-matrix associated with π. Thus, π is a regular k-SP-partition of M_0. It follows from the proofs of Lemma 4.3 and Theorem 4.4 that there exists QSSM's M_1 and M_2 such that $|M_2| \leqq 1 + \dim \mathscr{C}_\mu \leqq |M|$ and M_0 is a k-submachine of the cascade connection of M_1 and M_2.

Remark. In the above theorem, if $\bar{H}_0(M)$ is not in $\mathscr{C}_\mu$, then $|M_2| < |M|$. Moreover, $|M_1| = n$ and hence $|M_1|$ could be reduced by (i) omitting subspaces in $\mathscr{C}_\mu$ which are subsets of other subspaces in $\mathscr{C}_\mu$ if any, and/or (ii) replacing any subset of $\mathscr{C}_\mu$ by its direct sum. However, in case (ii), $|M_2|$ may be increased.

THEOREM 5.3. *If μ is a regular k-*SP-*cover of M, then there exist* QSSM's *M_1 and M_2 such that $|M_2| \leqq 1 + \dim \mathscr{C}_\mu \leqq |M|$ and the cascade connection of M_1 and M_2 is a regular strong k-realization of M.*

Proof. The proof follows from Theorems 3.2 and 5.2.

REFERENCES

[1] G. C. BACON, *The decomposition of stochastic automata*, Information and Control, 7 (1964), pp. 320–329.

[2] S. FUJIMOTO AND T. FUKAO, *The decomposition of probabilistic automata*, Denki Shikenjo Iho, 30 (1966), pp. 688–698.

[3] S. E. Gelenke, *On the loop-free decomposition of stochastic finite-state systems*, Information and Control, 17 (1970), pp. 474–484.

[4] J. HARTMANIS AND R. E. STERNS, *Algebraic Structure Theory of Sequential Machines*, Prentice-Hall, Englewood Cliffs, N.J., 1966.

[5] A. PAZ, *Introduction to Probabilistic Automata*, Academic Press, New York, 1971.

[6] C. A. RHODE, *Some results on generalized inverses*, SIAM Rev., 8 (1966), pp. 201–205.

[7] E. S. SANTOS, *First and second covering problems of quasi stochastic systems*, Information and Control, 20 (1972), pp. 20–37.

[8] ———, *Algebraic structure theory of stochastic machines*, Math. Systems Theory, 6 (1972), pp. 243–262.

SIAM J. COMPUT.
Vol. 4, No. 2, June 1975

COMPUTATIONAL COMPLEXITY AND NUMERICAL STABILITY*

WEBB MILLER†

Abstract. Limiting consideration to algorithms satisfying various numerical stability requirements may change lower bounds for computational complexity and/or make lower bounds easier to prove. We will show that under a sufficiently strong restriction upon numerical stability, any algorithm for multiplying two $n \times n$ matrices using only $+$, $-$ and $\times$ requires at least n^3 multiplications. We conclude with a survey of results concerning the numerical stability of several algorithms which have been considered by complexity theorists.

Key words. complexity, roundoff error, bilinear form, matrix multiplication

1. Introduction. Proofs of lower bounds for the computational complexity of a given problem must generally consider only a few measurements of cost and ignore the others. For example, a discussion of the complexity of the problem of sorting a list of items may count only the number of comparisons, disregarding issues like noncomparison operations, storage requirements and programming simplicity.

When the problem under consideration involves real (i.e., floating-point) arithmetic, then another factor is relevant, namely, the propagation of rounding errors. In a recent survey of arithmetic complexity, Borodin [3, p. 174] remarks:

But eventually we will have to develop results which simultaneously talk about arithmetic costs, and the "robustness" or "stability" of the algorithm. Perhaps if we were more formal about the numerical properties of an algorithm, then it might be easier to produce non trival lower bounds.

Of course, many people have given simultaneous consideration to complexity and stability. In particular, numerical analysts are daily faced with the trade-off stability and the operation count. Moreover, investigations have been made of the effects of rounding errors upon several algorithms of interest to complexity theorists (for a survey, see § 6).

Rather, the first sentence quoted from Borodin seems to suggest the possible existence of problems for which the fastest algorithms must be less than optimal with regard to stability. One motivation for this paper is to present several such examples. Thus we are interested in the effects upon complexity lower bounds of various stability requirements (essentially the opposite problem of maximizing stability subject to complexity constraints is investigated by Babuska [1] and Viten'ko [20]).

However, our primary motivation is to be found in Borodin's *second* sentence. When we limit consideration to all programs (in a given language) which evaluate a fixed function and *which satisfy a certain stability requirement*, it is entirely possible that the fastest programs are excluded. More intriguing to us is the possibility that the remaining programs have a simple structure which makes finding a program that is optimal with respect to some measurement of cost substantially easier than if arbitrary (and less stable) programs are allowed to compete.

* Received by the editors August 21, 1973, and in final revised form July 5, 1974.

† Computer Science Department, Pennsylvania State University, University Park, Pennsylvania 16802. This work was supported in part by the National Science Foundation under Grant GJ-42968.

Lower bounds for arithmetic complexity are often difficult to verify. In particular, the problem of determining the minimum number of multiplications needed to evaluate a bilinear form is equivalent to an ancient and seemingly intractable problem of ranking tensors (for a discussion and further references, see Dobkin [8], especially § 5). It seems reasonable to consider subcases, e.g., stability requirements, which are natural in the complexity framework (and perhaps not so natural in, e.g., the tensor framework) in the hope that they provide a handle on lower bound proofs. (However, it should be noted that lower bound proofs sometimes apply to arbitrary fields, whereas stability restrictions are only natural over the fields of real or complex numbers.)

Here we have focused on the evaluation of systems of bilinear forms by programs which apply only the operations $+$, $-$ and $\times$. For simplicity, we have not allowed constants, though almost everything carries over with minor modifications. One of our results is that if only such programs meeting a very restrictive stability requirement are considered, then n^3 multiplications are needed to find the product of two $n \times n$ matrices. Under a somewhat relaxed stability assumption, we will show that computing the product of a 2×2 matrix and a $2 \times n$ matrix requires at least $\lceil 7n/2 \rceil$ multiplications. In each case, if the stability requirement is dropped, then faster algorithms can be found and tight lower bounds seem to become harder to verify (they are not yet known).

The concepts of numerical stability which we employ are closely related to those currently used by experts in roundoff analysis (e.g., Wilkinson [21], [22]). They are idealized in at least two respects. First, they use very few of the actual properties of floating-point arithmetic. This creates a tendency for pessimistic results in that algorithms may be more stable than we can prove (see the remarks by Kahan [12, pp. 1232–1234] on Viten'ko [20]). Second, we consider only the first-order effects of rounding errors, so algorithms may be less stable than results like ours suggest. (In § 5 we will detail several more reasons why the practical significance of the results given here is not especially great.)

We hope that this paper offers supporting evidence for Borodin's cited belief and for our conviction that numerical analysts and complexity theorists can benefit from an exchange of ideas.

2. Numerical stability of polynomial programs. A *polynomial program* is a sequence of instructions of the form

$$V \leftarrow W \# X \tag{2.1}$$

where $\#$ is one of $+$, $-$ or $\times$. For simplicity, we require that each operand, W or X, is a variable (initial or defined).

We can think of the program as specifying a sequence of floating-point operations to be performed on the data $\mathbf{d} = (d_1, \cdots, d_n)$. In many contexts it is helpful to take the alternative point of view that the program specifies symbolic operations to be performed on multivariate polynomials in the initial variables $\mathbf{d}$. Any resulting polynomial $V(\mathbf{d})$ has integer coefficients and no constant term.

Let V be a defined variable of such a program, i.e., V appears on the left of a unique instruction (2.1). If we omit the instruction defining V and consider V an initial variable, then any variable Z can be considered a polynomial $Z(\mathbf{d}, V)$.

Define

$$\Delta Z_V(\mathbf{d}) \equiv \frac{\partial Z}{\partial V}(\mathbf{d}, V(\mathbf{d})) \cdot V(\mathbf{d}).$$

The polynomial $\Delta Z_V(\mathbf{d})$ measures the sensitivity of the numerical evaluation of $Z(\mathbf{d})$ to a floating-point rounding error committed in the operation producing $V(\mathbf{d})$. For suppose that the evaluation is performed exactly except that $V(\mathbf{d})(1 + \delta)$ is used in place of $V(\mathbf{d})$. The induced error is

$$Z(\mathbf{d}, V(\mathbf{d})(1 + \delta)) - Z(\mathbf{d}) = Z(\mathbf{d}, V(\mathbf{d})(1 + \delta)) - Z(\mathbf{d}, V(\mathbf{d})) \approx \delta \cdot \Delta Z_V(\mathbf{d})$$

for small δ.

For example, let $\mathbf{d} = (a_1, a_2, b_1, b_2)$ and consider

$$\begin{aligned} T &\leftarrow a_1 \times a_2, \\ U &\leftarrow b_1 \times b_2, \\ V &\leftarrow a_1 + b_2, \\ W &\leftarrow a_2 + b_1, \\ X &\leftarrow V \times W, \\ Y &\leftarrow X - T, \\ Z &\leftarrow Y - U. \end{aligned} \tag{2.2}$$

One easily computes

$$\begin{aligned} Z(\mathbf{d}) &\equiv a_1 b_1 + a_2 b_2, \\ Z(\mathbf{d}, W) &\equiv (a_1 + b_2)W - a_1 a_2 - b_1 b_2, \\ \frac{\partial Z}{\partial W}(\mathbf{d}, W) &\equiv a_1 + b_2, \\ \Delta Z_W(\mathbf{d}) &\equiv (a_1 + b_2)(a_2 + b_1). \end{aligned}$$

Stability conditions can sometimes be interpreted as restricting the form of certain polynomials $\Delta Z_V(\mathbf{d})$. If

$$\frac{\partial Z}{\partial V}(\mathbf{d}, V(\mathbf{d})) \not\equiv 0, \tag{2.3}$$

then the form of $V(\mathbf{d})$ is also restricted since

$$V(\mathbf{d}) \cdot \frac{\partial Z}{\partial V}(\mathbf{d}, V(\mathbf{d})) = \Delta Z_V(\mathbf{d}).$$

Our goal in the remainder of this section is to exhibit a useful condition which guarantees (2.3).

Let Z be a defined variable of a polynomial program. Consider the following process for marking certain instructions of the program. If Z is defined by an addition or subtraction, then mark that instruction. If an instruction

$$V \leftarrow X \# Y, \quad \text{where } \# \text{ is } + \text{ or } -,$$

is marked and if X and/or Y is defined by $+$ or $-$, then mark those instructions defining X and/or Y.

Now delete any unmarked instructions. The set $I(Z)$ of initial (i.e., undefined) variables of the resulting polynomial program contains only variables which were originally either initial or defined by multiplication (take $I(Z) = \{Z\}$ if Z is defined by multiplication).

Clearly Z is a linear combination of the U in $I(Z)$. Specifically, there exists a unique integer $i_Z(U)$ for each U in $I(Z)$ such that in the resulting program

$$Z \equiv \sum_I i_Z(U) \cdot U,$$

the dependence of Z on $I(Z)$ is essentially the same in the original program. In particular,

$$Z(\mathbf{d}) \equiv \sum_I i_Z(U) \cdot U(\mathbf{d}), \tag{2.4}$$

and for any V in $I(Z)$, we have

$$Z(\mathbf{d}, V) \equiv \sum_I i_Z(U) \cdot U(\mathbf{d}, V). \tag{2.5}$$

For an example of these notions, recall (2.2). We find that $I(Z) = \{T, U, X\}$ and

$$1 = i_Z(X) = -i_Z(T) = -i_Z(U).$$

THEOREM 2.1. *If V in $I(Z)$ is defined by a multiplication and if $i_Z(V) \neq 0$, then*

$$\frac{\partial Z}{\partial V}(\mathbf{d}, V(\mathbf{d})) \not\equiv 0.$$

Proof. If U in $I(Z)$ is originally defined by a multiplication, then U may depend on V in the sense that

$$\frac{\partial U}{\partial V}(\mathbf{d}, V(\mathbf{d})) \not\equiv 0.$$

The only other U in $I(Z)$ are originally initial variables and do not depend on V. From (2.5) we compute

$$\begin{aligned} \frac{\partial Z}{\partial V}(\mathbf{d}, V(\mathbf{d})) &= \sum_I i_Z(U) \cdot \frac{\partial U}{\partial V}(\mathbf{d}, V(\mathbf{d})) \\ &= i_Z(V) + \sum{}^{*} i_Z(U) \cdot \frac{\partial U}{\partial V}(\mathbf{d}, V(\mathbf{d})), \end{aligned} \tag{2.6}$$

where the sum $\sum^*$ is over all $U \neq V$ in $I(Z)$ which are defined by a multiplication $U \leftarrow X \times Y$. For such U we find

$$\frac{\partial U}{\partial V}(\mathbf{d}, V(\mathbf{d})) = \frac{\partial X}{\partial V}(\mathbf{d}, V(\mathbf{d})) \cdot Y(\mathbf{d}) + \frac{\partial Y}{\partial V}(\mathbf{d}, V(\mathbf{d})) \cdot X(\mathbf{d}).$$

Neither of these last two summands has a constant term. It follows that the constant term in (2.6) is $i_Z(V)$. This proves Theorem 2.1. $\square$

3. Stable evaluation of bilinear forms. Consider a polynomial program defining a variable B which evaluates a bilinear form

$$B(\mathbf{a},\mathbf{b}) \equiv \sum_{i=1}^{m} \sum_{j=1}^{n} \sigma_{ij} a_i b_j \tag{3.1}$$

Here the input has been partitioned as $\mathbf{d} = (\mathbf{a},\mathbf{b}) = (a_1, \cdots, a_m, b_1, \cdots, b_n)$ and the σ_{ij} are integer constants. In this section we will define and investigate four types of numerical stability which are applicable to such variables.

If f and g are real-valued functions of $(\mathbf{a},\mathbf{b})$, then

$$f(\mathbf{a},\mathbf{b}) = O(g(\mathbf{a},\mathbf{b}))$$

means that there exists a constant K such that

$$|f(\mathbf{a},\mathbf{b})| \leqq K \cdot |g(\mathbf{a},\mathbf{b})| \quad \text{for all } \mathbf{a},\mathbf{b}.$$

We also need the notation

$$|\mathbf{a}| = \max\{|a_i| : 1 \leqq i \leqq m\},$$
$$|\mathbf{b}| = \max\{|b_j| : 1 \leqq j \leqq n\},$$
$$|(\mathbf{a},\mathbf{b})|_B = \max\{|a_i b_j| : \sigma_{ij} \neq 0\}.$$

DEFINITION 1. The following four kinds of numerical stability are defined by the requirement that each defined variable V satisfies the corresponding equality.

(i) *Brent stability*:

$$\Delta B_V(\mathbf{a},\mathbf{b}) = O(|\mathbf{a}| \cdot |\mathbf{b}|).$$

(ii) *Restricted Brent stability*:

$$\Delta B_V(\mathbf{a},\mathbf{b}) = O(|(\mathbf{a},\mathbf{b})|_B).$$

(iii) *Weak stability*:

$$\Delta B_V(\mathbf{a},\mathbf{b}) = O\left(|\mathbf{a}| \cdot \sum_{i=1}^{m} \left|\frac{\partial B}{\partial a_i}(\mathbf{a},\mathbf{b})\right| + |\mathbf{b}| \cdot \sum_{j=1}^{n} \left|\frac{\partial B}{\partial b_j}(\mathbf{a},\mathbf{b})\right|\right).$$

(iv) *Strong stability*:

$$\Delta B_V(\mathbf{a},\mathbf{b}) = O\left(\sum_{i=1}^{m} \left|a_i \cdot \frac{\partial B}{\partial a_i}(\mathbf{a},\mathbf{b})\right| + \sum_{j=1}^{n} \left|b_j \cdot \frac{\partial B}{\partial b_j}(\mathbf{a},\mathbf{b})\right|\right).$$

Definition 1(i) is an idealization of a notion studied by Brent [4], [5]. It is idealized in that "second order" effects of rounding errors are obliterated by the differentiation in the definition of ΔB_V.

Definitions 1(iii) and 1(iv) are formalizations of ideas from "backward error analysis" (see Miller [16]). Their intuitive thrust is that the result computed with roundoff error is the exact result for slightly altered data. Here "slightly altered" means (in the case of strong stability) that each coordinate is accurate to within a few rounding errors or (in the case of weak stability) that the error in each a_i (respectively, b_j) is small compared to $|\mathbf{a}|$ (respectively, $|\mathbf{b}|$).

Of course B can have, e.g., Brent stability only in the context of a particular program. We will often omit specific mention of the underlying program.

Notice that (restricted) Brent stability is meaningful only for the evaluation of bilinear forms. However, strong stability and weak stability are meaningful when discussing *any rational program*, and we include them here so we can locate the Brent notions in a more general hierarchy of stability requirements.

PROPOSITION 3.1. *The following implications hold among the notions of Definition* 1: (iv) $\Rightarrow$ (ii) $\Rightarrow$ (i); (iv) $\Rightarrow$ (iii) $\Rightarrow$ (i).

Verifying Proposition 3.1 is easy. For instance, to show that restricted Brent stability implies Brent stability, one need only note that

$$|(\mathbf{a}, \mathbf{b})|_B = O(|\mathbf{a}| \cdot |\mathbf{b}|).$$

It is also easy to give conditions under which two of the stability requirements coalesce. We will use the following.

PROPOSITION 3.2. *Suppose* (3.1) *is a* permutation *bilinear form, i.e., suppose that whenever $\sigma_{ij} \neq 0$ and $\sigma_{IJ} \neq 0$, then either*

(i) *$i = I$ and $j = J$, or*

(ii) *$i \neq I$ and $j \neq J$.*

Then strong stability is equivalent to restricted Brent stability.

Proof. If $\sigma_{ij} \neq 0$, then $(\partial B/\partial a_i)(\mathbf{a}, \mathbf{b}) = \sigma_{ij} b_j$. Hence

$$|(\mathbf{a}, \mathbf{b})|_B = O\left(\sum \left| a_i \cdot \frac{\partial B}{\partial a_i}(\mathbf{a}, \mathbf{b}) \right| \right).$$

This shows that restricted Brent stability implies strong stability. Proposition 3.1 does the rest. □

Restricted Brent stability does not, *in general*, imply strong stability. Consider $B(\mathbf{a}, \mathbf{b}) = a_1 b_1 + a_1 b_2 + a_2 b_1 + a_2 b_2$ and

$$\begin{aligned} &U \leftarrow a_1 \times b_1, \\ &V \leftarrow a_1 \times b_2, \\ &W \leftarrow a_2 \times b_1, \\ &X \leftarrow a_2 \times b_2, \\ &Y \leftarrow U + Y, \\ &Z \leftarrow Y + W, \\ &B \leftarrow Z + X. \end{aligned} \tag{3.2}$$

One easily computes $\Delta B_U(\mathbf{a}, \mathbf{b}) = a_1 b_1$. But if $a_1 = b_1 = 1 = -a_2 = -b_2$, then

$$|\mathbf{a}| \cdot \sum \left| \frac{\partial B}{\partial a_i}(\mathbf{a}, \mathbf{b}) \right| + |\mathbf{b}| \cdot \sum \left| \frac{\partial B}{\partial b_j}(\mathbf{a}, \mathbf{b}) \right| = 0.$$

It follows that the program does not have weak stability; hence it lacks strong stability. On the other hand, one easily sees that it possesses restricted Brent stability. Of course, this B is not a *permutation* bilinear form.

THEOREM 3.3. *Let V in $I(B)$ be defined by a multiplication, and suppose that $i_B(V) \neq 0$ and that $V(\mathbf{a}, \mathbf{b}) \not\equiv 0$. If B has Brent stability, then the definition of V*

must be of the form

$$L_1(\mathbf{a}) \times L_2(\mathbf{b}) \quad (\text{or } L_2(\mathbf{b}) \times L_1(\mathbf{a})),$$

where L_1 and L_2 are linear (e.g., $L_1(\mathbf{a}) = \sum \alpha_i a_i$ with integers α_i).

Proof. We will show that ΔB_V is bilinear. Since, by Theorem 2.1, V is a divisor of ΔB_V, it will then follow that V must be bilinear. Now the result is immediate.

To show that ΔB_V is bilinear, let us assume that it is not bilinear, then show that the stability condition

$$(3.3) \qquad \Delta B_V(\mathbf{a}, \mathbf{b}) = O(|\mathbf{a}| \cdot |\mathbf{b}|)$$

is violated. We proceed by cases.

First suppose ΔB_V has a term involving none of b_j (or none of the a_i), say,

$$\alpha a_1^{p_1} a_2^{p_2} \cdots a_m^{p_m} \quad \text{with } \alpha \neq 0.$$

Fixing $b_1 = b_2 = \cdots = b_n = 0$ and allowing $\mathbf{a}$ to vary, we see that $\Delta B_V(\mathbf{a}, \mathbf{0})$ is a nonzero polynomial in $\mathbf{a}$. Thus for some $(\mathbf{a}, \mathbf{0}) = (\mathbf{a}, \mathbf{b})$, we have $\Delta B_V(\mathbf{a}, \mathbf{b}) \neq 0 = |\mathbf{a}| \cdot |\mathbf{b}|$, violating (3.3).

The only other possibility is that ΔB_V has a term of degree greater than two. A simple argument shows that (3.3) is again violated. □

Theorem 3.3 shows that example (2.2), an instance of Winograd's method for inner products, does not possess Brent stability (see also Brent [4], [5]).

The next result shows that if we add to the hypotheses of Theorem 3.3 the assumption of *restricted* Brent stability, then any term appearing in V must appear in B. The intuitive reason is that otherwise, a term, $\eta_{ij} a_i b_j$, must cancel algebraically in a later $\pm$ operation, and corresponding numerical cancellation creates an error not $O(|(\mathbf{a}, \mathbf{b})|_B)$.

THEOREM 3.4. *Let V in $I(B)$ be defined by a multiplication, and suppose $i_B(V) \neq 0$. Write*

$$V(\mathbf{a}, \mathbf{b}) = \sum \sum \eta_{ij} a_i b_j.$$

If B has restricted Brent stability, and if $\eta_{ij} \neq 0$, then the coefficient σ_{ij} in B (3.1) is nonzero.

Proof. From $\Delta B_V(\mathbf{a}, \mathbf{b}) = O(|(\mathbf{a}, \mathbf{b})|_B)$, we may conclude that if $a_i b_j$ appears in ΔB_V, then it appears in B, since otherwise there is an $(\mathbf{a}, \mathbf{b})$, where $|(\mathbf{a}, \mathbf{b})|_B$ is zero but ΔB_V is nonzero. V is a bilinear divisor of the bilinear form ΔB_V. The result follows since ΔB_V must be a constant multiple of V. □

THEOREM 3.5. *Suppose that B is a* permutation *bilinear form evaluated by a polynomial program and that B has strong stability (or equivalently, that B has restricted Brent stability—see Proposition 3.2). If $a_i b_j$ appears in B with nonzero coefficient, then there is a multiplication in the program of the form $(\alpha a_i) \times (\beta b_j)$.*

Proof. The term $a_i b_j$ must appear in some V in $I(B)$ satisfying $i_B(V) \neq 0$ (see (2.4)). If V is defined by $(\sum \alpha_I a_I) \times (\sum \beta_J b_J)$, where, say, α_i, α_I and β_j are nonzero, $i \neq I$, then a term $a_I b_j$ must appear in B by Theorem 3.4. This contradicts the assumption that B is a *permutation* bilinear form. □

Two remarks are in order. If a permutation bilinear form B is computed by directly finding its terms and adding and subtracting them to get B, then B has strong stability. This follows, e.g., from [16, Thm. 4.1] since NULL(U), as defined there, is trivial. Second, if B is not a *permutation* bilinear form, then such programs may lack strong stability (see (3.2)) and "fast" programs may possess it (for example, the evaluation $B = (a_1 + a_2)(b_1 + b_2)$).

4. Speed sacrifices stability. We will say that a polynomial program to evaluate a system of s bilinear forms

$$B_k(\mathbf{a}, \mathbf{b}) \equiv \sum_{i=1}^{m} \sum_{j=1}^{n} \sigma_{ijk} a_i b_j, \qquad k = 1, \cdots, s \tag{4.1}$$

possesses, e.g., simultaneous Brent stability if each B_k has Brent stability. In this section we will draw two conclusions from our previous results.

Conclusion 1. Any polynomial program for (4.1) which has simultaneous Brent stability can make use of multiplications only of the form $L_1(\mathbf{a}) \times L_2(\mathbf{b})$.

Example 4.1. Any polynomial program of the "$L_1(\mathbf{a}) \times L_2(\mathbf{b})$" form to multiply a 2×2 matrix times a $2 \times n$ matrix requires at least $\lceil 7n/2 \rceil$ multiplications, and this bound can be achieved (Hopcroft and Kerr [11]). However, Winograd's method requires only $3n + 2$ multiplications. Thus if $n \geqq 5$, then requiring simultaneous Brent stability increases the minimum number of multiplications. Moreover, the requirement seems to simplify the verification of lower bounds, since it is not known if Winograd's method is optimal.

Example 4.2. Often one does not count multiplications which are "preconditioning", i.e., which produce only polynomials in $\mathbf{a}$ alone or in $\mathbf{b}$ alone. To reduce the number of multiplications in a polynomial program by preconditioning, one must sacrifice simultaneous Brent stability.

Example 4.3. It can be shown that any polynomial program using only multiplication of the "$L_1(\mathbf{a}) \times L_2(\mathbf{b})$" form must exhibit simultaneous Brent stability (this is not hard to prove, but a proof does not belong in this paper). Thus Strassen's algorithm for matrix multiplication has simultaneous Brent stability (see also Brent [4]).

Conclusion 2. If each B_k is a *permutation* bilinear form, then any polynomial program with simultaneous strong stability must perform t multiplications, where t is the number of pairs (i, j) such that some σ_{ijk} in (4.1) is nonzero.

Example 4.4. The n^3 multiplications in the usual algorithm for multiplying $n \times n$ matrices make it optimal among polynomial programs with simultaneous strong stability. For general polynomial programs, the arithmetic complexity of matrix multiplication seems far from resolved.

Example 4.5. Conclusion 2 also applies to the multiplication of complex numbers or polynomials (i.e., computing the coefficients of the product of polynomials). Consider $(a_1 + a_2 i) \times (b_1 + b_2 i)$, i.e., computing

$$B_1(\mathbf{a}, \mathbf{b}) = a_1 b_1 - a_2 b_2 = \text{real part},$$

$$B_2(\mathbf{a}, \mathbf{b}) = a_1 b_2 + a_2 b_1 = \text{imaginary part}.$$

If we compute $B_1 = X - Y$ and $B_2 = X - Z$, where

$$X = (a_1 + a_2)b_1, \quad Y = a_2(b_1 + b_2), \quad Z = a_1(b_1 - b_2),$$

then our previous results indicate that the influence of roundoff error upon B_1 should be "unduly large" for **a**, **b** such that

$$|(\mathbf{a},\mathbf{b})|_{B_1} \ll |\mathbf{a}| \cdot |\mathbf{b}|.$$

The correct result for

$$(0.0015 + 1.01i) \times (1.01 + 0.001i)$$

is $B_1 = 0.000505$, whereas computing in "three-digit floating-point arithmetic", i.e., rounding symmetrically to three digits after each operation, gives 0.00051 with the usual method, and 0.0 with the "fast" method.

5. Caveat. The practical value of the above results is limited by several factors:

1. They consider only polynomial operations. If a system of bilinear forms can be evaluated in n multiplications, then it can be evaluated with n multiplications of the form $L_1(\mathbf{a},\mathbf{b}) \times L_2(\mathbf{a},\mathbf{b})$ for linear L_i (Winograd [23]). Such polynomial programs (with constants) are "nearly Brent stable" in the sense that

$$\Delta B_V(\mathbf{a},\mathbf{b}) = O([\max\{|a_i|, |b_j|\}]^2)$$

for all B and defined V. They can be given simultaneous Brent stability by scaling **a** and **b** to have roughly the same size (see Brent [4], [5] for a special case).

2. Sometimes it costs as much to run a "more stable" algorithm in single precision as it costs to run a fast algorithm in double precision.

3. For a particular method, range of data, machine and software ad hoc roundoff analyses will often override general results like ours.

6. Some known stability results. This section contains a brief review of stability results concerning algorithms of (possible) interest to complexity theorists.

(a) *Evaluation of a polynomial* $p(x) = \sum_{i=0}^{n} a_i x^i$ *given* $x, a_0, \cdots, a_n$. A popular stability condition is that the computed value be exactly $\sum a_i^* x^i$, where each relative difference $|a_i^* - a_i|/|a_i|$ is small ([21, pp. 36–37]). Following our approach, this might be formalized as

$$\Delta P_V(x,\mathbf{a}) = O(\sum |a_i x^i|)$$

for all defined variables V. There is nothing uniquely plausible about this requirement, and others have been considered, e.g., [13]. See also [2].

However, the condition does seem to be widely applicable. Both Horner's role and the naive method are easily seen to be stable in this sense. Also, Woźniakowski [24] has verified this property for a family of algorithms of Shaw and Traub [19] for evaluating a polynomial and its normalized derivatives.

(b) *Polynomial evaluation with preconditioning.* Workers involved in producing subroutines for evaluating common functions have considered the possible use of the Pan and Motzkin–Belaga forms. However, these procedures are too often contaminated by numerical errors (see Rice [18] or Hart, et al. [10, pp. 67–73]). On the other hand, preliminary work by M. Rabin and S. Winograd indicates the existence of stable preconditioning methods of practical value (see [14, p. 179]).

(c) *Interpolation.* A semiformal roundoff analysis of Lagrange's formula can be found in Dorn and McCracken [9, pp. 287–291]. It seems natural to compare

Lagrange's or Newton's form with fast methods [3], both in their use for evaluating the interpolating polynomial at a point and for finding its coefficients.

However, plausible stability conditions need to be agreed upon before one can formally discuss the propagation of rounding error in these methods. The problem here is more acute than for general polynomial evaluation or (especially) matrix multiplication. Much of the difficulty stems from the fact that one may well not care how a method performs with completely arbitrary data, e.g., when the polynomial assumes alternating values of 1 and -1. Moreover, in practice, only very low degree interpolating polynomials are used (with rare exceptions).

(d) *The fast Fourier transform.* Many studies have concluded that the FFT is reasonably stable (see Ramos [17] for results and references). With this example in mind, the reader might find it instructive to attempt an extension of our notions and results to programs with complex constants.

(e) *Parallel evaluation of arithmetic expressions.* Brent [6] proves the stability (in approximately the sense of our strong stability) of certain near-optimal schemes for the parallel evaluation of arithmetic expressions lacking division. For expression containing division his schemes may lose strong stability [7].

Acknowledgments. Allan Borodin convinced me that this subject should be explored and provided many helpful ideas. S. Winograd pointed out reference [11] and clarified the nature of his work with Rabin, mentioned in § 6. David Dobkin and a referee made several helpful suggestions. Support from H. R. Strong made this work possible. A preliminary version appeared as an IBM Technical Report [15].

REFERENCES

[1] I. Babuska, *Numerical stability in mathematical analysis*, Proc. 1968 IFIP Congress, vol. I, North-Holland, Amsterdam, 1969, pp. 11–23.

[2] N. Bakhvalov, *The stable calculation of polynomial values*, U.S.S.R. Computational Math. and Math. Phys., 11 (1971), no. 6, pp. 263–271.

[3] A. Borodin, *On the number of arithmetics required to compute certain functions—circa May 1973*, Complexity of Sequential and Parallel Numerical Algorithms, J. Traub, ed., Academic Press, New York, 1973, pp. 149–180.

[4] R. P. Brent, *Algorithms for matrix multiplication*, Rep. CS157, Computer Science Dept., Stanford Univ., Stanford, Calif., 1970.

[5] ———, *Error analysis of algorithms for matrix multiplication and triangular decomposition using Winograd's identity*, Numer. Math., 16 (1970), pp. 145–156.

[6] ———, *The parallel evaluation of arithmetic expressions in logarithmic time*, Complexity of Sequential and Parallel Numerical Algorithms, J. Traub, ed., Academic Press, New York, 1973, pp. 83–102.

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

[8] D. Dobkin, *On the optimal evaluation of a set of n-linear forms*, Proc. 14th Symposium of Switching and Automata Theory, Univ. of Iowa, 1973, pp. 92–102.

[9] W. Dorn and D. McCracken, *Numerical Methods with FORTRAN IV Case Studies*, John Wiley, New York, 1972.

[10] J. Hart et al., *Handbook of Computer Approximations*, John Wiley, New York, 1965.

[11] J. Hopcroft and L. Kerr, *On minimizing the number of multiplications necessary for matrix multiplication*, SIAM J. Appl. Math., 20 (1971), pp. 30–36.

[12] W. Kahan, *A survey of error analysis*, Proc. 1971 IFIP Congress, North-Holland, Amsterdam, 1972, pp. 1214–1239.

[13] C. MESZTENYI AND C. WITZGALL, *Stable evaluation of polynomials*, J. Res. Nat. Bur. Standards, Sect. B, 71B (1967), pp. 11–17.

[14] R. MILLER AND J. THATCHER, *Complexity of Computer Computations*, Plenum Press, New York, 1972.

[15] W. MILLER, *Computational complexity and numerical stability*, IBM Tech. Rep. RC4480, IBM T. J. Watson Res. Center, Yorktown Heights, N.Y., 1973.

[16] ———, *Remarks on the complexity of roundoff analysis*, Computing, 12 (1974), pp. 149–161.

[17] G. RAMOS, *Roundoff error analysis of the fast Fourier transform*, Math. Comp., 25 (1971), pp. 757–768.

[18] J. RICE, *On the conditioning of polynomial and rational forms*, Numer. Math., 7 (1965), 426–435.

[19] M. SHAW AND J. TRAUB, *On the number of multiplications for the evaluation of a polynomial and some of its derivatives*, J. Assoc. Comput. Mach., 21 (1974), pp. 161–167.

[20] I. VITEN'KO, *Optimal algorithms for adding and multiplying on computers with a floating point*, U.S.S.R. Comput. Math. and Math. Phys., 8 (1968), no. 5, pp. 183–195.

[21] J. WILKINSON, *Rounding Errors in Algebraic Processes*, Prentice-Hall, Englewood Cliffs, N.J., 1963.

[22] ———, *Modern error analysis*, SIAM Rev., 13 (1971), pp. 548–568.

[23] S. WINOGRAD, *On the number of multiplications necessary to compute certain functions*, Comm. Pure Appl. Math., 23 (1970), pp. 165–179.

[24] H. WOŹNIAKOWSKI, *Rounding error analysis for the evaluation of a polynomial and some of its derivatives*, SIAM J. Numer. Anal., 11 (1974), pp. 780–787.

SIAM J. COMPUT.
Vol. 4, No. 2, June 1975

DERIVATION OF CONFIDENCE INTERVALS FOR WORK RATE ESTIMATORS IN A CLOSED QUEUING NETWORK*

S. S. LAVENBERG† AND G. S. SHEDLER‡

Abstract. Closed queuing networks arise naturally as models of multiprogrammed computer systems and subsystems. Techniques for the efficient simulation of such models can be obtained from analytical results on the stochastic structure of the queuing networks. In this paper, confidence intervals are derived for a class of new work rate estimators in a closed queuing network. Numerical results are given which demonstrate that a substantial reduction in the length of confidence intervals is obtainable by use of the proposed estimators.

Key words. computer system modeling and analysis, confidence intervals, queuing networks, simulation

1. Introduction. In recent years, closed queuing networks have received considerable attention (e.g., [1]–[5]) as models of multiprogrammed computer system and subsystem structures. With few exceptions, the literature is concerned with the exact mathematical analysis of the congestion phenomena in such models. In most cases, the analysis given is under the usual queuing theoretic "independent identically (often exponentially) distributed" (i.i.d.) assumptions. In spite of the mathematical simplifications which such assumptions introduce, many queuing network models of interest to the computer systems community (e.g., models of multilevel storage hierarchies) are of sufficient structural complexity that they remain intractable analytically and/or limited computationally. Moreover, indications (cf. [6]) of significant departures from the i.i.d. assumptions of queuing theory have appeared in the literature.

For these reasons, one is led to consider alternatives to exact (analytical) solution of queuing networks. An obvious alternative is (Monte Carlo) simulation, although nontrivial questions concerning the efficiency and accuracy of simulation techniques arise. The literature on methods for the simulation of queuing networks has been primarily concerned with variance reduction techniques (e.g., [7]) or hybrid control variable simulation techniques [8], the evidence of the value of the proposed technique being empirical.

This paper is concerned with techniques for the efficient simulation of closed queuing networks based on the use of estimators suggested by the structure of the network. The main results of the paper, given in §§ 4 and 5, are the derivation and computation of asymptotic confidence intervals for a class of new work rate estimators in a closed queuing network. The derivation of the confidence intervals rests on the observation that particular stochastic processes associated with the queuing network are cumulative processes (cf. [9]). In addition, the computation draws heavily on the characterization of the stochastic structure of the queuing network as an imbedded semi-Markov process.

* Received by the editors October 23, 1973, and in final revised form June 24, 1974.

† IBM Research Laboratory, San Jose, California 95193.

‡ IBM Research Laboratory, San Jose, California 95193. This work was completed while this author was visiting at the Department of Operations Research, Stanford University, Stanford, California.

It is our experience that structural results frequently can be obtained for complex queuing networks under rather general distributional assumptions, even though exact numerical solution of the networks cannot be obtained in practice due to severe computational difficulties. By "structural results" we mean relationships between response variables in a network and characterizations of the stochastic structure of the network. We anticipate that relationships between response variables can be used in simulation to suggest estimators of response variables, and that efficient simulation methods based on characterizations of the stochastic structure of queuing networks can be developed.

A description of the closed queuing network considered in this paper is given in § 2. Section 3 provides the definition of the class of proposed work rate estimators for the network. Section 4 contains a derivation of asymptotic confidence intervals for these work rate estimators. A computation of confidence interval lengths appears in § 5. Some numerical results for confidence intervals based on this computation are given in § 6. In § 7 some empirical results on confidence intervals are reported for the network under more general distributional assumptions. The final section contains some concluding remarks.

2. Description of the network. Consider the 2-stage closed cyclic queuing network shown in Fig. 1. There are a fixed number N of customers in the network. Stage 1 service times X have nonzero finite mean and otherwise arbitrary distribution function $F_X(t)$, and stage 2 service times T have nonzero finite mean and otherwise arbitrary distribution function $F_T(t)$. All service times are mutually independent. Customers are served in the order of arrival at both stages.

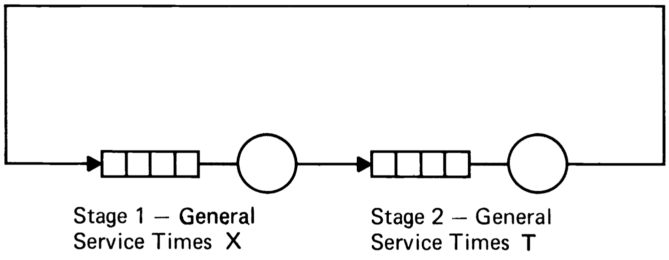


FIG. 1. *Closed 2-stage cyclic queuing network*

Suppose that at time $t = 0$ an initial location of customers in the network is specified with service(s) about to begin. For $i = 1, 2$, let $W_i(t)$ be the total server busy time at stage i in the time interval $(0, t]$, given this initial location of customers. (The dependence on the initial location of customers is suppressed in the notation.) It has been shown [11] that

$$U_i \equiv \lim_{t\to\infty} W_i(t)/t \quad \text{exists with probability 1.} \tag{1}$$

In this paper, U_i is called the *work rate* for stage i. U_i is a degenerate random variable, i.e., a constant, whose value is independent of the initial location of customers in the network. The estimation of work rates via simulation is discussed in the next section.

3. Estimation via simulation of work rates. A straightforward estimator of U_i is $U_i(\tau) = W_i(\tau)/\tau$, where τ is the fixed time at which a realization of the simulation is terminated. It follows from (1) that $U_i(\tau)$ is an asymptotically strongly consistent estimator of U_i, i.e., $\lim_{\tau\to\infty} U_i(\tau) = U_i$ with probability 1.

From conservation of flow arguments [11] it can be shown that

$$U_1/U_2 = \mu_1/\mu_2, \tag{2}$$

where μ_i is the mean service time at stage i, i.e., $\mu_1 = E[X]$ and $\mu_2 = E[T]$. Therefore,

$$U_1(\tau, \beta) = \beta U_1(\tau) + (1 - \beta)(\mu_1/\mu_2)U_2(\tau)$$

is also an asymptotically strongly consistent estimator of U_1, where β is fixed. The choice of β will be discussed later in §§ 6 and 7. Also,

$$U_2(\tau, \beta) = (\mu_2/\mu_1)U_1(\tau, \beta)$$

is an asymptotically strongly consistent estimator of U_2.

In simulation, it is desirable to have a confidence interval for the quantity being estimated. Usually, confidence intervals are determined empirically from multiple realizations of the simulation. (A noteworthy exception is the recent work of Crane and Iglehart [12].) A derivation of approximate confidence intervals centered about the estimators $U_i(\tau)$, and $U_i(\tau, \beta)$ is presented next under the assumption that stage 1 service times X are exponentially distributed. The lengths of these intervals provide a means by which the accuracy of these estimators can be assessed.

4. Derivation of confidence intervals. Assume that stage 1 service times X are exponentially distributed with rate parameter λ, i.e., $F_X(t) = 1 - \exp(-\lambda t)$, $t \geqq 0$. Also, assume that stage 2 service times T have finite variance $\mathrm{var}[T]$. Under the exponential stage 1 service time assumption, the work rates for the stages can be computed numerically (e.g., see (3) and § 5 of this paper). Thus in this case there is no need to estimate work rates via simulation. Nevertheless, the analytic results on confidence intervals which can be obtained with this assumption allow a preliminary comparison to be made of the estimators $U_i(\tau)$ and $U_i(\tau, \beta)$.

Let $n(t)$, $t \geqq 0$, be the number of customers in stage 1 at time t. We adopt the convention $n(t) = n(t+)$. Assume for convenience that $n(0) = N$, i.e., stage 2 is empty at $t = 0$, and that at $t = 0$ service is about to begin. Let $\{r_i : i = 1, 2, \cdots\}$ denote those epochs of departure from stage 2 at which stage 2 becomes empty. The r_i are regeneration points [10] in the process $n(t)$. The times between them, denoted by $\{Y_k : k = 1, 2, \cdots\}$ where $Y_1 = r_1$, $Y_k = r_k - r_{k-1}$, $k \geqq 2$, are independent random variables identically distributed as a random variable Y, i.e., the Y_k form a renewal process. Let $W_{i1} = W_i(r_1)$ and $W_{ik} = W_i(r_k) - W_i(r_{k-1})$, $k \geqq 2$. Observe that the W_{ik}, $k \geqq 1$, are independent random variables identically distributed as a random variable which is denoted by W_i. Thus, $W_i(t)$ is a cumulative process [10]. Note that $E[W_i] \leqq E[Y]$ and $E[W_i^2] \leqq E[Y^2]$.

It follows from cumulative process results [10] that if $E[Y] < \infty$, then

$$U_i = E[W_i]/E[Y]. \tag{3}$$

If, in addition, $E[Y^2] < \infty$, then

$$\operatorname{var}[W_i(t)/t] \sim \sigma_i^2/t, \tag{4}$$

where $\sim$ denotes asymptotic equality for large t and

$$\sigma_i^2 = (E[W_i^2] + E[Y^2](E[W_i]/E[Y])^2 - 2E[W_iY]E[W_i]/E[Y])/E[Y]. \tag{5}$$

Further, it is known [10] that $(W_i(t) - U_it)/\sigma_it^{1/2}$ is asymptotically normally distributed with mean zero and variance one, i.e.,

$$\lim_{t\to\infty} \Pr\{(W_i(t) - U_it)/\sigma_it^{1/2} \leqq \gamma\} = \phi(\gamma), \tag{6}$$

where

$$\phi(\gamma) = \int_{-\infty}^{\gamma} (2\pi)^{-1/2} \exp(-x^2/2)\,dx.$$

From (6) it follows that if $\gamma \geqq 0$, $\lim_{t\to\infty} \Pr\{|U_i - W_i(t)/t| \leqq \gamma\sigma_i/t^{1/2}\} = 2\phi(\gamma) - 1$. Therefore, for large τ and $\gamma \geqq 0$,

$$[U_i(\tau) - \gamma\sigma_i/\tau^{1/2},\ U_i(\tau) + \gamma\sigma_i/\tau^{1/2}] \tag{7}$$

is approximately a $100(2\phi(\gamma) - 1)\%$ confidence interval for U_i. The length of this interval is $2\gamma\sigma_i/\tau^{1/2}$.

Since $W_1(t)$ and $W_2(t)$ are cumulative processes defined with respect to the same sequence of regeneration points, $W_1(t) + W_2(t)$ is also a cumulative process. Thus, since $\operatorname{var}[W_1(t) + W_2(t)] = \operatorname{var}[W_1(t)] + \operatorname{var}[W_2(t)] + 2\operatorname{cov}[W_1(t), W_2(t)]$, cumulative process results yield, after some algebraic manipulation,

$$\operatorname{cov}[W_1(t)/t, W_2(t)/t] \sim \sigma_{12}/t, \tag{8}$$

where

$$\sigma_{12} = \left(E[W_1W_2] + \frac{E[Y^2]E[W_1]E[W_2]}{(E[Y])^2} - \frac{E[W_1Y]E[W_2] + E[W_2Y]E[W_1]}{E[Y]}\right)\Big/E[Y]. \tag{9}$$

Also, $tU_1(t,\beta) = \beta W_1(t) + (1-\beta)(\mu_1/\mu_2)W_2(t)$ is a cumulative process and $tU_2(t,\beta) = (\mu_2/\mu_1)tU_1(t,\beta)$ is a cumulative process. Using (4) and (8), it follows that

$$\operatorname{var}[U_i(t,\beta)] \sim \sigma_i^2(\beta)/t,$$

where

$$\sigma_1^2(\beta) = \beta^2\sigma_1^2 + (1-\beta)^2(\mu_1/\mu_2)^2\sigma_2^2 + 2\beta(1-\beta)(\mu_1/\mu_2)\sigma_{12} \tag{10}$$

and

$$\sigma_2^2(\beta) = (\mu_2/\mu_1)^2\sigma_1^2(\beta). \tag{11}$$

Therefore, it follows that for large τ and $\gamma \geqq 0$,

$$[U_i(\tau,\beta) - \gamma\sigma_i(\beta)/\tau^{1/2},\ U_i(\tau,\beta) + \gamma\sigma_i(\beta)/\tau^{1/2}] \tag{12}$$

is approximately a $100(2\phi(\gamma) - 1)\%$ confidence interval for U_i. In order to compare the lengths of the confidence intervals in (7) and (12), the expectations in (5) and (9) must be computed.

5. Computation of confidence interval lengths. The expectations in (5) and (9) can be computed directly using semi-Markov process analysis techniques [2], [13]. The computations, however, can be done more efficiently. The method employed in this section involves an application of Wald's equation [14], by means of which the expectations are expressed in terms of expectations which are simpler to compute using semi-Markov process analysis techniques.

In what follows, X and X_k, $k \geqq 1$, denote exponential random variables, each with rate parameter λ, and T and T_k, $k \geqq 1$, denote (nonnegative) random variables, each with the stage 2 service time distribution function $F_T(t)$; these random variables are mutually independent. The main result of this section is the demonstration that the expectations in (5) and (9) of § 4 are computable in terms of the quantities $N, \lambda, E[T], \operatorname{var}[T]$ and $\alpha_k = \Pr\{X_1 + X_2 + \cdots + X_k < T\}, 1 \leqq k \leqq N - 1$.

Let $R(t)$ denote the number of departures from stage 2 in the time interval $(0, t]$. Let $R_1 = R(r_1)$ and $R_k = R(r_k) - R(r_{k-1})$, $k \geqq 2$. The R_k are independent random variables identically distributed as a random variable R. The random variable R is the generic number of stage 2 departures between successive regeneration points. Recall that W_1, W_2 and Y are, respectively, the generic stage 1 busy time, the generic stage 2 busy time and the generic time between regeneration points. A generic interval between regeneration points is shown in Fig. 2. Let

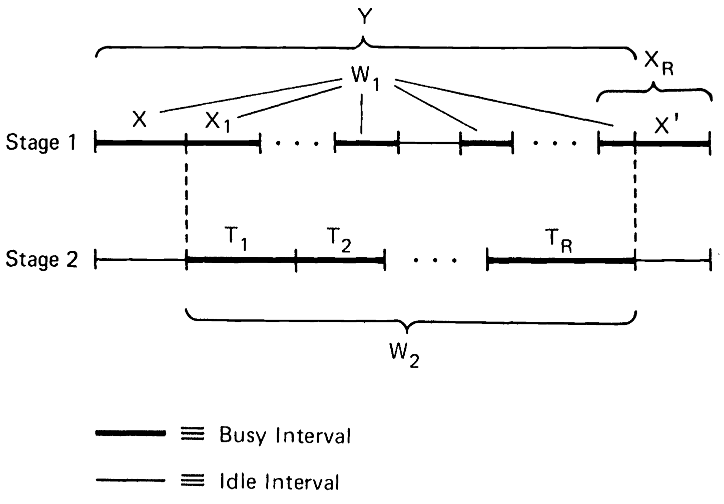


FIG. 2. *Generic interval between regeneration points*

$\hat{W}_1 = \sum_{k=1}^{R} X_k$. It is straightforward to show with reference to Fig. 2 that

$$W_1 + X' \overset{d}{=} \hat{W}_1 + X, \qquad W_2 \overset{d}{=} \sum_{k=1}^{R} T_k,$$

and

$$Y \overset{d}{=} X + W_2,$$

where X' is exponential with parameter λ, X' is independent of X, W_1 and W_2,

X is independent of $\hat{W}_1$ and W_2, and $\overset{d}{=}$ denotes equality in distribution. It follows that

$$\begin{aligned}
E[W_1] &= E[\hat{W}_1],\\
E[W_1^2] &= E[\hat{W}_1^2],\\
E[W_1W_2] &= E[\hat{W}_1W_2],\\
E[W_1Y] &= 1/\lambda^2 + E[\hat{W}_1]/\lambda + E[\hat{W}_1W_2],\\
E[W_2Y] &= E[W_2]/\lambda + E[W_2^2],\\
E[Y] &= 1/\lambda + E[W_2],\\
E[Y^2] &= 2/\lambda^2 + 2E[W_2]/\lambda + E[W_2^2].
\end{aligned}$$

Furthermore, R is independent of X_{R+1}, T_{R+1}, X_{R+2}, $T_{R+2}, \cdots$. This condition is sufficient for Wald's equation [14] to hold for the first two moments of $\hat{W}_1$ and W_2, i.e.,

$$\begin{aligned}
E[\hat{W}_1] &= E[R]/\lambda,\\
E[W_2] &= E[R]E[T],\\
E[\hat{W}_1^2] &= E[R]/\lambda^2 - E[R^2]/\lambda^2 + 2E[RW_1]/\lambda,\\
E[W_2^2] &= E[R]\operatorname{var}[T] - E[R^2](E[T])^2 + 2E[RW_2]E[T],
\end{aligned}$$

provided that all expectations on the right-hand side of these equations are finite. Appendix A contains a derivation of Wald's equation for the second moment, under more elementary assumptions than those in [14]. Wald's equation also holds for the second moment of $\hat{W}_1 + W_2$. Since

$$E[\hat{W}_1W_2] = (E[(\hat{W}_1 + W_2)^2] - E[\hat{W}_1^2] - E[W_2^2])/2,$$

after some algebraic manipulation, it follows that

$$E[\hat{W}_1W_2] = -E[R^2]E[T]/\lambda + E[RW_2]/\lambda + E[RW_1]E[T].$$

Thus, the expectations in (5) and (9) can be expressed explicitly in terms of λ, $E[T]$, $\operatorname{var}[T]$, $E[R]$, $E[R^2]$, $E[RW_1]$ and $E[RW_2]$. The quantities $E[R]$ and $E[R^2]$ can be computed in a straightforward manner by considering an imbedded Markov chain at stage 2 departure epochs. $E[RW_1]$ and $E[RW_2]$ can be computed using semi-Markov process analysis techniques. The imbedded Markov chain is discussed next.

Let $\{e_i : i = 0, 1, 2, \cdots\}$ be the epochs of departure from stage 2, where we assume $e_0 = 0$. Let $n_i = n(e_i+)$, i.e., n_i is the number of customers in stage 1 just after the ith departure from stage 2. Then $\{n_i : i = 0, 1, 2, \cdots\}$ is a finite state Markov chain over the integers $\{1, 2, \cdots, N\}$.

The transition probabilities of the Markov chain can be expressed simply in terms of the quantities α_k, where

$$\alpha_k = \Pr\{X_1 + X_2 + \cdots + X_k < T\}, \qquad k \geqq 1.$$

Since $X_1 + X_2 + \cdots + X_k$ has Erlang-k (gamma) distribution,

$$(13)\quad (1 - F_T(t))\alpha_k = \int_0^\infty (1 - F_T(t)) \frac{\lambda^k t^{k-1} \exp(-\lambda t)}{(k-1)!} dt, \qquad k \geqq 1.$$

Let p_{jk} denote the probability of transition from state j to state k in the imbedded chain. Straightforward computation reveals that for $1 \leqq j \leqq N - 1$, the nonzero transition probabilities are

$$p_{jk} = \begin{cases} 1 - \alpha_1, & k = j + 1, \\ \alpha_l - \alpha_{l+1}, & k = j + 1 - l,\ 1 \leqq l \leqq j - 1 \\ \alpha_j, & k = 1, \end{cases}$$

and that $p_{Nk} = p_{N-1,k}$, $1 \leqq k \leqq N$. The imbedded chain can be shown simply to be irreducible and aperiodic provided that T is not degenerate at zero, a case which is excluded by our assumptions. Let $\boldsymbol{\pi} = (\pi_1, \pi_2, \cdots, \pi_N)$ denote the steady state vector for the imbedded chain, i.e., $\boldsymbol{\pi}$ is the unique probability vector satisfying $\boldsymbol{\pi}\mathbf{P} = \boldsymbol{\pi}$, where $\mathbf{P} = (p_{jk})$. Direct substitution verifies that $\boldsymbol{\pi}$ is given by the following recursive procedure.

Let

$$\begin{aligned} a_0 &= 1, \\ a_1 &= \alpha_1/(1 - \alpha_1), \\ a_k &= \left(\sum_{l=1}^{k-1} (a_l \alpha_{k+1-l}) + \alpha_k \right) \Big/ (1 - \alpha_1), \qquad k \geqq 2, \\ A_k &= \sum_{j=0}^{k} a_j, \qquad k \geqq 0. \end{aligned}$$

Then

$$\pi_n = a_{N-n}/A_{N-1}, \qquad 1 \leqq n \leqq N.$$

The quantities $E[R]$ and $E[R^2]$ can be computed from $\mathbf{P}$ and $\boldsymbol{\pi}$ [13, p. 130]. The computation of $E[RW_1]$ and $E[RW_2]$ is described next.

Consider the sequence $\{W_m(e_{i+1}) - W_m(e_i) : i = 0, 1, 2, \cdots\}$ of stage m busy times between the epochs $\{e_i\}$. It is easy to verify that the stage m busy time between successive epochs e_i and e_{i+1} is, given the states at these epochs, a random variable which is independent of the stage m busy times between previous epochs and of the states at previous epochs. This allows computation of $E[RW_m]$ in a manner similar to that used to compute the moments of the first passage times for a semi-Markov process [2], [13]. This computation follows.

Let $Z_m(j, k)$ denote the stage m busy time between successive epochs, given that the state is j at the first epoch and k at the next epoch. Let $Z_m(j)$ denote the stage m busy time between successive epochs, given only that the state is j at the first of the epochs. Let $\tilde{Z}_m(j, k)$ denote the stage m busy time between an epoch at which the state is j and the *first* subsequent epoch at which the state is k. Similarly, let $\tilde{R}(j, k)$ denote the number of stage 2 departures, i.e., the number of epochs between an epoch at which the state is j and the first subsequent epoch at which the state is k. The departure at the first of the epochs is not included in $\tilde{R}(j,k)$. Suppose the state at an epoch is j and at the next epoch the state is l. Denote this event by $j \to l$. Then if $l \neq k$,

$$\begin{aligned} E[\tilde{R}(j,k)\tilde{Z}_m(j,k)|j \to l] &= E[(1+\tilde{R}(l,k))(Z_m(j,l)+\tilde{Z}_m(l,k))|j \to l] \\ &= (1+E[\tilde{R}(l,k)])E[Z_m(j,l)] + E[\tilde{Z}_m(l,k)] + E[\tilde{R}(l,k)\tilde{Z}_m(l,k)], \end{aligned}$$

and if $l = k$,

$$E[\tilde{R}(j,k)\tilde{Z}_m(j,k)|j \to k] = E[Z_m(j,k)].$$

Therefore,

$$\begin{aligned} E[\tilde{R}(j,k)\tilde{Z}_m(j,k)] &= E[Z_m(j)] \\ &\quad + \sum_{l\neq k} p_{jl}(E[\tilde{R}(l,k)]E[Z_m(j,l)] + E[\tilde{Z}_m(l,k)] \\ &\quad + E[\tilde{R}(l,k)\tilde{Z}_m(l,k)]). \end{aligned} \tag{14}$$

Multiplying both sides of (14) by π_j and summing over j yields

$$\begin{aligned} E[\tilde{R}(k,k)\tilde{Z}_m(k,k)] &= E[\tilde{Z}_m(k,k)] \\ &\quad + \left(\sum_{l\neq k} E[\tilde{R}(l,k)] \sum_{j=1}^{N} \pi_j p_{jl} E[Z_m(j,l)] + \sum_{l\neq k} \pi_l E[\tilde{Z}_m(l,k)]\right) \Big/ \pi_k, \end{aligned}$$

where we have used the equation [13, p. 133]

$$E[\tilde{Z}_m(k,k)] = \sum_{j=1}^{N} \pi_j E[Z_m(j)]/\pi_k.$$

The $E[\tilde{R}(l,k)]$ can be computed from $\mathbf{P}$, and the $E[\tilde{Z}_m(l,k)]$ can be computed from $\mathbf{P}$ and the $E[Z_m(l)]$ [13, pp. 130, 132]. Note that $E[W_m] = E[\tilde{Z}_m(N,N)]$ and $E[RW_m] = E[\tilde{R}(N,N)\tilde{Z}_m(N,N)]$. Thus, if expressions are obtained for $E[Z_m(l,k)]$, $1 \leqq l,k \leqq N$, and $m = 1,2$, then $E[RW_1]$ and $E[RW_2]$ can be computed.

It can be shown that for $1 \leqq l \leqq N-1$,

$$E[Z_2(l,k)] = \begin{cases} E[T|T<X], & k = l+1, \\ E[T|X_1+\cdots+X_j < T < X_1+\cdots+X_{j+1}], & k = l+1-j,\ 1 \leqq j \leqq l-1, \\ E[T|X_1+\cdots+X_l < T], & k = 1, \end{cases}$$

and for $l = N$,

$$E[Z_2(N,k)] = E[Z_2(N-1,k)], \qquad 1 \leqq k \leqq N.$$

Also for $1 \leqq l \leqq N-1$,

$$E[Z_1(l,k)] = \begin{cases} E[Z_2(l,k)], & 2 \leqq k \leqq l+1, \\ E[X_1+\cdots+X_l|X_1+\cdots+X_l<T], & k = 1, \end{cases}$$

and for $l = N$,

$$E[Z_1(N,k)] = E[Z_1(N-1,k)] + 1/\lambda, \qquad 1 \leqq k \leqq N.$$

The conditional expectations above are expressible directly in terms of the α_k in (13) as follows:

$$E[T|T < X] = (\alpha_1 - \alpha_2)/(1 - \alpha_1)\lambda,$$

$$E[T|X_1 + \cdots + X_j < T < X_1 + \cdots + X_{j+1}] = (j + 1)(\alpha_{j+1} - \alpha_{j+2})/(\alpha_j - \alpha_{j+1})\lambda,$$

$$E[T|X_1 + \cdots + X_l < T] = \left(\lambda E[T] - \sum_{k=1}^{l} \alpha_k + l\alpha_{l+1}\right)\Big/\alpha_l\lambda,$$

$$E[X_1 + \cdots + X_l|X_1 + \cdots + X_l < T] = l\alpha_{l+1}/\alpha_l\lambda.$$

6. Numerical results. In this section, the length of the confidence interval centered at $U_i(\tau)$ in (7) and the length of the confidence interval centered at $U_i(\tau, \beta)$ in (12) are compared numerically for a fixed level of confidence and a fixed realization length τ by comparing numerically σ_i and $\sigma_i(\beta)$, where $\sigma_i(\beta)$ is expressed in terms of σ_1, σ_2 and σ_{12} in (10) and (11). In this section, β is taken equal to β^*, where β^* denotes the optimum value of β, i.e., the value of β which minimizes $\sigma_1(\beta)$ and hence minimizes the length of the confidence interval centered at $U_1(\tau, \beta)$. A simple computation using (10) yields

$$(15) \qquad \beta^* = ((\mu_1/\mu_2)^2\sigma_2^2 - (\mu_1/\mu_2)\sigma_{12})/(\sigma_1^2 + (\mu_1/\mu_2)^2\sigma_2^2 - 2(\mu_1/\mu_2)\sigma_{12}).$$

Note from (11) that β^* also minimizes $\sigma_2(\beta)$. It follows from (15) that if $\sigma_2/\mu_2 \gg \sigma_1/\mu_1$, then $\beta^* \approx 1$; if $\sigma_2/\mu_2 \approx \sigma_1/\mu_1$, then $\beta^* \approx .5$; and if $\sigma_2/\mu_2 \ll \sigma_1/\mu_1$, then $\beta^* \approx 0$.

When estimating U_i via simulation using the estimator $U_i(\tau, \beta)$, β^* is not known. Either a value of β which is expected to be near β^* is chosen a priori or β^* is estimated empirically from multiple realizations of the simulation. Thus the accuracy obtained in practice when using the estimator $U_i(\tau, \beta)$ will not be as great as is indicated in this section.

Numerical values for σ_1, σ_2, β^*, $\sigma_1^* = \sigma_1(\beta^*)$, $\sigma_2^* = \sigma_2(\beta^*)$, σ_1^*/σ_1 and σ_2^*/σ_2 are given in Tables 1–5 for various service time distributions at stage 2. (From (11), $\sigma_2^* = (\mu_2/\mu_1)\sigma_1^*$.) Also given are numerical values for U_1 and U_2 which are computed from (3) and numerical values for $\rho = \sigma_{12}/\sigma_1\sigma_2$, the asymptotic correlation of $U_1(\tau)$ and $U_2(\tau)$. The coefficient of variation of the stage 2 service time distribution increases from 0 in Table 1 to 5 in Table 5. Observe from the tables that the estimators $U_1(\tau)$ and $(\mu_1/\mu_2)U_2(\tau)$ have, in general, unequal variances and are negatively correlated, i.e., $\sigma_1 \neq (\mu_1/\mu_2)\sigma_2$ and $\rho < 0$. (It follows from (15) that if $\rho < 0$, then $0 < \beta^* < 1$.) The optimum parameter value β^* represents a compromise between putting all the weight ($\beta^* = 0$ or $\beta^* = 1$) on whichever of these two estimators has the smaller variance and putting equal weights ($\beta^* = .5$) on the two estimators in order to take advantage of the negative correlation between them. From the tables, when $\mu_2 \neq \mu_1$, the stage with the larger mean service time—and hence the larger work rate—has a smaller value of σ_i/μ_i and β^* is usually closer to either 0 or 1 than to .5. When $\mu_2 = \mu_1$, the stage with the larger service time variance has the smaller value of σ_i/μ_i, although the values for the two stages do not differ greatly, and β^* is closer to .5 than to either 0 or 1.

TABLE 1
Comparison of asymptotic confidence interval lengths.
Stage 2 *service times*: const., $\mu_2 = E[T] = 1.$, var $[T] = 0$

μ_1	N	U_1	U_2	σ_1	σ_2	ρ	β^*	σ_1^*	σ_2^*	σ_1^*/σ_1	σ_2^*/σ_2
0.5	2	.468	.937	.398	.228	−.697	.190	.067	.134	.169	.589
	4	.499	.998	.490	.055	−.311	.020	.026	.051	.052	.933
	6	.500	1.000	.499	.011	−.109	.001	.006	.011	.011	.993
1.0	2	.731	.731	.423	.484	−.774	.538	.152	.152	.359	.314
	4	.870	.870	.501	.537	−.615	.522	.227	.227	.454	.423
	6	.914	.914	.527	.551	−.571	.514	.249	.249	.473	.452
2.0	2	.904	.452	.288	.558	−.741	.822	.161	.080	.557	.144
	4	.992	.496	.105	.681	−.413	.965	.092	.046	.880	.068
	6	.999	.500	.031	.704	−.199	.995	.030	.015	.975	.022

TABLE 2
Comparison of asympotic confidence interval lengths.
Stage 2 *service times*: Erlang-3, $\mu_2 = E[T] = 1.$, var $[T] = .333$

μ_1	N	U_1	U_2	σ_1	σ_2	ρ	β^*	σ_1^*	σ_2^*	σ_1^*/σ_1	σ_2^*/σ_2
0.5	2	.451	.903	.438	.284	−.714	.214	.079	.159	.181	.559
	4	.495	.990	.547	.115	−.429	.051	.050	.099	.091	.861
	6	.499	.999	.572	.040	−.232	.009	.019	.038	.033	.964
1.0	2	.703	.703	.497	.532	−.770	.519	.174	.174	.351	.328
	4	.842	.842	.574	.599	−.632	.513	.252	.252	.438	.420
	6	.893	.893	.604	.621	−.586	.509	.278	.278	.461	.448
2.0	2	.885	.443	.372	.620	−.741	.796	.202	.101	.542	.163
	4	.985	.492	.176	.765	−.474	.940	.146	.073	.831	.095
	6	.998	.499	.069	.805	−.283	.986	.066	.033	.947	.041

TABLE 3
Comparison of asymptotic confidence interval lengths.
Stage 2 *service times*: exponential, $\mu_2 = E[T] = 1.$, var $[T] = 1$

μ_1	N	U_1	U_2	σ_1	σ_2	ρ	β^*	σ_1^*	σ_2^*	σ_1^*/σ_1	σ_2^*/σ_2
0.5	2	.429	.857	.507	.358	−.771	.238	.088	.176	.174	.492
	4	.484	.968	.628	.222	−.555	.105	.083	.167	.133	.751
	6	.496	.992	.678	.120	−.401	.040	.053	.106	.078	.882
1.0	2	.667	.667	.609	.609	−.800	.5	.192	.192	.316	.316
	4	.800	.800	.693	.693	−.667	.5	.283	.283	.408	.408
	6	.857	.857	.728	.728	−.615	.5	.319	.319	.439	.439
2.0	2	.857	.429	.507	.716	−.771	.762	.249	.125	.492	.174
	4	.968	.484	.314	.888	−.555	.895	.236	.118	.751	.133
	6	.992	.496	.170	.959	−.401	.960	.150	.075	.882	.078

TABLE 4
Comparison of asymptotic confidence interval lengths.
Stage 2 *service times*: hyperexponential,
$F_T(t) = \gamma(1 - \exp(-t/\delta_1)) + (1 - \gamma)(1 - \exp(-t/\delta_2)), t \geqq 0,$
$\delta_1 = 4., \delta_2 = .5, \gamma = .143,$
$\mu_2 = E[T] = 1., \operatorname{var}[T] = 4$

μ_1	N	U_1	U_2	σ_1	σ_2	ρ	β^*	σ_1^*	σ_2^*	σ_1^*/σ_1	σ_2^*/σ_2
0.5	2	.409	.818	.786	.493	−.887	.228	.088	.177	.112	.359
	4	.463	.925	.934	.376	−.708	.138	.115	.231	.123	.613
	6	.483	.965	1.01	.289	−.590	.088	.107	.214	.106	.741
1.0	2	.625	.625	.991	.818	−.908	.450	.193	.193	.194	.235
	4	.725	.725	1.11	.896	−.794	.440	.319	.319	.286	.356
	6	.776	.776	1.16	.962	−.730	.447	.386	.386	.334	.401
2.0	2	.811	.405	.940	.988	−.895	.686	.291	.146	.310	.148
	4	.907	.454	.827	1.17	−.759	.763	.417	.209	.505	.179
	6	.949	.474	.677	1.29	−.663	.829	.426	.213	.629	.165

TABLE 5
Comparison of asymptotic confidence interval lengths.
Stage 2 *service times*: hyperexponential
$F_T(t) = \gamma(1 - \exp(-t/\delta_1)) + (1 - \gamma)(1 - \exp(-t/\delta_2)), t \geqq 0,$
$\delta_1 = 25., \delta_2 = .5, \gamma = .02,$
$\mu_2 = E[T] = 1., \operatorname{var}[T] = 25$

μ_1	N	U_1	U_2	σ_1	σ_2	ρ	β^*	σ_1^*	σ_2^*	σ_1^*/σ_1	σ_2^*/σ_2
0.5	2	.402	.803	1.82	.959	−.972	.206	.089	.178	.049	.186
	4	.448	.896	2.13	.640	−.881	.121	.133	.267	.063	.417
	6	.467	.934	2.26	.516	−.786	.086	.146	.292	.065	.566
1.0	2	.605	.605	2.34	1.64	−.982	.411	.182	.182	.078	.111
	4	.668	.668	2.64	1.57	−.949	.370	.315	.315	.119	.200
	6	.692	.692	2.71	1.60	−.927	.367	.385	.385	.142	.240
2.0	2	.778	.389	2.37	2.01	−.984	.630	.264	.132	.111	.066
	4	.827	.414	2.43	2.10	−.962	.636	.425	.212	.175	.101
	6	.848	.424	2.36	2.19	−.941	.654	.523	.262	.222	.120

The tables indicate that for a fixed level of confidence and a fixed realization length, a substantial reduction in the confidence interval length can be obtained by using the estimator $U_i(\tau, \beta^*)$ instead of the estimator $U_i(\tau)$. Equivalently, for a fixed level of confidence and a fixed confidence interval length, a substantially smaller realization length can be used. (For a fixed confidence interval length, $\sigma_i^2(\beta)$ and τ are inversely proportional.)

7. Empirical results. In order to compare analytically the estimators $U_i(\tau)$ and $U_i(\tau, \beta)$, it was assumed in the preceding sections that stage 1 service times are exponentially distributed. In this section, this exponential assumption is removed and the estimators $U_i(\tau)$ and $U_i(\tau, \beta)$ are compared empirically.

When using the estimator $U_i(\tau, \beta)$ in practice, it is necessary either to choose a priori the parameter β or to estimate β via simulation. In addition, it is desirable to estimate via simulation a valid confidence interval based on this estimator. The use of the estimator $U_i(\tau, \beta)$ when β is chosen a priori is considered first.

If β is chosen a priori, then the following theorem provides a distributional theory for estimating confidence intervals based on $U_i(\tau, \beta)$ (or based on $U_i(\tau)$).

THEOREM. *Assume $F_X(t)$ (resp. $F_T(t)$) is a finite mixture of Erlangian distributions, i.e., $F_X(t)$(resp. $F_T(t)$) $= \sum_{j=1}^{J} \delta_j F(k_j, t)$, $t \geqq 0$, where $\delta_j \geqq 0$, $1 \leqq j \leqq J$, $\sum_{j=1}^{J} \delta_j = 1$, k_j is a positive integer and $F(k, t)$ is an Erlang-k distribution. Assume T (resp. X) has nonzero finite mean and finite variance. Then there exists an increasing sequence of random times which are regeneration points in the stochastic process describing the evolution of the network in time. Furthermore, the first two moments of the time between successive regeneration points are finite.*

The proof of this theorem is straightforward and proceeds by considering a finite state imbedded semi-Markov process at the epochs of departure from the non-Erlangian stage. It can be simply shown that the imbedded Markov chain at these epochs is irreducible and that the unconditional waiting times of the imbedded semi-Markov process have finite first and second moments. Thus all states of the imbedded semi-Markov process are recurrent and the first two moments of the recurrence times are finite.

Using this theorem, it follows from cumulative process results [10] that $(U_i(t, \beta) - U_i)/(\mathrm{var}\,[U_i(t, \beta)])^{1/2}$ is asymptotically normally distributed with mean zero and variance one. Therefore a confidence interval for U_i can be estimated using the t-statistic from multiple independent realizations of a simulation as described below. (The theorem also provides sufficient conditions for estimating confidence intervals using the method for regenerative processes proposed by Crane and Iglehart [12]. This method is not pursued in this paper, however.) Note that under the assumptions of the theorem, one could in principle use semi-Markov process analysis techniques to compute the work rates. However, due to the complexity of the computations, particularly as $\sum_{j=1}^{J} k_j$ increases, simulation is a viable alternative.

Let $M > 1$ be the number of independent realizations. Each realization starts at simulated time zero with all N customers in stage 1 and service about to begin. The realization is stopped when simulated time τ is reached. The superscript k will denote the value of an estimator observed on the kth realization. Let

$$\hat{U}_i(\tau, \beta) = \sum_{k=1}^{M} U_i^k(\tau, \beta)/M,$$

$$\hat{V}_i^2(\tau, \beta) = \sum_{k=1}^{M} (U_i^k(\tau, \beta) - \hat{U}_i(\tau, \beta))^2/(M - 1).$$

Then $M^{1/2}(U_i(\tau, \beta) - U_i)/\hat{V}_i(\tau, \beta)$ is asymptotically (i.e., for τ large) distributed as the standardized t-statistic with $M - 1$ degrees of freedom. Therefore, for τ large and $\gamma \geqq 0$,

$$[\hat{U}_i(\tau, \beta) - \gamma \hat{V}_i(\tau, \beta)/M^{1/2}, \hat{U}_i(\tau, \beta) + \gamma \hat{V}_i(\tau, \beta)/M^{1/2}]$$

is approximately a $100(2\theta_{M-1}(\gamma) - 1)\%$ confidence interval for U_i based on the estimator $U_i(\tau, \beta)$, where $\theta_{M-1}(\gamma)$ is the distribution of the standardized t-statistic with $M - 1$ degrees of freedom. Similarly, let

$$\hat{U}_i(\tau) = \sum_{k=1}^{M} U_i^k(\tau)/M,$$

$$\hat{V}_i^2(\tau) = \sum_{k=1}^{M} (U_i^k(\tau) - \hat{U}_i(\tau))^2/(M - 1).$$

Then for τ large and $\gamma \geqq 0$,

$$[\hat{U}_i(\tau) - \gamma\hat{V}_i(\tau)/M^{1/2}, \hat{U}_i(\tau) + \gamma\hat{V}_i(\tau)/M^{1/2}]$$

is approximately a $100(2\theta_{M-1}(\gamma) - 1)\%$ confidence interval for U_i based on the estimator $U_i(\tau)$.

The question of choosing β a priori has not yet been addressed. One method of choosing β a priori for a network with service time distributions $F_X(t)$ and $F_T(t)$ is to choose β equal to $\beta_{\exp}$, where $\beta_{\exp}$ is the optimum value of β (see (15)) computed under the assumption that X and T are exponentially distributed. In order to investigate whether this choice of β is good, results from simulating the queuing network are presented in Tables 6 and 7. Stage 1 service times are constant and stage 2 service times have hyperexponential distribution (i.e., a mixture of two exponential distributions) with coefficient of variation equal to 2 in Table 6 and equal to 5 in Table 7. Each row of the tables is based on 25 independent experiments where each experiment consists of 5 independent realizations of a simulation with realization length 1,000. In the tables, $\bar{U}_i$, $\bar{\sigma}_i$ and $\bar{\sigma}_i(\beta)$ are averages over the 25 experiments; $\bar{U}_i$ is the average of $U_i(\tau)$, $\bar{\sigma}_i$ is the average of $\tau^{1/2}\hat{V}_i(\tau)$ and $\bar{\sigma}_i(\beta)$ is the average of $\tau^{1/2}\hat{V}_i(\tau, \beta)$.

The results in the last 2 columns of Table 6 and 7 indicate that substantial reductions in confidence interval lengths are obtained in practice using the estimator $U_i(\tau, \beta)$ with $\beta = \beta_{\exp}$ instead of the estimator $U_i(\tau)$. For example, if $\mu_1 = 1$ and $N = 4$ in Table 6, then $\bar{\sigma}_1(\beta)/\bar{\sigma}_1 = .279$ and $\bar{\sigma}_2(\beta)/\bar{\sigma}_2 = .408$, which corresponds to a reduction in confidence interval length by a factor of $1/.279 = 3.58$ for the stage 1 work rate and by a factor of $1/.408 = 2.45$ for the stage 2 work rate. Reductions in confidence interval lengths by factors of 2 or more are usual. Two other a priori choices for β are $\beta = \beta_{X,\exp}$ or $\beta_{T,\exp}$ where $\beta_{X,\exp}$ (resp., $\beta_{T,\exp}$) is the optimum value of β computed under the assumption that X (resp., T) is exponentially distributed and T (resp., X) has distribution $F_T(t)$ (resp., $F_X(t)$). These choices are briefly investigated in Table 8, which is self-explanatory. The results of Table 8 should be compared with the results in the middle three rows of Table 7 to see that further reductions in confidence interval lengths are obtained.

The use of the estimator $U_i(\tau, \beta)$ when β is estimated via simulation is now briefly discussed. An estimate $\hat{\beta}$ of the optimum value for the parameter β could be obtained from the M independent realizations by substituting $\hat{V}_i^2(\tau)$ for σ_i^2 and $\hat{V}_{12}(\tau)$ for σ_{12} in (15), where

$$\hat{V}_{12}(\tau) = \sum_{k=1}^{M} (U_1^k(\tau) - \hat{U}_1(\tau))(U_2^k(\tau) - \hat{U}_2(\tau))/(M - 1).$$

The resulting estimator $\hat{U}_i(\tau, \hat{\beta})$ introduces additional bias due to the correlation

TABLE 6

Empirical comparison of asymptotic confidence interval lengths; $\tau = 1{,}000$, $M = 5$, 25 *experiments.*
Stage 1 *service times*: const.
Stage 2 *service times*: hyperexponential
(*same as Table* 4)

μ_1	N	$\bar{U}_1$	$\bar{U}_2$	$\bar{\sigma}_1$	$\bar{\sigma}_2$	$\beta = \beta_{\text{exp}}$	$\bar{\sigma}_1(\beta)$	$\bar{\sigma}_2(\beta)$	$\bar{\sigma}_1(\beta)/\bar{\sigma}_1$	$\bar{\sigma}_2(\beta)/\bar{\sigma}_2$
0.5	2	.434	.858	.720	.312	.238	.086	.172	.120	.553
	4	.489	.959	.892	.196	.105	.080	.159	.089	.814
	6	.503	.986	.836	.133	.040	.060	.123	.071	.925
1.0	2	.671	.661	1.09	.713	.5	.234	.234	.215	.328
	4	.764	.753	1.07	.731	.5	.298	.298	.279	.408
	6	.812	.800	1.05	.811	.5	.333	.333	.317	.411
2.0	2	.852	.420	.894	.847	.762	.352	.174	.393	.205
	4	.934	.461	.788	1.11	.895	.524	.262	.665	.235
	6	.969	.478	.552	1.16	.960	.463	.231	.838	.198

TABLE 7

Empirical comparison of asymptotic confidence interval lengths; $\tau = 1{,}000$, $M = 5$, 25 *experiments.*
Stage 1 *service times*: const.
Stage 2 *service times*: hyperexponential
(*same as Table* 5)

μ_1	N	$\bar{U}_1$	$\bar{U}_2$	$\bar{\sigma}_1$	$\bar{\sigma}_2$	$\beta = \beta_{\text{exp}}$	$\bar{\sigma}_1(\beta)$	$\bar{\sigma}_2(\beta)$	$\bar{\sigma}_1(\beta)/\bar{\sigma}_1$	$\bar{\sigma}_2(\beta)/\bar{\sigma}_2$
0.5	2	.429	.844	2.00	.760	.238	.202	.405	.101	.533
	4	.475	.934	2.05	.405	.105	.113	.226	.055	.557
	6	.490	.962	2.25	.302	.040	.101	.202	.045	.670
1.0	2	.651	.636	2.23	1.27	.5	.494	.494	.222	.391
	4	.688	.680	2.56	1.40	.5	.640	.640	.250	.457
	6	.707	.695	2.45	1.42	.5	.594	.594	.242	.418
2.0	2	.808	.401	2.27	1.80	.762	.899	.449	.396	.250
	4	.831	.415	2.10	1.76	.895	1.52	.763	.725	.434
	6	.853	.423	1.97	1.84	.960	1.75	.877	.891	.477

TABLE 8

Empirical comparison of asymptotic confidence interval lengths; $\tau = 1{,}000$, $M = 5$, 25 *experiments.*
Stage 1 *service times*: const., $\mu_1 = 1$.
Stage 2 *service times*: hyperexponential
(*same as Table* 5)

N	$\beta = \beta_{X,\text{exp}}$	$\bar{\sigma}_1(\beta)$	$\bar{\sigma}_2(\beta)$	$\bar{\sigma}_1(\beta)/\bar{\sigma}_1$	$\bar{\sigma}_2(\beta)/\bar{\sigma}_2$
2	.411	.206	.206	.092	.162
4	.370	.265	.265	.104	.190
6	.367	.275	.275	.112	.194
	$\beta = \beta_{T,\text{exp}}$				
2	.462	.366	.366	.165	.290
4	.478	.562	.562	.219	.402
6	.486	.547	.547	.223	.386

between $\hat{\beta}$ and $U_i^k(\tau)$. (Note that $\hat{U}_i(\tau)$ already contains bias due to τ being finite, i.e., $E[\hat{U}_i(\tau)] \neq U_i$.) This additional bias can be removed by the standard technique of splitting, e.g., [8]. However, since $\hat{\beta}$ is a random variable, it no longer follows that $U_i^k(\tau, \hat{\beta})$ is asymptotically normal. Therefore, it is an open question whether a valid confidence interval for U_i based on $\hat{U}_i(\tau, \hat{\beta})$ can be estimated from a small number of independent realizations, as is the case for $\hat{U}_i(\tau)$. If not, then any reduction in confidence interval length obtained by using $\hat{U}_i(\tau, \hat{\beta})$ could be more than offset by the extra realizations required. This question is not pursued in this paper.

Remarks. (i) Another way of comparing the accuracy of the estimators $U_i(\tau)$ and $U_i(\tau, \beta)$ is to compare the mean-square errors for fixed τ. The mean-square error of the estimator $U_i(\tau)$ is given by

$$(16)\qquad \mathrm{MSE}\,[U_i(\tau)] = E[(U_i(\tau) - U_i)^2] = \mathrm{var}\,[U_i(\tau)] + (E[U_i(\tau)] - U_i)^2.$$

The second term in (16) is the square of the bias of the estimator. In Appendix B it is shown that since $W_i(t)$ is a cumulative process and $0 \leqq W_i(t_2) - W_i(t_1) \leqq t_2 - t_1$ for all $0 \leqq t_1 \leqq t_2$,

$$E[U_i(\tau)] \sim U_i + b_i/\tau.$$

Therefore, from (4) and (16) it follows that $\mathrm{MSE}\,[U_i(\tau)] \sim \sigma_i^2/\tau$. Similarly, $\mathrm{MSE}\,[U_i(\tau, \beta)] \sim \sigma_i^2(\beta)/\tau$. Thus, by comparing σ_i and $\sigma_i(\beta)$, $\mathrm{MSE}\,[U_i(\tau)]$ and $\mathrm{MSE}\,[U_i(\tau, \beta)]$ are compared for fixed τ.

(ii) It was assumed in § 4 that $n(0) = N$, i.e., stage 2 is empty at $t = 0$. However, if $n(0) = n$, $0 \leqq n < N$, then the sequence $\{Y_k : k = 1, 2, \cdots\}$ forms a general renewal process [10], and all the results in §§ 4–6 still hold. These results also hold for queuing disciplines other than first-come first-served, such as last-come first-served and random, which are also independent of service time.

(iii) The estimator for work rates proposed in this paper is an asymptotically strongly consistent estimator which is a linear combination of the straightforward estimators of the work rates at different stages. This new estimator was suggested by the known relation between the work rates for different stages given in (2). Similar relations, due to conservation of flow, exist between the work rates for different stages in more complex closed queuing networks than the one considered in this paper [11]. Asymptotically strongly consistent estimators of work rates in these networks are similarly obtained by taking appropriate linear combinations of the straightforward estimators of work rates for different stages. The results of this paper suggest that these estimators for more complex networks are worth investigating, analytically where possible and otherwise empirically.

Appendix A.

THEOREM. *Let V_k, $k \geqq 1$, be independent and identically distributed nonnegative random variables. Let K be a positive integer-valued random variable such that K and $V_{K+1}, V_{K+2}, \cdots$ are mutually independent. Let $S_K = \sum_{k=1}^K V_k$. Then*

$$(\mathrm{A}.1)\qquad E[S_K^2] = E[K]\,\mathrm{var}\,[V] - E[K^2](E[V])^2 + 2E[V]E[KS_K],$$

provided the expectations and variance on the right side of (A.1) *are finite.*

Proof. Let

$$\psi_k(K) = \begin{cases} 1, & k \leqq K, \\ 0, & k > K. \end{cases}$$

Then $\psi_k(K)$ and V_k are independent and $\psi_k(K)\psi_j(K) = \psi_k(K)$, $k \geqq j$.

$$E[S_K^2] = E\left[\sum_{k=1}^{\infty}\sum_{j=1}^{\infty} V_k\psi_k(K)V_j\psi_j(K)\right]$$

$$\text{(A.2)} \qquad = \sum_{k=1}^{\infty} E[V_k^2\psi_k(K)] + \sum_{k=2}^{\infty}\sum_{j=1}^{k-1} E[V_k\psi_k(K)V_j] + \sum_{k=1}^{\infty}\sum_{j=k+1}^{\infty} E[V_kV_j\psi_j(K)]$$

$$= E[V^2]\sum_{k=1}^{\infty}\Pr\{K \geqq k\} + 2E[V]\sum_{k=1}^{\infty}\sum_{j=k+1}^{\infty} E[V_k\psi_j(K)]$$

$$\text{(A.3)} \qquad = E[V^2]E[K] + 2E[V]E\left[\sum_{k=1}^{\infty} V_k \sum_{j=k+1}^{\infty}\psi_j(k)\right],$$

where the interchanges of summation and integration are justified by the monotone convergence theorem and the double summations in (A.2) are equal by Fubini's theorem.

Observe that $\sum_{j=k+1}^{\infty}\psi_j(K) = \max(K - k, 0)$. Therefore,

$$\text{(A.4)} \quad E\left[\sum_{k=1}^{\infty} V_k \sum_{j=k+1}^{\infty}\psi_j(K)\right] = E\left[\sum_{k=1}^{K} V_k(K-k)\right] = E[KS_K] - E\left[\sum_{k=1}^{K} kV_k\right],$$

where

$$E\left[\sum_{k=1}^{K} kV_k\right] = E\left[\sum_{k=1}^{\infty} kV_k\psi_k(K)\right] = E[V]\sum_{k=1}^{\infty} k\Pr\{K \geqq k\}$$

$$\text{(A.5)} \qquad = E[V](E[K^2] + E[K])/2.$$

Equation (A.1) follows from (A.3)–(A.5), completing the proof.

Appendix B. Let $J(t)$ be the number of regeneration epochs in the time interval $(0, t]$, i.e., $J(t) = \sup\{j : r_j \leqq t\}$, where $r_0 = 0$. Then writing

$$W_i(t) = \sum_{j=1}^{J(t)+1} W_{ij} - (W_i(r_{J(t)+1}) - W_i(t))$$

and noting that $0 \leqq W_i(t_2) - W_i(t_1) \leqq t_2 - t_1$ for all t_1, t_2 such that $0 \leqq t_1 \leqq t_2$ yields

$$\text{(B.1)} \qquad \sum_{j=1}^{J(t)+1} W_{ij} - (r_{J(t)+1} - t) \leqq W_i(t) \leqq \sum_{j=1}^{J(t)+1} W_{ij}.$$

It follows directly from results in [15] that

$$\text{(B.2)} \qquad E\left[\sum_{j=1}^{J(t)+1} W_{ij}\right] \sim tE[W_i]/E[Y] + E[W_i]E[Y^2]/2(E[Y])^2$$

and

$$E[r_{J(t)+1}] \sim t + E[Y^2]/2E[Y] \tag{B.3}$$

if $E[Y] < \infty$, $E[Y^2] < \infty$ and the random variable Y is not lattice, conditions which hold for the queuing network considered in this paper. Combining (B.1)–(B.3) yields

$$E[W_i(t)/t] \sim E[W_i]/E[Y] + b_i/t,$$

where

$$(E[W_i] - E[Y])E[Y^2]/2(E[Y])^2 \leqq b_i \leqq E[W_i]E[Y^2]/2(E[Y])^2.$$

REFERENCES

[1] D. P. Gaver, *Probability models for multiprogramming computer systems*, J. Assoc. Comput. Mach., 14 (1967), pp. 423–438.
[2] G. S. Shedler, *A cyclic-queue model of a paging machine*, IBM Res. Rep. RC-2814, Yorktown Heights, N.Y., 1970.
[3] P. A. W. Lewis and G. S. Shedler, *A cyclic-queue model of system overhead in multiprogrammed computer systems*, J. Assoc. Comput. Mach., 18 (1971), pp. 199–220
[4] J. P. Buzen, *Queueing network models of multiprogramming*, Ph.D. thesis, Div. of Engineering and Applied Physics, Harvard University, Cambridge, Mass., 1971.
[5] S. S. Lavenberg, *Queueing analysis of a multiprogrammed computer system having a multilevel storage hierarchy*, this Journal, 2 (1973), pp. 232–252.
[6] P. A. W. Lewis and G. S. Shedler, *Empirically derived micromodels of sequences of page exceptions*, IBM J. Res. Develop., 17 (1973), pp. 86–100.
[7] G. S. Shedler and S. C. Yang, *Simulation of a model of paging system performance*, IBM Systems J., 10 (1971), pp. 113–128.
[8] D. P. Gaver and G. S. Shedler, *Control variable methods in the simulation of a model of a multiprogrammed computer system*, Naval Res. Logist. Quart., 18 (1971), pp. 435–450.
[9] D. G. Polyak, *Precision of statistical simulation of queueing systems*, Engrg. Cybernetics 1 (1970), pp. 72–80.
[10] W. L. Smith, *Renewal theory and its ramifications*, J. Roy. Statist. Soc. Ser. B, 20 (1958), pp. 243–302.
[11] A. Chang and S. S. Lavenberg, *Work rates in closed queuing networks with general independent servers*, Operations Res., 22 (1974), pp. 838–847.
[12] M. A. Crane and D. L. Iglehart, *Statistical analysis of discrete even simulations*, Proc. 1974 Winter Simulation Conf., pp. 513–521, Washington, D.C.
[13] R. E. Barlow and F. Proschan, *Mathematical Theory of Reliability*, John Wiley, New York, 1967.
[14] N. L. Johnson, *A proof of Wald's theorem on cumulative sums*, Ann. Math. Statist., 30 (1959), pp. 1245–1247.
[15] W. L. Smith, *Regenerative stochastic processes*, Proc. Roy. Soc. Ser. A, 232 (1955), pp. 6–31.

SIAM J. COMPUT.
Vol. 4, No. 2, June 1975

TRANSLATING PROGRAM SCHEMAS TO WHILE-SCHEMAS*

EDWARD ASHCROFT† AND ZOHAR MANNA‡

Abstract. While-schemas are defined as program schemas without goto statements, in which iteration is achieved using while statements. We present two translations of program schemas into equivalent while-schemas, the first one by adding extra program variables, and the second one by adding extra logical variables. In both cases we aim to preserve as much of the structure of the original program schemas as possible.

We also show that, in general, any translation must add variables.

Key words. program schemas, removing goto statements, while-schemas, flowchart transformations.

Introduction. The program schema approach makes it meaningful to consider the relative "power" of programming language constructs. Most work in this area [13], [4], [5], [14], [15] has considered adding features to program schemas such as recursion and arrays. Here we consider removing, or at least restricting, a feature of program schemas: the goto statement.

There has been much interest lately, following observations by Dijkstra [8], in the possibility and desirability of removing goto statements from programming languages, using instead such statements as the while statement. Programs in such languages should be better structured, easier to understand and, hopefully, easier to prove correct. For example, the elegant formal system of Hoare [10] for proving programs correct requires programs with the sort of "nested" structure that while statements provide. Goto-less programs are clearly an interesting class of programs to study.

We therefore define a class of while-schemas in which iteration is achieved with while statements: **while** ψ **do** S. The tests ψ may be arbitrarily complicated; this feature of our while-schemas is crucial. We show that while-schemas are as powerful as program schemas by giving a translation, Algorithm 1, of program schemas to equivalent while-schemas. This translation is interesting in that it preserves most of the "loop structure" of the program schemas, and gives while-schemas of the same order of efficiency. The translation allows the addition of extra program variables.

Bohm and Jacopini [3] have shown that program schemas can be translated into while-schemas, with the addition of extra *logical variables*. A modification of a technique in Brown et al. [2] would show (by a further translation) that the additional logical variables add no extra power to while-schemas. We present an improvement on Bohm and Jacopini's reduction to while-schemas with logical variables, which we call Algorithm 2. This doesn't give us "pure" while-schemas, since logical variables are used, but the schemas produced are often more "readable"

* Received by the editors November 2, 1973, and in revised form March 1, 1974. This research was supported in part by the Advanced Research Projects Agency of the Office of the U.S. Secretary of Defense under Contract SD-183, and in part by the National Research Council of Canada.

† Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada.

‡ Department of Applied Mathematics, Weizmann Institute, Rehovot, Israel.

than those produced by Algorithm 1. The method preserves whatever "while-structure" already exists in a program schema, and when applied to a program schema corresponding directly to a while-schema, Algorithm 2 gives us back that while-schema.

Both Algorithm 1 and Algorithm 2 give while-schemas that use more variables in general than the original program schemas. It is natural to ask whether this is a necessary feature of any translation. We show that this is the case by giving a program schema, with one variable, for which there is no equivalent one-variable while-schema. This also means, of course, that program schemas *are* more powerful than while-schemas in the restricted sense that they need fewer variables in general.

The construction and proofs presented here first appeared in abbreviated form in a paper by the authors presented at IFIP Congress 1971 (Ljubliana, Yugoslavia).

1. Program schemas. A *program schema* consists of a finite sequence of *statements*, separated by semicolons. This sequence must start with a *start statement*, e.g., START(x_2, x_4), designating *input variables*, and end with a *halt statement*, e.g., HALT(x_1, x_3), designating *output variables*. The other statements may be of the following types:

(i) *null statements*, i.e., **null**;

(ii) *assignment statements*, i.e.,

$$x_i \leftarrow \tau,$$

where τ is a *term*;

(iii) *conditional statements*, i.e.,

$$\textbf{if } \psi \textbf{ then } S_1 \textbf{ else } S_2,$$

where S_1 and S_2 are statements and ψ is a *formula*;

(iv) *compound statements*, i.e.,

$$[S_1; S_2; \cdots; S_n],$$

where $S_1, S_2, \cdots, S_n$ are statements;

(v) *goto statements*, i.e.,

$$\textbf{goto } L_i,$$

where L_i is a *label*.

Any statement can be labeled by preceding it with a label followed by a colon. A *formula* is any quantifier-free formula of predicate calculus. A *term* is any composition of variables, constants and function symbols.

The statement **if** $\cdots$ **then** $\cdots$ **else null** can be written **if** $\cdots$ **then** $\cdots$ provided no confusion results.

Example. The following is a program schema P_1 with one variable, that will be used often throughout the paper:

```
SCHEMA $P_1$. START($x$);
        $x \leftarrow a(x)$;
        $L$: **if** $p(x)$ **then** [$x \leftarrow e(x)$; **goto** $L$];
        $A$: **if** $q(x)$ **then** $x \leftarrow b(x)$ **else** [$x \leftarrow g(x)$; **goto** $N$];
        $M$: **if** $r(x)$ **then** [$x \leftarrow d(x)$; **goto** $M$];
        $B$: **if** $s(x)$ **then** [$x \leftarrow c(x)$; **goto** $L$] **else** $x \leftarrow f(x)$;
        $N$: **null**;
        HALT($x$).
```

A, B, L, M and N are labels (A and B will be used in later discussions). The symbols a, b, c, d, e, f and g denote functions and the symbols p, q, r and s denote predicates or tests. The expressions $p(x)$, $q(x)$, etc. are (simple) formulas.

2. While-schemas. A *while-schema* is a program schema using only statements of types (i), (ii), (iii) and (iv) and type (vi) below:

(vi) *while statement*, i.e.,

while ψ **do** S,

where S is a statement and ψ is a formula.

Such statements are to be considered as abbreviations for the equivalent statements

L: **if** ψ **then** [S; **goto** L] **else null**.

Example. The following is a while-schema P_2 with two variables.

```
SCHEMA $P_2$. START($x$);
        $x \leftarrow a(x)$;
        **while** $p(x)$ **do** $x \leftarrow e(x)$;
        $y \leftarrow x$;
        **if** $q(x)$ **then** [$x \leftarrow b(x)$; **while** $r(x)$ **do** $x \leftarrow d(x)$];
        **while** $q(y) \wedge s(x)$ **do**
            [$x \leftarrow c(x)$;
            **while** $p(x)$ **do** $x \leftarrow e(x)$;
            $y \leftarrow x$;
            **if** $q(x)$ **then** [$x \leftarrow b(x)$; **while** $r(x)$ **do** $x \leftarrow d(x)$]];
        **if** $q(y)$ **then** $x \leftarrow f(x)$ **else** $x \leftarrow g(x)$;
        HALT($x$).
```

The schema P_2 uses the same symbols as P_1, denoting functions and predicates, and here we have the (more complicated) formula $q(y) \wedge s(x)$.

3. While-schemas with logical variables. A *while-schema with logical variables* is a program schema using only unlabeled statements of types (i), (ii), (iii), (iv) and (vi), and type (vii) below:

(vii) *logical assignment statements*, i.e.,

$t_i \leftarrow$ **true**

or

$t_i \leftarrow$ **false**.

The variables appearing in logical assignment statements are called *logical variables*, and they may not appear in ordinary assignments. They may appear, however, in formulas, as if they were *propositions*, i.e., 0-ary predicates.

Example. The following schema P_3 is a while-schema with one logical variable (and one program variable).

```
SCHEMA $P_3$. START($x$);
          $x \leftarrow a(x)$;
          $t \leftarrow$ true;
          while $t$ do
               [while $p(x)$ do $x \leftarrow e(x)$;
                if $q(x)$ then [$x \leftarrow b(x)$;
                               while $r(x)$ do $x \leftarrow d(x)$;
                               if $s(x)$ then $x \leftarrow c(x)$
                                         else [$x \leftarrow f(x)$; $t \leftarrow$ false]]
                          else [$x \leftarrow g(x)$; $t \leftarrow$ false]];
          HALT($x$).
```

P_3 uses the same symbols as P_1 and P_2 denoting functions and predicates, and here t is a logical variable used also as a formula.

4. Equivalence of schemas.

4. Equivalence of schemas. Two schemas having the same input variables and the same output variables are said to be *equivalent* if they compute the same function (from input variable values to output variable values), *no matter what functions or predicates are denoted by the symbols in the schema.* (Of course, the same symbol appearing in the two schemas must denote the same function or predicate.)

More formally, we can first give meaning to the symbols in a schema by using an interpretation. An *interpretation I* consists of a domain D_I from which the variables in the schema may take values and a specification of the functions and predicates over D_I denoted by the function and predicate symbols in the schema. The interpretation also supplies initial values (from D_I) for the input variables. Given an interpretation I, a schema S becomes a *program* (S, I). The program has a finite or infinite computation in the usual way, and if this is finite we let val(S, I) denote the final values of the output variables. If the computation is infinite, val(S, I) is undefined.

Two schemas S_1 and S_2 are then equivalent if, for all interpretations I, val(S_1, I) = val(S_2, I), i.e., both are undefined, or both are defined and have the same values.

In most of the paper we do not need the formal definition of equivalence. In these sections we will use simple equivalence-preserving transformations which are clearly correct. However, we do use the formal definition using interpretations in the last section.

Examples. The schemas P_1, P_2 and P_3 are all equivalent. (In fact, P_2 is the result of applying Algorithm 1 to P_1, and P_3 is the result of applying Algorithm 2 to P_1, as we will see later.

To see informally that P_1 is equivalent to P_2, note that each iteration of the main while statement in P_2 corresponds in P_1 to going from label B back to label B. The variable y in P_2, at the beginning of each iteration, holds the value that x previously held in P_1, the last time computation reached label A.

To see informally that P_1 is equivalent to P_3, note that each iteration of the main while statement in P_3 corresponds in P_1 to going from label L back to label L (the long way, via statement labeled B) or to label N. In the latter case, t is made **false** in P_3, and we subsequently exit from the main while statement.

5. Flowcharts. We will find it useful to consider the flowchart representations of schemas. Program schemas clearly correspond to arbitrary flowcharts, with one start node and one halt node, using assignment and test statements as shown in Fig. 1; ψ is a formula and τ is a term.

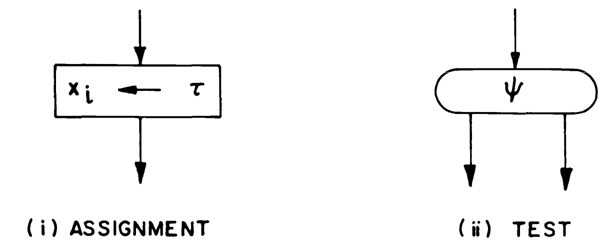


FIG. 1. *Flowchart statements*

We shall be more concerned with *normal forms* for such flowcharts.

5.1. While-chart form. Firstly, it is clear that while-schemas have more restricted "structure" than program schemas, and we define below (inductively) a correspondingly restricted class of flowcharts.

A *while-chart* is a *one-entrance*, *one-exit* piece of flowchart of one of the five types shown in Fig. 2. The boxes $A_1, A_2, \cdots$ represent while-charts, ψ is a formula and τ is a term. The various cases correspond to the types of statement allowed in while-schemas. Any flowchart of the form of Fig. 3, where A is a while-chart, is said to be in *while-chart form*. For every while-chart form flowchart, there is an equivalent while-schema, and vice versa.

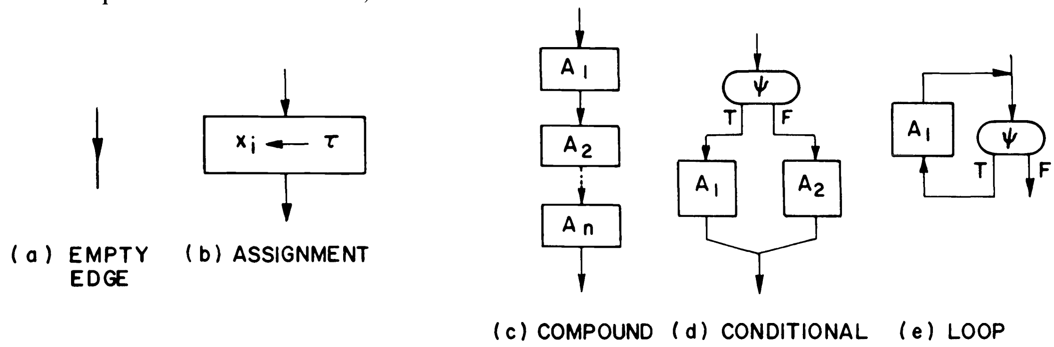


FIG. 2. *While chart constructs*

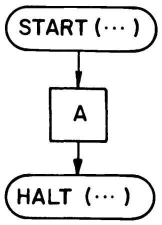


FIG. 3. *While-chart form flowchart*

5.2. Block form. Even general flowcharts can be put into normal forms, by such methods as duplicating nodes, unwinding loops, etc. One such normal form is the block form of Cooper [7] and Engeler [9].

A *block* is a *one-entrance, many-exit* piece of flowchart constructed inductively as follows (we occasionally number the exits from a block, starting at the left):

(i) A *basic block* is a block. A basic block is a one-entrance, many-exit *tree-like* piece of flowchart. An example is shown in Fig. 4.

(ii) The flowcharts in Fig. 5 are blocks, where B_1 and B_2 are blocks.

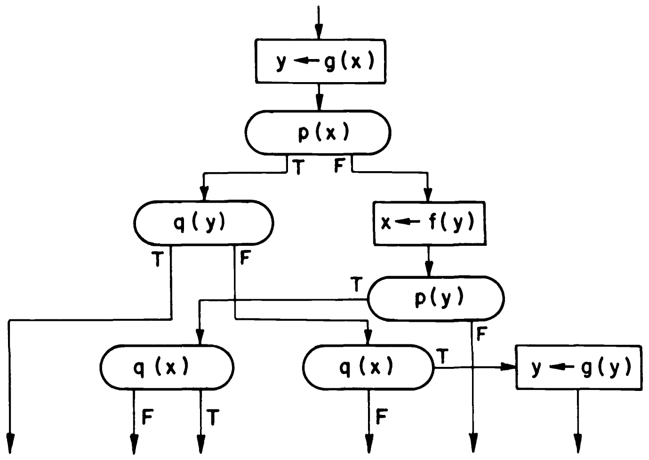


FIG. 4. *Example of a basic block*

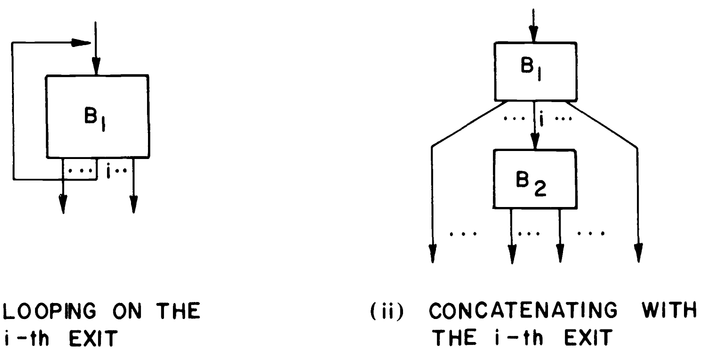

B1
... i ..
B1
... i ...
B2
...
...
...
(i) LOOPING ON THE i-th EXIT
(ii) CONCATENATING WITH THE i-th EXIT

FIG. 5. *Nonbasic constructs for blocks*

A flowchart is in *block form* if it is of the form shown in Fig. 6, where B is a block.

Clearly every flowchart in block form is equivalent to some program schema. The result of Engeler and Cooper is that for every program schema we can find an equivalent flowchart in block form (see for example, Manna [14]).

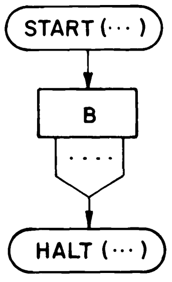


FIG. 6. *Block form flowchart*

Example. Figure 7 shows a flow chart P'_1 in block form which is equivalent to the program schema P_1. The blocks are indicated by broken lines. B_0, B_1 and B_3 are basic blocks. B_2, B_4 and B_6 are constructed by looping, and B_5 and B_7 by concatenation.

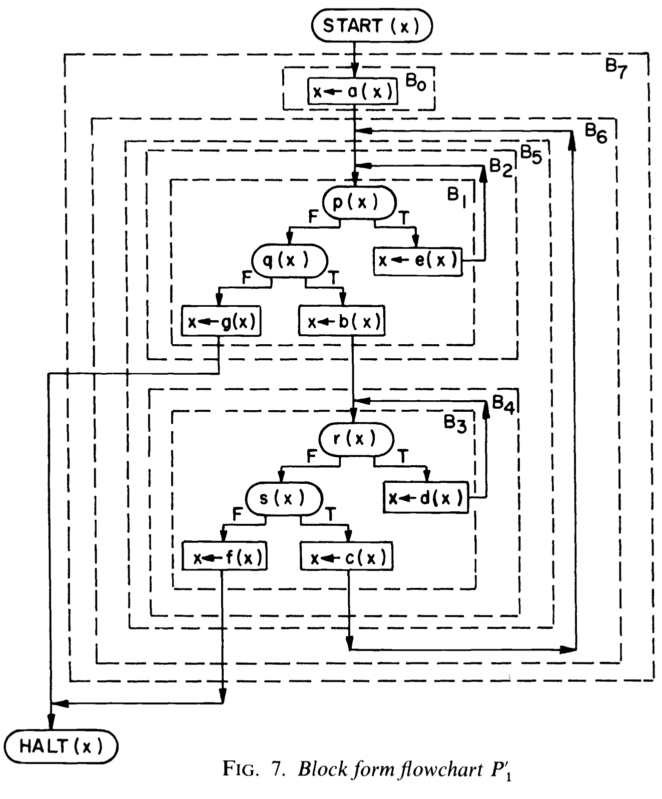


FIG. 7. *Block form flowchart* P'_1

5.3. Properties of basic blocks. Before we consider our next normal form, we observe two useful properties of basic blocks.

Property 1. Given any basic block B (with n exits) and some ith exit of B, there exist a formula i-test(B), a basic block i-pruned(B) (with $n - 1$ exits) and a sequence of assignment statements i-ops(B) such that the flowcharts in Fig. 8 are equivalent.

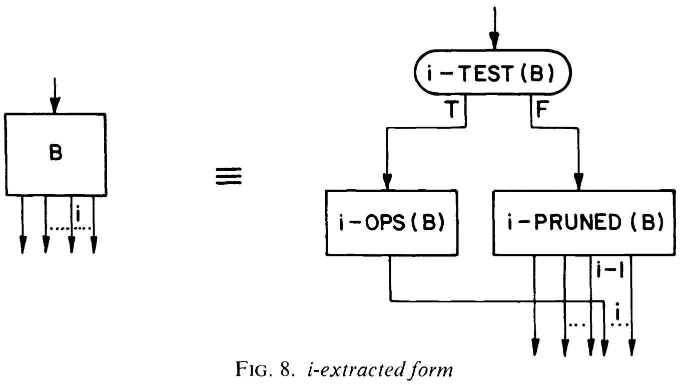


FIG. 8. *i-extracted form*

To see this, note first that the basic block can be put into a form in which the tests on the path on the *i*th exit precede the assignments, by repeated application of the transformation shown in Fig. 9, where ψ' is like ψ but with x_i replaced by τ. It is then a simple matter to find a single test to "extract" the *i*th path by working up the path from the bottom, repeatedly applying the transformations shown in Fig. 10 (or their mirror images). The upper transformation extracts the first path,

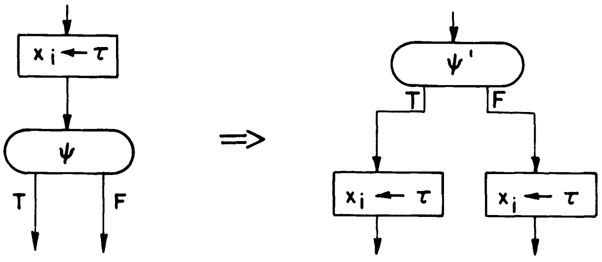


FIG. 9. *Moving tests up*

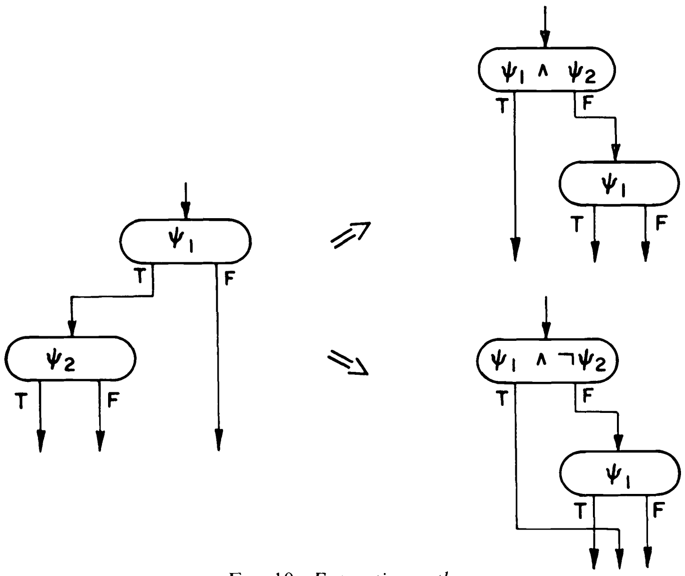


FIG. 10. *Extracting paths*

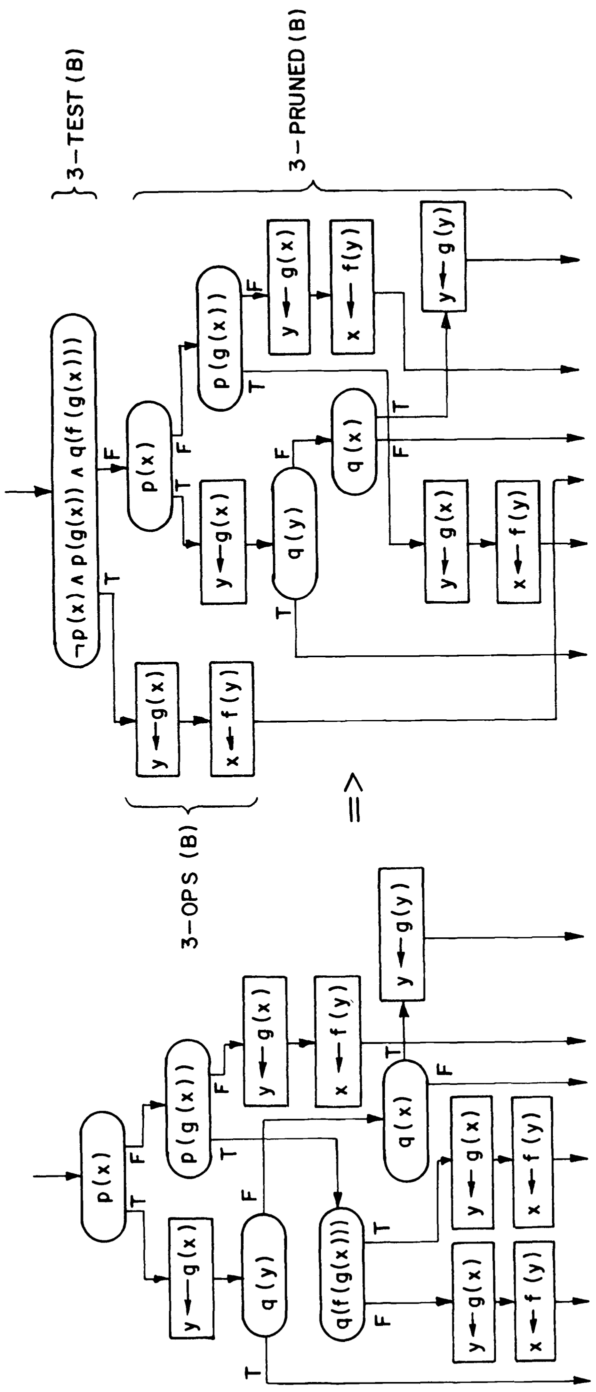


FIG. 11. *The 3-extracted form of the basic block of Fig. 4*

and the lower transformation extracts the second path (the third path is already extracted).

This eventually gives us the desired form for the basic block. It will be called the *i-extracted form.*

Example. Figure 11 shows the two stages in obtaining the 3-extracted form of the basic block B of Fig. 4.

Property 2. The second property of basic blocks that we need is that every piece of flowchart of the form shown in Fig. 12(a), where B is a basic block, is a while-chart. This can be seen very easily by induction on the number of statements in B. If there are no statements, we have an empty edge, which is a while-chart. If there are $n > 0$ statements, we have either Fig. 12(b) or Fig. 12(c), where B_1 and B_2 are basic blocks. In both cases we have while-charts, since the lower parts are while-charts by the induction hypothesis.

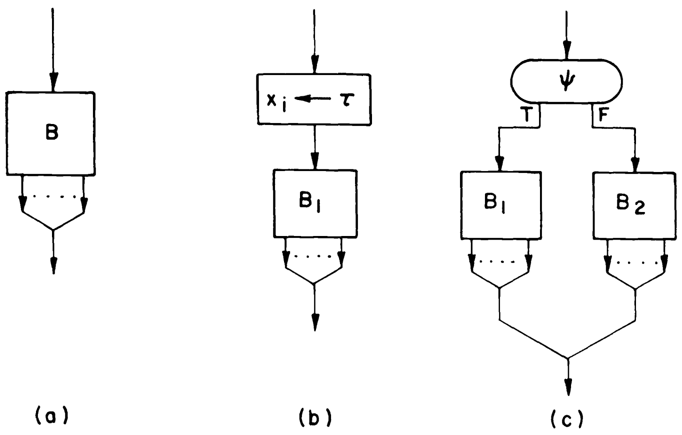


FIG. 12

5.4. Module form. The final normal form for flowcharts which we will consider is module form.

A *module* is either

(i) an assignment statement, or

(ii) a *one-entrance, one-exit* piece of flowchart constructed from modules and tests; these tests are called the tests of the module.

A flowchart is in *module form* if it is of the form shown in Fig. 13, where M is a module.

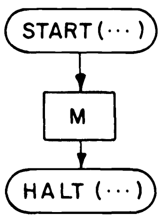


FIG. 13. *Module form flowchart*

This definition may appear surprising since we immediately have that *any* flowchart is in module form, by taking each assignment statement as a module, at one level, and then taking the whole flowchart as a module at the next level. However, we can find an interesting subclass of module form flowcharts:

A *simple module* is either

(i) an assignment statement, or

(ii) a *one-entrance, one-exit* piece of flowchart constructed from modules and *at most one test*.

A flowchart is in *simple module form* if it is in module form and each module is simple.

A simple module either has no tests (and is thus either an empty edge, an assignment statement or the concatenation of modules) or it has one test and can only be of the forms shown in Fig. 14, where A, B, C and D are modules.

The analogy with while-schemas is obvious:

Fig. 14(a) is equivalent to $[C$; **if** ψ **then** A **else** B; $D]$;

Fig. 14(b) is equivalent to $[C$; B; **while** ψ **do** $[A; B]$; $D]$.

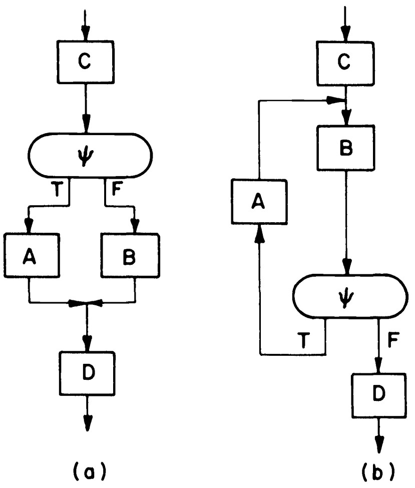


FIG. 14. *Simple modules*

In general, for every flowchart in simple module form there is an equivalent while-schema, and vice versa.

The motivation for module form now becomes clear. At one extreme we can take any flowchart as a module whose submodules are simply assignment statements. If, however, by ingenuity and equivalence-preserving transformations we can get many levels of modules, with fewer tests per module, then we get closer to simple module form and hence closer to while-schemas.

Example. In Fig. 15 we give a module form flowchart P_1'' for the program schema P_1. Modules M_1 and M_2 are simple, but module M_3 is nonsimple since it contains two tests $q(x)$ and $s(x)$.

6. Algorithm 1. To translate program schemas to while-schemas it suffices to consider flowcharts in block form. We show how to transform each block B into an equivalent piece of flowchart consisting of a while-chart W_B followed by a

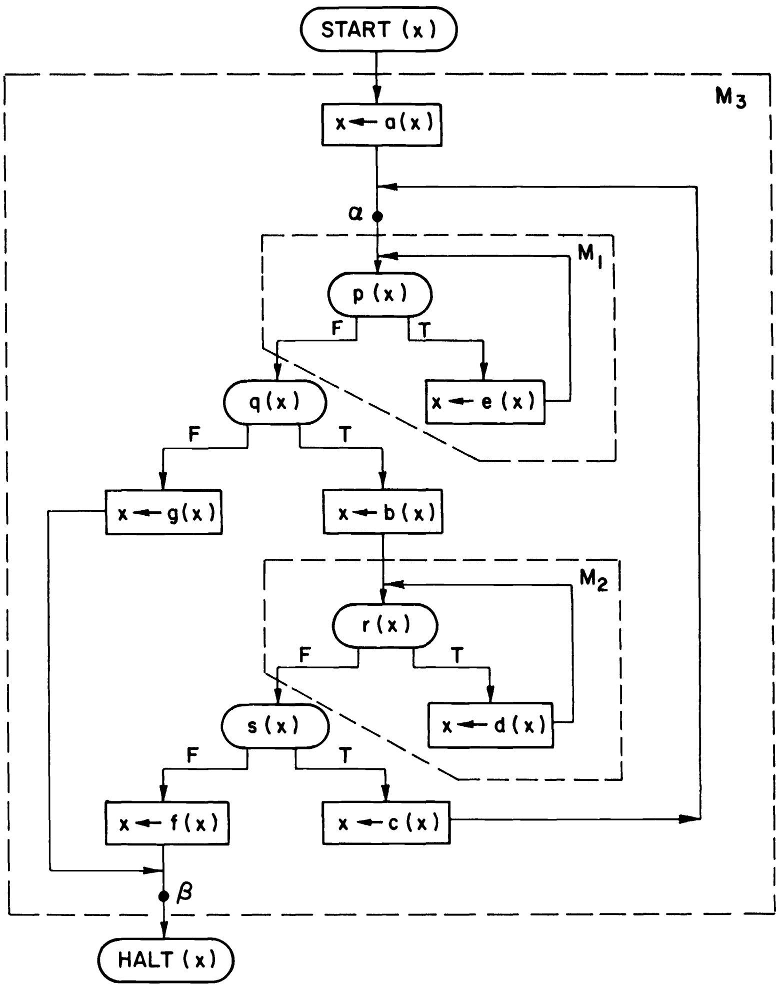


FIG. 15. *Module form flowchart* P_1''

basic block $\bar{B}$. We do this by induction on the block structure as follows:

(i) B is a basic block: we use the transformation of Fig. 16.

(ii) B is constructed by looping on the ith exit of B_1: we first transform B_1; and then extract the ith path of $\bar{B}_1$, as shown in Fig. 17.

(iii) B is the concatenation of B_1 and B_2, using the ith exit of B_1: we first transform B_1 and B_2, and extract the ith path of $\bar{B}_1$. It is then possible to move up W_{B_2} past i-pruned($\bar{B}_1$), as shown in Fig. 18.

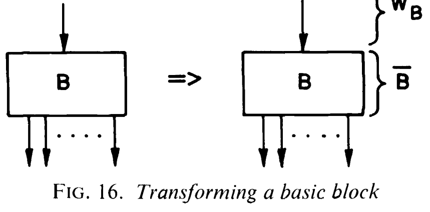


FIG. 16. *Transforming a basic block*

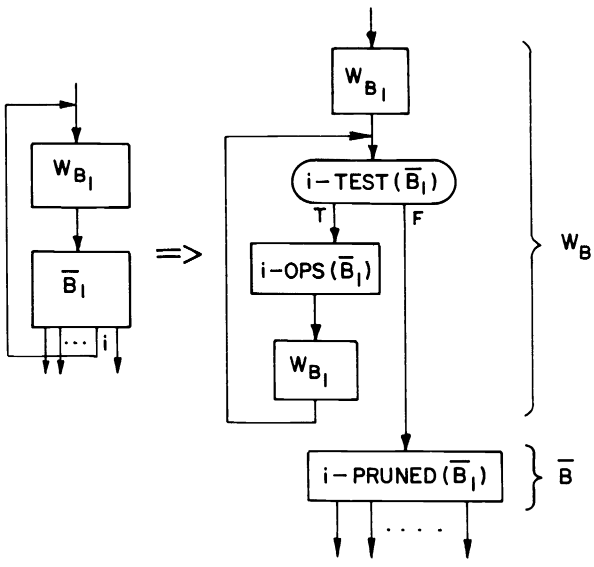


FIG. 17. *Transforming a looping block*

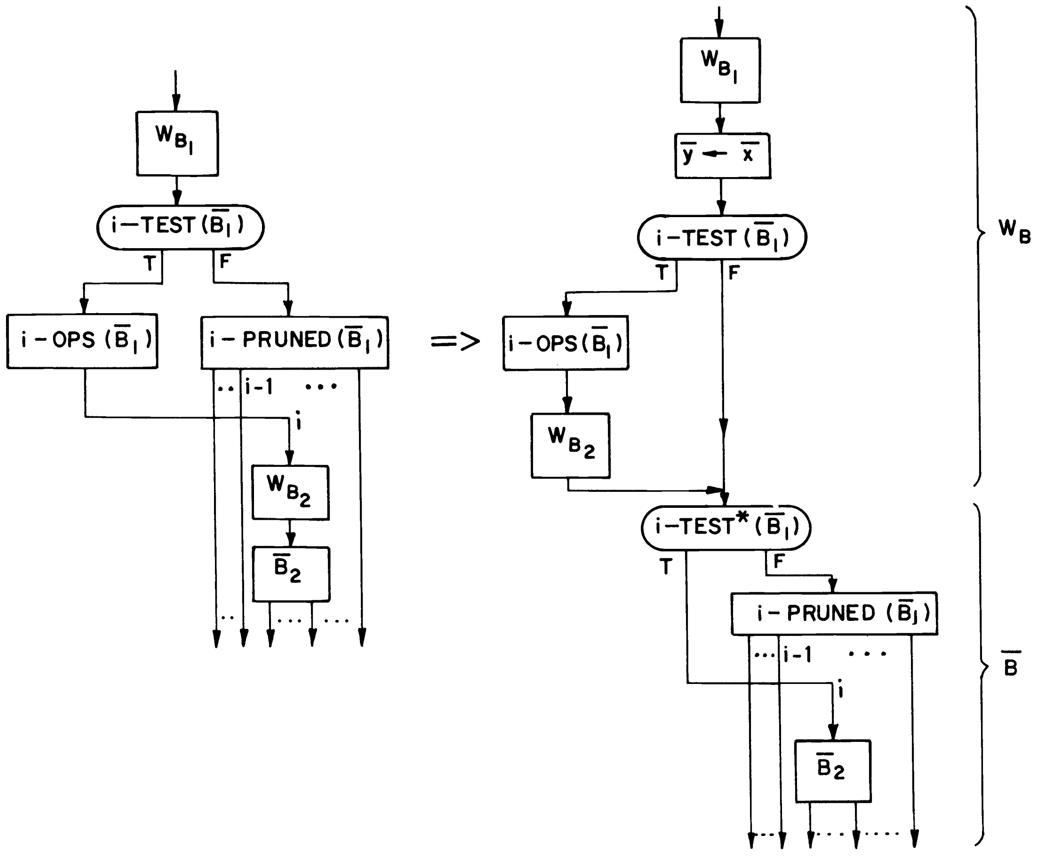


FIG. 18. *Transforming a concatenation block (general case)*

In Fig. 18, $\bar{x}$ is a vector of the variables occurring in i-test($\bar{B}_1$); $\bar{y}$ is a vector of the same length of *new* variables; and i-test*($\bar{B}_1$) is the same as i-test($\bar{B}_1$) but with variables $\bar{x}$ replaced by $\bar{y}$, so that any computation must take the same branch out of the two tests.

Note. If B_2 is a basic block, we can simply make the transformation shown in Fig. 19. No new variables are needed in this case.

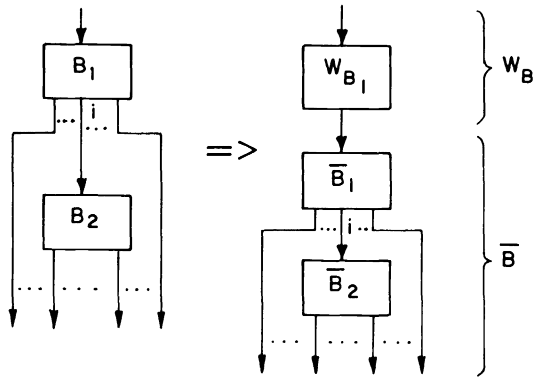


FIG. 19. *Transforming a concatenation block, B_2 basic*

Thus for every block form flowchart, as in Fig. 6, we get a flowchart as in Fig. 20. This flowchart is in while-chart form, by the second property of basic blocks proved earlier, and thus can be simply written as a while-schema.

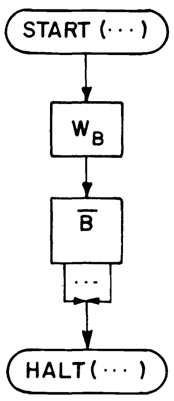


FIG. 20. *Transformed flowchart*

Example. We take flowchart P'_1 of Fig. 7. Blocks B_2 and B_4 are already of the required form; for example, B_2 is shown in Fig. 21. (This is the decomposition that Algorithm 1 yields for looping on the third exit of B_1). The transformed version of B_5 is shown in Fig. 22; note that $q(x)$ comes from 2-test($\bar{B}_2$), $x \leftarrow b(x)$ comes from 2-ops ($\bar{B}_2$) and $x \leftarrow g(x)$ comes from 2-pruned($\bar{B}_2$). The transformed version of B_6 is shown in Fig. 23; note that $q(y) \wedge s(x)$ comes from 3-test($\bar{B}_5$), $x \leftarrow c(x)$ comes from 3-ops($\bar{B}_5$) and $\bar{B}_6$ is simply 3-pruned($\bar{B}_5$). The final while-chart form flowchart is shown in Fig. 24, and it corresponds exactly to while-schema P_2.

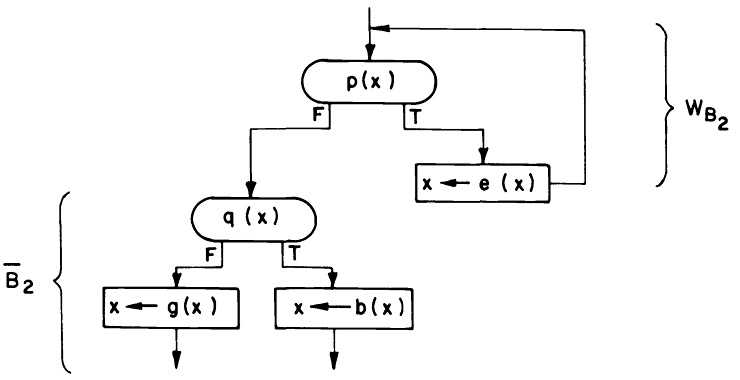


FIG. 21. B_2

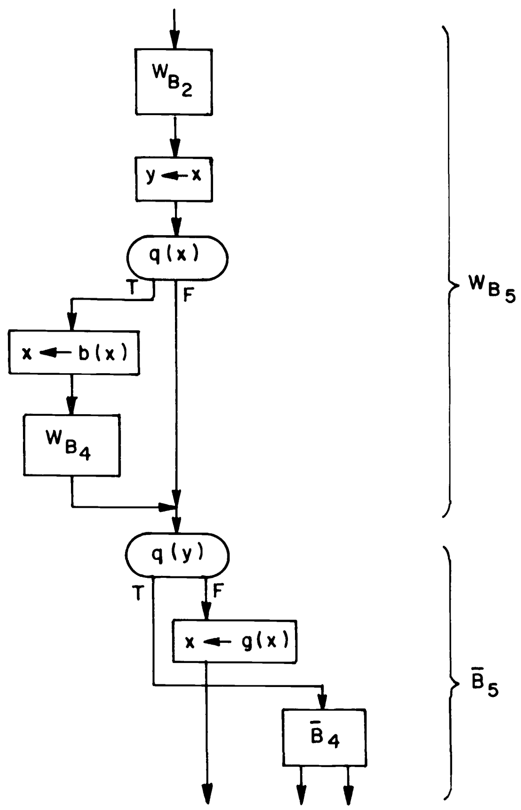


FIG. 22. B_5 *(transformed)*

Comments. (i) Putting a flowchart into block form in general requires some increase in the size of the flowchart. This can be avoided by allowing the exits of any block to be joined together in arbitrary ways and still be a block. In the same spirit we would allow basic blocks that were not tree-like but merely loop-free. Algorithm 1 will work just as well for such block form. The only change needed is in defining the i-extracted form for basic blocks; for example, i-ops(B) becomes a

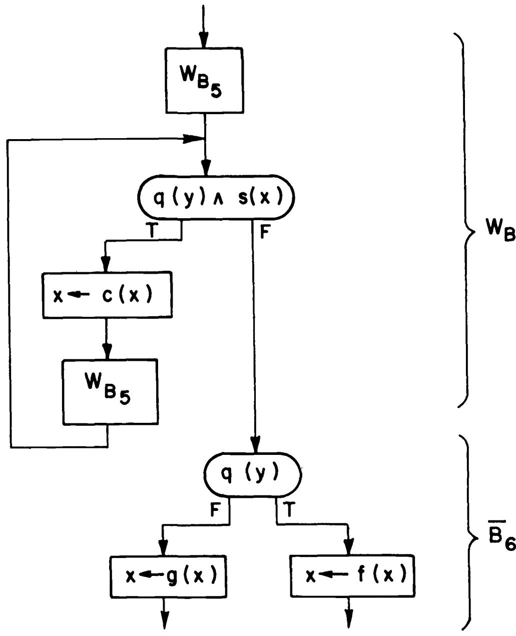


FIG. 23. B_6 (*transformed*)

one-exit basic block rather than a sequence of assignment statements. This modified block form corresponds to interval analysis (see [1]).

(ii) To minimize the number of new variables added by Algorithm 1, we must find block form flowcharts which avoid concatenating blocks except when the second block is basic. Even for while-schemas it is not clear how to do this, and so Algorithm 1 is not an identity mapping on while-schemas. We could avoid this by allowing while-charts to be special cases of blocks. The algorithm is easily modified to deal with this.

(iii) The duplication of W_{B_1} produced by transforming looping blocks (Fig. 17) could be avoided by using a new control construct **repeat** $[S_1$; **exit on** ψ ; $S_2]$ instead of while statements.

7. Algorithm 2. The idea of Bohm and Jacopini's translation of program schemas to while-schemas with logical variables [3] (see also [6]) can be expressed as follows. Suppose the given program schema has n statements including the halt statement, numbered, for our convenience, 1 to n. We construct a while-schema using k additional logical variables, where $2^{k-1} < n \leqq 2^k$. Each statement of the original program schema then corresponds to a particular pattern of values for the k logical variables, e.g., the number of the statement written in binary notation. The while-schema consists of a single while statement, the formula of which will be true provided the "pattern" of logical variable values does *not* correspond to the halt statement. If the formula is true, we enter the body of the while statement, where a series of tests decides to which statement in the program schema the logical variable values correspond. The operation of that statement is then performed, and the values of the logical variables are changed so that their "pattern" corresponds to the next statement to be executed in the program schema. The body of the while statement is repeatedly executed, until we reach the pattern

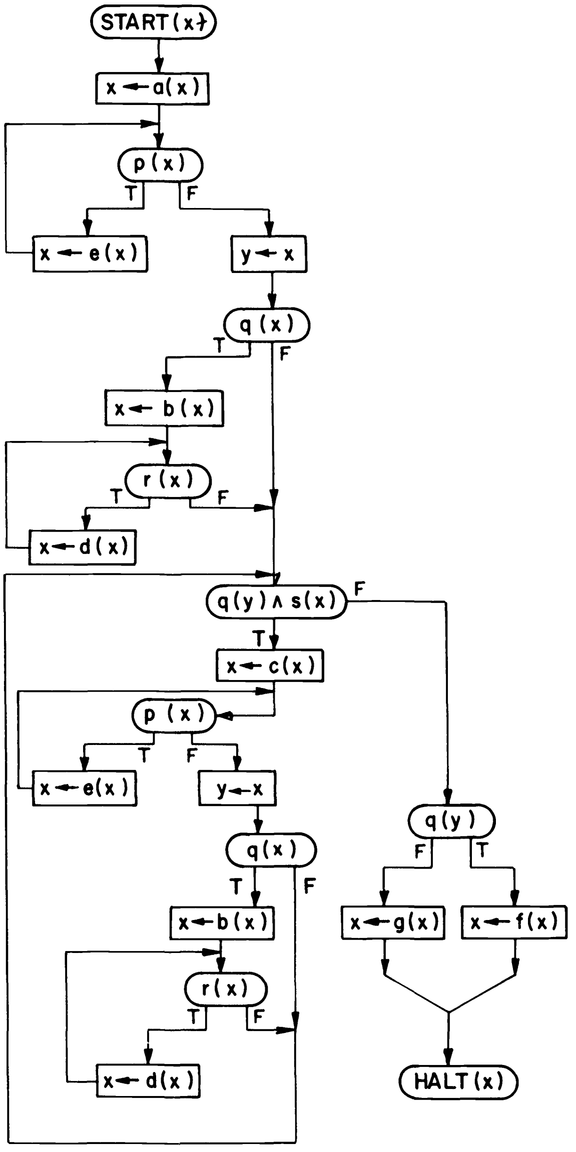


FIG. 24. *Transformed version of* P_1''

for the halt statement in the program schema. When this happens, we exit from the while statement, and reach the halt statement of the while-schema.

The while statement simply acts as a one-loop interpreter, performing one operation of the original program schema on each iteration. The logical variables simply represent a "program counter".

An improvement upon this method, due to Cooper [private communication], reduces the number of logical variables required. We take the flowchart representation of the program schema and choose a "cut set" of the edges between the assignment and test statements from which it is composed, i.e., we choose at least one edge per loop. We add the edge leading to the halt statement to this set. These edges are then numbered, and coded up as logical variable value patterns as before. The while statement "interpreter", on each iteration, now performs the operations of the original program schema from one cut set edge to the next cut set edge, and updates the logical variables accordingly. This technique is used in Algorithm 2 below.

We consider flowcharts in module form, and, for good results, we try to get as many simple modules as possible. We then translate each module M into a statement of while-schema W_M by induction on the module structure.

(i) If M is an assignment statement, W_M is that assignment statement.

(ii) If M is a simple module, W_M is the corresponding statement of while-schema (see Fig. 14).

(iii) If M is a nonsimple module, then we apply Cooper's version of the Bohm and Jacopini reduction. We choose a cut set of the edges between modules and tests comprising the module M, and add the single exit edge of M. We then take sufficient "new" logical variables to represent these positions in M, and construct a statement of while-schema. This statement will be a compound statement $[S_1; S_2]$. Statement S_1 will perform the operations from the entrance of M up to the first cut set edge, and set the logical variables to correspond to that edge. Statement S_2 is then the while statement which "interprets" the module M. Its formula checks that the current pattern of logical variable values does *not* correspond to the exit edge. The body determines the current cut set edge, performs the operations to the next cut set edge (using the while-schema statements corresponding to the modules of which M is composed) and updates the logical variable values accordingly. This is possible as a statement of while-schema since the use of a *cut set* of edges ensures that there is a bound on the number of tests and modules that can be performed between one cut set edge and the next.

Example. The modules of flow chart P_1'' (Fig. 15) correspond to statements of while-schema as follows: M_1 and M_2 are simple modules and correspond to

while $p(x)$ **do** $x \leftarrow e(x)$

and

while $r(x)$ **do** $x \leftarrow d(x)$,

respectively. M_3 is nonsimple, so we choose cut set edges, for example α and β in Fig. 15 (there is only one loop in M_3 and we must add the exit edge of M_3). We then need one logical variable t, say, to keep track of the cut set edge—**true** corresponds to α, **false** corresponds to β. W_{M_3} is then the following statement of while-schema

```
[[$x \leftarrow a(x)$; $t \leftarrow$ true];
  while $t$ do [$W_{M_1}$;
                if $q(x)$ then [$x \leftarrow b(x)$;
                                $W_{M_2}$;
                                if $s(x)$ then $x \leftarrow c(x)$
                                          else [$x \leftarrow f(x)$;
                                                $t \leftarrow$ false]]
                          else [$x \leftarrow g(x)$; $t \leftarrow$ false]]].
```

Enclosing W_{M_3} between start and halt statements then gives us the while-schema P_3.

Comment. No reasonable algorithm is known for finding the *optimal* equivalent module form for a program schema, optimal in the sense that Algorithm 2 adds the smallest number of logical variables. However, it is clear that the flow charts of while-schemas (i.e., while-charts) are in simple module form, so that Algorithm 2 is the identity mapping on while-schemas.

8. The necessity of adding variables.

We show that any translation from program schemas to while-schemas must in general add variables. We prove that for a particular one-variable program schema there is no equivalent while-schema that also uses only one variable.

Similar results have been demonstrated by several authors: Knuth and Floyd [11], Scott [private communication] and Kosaraju [12], for example. However, these results are weaker than ours, either because the notion of equivalence used is more restrictive than ours, requiring the equivalence of computation sequences (i.e., the sequences of assignments and tests in order of execution) and not just the equivalence of final results, or because complex formulas are not allowed in while statements. For example, the following program schema has no "equivalent" while-schema if we consider execution sequences, or disallow compound tests:

```
START($x$);
$x \leftarrow a(x)$;
$L$: **if** $p(x)$ **then** [$x \leftarrow b(x)$; **if** $q(x)$ **then** [$x \leftarrow c(x)$; **go to** $L$]
                                          **else** $x \leftarrow d(x)$]
            **else** $x \leftarrow e(x)$;
HALT($x$).
```

However, if we apply Algorithm 1, we get an equivalent while-schema, which happens to use only one variable:

```
START($x$);
$x \leftarrow a(x)$;
**while** $p(x) \wedge q(b(x))$ **do** $x \leftarrow c(b(x))$;
**if** $p(x)$ **then** $x \leftarrow d(b(x))$ **else** $x \leftarrow e(x)$;
HALT($x$).
```

Since our result is stronger than the previous results, it needs a more complicated program schema to demonstrate it. The one we use is the following program schema P_5.

```
SCHEMA $P_5$. START($x$);
        $L$: **if** $p(x)$ **then** [$x \leftarrow e(x)$; **go to** $L$];
             **if** $q(x)$ **then** $x \leftarrow e(x)$ **else** [$x \leftarrow e(x)$; **go to** $N$];
        $M$: **if** $q(x)$ **then** [$x \leftarrow d(x)$; **go to** $M$];
             **if** $p(x)$ **then** [$x \leftarrow d(x)$; **go to** $L$] **else** $x \leftarrow d(x)$;
        $N$: **null**;
        HALT($x$).
```

This schema is similar to P_1, but is simpler since it only uses two functions and two predicates. It is especially interesting because for most other simpler versions of P_1 there *are* equivalent one-variable while schemas. For example, the program schema

```
START($x$);
$x \leftarrow a(x)$;
$L$: **if** $p(x)$ **then** [$x \leftarrow e(x)$; **go to** $L$];
    **if** $q(x)$ **then** $x \leftarrow b(x)$ **else** [$x \leftarrow g(x)$; **go to** $N$];
$M$: **if** $q(x)$ **then** [$x \leftarrow b(x)$; **go to** $M$]; .
    **if** $s(x)$ **then** [$x \leftarrow c(x)$; **go to** $L$] **else** $x \leftarrow f(x)$;
$N$: **null**;
HALT($x$)
```

is equivalent to the following one-variable while-schema:

```
START($x$);
$x \leftarrow a(x)$;
**while** $p(x)$ **do** $x \leftarrow e(x)$;
**while** $q(x) \wedge q(b(x))$ **do** $x \leftarrow b(x)$·
**while** $q(x) \wedge s(b(x))$ **do**
        [$x \leftarrow c(b(x))$;
         **while** $p(x)$ **do** $x \leftarrow e(x)$;
         **while** $q(x) \wedge q(b(x))$ **do** $x \leftarrow b(x)$];
**if** $q(x)$ **then** $x \leftarrow f(b(x))$ **else** $x \leftarrow g(x)$;
HALT($x$).
```

Our proof that there is no one-variable while-schema P'_5 equivalent to P_5 must therefore depend crucially on special features of P_5. The essential property of P_5 is the following:

In any unfinished computation of P_5, if p is true and q is false, then the *next-but-one* function that will be applied is e, whereas if q is true and p is false, then the *next-but-one* function that will be applied is d. If both p and q are false, the computation will terminate after applying one more function.

Let d, e and h be symbols and $D = \{d, e\}^*$. We shall consider the interpretations I_z, where $z \in D \cdot h \cdot D$, defined as follows:

(i) $D_{I_z} = D$

(ii) for $y \in D_{I_z}$, $d(y) = yd$,
$e(y) = ye$,
$p(y) \equiv [|y| < |z| \wedge z(|y| + 1) = e]$[1],
$q(y) \equiv [|y| < |z| \wedge z(|y| + 1) = d]$,

(iii) the initial value of the input variable x is Λ, the empty string.[2]

Note that the predicates p and q are mutually exclusive, and from the essential property of P_5, the computation of (P_5, I_z), where $z = uhv$ ($u, v \in D$), must terminate with val$(P_5, I_z) = eu$ (the symbol h makes both p and q false and makes the computation halt). Also for any interpretation $I_{w\alpha uhv}$ ($\alpha \in \{d, e\}, w, u, v \in D$), when the value of x becomes ew, the future course of the computation is determined by uhv, since this substring will determine the possible future values of the predicates p and q. This property also holds for any *one-variable* schema P'_5 equivalent to P_5 (it will be called the main property of P'_5), and will be used to show that such a schema cannot exist.

Let us assume therefore that there exists a one-variable while-schema P'_5

[1] Here $|y|$ denotes the length of string y; $z(i)$ denotes the ith symbol in string z.

[2] This is a "Herbrand" or "free" interpretation (see [13]).

equivalent to P_5. Without loss of generality we can assume there is some while statement S in P'_5, say **while** ψ' **do** S_1, which is not contained in or followed by any other while statement, and for which S_1 is executed in the computation for some I_z. We can also assume that there is no bound on the number of iterations of S for computations for such interpretations I_z. (All such bounded while statements could be "unwound" the corresponding number of times, leaving only "unbounded" while statements and while statements never entered for any I_z.)

Let the maximum "depth" of functional composition in any formula in P'_5 be M. Then in computation of (P'_5, I_z), if we evaluate a formula ψ for value w of variable x, then the outcome of ψ is determined by $z(|w| + 1)$, $z(|w| + 2)$, $\cdots$, $z(|w| + M + 2)$. We define visible(z, w) as this substring of z starting at $z(|w| + 1)$ and ending at $z(|w| + M + 2)$.

LEMMA. *For all $n \geqq 0$ there exist strings u, w, $y \in D$, $|w| = n$, such that for all $v \in D$, the computation of (P'_5, I_{uvwhy}) exits from S with a proper prefix of euv as the value of variable x.*

This technical lemma has the following informal corollary.

COROLLARY. *For every $n \geqq 0$ there exists a computation of P'_5 which exits from S with more than n functions still to be applied.*

This corollary contradicts the fact that S is not followed by or contained in another while-statement; the number of functions that can be applied after exiting from S is bounded. Hence while-schema P'_5 cannot exist.

Proof of lemma. The proof is by induction on n.

Base step $(n = 0)$. Since S is unbounded, there exists an interpretation $I_{z'}$ whose computation enters S_1 before the end of the computation is "visible", i.e., more than M function applications from the end. In other words, $z' = u'\alpha yv'hy'$ where $u', y, v', y' \in D$, $\alpha \in \{d, e\}$ and $|y| = M + 1$, and the computation of $(P'_5, I_{z'})$ reaches S with eu' as the value of x. Moreover, since S_1 is entered, the formula ψ' must be true for this value of x. Note that the truth of ψ' is determined by visible$(z', eu') = y$.

Consider now the interpretation $I_z = I_{uvhy}$, where $u = u'\alpha y$ and v is any string from D. The computation must reach S as for $I_{z'}$, i.e., with value eu' for variable x, since the changes in the interpretation are not "visible" by this point. However, when it subsequently exits from S, it can not do so with value euv (the final value) for x, since visible$(z, euv) = y$, and for this value the formula ψ' must be true. Thus it must exit from S with a *proper prefix* of euv as the value of x.

Induction step. Assume we have strings $u, w, y \in D$, with $|w| = n$, such that for all $v \in D$, the computation of (P'_5, I_{uvwhy}) exits from S with a proper prefix of euv as the value of x.

We shall find a string $w' \in D$, $|w'| = n + 1$, such that for all $v' \in D$, the computation of $(P'_5, I_{uv'w'hy})$ exits from S with a proper prefix of euv' as the value of x.

There are three cases to consider (in order):

(i) For all $v = v'e$ (for all $v' \in D$) in the induction hypothesis, the corresponding proper prefix of euv is also a proper prefix of euv'. In this case we take $w' = ew$.

(ii) For all $v = v'd$ (for all $v' \in D$) in the induction hypothesis, the corresponding proper prefix of euv is also a proper prefix of euv'. In this case we take $w' = dw$.

(iii) For some $v = v''e$ in the induction hypothesis, the corresponding proper prefix of euv is euv'', i.e., the computation C of $(P'_5, I_{uv''ewhy})$ exits from S with value euv'' for variable x. Note that the rest of the computation adds ew to the value of x.

Consider now the interpretations $I_{uv'dwhy}$, for all $v' \in D$. By the induction hypothesis, the value of x on exiting from S must in each case be a proper prefix of $euv'd$. But the main property of P'_5 ensures that in no case can this value of x be euv', otherwise the future course of this computation, being determined by why, would be the same as for C, giving x a final value of $euv'ew$ instead of $euv'dw$. Thus with $w' = dw$, the computations of $(P'_5, I_{uv'w'hy})$ (for all $v' \in D$) exit from S with a proper prefix of euv' as the value of x. Q.E.D.

Acknowledgments. We are indebted to David Cooper for stimulating discussions and for his modification of the Bohm and Jacopini's reduction, which we have used in Algorithm 2. We are also grateful to Donald Knuth for his critical reading of an earlier version of this paper, and subsequent helpful suggestions.

REFERENCES

[1] F. E. Allen, *A basis for program optimization*, Proc. IFIP Congress, Ljubliana, Yugoslavia, 1971.

[2] S. Brown, D. Gries and T. Szymanski, *Program schemas with pushdown stores*, this Journal, 1 (1972), pp. 242–268.

[3] C. Bohm and G. Jacopini, *Flow diagrams, Turing machines and languages with only two formation rules*, Comm. ACM, 9 (1966), pp. 366–371.

[4] A. K. Chandra, *On the properties and applications of program schemas*, Ph.D. thesis, Computer Science Dept., Stanford Univ., Stanford, Calif., 1973.

[5] R. L. Constable and D. Gries, *On classes of program schemata*, this Journal, 1 (1972), pp. 66–118.

[6] D. C. Cooper, *Bohm and Jacopini's reduction of flowcharts*, letter to the Editor, Comm. ACM, 10 (1967), p. 463, p. 473.

[7] ———, *Programs for mechanical program verification*, Machine Intelligence 6, Edinburgh Univ. Press, 1970.

[8] E. Dijkstra, *Goto statement considered harmful*, Comm. ACM, 11 (1968), pp. 147–148.

[9] E. Engeler, *Structure and Meaning of Elementary Programs*, Symp. on Semantics of Algorithmic Languages, Springer-Verlag, Berlin, 1971.

[10] C. A. R. Hoare, *An axiomatic approach to computer programming*, Comm. ACM, 12 (1969), pp. 576–580, p. 583.

[11] D. E. Knuth and R. W. Floyd, *Notes on avoiding goto statements*, Information Processing Letters, 1 (1971), pp. 23–31.

[12] S. Kosaraju, *Analysis of structured programs*, Proc. of 5th SIGACT Conf., 1973, pp. 240–252.

[13] D. Luckham, D. Park and M. Paterson, *On formalized computer programs*. J. Comput. Systems Sci., 4 (1970), pp. 220–249.

[14] Z. Manna, *Introduction to Mathematical Theory of Computation*, McGraw-Hill, New York, 1974.

[15] M. Paterson and C. Hewitt, *Comparative schematology*, Conf. Record of Project MAC Conf. on Concurrent Systems and Parallel Computation, ACM, New York, 1970.

SIAM J. COMPUT.
Vol. 4, No. 2, June 1975

TIME BOUNDS ON THE PARALLEL EVALUATION OF ARITHMETIC EXPRESSIONS*

D. J. KUCK† AND K. MARUYAMA‡

Abstract. This paper presents a number of bounds on the parallel processor evaluation of arithmetic expressions. Several previous papers show that if the evaluation of an expression using a serial computer requires t operations, by using a number of processors in parallel, the expression may be evaluated in time proportional to $\log_2 t$. Since $\log_2 t$ is an obvious lower bound, it is of interest to attempt to approach this bound.

The present paper shows that if more information than the number of operations (or operands) is known, sharper bounds may be given in certain cases. Thus if the number of parenthesis pairs is small or if the depth of parenthesis nesting is small, we may approach the lower bound. A new bound is also given for expressions which have few division operations.

Similarly, if the expression's form is restricted, sharper bounds may be found. Thus generalizations of polynomials and generalizations of continued fractions are shown to have improved bounds. We also give a new bound for expressions without division operations which have a limited number of parenthesis pairs. Finally, we give an upper bound on the time to evaluate expressions in which multiplication is not commutative.

Key words. arithmetic expressions, computational complexity, continued fraction, parallel evaluation, polynomial, processing time, tree height reduction, upper bound

1. Introduction.

Many computers now exist which are capable of executing more than one operation simultaneously. As parallel and pipeline processors continue to be developed, the question of how fast an arithmetic expression can be evaluated becomes more interesting. The problem of tree height reduction for the fast evaluation of an arithmetic expression has been studied by a number of people.

The goal of a number of papers has been to present tree height reduction algorithms which provide substantial speedup of expression evaluation. Recently several papers have included upper bounds on the number of steps required to evaluate the transformed expression. The three main transformation techniques used are the laws of associativity, commutativity and distributivity. Early papers concentrated on the first two of these. Later papers have used all three to obtain upper bounds which are fairly close to the lower bound. By a simple fan-in argument, it is clear that a lower bound on the evaluation time for any arithmetic expression of 2^k constants and variables is k steps.

The present paper presents several new upper bounds which are sharper than previous time bounds in certain cases. These include expressions which have few parenthesis pairs and expressions which have few division operations. We also give an upper bound which holds when multiplication is not commutative. This provides a time bound on the evaluation of expressions of matrices and

* Received by the editors December 27, 1973, and in revised form June 24, 1974. This work was supported in part by the National Science Foundation under Grant GJ-36936 and by the IBM Thomas J. Watson Research Center.

† Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801.

‡ IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.

vectors. Finally, we prove sharper upper bounds on two special classes of expressions; one is a generalization of polynomials in Horner's rule form and the other is a generalization of continued fractions. The paper begins with a brief survey of a number of previous results. This is followed by a presentation of new results.

2. Definitions and background. The following definitions and assumptions will hold throughout the paper. An *atom* is a variable or constant and is denoted by a lower case italic letter. A binary *operator* θ is either an addition, subtraction, multiplication or division, denoted by $+$, $-$, $*$ and $/$, respectively.[1] An *arithmetic expression* is a well-formed string consisting of atoms and operators and is denoted by an upper case italic letter. A subexpression of an expression E is defined as follows: let E_1 be a substring which appears in E, and let E' be an expression which is derived from E by replacing E_1 by an expression (E_1). Then E_1 is a *subexpression* of E if the value of E equals the value of E'.

We write $E(n)$ to denote an arithmetic expression of at most n distinct atoms.[2] The exact number of atoms in E is denoted by $|E|$. The exact number of pairs of parentheses which appear in an expression E is denoted by $\|E\|$. We write $E(n|p)$ to denote an arithmetic expression of at most n distinct atoms with at most p pairs of parentheses. We write $E(n|d)$ to denote an arithmetic expression of at most n distinct atoms in which the maximum depth of parenthesis nesting is d. The exact number of division operations which appear in an expression E is denoted by $/E/$. We write $E(n|q)$ to denote an arithmetic expression of at most n distinct atoms with at most q division operations.

The following machine idealizations are assumed (unless otherwise stated):

1. Each processor may perform any of the four binary arithmetic operations at any time, but all processors need not perform the same operation at any time.
2. Each operation takes one unit of time, which we refer to as a *step*.[3]
3. No time is required to communicate data between processors.
4. Any number of processors may be used at any time.

We use the notation $T[E]$ to denote the number of steps required to evaluate expression E after some given transformation has been performed. Throughout the paper, for any real number x, $\lceil x \rceil$ denotes the smallest integer greater than or equal to x, and $\lfloor x \rfloor$ denotes the largest integer less than or equal to x.

Now we present a brief survey of previous results concerning the evaluation time for arithmetic expressions. Details and further references may be found in the papers cited. Assuming that only associativity and commutativity are used to transform expressions, Baer and Bovet [1] gave a comprehensive tree height reduction algorithm based on a number of earlier papers. Beatty [2] showed the

[1] We assume that all unary plus operators are dropped. Sequences of unary minus operators of even length may be dropped and sequences of odd length may be replaced by a single unary minus operator. We assume that these are distributed such that no unary minus operator appears except at the level of atoms.

[2] By distinct we mean algebraically independent. Thus each atom appears just once in an expression, so if we are presented with an expression containing multiple occurrences of some atoms, the expression may be relabeled such that each atom has a unique label.

[3] We assume that the processors are capable of complementing a number in negligible time before computing with it. This means, for example, that if values a and b are available, either $a * b$ or $a * (-b)$ may be evaluated in one step. See footnote 1.

optimality of this method. An upper bound on the reduced tree height assuming only associativity and commutativity are used, given by Kuck and Muraoka [12], is the following.

THEOREM 1. *Let $E(n|d)$ be any arithmetic expression with depth d of parenthesis nesting. By the use of associativity and commutativity only, $E(n, d)$ can be transformed such that*

$$T[E(n|d)] \leqq \lceil \log_2 n \rceil + 2d + 1.$$

Note that if the depth of parenthesis nesting d, is small, then this bound is quite close to the lower bound of $\lceil \log_2 n \rceil$. Unfortunately, there are classes of expressions, e.g., Horner's rule polynomials or continued fractions, for which no speed increase can be achieved by using more than one processor under the assumption that only associativity and commutativity are used in tree height reduction.

Muraoka [17] studied the use of distributivity as well as associativity and commutativity for tree height reduction and developed comprehensive tree height reduction algorithms using all three transformations. These algorithms were programmed and used in the analysis of a number of real FORTRAN programs. The algorithms as well as some numerical results of program analysis are contained in [13], and further numerical results are given in [10]. An algorithm which considers operations which take different amounts of time is presented by Kraska [8]. While the arithmetic expressions found in most real programs are quite simple, more complex ones may be derived by the substitution of one expression into another, as discussed in [13]. Recurrence relations, for example, can lead to an expression whose length is proportional to the number of times a loop is executed. Thus it seems that an important fundamental issue in the design of future computers and their compilers is the potential for program speedup by tree height reduction techniques.

The following bounds show that distribution is indeed effective in transforming an expression so that its reduced tree height approaches the lower bound. It was shown by Brent [3] that arithmetic expressions of the form $((\cdots(a_n x_n + a_{n-1})x_{n-1} + \cdots + a_2)x_2 + a_1)x_1 + a_0$ can be evaluated in $\log_2 n + O_2(\sqrt{\log_2 n})$ steps. The special case of polynomial evaluation ($x_1 = x_2 = \cdots = x_n$) has been studied by Maruyama [14] and by Munro and Paterson [16]. They have shown that a polynomial of degree n can be evaluated in $\log_2 n + O_2(\sqrt{\log_2 n})$ steps if n processors are available. They also have deduced some results which apply when a fixed number of processors are available. In [7], Kogge and Stone discussed a class of linear recurrence relations and showed that $O_2(\log_2 n)$ steps are sufficient for their evaluation. Certain special recurrences, including continued fractions, were also discussed in [19].

The class of all arithmetic expressions without the division operation was studied by Brent, Kuck and Maruyama [6], who proved the following.

THEOREM 2. *Let $E(n|q = 0)$ be any arithmetic expression with no division operations. Then by the use of associativity, commutativity and distributivity, $E(n|q = 0)$ can be transformed such that:*[4]

$$T[E(n|g = 0) \leqq O_1(2.465 \log_2 n).$$

[4] We find it convenient to use two different order of magnitude notations. If a and b are some constants, we represent $x + a$ by $O_1(x)$ and $ax + b$ by $O_2(x)$.

While this is fairly close to the lower bound, it is likely that some improvement is possible. For later use, we simplify the notation by writing α for the best known coefficient (2.465 presently) such that $T[E(n|q = 0)] \leqq O_1(\alpha \log_2 n)$. Recently, Brent [4], [5] has proved the following bound for any arithmetic expression $E(n)$.

THEOREM 3. *Let $E(n)$ be any arithmetic expression. By the use of associativity, commutativity and distributivity, $E(n)$ can be transformed such that*

$$T[E(n)] \leqq O_1(4 \log_2 n).$$

The above results as well as the main results of this paper are summarized in Table 1. It may be seen that Brent's bound of Theorem 3 is the best known for general arithmetic expressions, assuming associativity, commutativity and distributivity are used. But for various special classes of expressions, better bounds are possible. We also show that even if multiplication is not commutative, a bound which is fairly sharp (Theorem 8) may be given.

TABLE 1
Summary of time bounds;
n = number of atoms,
d = depth of parenthesis nesting,
p = number of parenthesis pairs,
q = number of division operators

Transformations allowed	Class of expressions	Best known time bound	Reference in paper
associativity and commutativity	general	$\log_2 n + 2d$ + const.	Theorem 1
associativity, distributivity and commutative addition	general	$6 \log_2 n$ + const.	Theorem 8
associativity, commutativity and distributivity	polynomial forms	$\log_2 n + \sqrt{8 \log_2 n}$ + const.	Theorem 10
	continued parenthesis forms	$2 \log_2 n$ + const.	Theorem 9
	general without division	$2.465 \log_2 n$ + const.	Theorem 2
		$\log_2 n + 2.465 \log_2 p$ + const.	Theorem 4
	general	$4 \log_2 n$ + const.	Theorem 3
		$\log_2 n + 4 \log_2 p$ + const.	Theorem 5
		$2.465 \log_2 n + 4 \log_2 q$ + const	Theorem 6

3. New time bounds using associativity, commutativity and distributivity. We begin by presenting four lemmas. Lemmas 1 and 2 are basic to a number of the following proofs, while Lemmas 3 and 4 lead to the proof of Theorem 4. Throughout the rest of the paper, we assume that expressions may be transformed using associativity, commutativity and distributivity and that redundant parentheses have been removed from all expressions.

Lemma 1 is a direct generalization of [6, Lemma 2], and we state it without proof.

LEMMA 1. (a) *Let $E(n)$ be any arithmetic expression and let $n \geqq m > 1$. Then we can always find a subexpression $L\ \theta\ R$ such that*

$$|L| < m, \quad |R| < m \quad \text{and} \quad |L| + |R| \geqq m,$$

where $\theta \in \{+, -, *, /\}$.

(b) *Let $E(n|p)$ be any arithmetic expression with p pairs of parentheses. Then for any m, $p \geqq m \geqq 0$, we can always find a subexpression $L\ \theta\ R$ or $(L\ \theta\ R)$ such that*

$$\|L\| \leqq m, \quad \|R\| \leqq m \quad \text{and} \quad \|L\| + \|R\| \geqq m,$$

where $\theta \in \{+, -, *, /\}$.

(c) *Let $E(n|q)$ be any arithmetic expression with q division operations. Then for any m, $q \geqq m \geqq 0$, we can always find a subexpression $L\ \theta\ R$ such that*

$$/L/ \leqq m, \quad /R/ \leqq m \quad \text{and} \quad /L/ + /R/ \geqq m,$$

where $\theta \in \{+, -, *, /\}$.

Lemma 2 is a direct generalization of [5, Lemma 1], and we state it without proof.

LEMMA 2. (a) *Let $E(n)$ be any arithmetic expression, and let x be any one of its n atoms. For any m, $n \geqq m > 1$, we can always find a subexpression $X = L\ \theta\ R$ such that x is in X and $|X| \geqq m$, and either*

1. *x is an atom of L and $|L| < m$; or*
2. *x is an atom of R and $|R| < m$, where $\theta \in \{+, -, *, /\}$.*

(b) *Let $E(n|p)$ be any arithmetic expression with p pairs of parentheses, and let x be any one of its n atoms. Then for any m, $p \geqq m \geqq 0$, we can always find a subexpression $X = L\ \theta\ R$ or $X = (L\ \theta\ R)$ such that x is in X and $\|X\| \geqq m$, and either*

1. *x is an atom of L and $\|L\| \leqq m$; or*
2. *x is an atom of R and $\|R\| \leqq m$, where $\theta \in \{+, -, *, /\}$.*

(c) *Let $E(n|q)$ be any arithmetic expression with q division operations, and let x be any one of its n atoms. Then for any m, $q \geqq m \geqq 0$, we can always find a subexpression $X = L\ \theta\ R$ such that x is in X and $/X/ \geqq m$, and either*

1. *x is an atom of L and $/L/ \leqq m$; or*
2. *x is an atom of R and $/R/ \leqq m$, where $\theta \in \{+, -, *, /\}$.*

Now we turn our attention to the proof of a new bound on the time required to evaluate expressions without division, assuming that the number of parenthesis pairs in the expression is known.

LEMMA 3. *Let $E(n|p)$ be any arithmetic expression with p pairs of parentheses but with no division operations. Then the following hold:*

(a) $T[E(n|0)] \leqq \lceil \log_2 n \rceil + 1$ *for* $n \geqq 2$,

(b) $T[E(n|1)] \leqq \lceil \log_2 n \rceil + 3$ *for* $n \geqq 3$,

(c) $T[E(n|2)] \leqq \lceil \log_2 n \rceil + 4$ *for* $n \geqq 4$,

(d) $T[E(n|3)] \leqq \lceil \log_2 n \rceil + 5$ *for* $n \geqq 6$.

Proof. Parts (a) and (b) follow immediately from Theorem 1 and the definition of $E(n|p)$ for $P = 0, 1$.

To prove (c), we consider the following two cases.

Case 1. The depth of parenthesis nesting for $E(n|2)$ is 1.

Case 2. The depth of parenthesis nesting for $E(n|2)$ is 2.

In Case 1, by taking X and Y as subexpressions nested by pairs of parentheses in $E(n|2)$, we have $X = X(n|0)$ and $Y = Y(n|0)$. Furthermore, we have either

$$E(n|2) = A(n|0)X(n|0) + B(n|0)Y(n|0) + C(n|0)$$

or

$$E(n|2) = A(n|0)X(n|0)Y(n|0) + C(n|0).$$

Since each of A, B, C, X and Y can be evaluated in $(\lceil \log_2 n \rceil + 1)$ steps by (a), $E(n|2)$ can be evaluated in another 3 steps.

In Case 2, let X be the subexpression nested by the first pair of parentheses and let Y be the subexpression nested by the second pair of parentheses which appears in X. Then we have $Y = Y(n|0)$ and $X = X(n|1) = A(n|0)Y(n|0) + B(n|0)$. Furthermore, $E(n|2) = A'(n|0)X(n|1) + B'(n|0)$. By substitution, we have

$$\begin{aligned} E(n|2) &= A'(n|0)(A(n|0)Y(n|0) + B(n|0)) + B'(n|0) \\ &= A'(n|0)A(n|0)Y(n|0) + A'(n|0)B(n|0) + B'(n|0). \end{aligned}$$

Since each of A, B, A', B' and Y can be evaluated in $(\lceil \log_2 n \rceil + 1)$ steps, $E(n|2)$ can be evaluated in another 3 steps. We can prove (d) by an argument similar to the proof of (c). Q.E.D.

Using Lemma 3, we prove the following.

LEMMA 4. *Let* $p_0 = 1$, $p_1 = 2$, $p_2 = 3$ *and* $p_{k+3} = p_k + p_{k+1}$ *for* $k \geqq 0$. *Then for any arithmetic expression* $E(n|p_k)$ *with* p_k *pairs of parentheses and with no division operations,*

$$T[E(n|p_k)] \leqq O_1(\log_2 n + k).$$

Proof. We prove by induction that $T[E(n|p_k)] \leqq \lceil \log_2 n \rceil + k + 3$. By Lemma 3, $E(n|1)$, $E(n|2)$ and $E(n|3)$ can be evaluated in $(\lceil \log_2 n \rceil + 3)$ steps, $(\lceil \log_2 n \rceil + 4)$ steps and $(\lceil \log_2 n \rceil + 5)$ steps, respectively. Thus the lemma holds for $k \leqq 2$, since $p_3 = p_2$.

We assume that the lemma holds for $k \leqq r + 2$, $r \geqq 0$, and we prove it for $k = r + 3$. For any given $E(n|p_{r+3})$, using Lemma 1(b), find a subexpression $L \, \theta \, R$ or $(L \, \theta \, R)$ of $E(n|p_{r+3})$ such that

$$\|L\| \leqq p_r, \quad \|R\| \leqq p_r \quad \text{and} \quad \|L\| + \|R\| \geqq p_r,$$

where $\theta \in \{+, -, *, /\}$. Let us denote such a subexpression by X, and let E' be the expression derived from $E(n|p_{r+3})$ by replacing X by an atom x. Then $|X| \geqq 1$ and $|E'| \leqq n + 1 - |X| \leqq n$. Since $\|E'\| \leqq p_{r+3} - \|X\| \leqq p_{r+3} - p_r = p_{r+1}$, we have

$$E' = A(n|p_{r+1})x + B(n|p_{r+1}).$$

Moreover, since x corresponds to the expression X, we have

$$E(n|p_{r+3}) = A(n|p_{r+1})(L(n|p_r) \, \theta \, R(n|p_r)) + B(n|p_{r+1}).$$

By the induction hypothesis, each of $A(n|p_{r+1})$ and $B(n|p_{r+1})$ can be evaluated in $((r + 1) + \lceil \log_2 n \rceil + 3)$ steps and each of $L(n|p_r)$ and $R(n|p_r)$ can be evaluated in $(r + \lceil \log_2 n \rceil + 3)$ steps. Therefore, $E(n|p_{r+3})$ can be evaluated in $(r + \lceil \log_2 n \rceil + 3) + 3 = ((r + 3) + \lceil \log_2 n \rceil + 3)$ steps. From this, the lemma follows. Q.E.D.

Now the following theorem can easily be proved (cf. [6, Lemma 4]).

THEOREM 4. *For any arithmetic expression $E(n|p)$ with p pairs of parentheses and with no division operations,*

$$T[E(n|p)] \leqq O_1(\log_2 n + 2.465 \log_2 p).$$

Proof. The general solution of the linear recurrence relation in Lemma 4 is

$$p_k = c_1(\lambda_1)^k + c_2(\lambda_2)^k + c_3(\lambda_3)^k,$$

where λ_1, λ_2 and λ_3 are roots of $z^3 = 1 + z$ and c_1, c_2 and c_3 are arbitrary constants. Let λ_1 be the real root and λ_2 and λ_3 be complex roots; then by inspection $c_2(\lambda_2)^k$ and $c_3(\lambda_3)^k$ vanish as k gets large. Thus, we have $p_k \approx c_1(\lambda_1)^k$, where $\lambda_1 = 1.3247$. From this, the theorem follows. Q.E.D.

Next we turn our attention to several new bounds on the time required for the evaluation of general arithmetic expressions of n atoms, assuming additional information is available. Theorem 5 assumes the number of parenthesis pairs is known, and Theorem 6 assumes the number of division operations is known. Since commonly occurring expressions may have few parentheses or few divisions, in practice these bounds may be better than that of Theorem 3.

THEOREM 5. *For any arithmetic expression $E(n|p)$ with p pairs of parentheses,*

$$T[E(n|p)] \leqq O_1(\log_2 n + 4 \log_2 p),$$

and only one division operation need be performed.

Proof. To prove the theorem, we prove the following claim, so that $E(n|p)$ can be evaluated in $\lceil \log_2 n \rceil + 4\lceil \log_2 p \rceil + 4$ steps by the first part of the claim.

CLAIM. *$E(n|p)$ can be transformed into the form G/H, and each of G and H contains no division operators and each can be evaluated in $4\lceil \log_2 p \rceil + \lceil \log_2 n \rceil + 3$ steps. Furthermore, $E(n|p)$ can be transformed into the form $(Ax + B)/(Cx + D)$, where x is any atom in $E(n|p)$ and each of A, B, C and D contains no division operators and each can be evaluated in $4\lceil \log_2 p \rceil + \lceil \log_2 n \rceil + 5$ steps.*

We prove the claim by induction on p. Let $r = \lceil \log_2 p \rceil$. For $r = 0$, i.e., $E(n|1)$, let X_0 be the expression which is nested by the pair of parentheses. Then it can be seen easily that $E(n|1)$ can be transformed into the form

$$E(n|1) = A_0 \, \theta_0 \, X_0 + B_0, \qquad \theta_0 \in \{*, /\},$$

where A_0, B_0 and X_0 are expressions with no parentheses and A_0 is an expression with only multiplication and/or division operations. Since $|A_0| \leqq n - 1$, $|B_0| \leqq n - 1$ and $|X_0| \leqq n - 1$, by Theorem 1, each of A_0, B_0 and X_0 can be evaluated in $\lceil \log_2 n \rceil + 1$ steps. Now $E(n|1) = G/H$, where $G = A_0 X_0 + B_0$ and $H = 1$ for $\theta_0 = *$, and $G = A_0 + B_0 X_0$ and $H = X_0$ for $\theta_0 = /$. Hence each of G and H can be evaluated in $\lceil \log_2 n \rceil + 3$ steps. So the first half of the proof is complete for $r = 0$.

Let x be any atom of $E(n|1)$. There are three cases to be examined for $\theta_0 = *$.

Case 1. *x is in A*. Since A_0 is an expression with only multiplication and/or division operations, we can rewrite A_0 as

$$A_0 = (A_1/C_1)\theta_1 x, \qquad \theta_1 \in \{*, /\}$$

for any x in A_0, and each of A_1 and C_1 can be evaluated in $\lceil \log_2 n \rceil + 1$ steps, by Theorem 1. Similarly, we rewrite X_0 and B_0 as

$$X_0 = (A_2 y + B_2)/(C_2 y + D_2), \qquad B_0 = (A_3 z + B_3)/(C_3 z + D_3),$$

where y and z are any atoms in X_0 and B_0, respectively. Further, A_2, B_2, C_2, D_2, A_3, B_3, C_3 and D_3 are expressions of at most $n - 2$ atoms; hence each of them can be evaluated in $\lceil \log_2 n \rceil + 1$ steps, by Theorem 1.

Now for

$$E(n|1) = (Ax + B)/(Cx + D),$$

by substituting expressions of A_0, X_0 and B_0 into $E(n|1)$, we find for $\theta_0 = *$ and for $\theta_1 = /$,

$$A = C_1(C_2 y + D_2)(A_3 z + B_3).$$

B and C are similar expressions and $D = 0$. Thus A, B, C and D can be evaluated in $\lceil \log_2 n \rceil + 5$ steps for $\theta_0 = *$.

By an argument similar to the above, Case 2 (x is in X_0) and Case 3 (x is in B_0) can be examined, as can the case of $\theta_0 = /$. This completes the proof of the second half of the claim for $r = 0$.

Now assuming that the claim holds for $0 \leqq r \leqq k - 1$, the claim can be proved for $r = k$ by using Lemma 1(b), Lemma 2(b) and an argument similar to that used by Brent [5] in proving Theorem 3. Q.E.D.

THEOREM 6. *Assume we have a procedure to evaluate any expression $E(n|q = 0)$, which has no division operations, in $O_1(\alpha \log_2 n)$ steps. Then for any arithmetic expression $E(n|q)$ with $q \geqq 1$ division operations,*

$$T[E(n|q)] \leqq O_1(\alpha \log_2 n + 4 \log_2 q),$$

and at most one division need be performed.

Proof. As in the case of Theorem 5, this theorem can be proved by proving the following claim.

CLAIM. *$E(n|q)$ can be transformed into the form G/H, and each of G and H contains no division operators and each can be evaluated in $O_1(\alpha \log_2 n) + 4 \log_2 q + 4$ steps. Furthermore, $E(n|q)$ can be transformed into the form $(Ax + B)/(Cx + D)$, where x is an atom in $E(n|q)$ and each of A, B, C and D contains no division operators and each can be evaluated in $O_1(\alpha \log_2 n) + 4\lceil \log_2 q \rceil + 6$ steps.*

This claim can be proved by induction on q using an argument similar to the proof of Theorem 5, together with Lemma 1(c) and Lemma 2(c). Q.E.D.

Note that by using the value $\alpha = 2.465$ from Theorem 2, we can immediately obtain the entry shown in Table 1.

The next theorem presents a bound which is worse than that of Theorem 6. However, since it is expressed in terms of the coefficient α (where an expression without division can be evaluated in $O_1(\alpha \log_2 n)$ steps), if the best present $\alpha = 2.465$ (Theorem 2) can be reduced, then Theorem 7 will provide a better bound than

Theorem 6. In fact, if it can be shown that $\alpha < 2$, then Theorem 7 is an improvement on Theorem 6, and if it can be shown that $\alpha < \frac{4}{3}$, Theorem 7 is an improvement on Theorem 3, since in the worst case, q approaches n.

THEOREM 7. *Assume we have a procedure to evaluate any $E(n|q = 0)$, which has no division operations, in $O_1(\alpha \log_2 n)$ steps. Then for any $E(n|q)$ with $q \geqq 1$ division operations,*

$$T[E(n|q)] \leqq O_1(\alpha(\log_2 n + 2 \log_2 q)),$$

and just one division need be performed.

Proof. A detailed proof is given in Kuck [9], but since it involves a rather long and straightforward argument, we will just sketch it here. The proof proceeds by breaking any given expression $E(n|q)$ into a set of $r \leqq 2q$ expressions of the form

$$(1) \qquad D_i = \frac{D_jF_1 + F_2}{D_jF_3 + F_4}\theta_i\frac{E_1}{E_2},$$

where the F_j are free of division operations, $\theta_1 \in \{\pm, *, /\}$, and D_j has the same form as D_i. As a tree of such expressions is traced from its root to its leaves, eventually expressions occur in which E_1 and E_2 are free of division operations. We represent the situation by assuming D_j has F_9 and F_{10} as follows:

$$(2) \qquad D_j = \frac{D_kF_5 + F_6}{D_kF_7 + F_8}\theta_j\frac{F_9}{F_{10}},$$

where all the F_j are free of division operators, $\theta_j \in \{\pm, *, /\}$, and D_k has the same form as D_i. By substituting the right-hand side of (2) into (1) and factoring it, we obtain (assuming $\theta_j = +$)

$$(3) \quad D_i = \frac{D_k[(F_1F_5 + F_2F_7)F_{10} + F_1F_7F_9] + [(F_1F_6 + F_2F_8)F_{10} + F_1F_8F_9]}{D_k[(F_3F_5 + F_4F_7)F_{10} + F_3F_7F_9] + [(F_3F_6 + F_4F_8)F_{10} + F_3F_8F_9]}\theta_i\frac{E_1}{E_2}.$$

Note that (3) has the same form as (1) and has at most four occurrences of any F_i on its right-hand side. Furthermore, we have eliminated two division operators from the set of expressions derived from the given $E(n|q)$. By performing a number of such transformations in parallel, we can show that in $\lceil \log_2 r \rceil + 1$ steps, all but one division (at the last step) can be eliminated. Thus an upper bound on the number of atoms in the final expression is

$$O_2(n4^{\log_2 r}) = O_2(n2^{\log_2 r^2}) = O_2(nr^2).$$

Hence, since $r \leqq 2q$, if any expression without division can be evaluated in $\alpha \log_2 n$ steps, the theorem follows. Q.E.D.

4. Evaluating expressions with noncommutative multiplication. In this section we consider the class of general arithmetic expressions in which multiplication does not commute, i.e., $ab \neq ba$ for arbitrary atoms a and b. Our interest in such expressions arises from the study of well-formed expressions of scalars, vectors and matrices. In [18], algorithms were presented for the fast serial and parallel evaluation of products of scalars, vectors and matrices, but no bounds were given. Here we prove an upper bound on the time required for the parallel evaluation of general expressions with noncommutative multiplication and follow it with some discussion of the result in the context of matrix expressions.

THEOREM 8. *For any arithmetic expression $E(n)$ whose multiplication operator is noncommutative,*

$$T[E(n)] \leqq O_1(6 \log_2 n).$$

Proof. In the proof of the theorem, we denote addition time by t_A, multiplication time by t_M and division time by t_D. We prove the theorem by proving the following claim.

CLAIM. *$E(n)$ can be transformed into the form GH^{-1} and each of G and H can be evaluated in time $\lceil \log_2 n \rceil(3t_M + 2t_A + t_D) - 2t_M - t_A$. Furthermore, $E(n)$ can be transformed into the form $(Ax + B)(Cx + D)^{-1}$, where x is an atom in $E(n)$ and each of A, B, C and D can be evaluated in time $\lceil \log_2 n \rceil(3t_M + 2t_A + t_D) + t_D$.*

Thus the theorem follows since $E(n)$ can be evaluated in $\lceil \log_2 n \rceil(3t_M + 2t_A + t_D) - t_M - t_A + t_D$ steps, by the first part of the claim.

Proof of the claim. We prove the claim by induction on n.

Let $\beta = 3t_M + 2t_A + t_D$ and $r = \lceil \log_2 n \rceil$. For $r = 1$, it can be seen easily that G and H can be evaluated in $t_M + t_A + t_D = \beta - 2t_M - t_A$ steps. Furthermore, A, B, C and D of $E(2)$ can be evaluated in $\beta + t_D$ steps. Thus, the claim holds for $r = 1$. We assume that the claim holds for $1 \leqq r \leqq k - 1$, and we prove the claim for $r = k$.

We prove the first part of the claim first. For any given $E(2^k)$, we apply Lemma 1(a) with $m = 2^{k-1} + 1$ and find a subexpression $X_1 = L_1\ \theta_1\ R_1$ of $E(2^k)$ such that $|X_1| \geqq 2^{k-1} + 1$, $|L_1| \leqq 2^{k-1}$ and $|R_1| \leqq 2^{k-1}$, where $\theta_1 \in \{+, -, *\}$. Thus the second part of the inductive hypothesis gives

$$E(2^k) = (A_1X_1 + B_1)(C_1X_1 + D_1)^{-1},$$

where A_1, B_1, C_1 and D_1 can be evaluated simultaneously in $\beta(k - 1) + t_D$ steps. Also, the first half of the inductive hypothesis applied to L_1 and R_1 gives $L_1 = G_1H_1^{-1}$ and $R_1 = G_2H_2^{-1}$, where G_1, H_1, G_2 and H_2 can be evaluated simultaneously in $\beta(k - 1) - 2t_M - t_A$ steps.

By substituting expressions L_1 and R_1 into $E(2^k)$, we can derive $E(2^k) = GH^{-1}$, and whatever θ_1 is, we can show that each of G and H can be evaluated in $\beta(k - 1) + t_D + t_M + t_A = \beta k - 2t_M - t_A$ steps. This completes the proof of the first part of the claim.

Now we prove the second part of the claim. Let x be an atom of $E(2^k)$. Applying Lemma 2(a) with $m = 2^{k-1} + 1$ on $E(2^k)$, we see that there is a subexpression $X_2 = L_2\ \theta_2\ R_2$ of $E(2^k)$ such that $|X_2| \geqq 2^{k-1} + 1$, and either x is an atom of L_2 and $|L_2| \leqq 2^{k-1}$ or x is an atom of R_2 and $|R_2| \leqq 2^{k-1}$, where $\theta_2 \in \{+, -, *\}$.

Without loss of generality, suppose the former holds. Thus the second part of the inductive hypothesis gives

$$E(2^k) = (A_2X_2 + B_2)(C_2X_2 + D_2)^{-1},$$

where A_2, B_2, C_2 and D_2 can be evaluated simultaneously in $\beta(k - 1) + t_D$ steps. Similarly,

$$L_2 = (A_3x + B_3)(C_3x + D_3)^{-1},$$

where A_3, B_3, C_3 and D_3 can be evaluated simultaneously in $\beta(k - 1) + t_D$ steps.

Since $|R_2| < 2^k$, the first half of the inductive hypothesis shows that $R_2 = G_3H_3^{-1}$, where G_3 and H_3 can be evaluated simultaneously in $\beta k - 2t_M - t_A$ steps.

By substituting expressions L_2 and R_2 into $E(2^k)$, we can derive

$$E(2^k) = (Ax + B)(Cx + D)^{-1},$$

and whatever θ_2 is, we can show that each of A, B, C and D can be evaluated in $\beta k - 2t_M - t_A + (2t_M + t_A + t_D) = \beta k + t_D$ steps. Hence the second half of the proof is complete. Q.E.D.

This bound may be of interest in studying the time required to evaluate any well-formed expression of scalars, vectors, and matrices. Given a sufficient number of processors, the times required to add or multiply pairs of such operands are easy to derive (see [18]). However, matrix inversion time is rather difficult to bound, since a variety of inversion methods are available, and the one used may depend on the numerical details of the matrix. Thus various bounds could be derived for different matrix inversion methods. In [15], matrix bounds similar to the above are proved for various restricted classes of matrix expressions.

5. Evaluating continued parenthesis forms. In this section we consider a class of special arithmetic expressions which we call continued parenthesis forms. Our interest in these forms arises from the study of polynomials in the form of Horner's rule and from continued fractions; both of these are continued parenthesis forms. Theorem 9 shows that any continued parenthesis form of degree n (with $2n + 1$ atoms) may be evaluated in at most $2\lceil \log_2 n \rceil + 3$ steps.

In order to simplify our proofs, we make some small notational departures from above. We write $E_{CP}(n)$ to denote the *continued parenthesis forms of degree n*, $CF(n)$ and $PF(n)$, which contain at most $2n + 1$ atoms, i.e., $|E_{CP}(n)| \leqq 2n + 1$. The definitions are:

$$CF(n) = a_0\,\theta_1\,(b_1\,\theta_2\,(a_1\,\theta_3 \cdots (b_i\,\theta_{2i}\,(a_i\,\theta_{2i+1}\,(\cdots\,\theta_{2n-1}\,(b_n\,\theta_{2n}\,a_n)\cdots)))\cdots)),$$

$$PF(n) = ((\cdots(((\cdots(a_n\,\theta_{2n}\,b_n)\,\theta_{2n-1}\,\cdots)\,\theta_{2i+1}\,a_i)\,\theta_{2i}\,b_i)\cdots\,\theta_3\,a_1)\,\theta_2\,b_1)\,\theta_1\,a_0,$$

where if $\theta_{2i} \in \{*, /\}$ and $\theta_{2i-1} \in \{+, -\}$ for $i = 1, 2, \cdots, n$, then we call $CF(n)$ a *continued fraction form of degree n*, and we call $PF(n)$ a *polynomial form of degree n*. We write $E_{CP}(n|X)$ to denote a continued parenthesis form which is derived by replacing a_n of $E_{CP}(n)$ by another arithmetic expression X.

THEOREM 9. *If* $4n$ *processors are available, then for any continued parenthesis form* $E_{CP}(n)$ *of degree n,*

$$T[CP(n)] \leqq O_1(2\log_2 n),$$

where at most $24n$ *operations, including just one division, need be performed.*

Proof. We know that a continued parenthesis form $E_{CP}(n|x)$ of degree n (i.e., at most $2n + 1$, $n \geqq 1$, atoms) can be rewritten in the form of $(Ax + B)/(Cx + D)$. To prove the theorem, we prove the statement that each of A, B, C and D which contains no division operators can be evaluated in $2\lceil \log_2 n \rceil$ steps. Thus the final result can be evaluated in another 3 steps, for a total of $2\lceil \log_2 n \rceil + 3$ steps.

Let $r = \lceil \log_2 n \rceil$. The statement holds for $r = 0, 1$. Let us assume that it holds for $r \leqq k - 1$, and prove the statement for $r = k$ by induction. For $E_{CP}(2^k|x)$, which can be rewritten as

(4) $$(A'x + B')/(C'x + D'),$$

we apply Lemma 1(a) by setting $m = 2^k + 1$, and we can always find $X = (L\,\theta\,R)$ such that $|X| = m$, and $X = X_{CP}(2^{k-1}|x)$. Therefore we have $E_{CP}(2^k|x) = E_{CP}(2^{k-1}|X)$.

Furthermore, $E_{CP}(2^{k-1}|X)$ and $X_{CP}(2^{k-1}|x)$ can be rewritten as

(5) $$(A_0X + B_0)/(C_0X + D_0)$$

and

(6) $$(A_1x + B_1)/(C_1x + D_1),$$

respectively. Thus, by substituting (6) into (5), we can find each of A', B', C' and D' for (4) as follows:

$$A' = A_0A_1 + B_0C_1, \qquad B' = B_0D_1 + A_0B_1,$$
$$C' = C_0A_1 + D_0C_1, \qquad D' = D_0D_1 + C_0B_1.$$

By our induction hypothesis, each of A_i, B_i, C_i and D_i for $i = 0, 1$ can be evaluated in $2(k - 1)$ steps. We can evaluate each of A', B', C' and D' in another 2 steps, which complete our inductive proof of the statement. Therefore, the final result of (4) can be evaluated in another 3 steps, from which we have $T[CP(n)] \leqq 2\lceil\log_2 n\rceil + 3$.

Let $k = \lceil\log_2 n\rceil$, and let $P(2^k)$ denote the number of processors required to evaluate either A', B', C' or D' of $E_{CP}(2^k)$. To evaluate A', B', C' and D' simultaneously, we evaluate A_0, B_0, C_0, D_0, A_1, B_1, C_1 and D_1 simultaneously. Thus, the total number of processors is

$$4P(2^k) \leqq 4 \cdot 2P(2^{k-1}) \leqq \cdots \leqq 4 \cdot 2^{k-1}P(2).$$

Since $P(2) \leqq 1$, we get

$$4 \cdot P(2^k) \leqq 2 \cdot 2^k \leqq 2 \cdot 2^{\lceil\log_2 n\rceil} < 2 \cdot 2^{(\log_2 n + 1)} = 4n$$

processors.

Let $Q(2^k)$ denote the number of operations required to evaluate either A', B', C' or D' of $E_{CP}(2^k)$. Then we have $Q(2^k) \leqq 2Q(2^{k-1}) + 3 \leqq \cdots \leqq 3 \cdot 2^k - 3 \leqq 6n - 3$. Thus $4Q(2^k) + 5 \leqq 24n - 7$ operations are sufficient. From these, the theorem follows. Q.E.D.

In Theorem 9, let $m = 2n + 1$ be the number of atoms in a continued parenthesis form. Then $E_{CP}(\lceil m - 1)/2\rceil)$ can be evaluated in $2\lceil\log_2 m\rceil + 1$ steps if $4\lceil(m - 1)/2\rceil$ processors are available. Thus we can say that continued parenthesis forms of m atoms can be evaluated in $2\lceil\log_2 m\rceil + 1$ steps using $12m$ operations if $2m$ processors are available. This may be regarded as a generalization of the continued fraction result in [19] but with a sharper processor bound here.

6. Evaluating polynomial forms.

In this section we continue our discussion of continued parenthesis forms, but here we deal exclusively with polynomial forms of degree n, denoted by $PF(n)$. We repeat the definition given in § 5:

$$PF(n) = (\cdots(a_n\,\theta_{2n}\,x_n\,\theta_{2n-1}\,a_{n-1})\,\theta_{2n-2}\,x_{n-1}\,\theta_{2n-3}\cdots\theta_3\,a_1)\,\theta_2\,x_1\,\theta_1\,a_0,$$

where $\theta_{2i} \in \{*, /\}$ and $\theta_{2i-1} \in \{+, -\}$ for $i = 1, 2, \cdots, n$. Results similar to those in this section were given in [3], [14] and [16], but in those papers the restrictions were made that $\theta_{2i} \in \{*\}$, $\theta_{2i-1} \in \{+\}$. The main result of this section is Theorem 10, which states that any polynomial form of degree n may be evaluated in $O_1(\log_2 n) + \sqrt{8 \log_2 n})$ steps using at most $2n$ processors. Theorem 10 follows from three lemmas which we now prove.

First, we find it convenient to introduce some new notation. For an expression

$$x_1 \, \theta_1 \, x_2 \, \theta_2 \cdots \theta_{i-1} \, x_i \, \theta_i \cdots \theta_{n-1} \, x_n, \tag{7}$$

where $\theta_i \in \{+, -, *, /\}$ for $i = 1, 2, \cdots, n - 1$, we introduce the following:

(i) expression (7) is denoted by $\sum_{i=1}^{n} x_i$ if $\theta_1 \in \{+, -\}$ for $i = 1, 2, \cdots, n - 1$.

(ii) expression (7) is denoted by $\prod_{i=1}^{n} x_i$ if $\theta_i \in \{*, /\}$ for $i = 1, 2, \cdots, n - 1$.

Now we can expand $PF(n)$ to the form:

$$\sum_{i=1}^{n} a_i \prod_{j=1}^{i} x_j \pm a_0 = \sum_{i=1}^{n} a_i X(i) \pm a_0,$$

where $X(i)$ denotes $\prod_{j=1}^{i} x_j$.

We state without proof the following simple lemma which can be proved easily by induction.

Lemma 5. *Either* $\{\sum_{i=1}^{k} x_i | 1 \leqq k \leqq n\}$ *or* $\{\prod_{i=1}^{k} x_i | 1 \leqq k \leqq n\}$ *can be computed from* $\{x_j | 1 \leqq j \leqq n\}$ *in* $\lceil \log_2 n \rceil$ *steps using* $\lfloor n/2 \rfloor$ *processors.*

Using Lemma 5, the following lemma can be proved as shown in [14] and [16].

Lemma 6. *Let* $n(d)$ *denote the degree of a polynomial form,* $PF(n(d))$, *which can be evaluated in* d *steps. If an unlimited number of processors are available, then* $n(d_r) \geqq 2^{d_{r-1}}$ *holds, where* $d_r = r(r + 1)/2$, $r \geqq 2$.

In order to prove Lemma 7 and Theorem 10, we will find the following notation useful. Given any polynomial form $PF(2^{d_k})$, assume it is rewritten in the form

$$PF_0(2^{d_{k-1}}) \, \theta_0 \sum_{j=1}^{2^k - 1} (PF_j'(2^{d_{k-1}}) \, \theta_j \, X(j \cdot 2^{d_{k-1}})), \tag{8}$$

where $\theta_0 \in \{+, -\}$, $\theta_j \in \{*, /\}$, $1 \leqq j \leqq 2^k - 1$, and the $PF_j'(2^{d_{k-1}})$, $0 \leqq j \leqq 2^k - 1$, are polynomial forms of degree less than or equal to $2^{d_{k-1}}$.

We then write $\{X | PF(2^{d_k})\}$ to denote the set of all products and quotients of x_i which appear in this form. For example, if we are given some $PF(8)$ which is rewritten as

$$(a_0 + a_1 x_1 + a_2 x_1 x_2) + (a_3 + a_4 x_4 + a_5 x_4 x_5) x_1 x_2 x_3$$
$$+ (a_6 + a_7 x_7 + a_8 x_7 x_8) x_1 x_2 x_3 x_4 x_5 x_6,$$

then

$$\{X | PF(8)\} = \{x_1, x_1 x_2, x_4, x_4 x_5, x_1 x_2 x_3, x_7, x_7 x_8, x_1 x_2 x_3 x_4 x_5 x_6\}.$$

Now we turn to Lemma 7, which provides the key to bounding the number of processors required to evaluate a polynomial form.

Lemma 7. *For any* $k \geqq 2$, *both* $X(2^{d_k}) = \prod_{j=1}^{2^{d_k}} x_j$ *and* $\{X | PF(2^{d_k})\}$, *the set of* x_i *products and quotients corresponding to* $PF(2^{d_k})$, *can be evaluated in* d_k *steps using* $2^{d_k - 1}$ *processors.*

Proof. We prove the lemma by induction on k. For $k = 2$, $\{X|PF(8)\}$ is

$$\{x_1, x_1\,\theta_1\,x_2, x_4, x_4\,\theta_4\,x_5, x_1\,\theta_1\,x_2\,\theta_2\,x_3, x_7, x_7\,\theta_7\,x_8, \\ x_1\,\theta_1\,x_2\,\theta_2\,x_3\,\theta_3\,x_4\,\theta_4\,x_5\,\theta_5\,x_6\}$$

and

$$X(8) = x_1\,\theta_1\,x_2\,\theta_2\,x_3\,\theta_3\,x_4\,\theta_4\,x_5\,\theta_5\,x_6\,\theta_6\,x_7\,\theta_7\,x_8,$$

where $\theta_i \in \{*, /\}$, $1 \leqq i \leqq 7$. By drawing computation trees, it can be seen that $\{x|PF(8)\}$ and $X(8)$ can be computed in $d_2 = 3$ steps using $2^{d_2-1} = 4$ processors.

Now we assume that the lemma holds for $k \leqq r - 1$, and we prove the lemma for $k = r$. Given any $PF(2^{d_r})$, we rewrite it in the form of (8) as

$$PF_0(2^{d_{r-1}})\,\theta_1 \sum_{j=1}^{2^r-1} PF'_j(2^{d_{r-1}})\,\theta_2\,X(j\cdot 2^{d_{r-1}}).$$

By induction hypotheses, we know that each $\{X|PF'_j(2^{d_{r-1}})\}$ of $PF'_j(2^{d_{r-1}})$ and $X_j(2^{d_{r-1}})$ for $j = 0, 1, \cdots, 2^r - 1$, can be computed in d_{r-1} steps using $2^{d_{r-1}-1}$ processors. Thus we need a total of $2^r \cdot 2^{d_{r-1}-1} = 2^{d_r-1}$ processors. Furthermore, using the $X_j(2^{d_{r-1}})$, all of the $X(j\cdot 2^{d_{r-1}})$ for $j = 1, 2, \cdots, 2^r$, can be computed in another r steps using 2^{r-1} processors, by Lemma 5. Note that for $j = 2^r$, $X(j\cdot 2^{d_{r-1}}) = X(2^{d_r})$. Further, the union of the remaining $X(j\cdot 2^{d_{r-1}})$ terms and the terms previously evaluated give us $\{X|PF(2^{d_r})\}$, that is,

$$\{X|PF(2^{d_r})\} = \bigcup_{j=0}^{2^r-1} [\{X|PF'_j(2^{d_{r-1}})\} \cup X(j\cdot 2^{d_{r-1}})].$$

Therefore, both $\{X|PF(2^{d_r})\}$ and $X(2^{d_r})$ can be computed in $r + d_{r-1} = d_r$ steps using 2^{d_r-1} processors. From this the lemma follows. Q.E.D.

THEOREM 10. *If* $2n$ *processors are available, then any polynomial form of degree* n, $PF(n)$, *can be evaluated in* $O_1(\log_2 n + \sqrt{8\log_2 n})$ *steps.*

Proof. If $P_1(2^{d_k})$ processors are required to evaluate $\{X|PF(2^{d_k})\}$ and $X(2^{d_k})$ in form (8) of the proof of Lemma 6, then by Lemma 7 we have $P_1(2^{d_k}) \leqq 2^{d_k-1}$. Moreover, if $P_2(2^{d_k})$ processors are required to evaluate $PF(2^{d_k})$ without computing $\{X|PF(2^{d_k})\}$, then we get

$$P_2(2^{d_k}) \leqq 2^k \cdot P_2(2^{d_{k-1}}) \leqq \cdots \leqq 2^k \cdot 2^{k-1} \cdots 2^3 \cdot P_2(2^{d_2}).$$

Since $P_2(2^{d_2}) = P_2(8) \leqq 4$, we have

$$P_2(2^{d_k}) \leqq 2^{k(k+1)/2-1} = 2^{d_k-1}.$$

Thus $P_1(2^{d_k}) + P_2(2^{d_k}) = 2^{d_k}$ processors are sufficient to evaluate $PF(2^{d_k})$. By noticing that 2^{d_k} covers discrete integers and from Lemma 6, the theorem follows. Q.E.D.

Let m be the number of atoms in a polynomial form. By Theorem 10, since $m = 2n + 1$, $PF(\lceil(m-1)/2\rceil)$ can be evaluated in

$$O_1(\log_2(m-1) + \sqrt{8\log_2\lceil(m-1)/2\rceil})$$

steps if $2\lceil(m-1)/2\rceil$ processors are available. Thus, we may say that a polynomial form of m distinct atoms, $m \geqq 2$, can be evaluated in $O_1(\log_2 m + \sqrt{8\log_2 m})$

steps if m processors are available. We observe that for the points $n = d_r$, the coefficient 8 may be reduced to 2, for a bound of $O_1(\log_2 n + \sqrt{2 \log_2 n})$ steps in Theorem 10.

7. Conclusion. This paper has presented several upper bounds on the time required to evaluate various classes of arithmetic expressions. Such bounds seem to be of fundamental importance in bounding the time required to evaluate various numerical algorithms using computers with multiple arithmetic units. Since the proofs are all constructive, they may also suggest techniques which could be used in compiling for such computers in the future.

Although the time bounds discussed in this paper are all within a constant factor of the obvious lower bound, it is likely that these results can be improved. It was conjectured in [17] that any arithmetic expression of n distinct atoms whose operations are addition and multiplication can be evaluated in at most $2\lceil \log_2 n \rceil$ steps. Here, as the summary of our experience, we propose the following two conjectures.

Conjecture 1. Any arithmetic expression of n distinct atoms whose operations are addition, subtraction and multiplication may be evaluated in at most $\log_2 n + O_2(\sqrt{\log n})$ steps using $O_2(n)$ processors (cf. Theorem 10).

Conjecture 2. Any arithmetic expression of n distinct atoms whose operations are addition, subtraction, multiplication and division may be evaluated in at most $O_1(2 \log_2 n)$ steps using $O_2(n)$ processors (cf. Theorem 9).

REFERENCES

[1] J. L. BAER AND D. P. BOVET, *Compilation of arithmetic expressions for parallel computations*, Proc. of IFIP Congress, North-Holland, Amsterdam, 1968, pp. 340–346.

[2] J. C. BEATTY, *An axiomatic approach to code optimization for expressions*, J. Assoc. Comput. Mach., 19 (1972), pp. 613–640.

[3] R. P. BRENT, *On the addition of binary numbers*, IEEE Trans. Computers, C-19 (1970), pp. 758–759.

[4] ———, *The parallel evaluation of arithmetic expressions in logarithmic time*, Complexity of Sequential and Parallel Numerical Algorithms, J. F. Traub, ed., Academic Press, New York, 1973, pp. 83–102.

[5] ———, *The parallel evaluation of general arithmetic expressions*, J. Assoc. Comput. Mach., 21 (1974), pp. 201–206.

[6] R. P. BRENT, D. J. KUCK AND K. MARUYAMA, *The parallel evaluation of arithmetic expressions without division*, IEEE Trans. Computers, C-22 (1973), pp. 532–534.

[7] P. M. KOGGE AND H. S. STONE, *A parallel algorithm for the efficient solution of a general class of recurrence equations*, Ibid., C-22 (1973), pp. 786–793.

[8] P. W. KRASKA, *Parallelism exploitation and scheduling*, Ph.D. thesis, Univ. of Illinois at Urbana-Champaign, 1972; Dept. Computer Sci. Rep. 518.

[9] D. J. KUCK, *Evaluating arithmetic expressions of n atoms and k divisions in* $\alpha(\log n + 2 \log k) + c$ *steps*, Doc. 69, Dept. Computer Sci., Univ. of Illinois at Urbana-Champaign, 1973.

[10] ———, *Multioperation machine computational complexity*, Complexity of Sequential and Parallel Numerical Algorithms, J. F. Traub, ed., Academic Press, New York, 1963, pp. 17–47.

[11] D. J. KUCK AND K. MARUYAMA, *The parallel evaluation of arithmetic expressions of special forms*, Rep. RC-4276, IBM T. J. Watson Research Center, Yorktown Heights, N.Y., 1973.

[12] D. J. KUCK AND Y. MURAOKA, *Bounds on the parallel evaluation of arithmetic expressions using associativity and commutativity*, Ann. Princeton Conf. on Information Sciences and Systems, Princeton, N.J., March 1973, pp. 161–168.

[13] D. J. KUCK, Y. MURAOKA AND S. CHEN, *On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up*, IEEE Trans. Computers, C-21 (1972), pp. 1293–1310.

[14] K. MARUYAMA, *On the parallel evaluation of polynomials*, Ibid., C-22 (1973), pp. 2–5.

[15] ———, *The parallel evaluation of matrix expressions*, RC 4380 (#19612), IBM T. J. Watson Research Center, Yorktown Heights, N.Y., June 1973.

[16] I. MUNRO AND M. PATERSON, *Optimal algorithm for parallel polynomial evaluation*, Proc. IEEE 12th Ann. Symp. on Switching and Automata Theory, Oct. 1971, pp. 132–139.

[17] Y. MURAOKA, *Parallelism exposure and exploitation in programs*, Ph.D. thesis, Univ. of Illinois at Urbana-Champaign, 1971; Dept. Computer Sci. Rep. 424.

[18] Y. MURAOKA AND D. J. KUCK, *On the time required for a sequence of matrix products*, Comm. ACM, 16 (1973), pp. 22–26.

[19] H. S. STONE, *An efficient parallel algorithm for the solution of a tridiagonal linear system of equations*, J. Assoc. Comput. Mach., 20 (1973), pp. 27–38.

SIAM J. COMPUT.
Vol. 4, No. 2, June 1975

SOLVING A PROBLEM IN EIGENVALUE APPROXIMATION WITH A SYMBOLIC ALGEBRA SYSTEM*

ANDREW D. HALL, JR.†

Abstract. In a recent paper by R. A. Handelsman and J. S. Lew [1], it is shown that for a certain family of potentials, the eigenvalues of the one-dimensional time-independent Schrödinger equation are proportional to $u(x, y)^{-2}$ with u determined by $u^{1/\beta} = f(xu, yu^{2/\beta})$, where $f(x, y) = \sum_{r=0}^{\infty}\sum_{s=0}^{\infty} f_{rs}x^s y^r$. Both β and the f_{rs} are known constants, with $f_{00} = 1$.

In [2], Lew proposed that a system for symbolic algebra be used to compute the Taylor series expansion for $u(x, y)$, although in [1] what is desired is the expansion of $c(x, y) = u(x, y)^{-2}$. To illustrate how such a system can be used to solve this and similar problems, three solution methods are described and corresponding programs given.

During the attempt to put the output resulting from these programs into a form similar to that given by Lew [1], the general solution was found. We give this solution here but defer the proof to another paper [3].

Key words. symbolic computation, power series

1. Introduction.

In a recent paper by R. A. Handelsman and J. S. Lew [1], it is shown that for the family of potentials

$$V(\chi) = A\chi^m + B\chi^{2m}, \qquad B > 0, \quad m = 2, 4, 6, \cdots, \tag{1}$$

the eigenvalues E_n of the one-dimensional time-independent Schrödinger equation are proportional to $u(x, y)^{-2}$, with u determined by

$$u^{1/\beta} = f(xu, yu^{2/\beta}). \tag{2}$$

Here $\beta = m/(m + 1)$, and the parameters x and y involve negative powers of $(n + \frac{1}{2})$. The function f is defined by

$$f(x, y) = \sum_{r=0}^{\infty}\sum_{s=0}^{\infty} f_{rs}x^s y^r, \tag{3}$$

where the f_{rs} are known constants. In particular, $f_{00} = 1$.

Handelsman and Lew define

$$c(x, y) = u(x, y)^{-2} = \sum_{p=0}^{\infty}\sum_{q=0}^{\infty} c_{pq}x^q y^p, \tag{4}$$

and the problem is to express the c_{pq} in terms of β and the f_{rs}. Handelsman and Lew were able to compute the c_{pq} for $p + q \leqq 4$ "by a hand computation lasting about a week" [2].

In this paper we present three methods for solving this problem and illustrate each with a program written in ALTRAN [4], [5]. Alternative methods and solutions are described in [6] and [7].

During the attempt to put the output resulting from these programs in a readable form, the general solution was discovered. We give this solution here but defer the proof to another paper [3].

* Received by the editors May 28, 1973, and in revised form June 11, 1974.
† Bell Telephone Laboratories, Inc., Murray Hill, New Jersey 07974.

2. Method 1. Method 1 is a brute force solution designed to take advantage of the ALTRAN library procedures for manipulating one-dimensional truncated power series. The method illustrates an often useful technique for converting a multidimensional power series problem into a one-dimensional problem that is easily solved.

The problem is considerably simplified if we let $u = v^\beta$ so that (2) becomes

$$v = f(xv^\beta, yv^2). \tag{5}$$

We wish now to determine the coefficients v_{pq} in the expansion

$$v(x, y) = \sum_{p=0}^{\infty} \sum_{q=0}^{\infty} v_{pq} x^q y^p, \tag{6}$$

and then use the relation $c = u^{-2} = v^{-2\beta}$ to determine the coefficients c_{pq} in (4).

We can make the problem one-dimensional by grouping terms according to the sum of the exponents of x and y. Formally, this is accomplished by setting $x = \lambda x$ and $y = \lambda y$ and viewing (3), (4) and (6) as power series in λ. These become, respectively,

$$f(\lambda x, \lambda y) = \sum_{k=0}^{\infty} f_k(x, y)\lambda^k, \qquad f_k(x, y) = \sum_{s=0}^{k} f_{k-s,s} x^s y^{k-s} \tag{7}$$

$$c(\lambda x, \lambda y) = \sum_{k=0}^{\infty} c_k(x, y)\lambda^k, \qquad c_k(x, y) = \sum_{s=0}^{k} c_{k-s,s} x^s y^{k-s} \tag{8}$$

$$v(\lambda x, \lambda y) = \sum_{k=0}^{\infty} v_k(x, y)\lambda^k, \qquad v_k(x, y) = \sum_{s=0}^{k} v_{k-s,s} x^s y^{k-s}. \tag{9}$$

Equation (5) becomes

$$v(\lambda x, \lambda y) = f(\lambda xv^\beta, \lambda yv^2). \tag{10}$$

The problem can now be solved in the following way. First represent $v(\lambda x, \lambda y)$ as a power series in λ to order n with unknown coefficients $v_0, v_1, \cdots, v_n$. Letting $\lambda = 0$ in (9) and (10), we find immediately that $v_0 = 1$. Now use (7) to compute to order n the power series for $f(\lambda x, \lambda y)$. Then substitute xv^β and yv^2 for x and y, respectively. The result will be a power series for $f(\lambda xv^\beta, \lambda yv^2) = v(\lambda x, \lambda v)$. Normally, we would have to equate the coefficients of these two series and solve the resulting system of equations for the unknowns $v_0, v_1, \cdots, v_n$. However, from (10) it is easy to show that the coefficient of λ^k in $f(\lambda xv^\beta, \lambda yv^2)$ depends only on x, y, β, f_{rs} and the unknown coefficients $v_1, \cdots, v_{k-1}$. Since we also know from (10) that these are precisely the previous coefficients in the series, simple substitution can be used to eliminate them. The result will be the desired power series for $v(\lambda x, \lambda y)$.

Using the relation $c = v^{-2\beta}$, we can now compute $c(\lambda x, \lambda y)$. From (8) we see that we can easily extract the c_{pq} of (4) from the coefficients of $c(\lambda x, \lambda y)$.

An ALTRAN program for carrying out this computation to order n is shown in Fig. 1. The program is straightforward except for the integer array-valued function maxexp which is used to compute the maximum exponent of each f_{rs} that can occur. For small n, we could have simply used n in place of the procedure call maxexp (n, n), but for large n we need a better bound to conserve space. For completeness, the procedure maxexp is discussed in detail in Appendix A.

```
        procedure main
        integer n = 4
        integer k, s
        integer array altran maxexp
# Declare the indeterminates and their maximum exponents.
        algebraic ( x:n, y:n, f(0:n,0:n):maxexp(n,n), b:n, v(0:n):n )
# Declare arrays for the coefficients of the truncated power series.
# In ALTRAN, the power series variable is not explicitly represented.
        long algebraic array (0:n) vtps, ctps, ftps = 0
        long algebraic array altran tpspwr, tpssbs
# Step 1.  As in (9), let vtps be a series with unknown coefficients,
#          v(0), v(1), . . . , v(n), but noting that v(0) = 1.
        vtps = v; vtps(0) = 1
# Step 2.  Using (7), compute the power series for f. Note that
#          f(0) = 1 and that ftps(k) is initially 0.
        ftps(0) = 1
        do k = 1, n
           do s = 0, k
              ftps(k) = ftps(k) + f(k-s,s) * x**s * y**(k-s)
           doend
        doend
# Step 3.  Compute the right side of (10), using truncated power series
#          substitution to replace x and y by x*vtps**b and y*vtps**2.
        ftps = tpssbs( ftps, x*tpspwr(vtps,b) , x )
        ftps = tpssbs( ftps, y*tpspwr(vtps,2) , y )
# Step 4.  Form vtps(k) by replacing the v(1), v(2) , . . . , v(k-1) in
#          ftps(k) with vtps(1), vtps(2), . . . , vtps(k-1).
        do k = 1, n
           vtps(k) = ftps(k) (v = vtps)
        doend
# Step 5.   Compute c and write out the coefficients.
        ctps = tpspwr( vtps, -2*b )
        do k = 0, n
           write ctps(k)
        doend
        end
```

FIG. 1. *Method* 1

3. Method 2. In Method 2, we derive a recurrence relation for the v_k defined by (9). We first replace x and y in (7) by xv^β and yv^2, respectively, to arrive at

$$v(\lambda x, \lambda y) = \sum_{k=0}^{\infty} \left(\sum_{s=0}^{k} f_{k-s,s} x^s y^{k-s} v^{\beta s + 2(k-s)} \right) \lambda^k. \tag{11}$$

Now let the γth power of v be represented by

$$v^\gamma = \sum_{k=0}^{\infty} (v^\gamma)_k \lambda^k, \tag{12}$$

where $(v^\gamma)_k$ simply denotes the coefficient of λ^k in the power series expansion of v^γ.

Substituting (12) into the right side of (11), we have

$$\sum_{k=0}^{\infty} v_k \lambda^k = \sum_{k=0}^{\infty} \left[\sum_{s=0}^{k} f_{k-s,s} x^s y^{k-s} \sum_{i=0}^{\infty} (v^{\beta s + 2(k-s)})_i \lambda^i \right] \lambda^k. \tag{13}$$

Equating terms of order n in λ, we have

$$v_n = \sum_{k=0}^{n} \sum_{s=0}^{k} f_{k-s,s} x^s y^{k-s} (v^{\beta s + 2(k-s)})_{n-k}. \tag{14}$$

Note that the terms with $k = 0$ do not contribute to the sum when $n > 0$, because $\beta s + 2(k - s) = 0$ and $(1)_n = 0$. Thus for $n > 0$, (14) expresses v_n in terms of the first $n - 1$ coefficients of the expansion of v^γ for various values of γ.

Furthermore, for $n > 0$ and γ arbitrary, $(v^\gamma)_n$ can be computed from $(v^\gamma)_0, \cdots, (v^\gamma)_{n-1}$ and $v_0, \cdots, v_n$ by the powering formula (see Appendix B)

$$(v^\gamma)_n = \frac{1}{nv_0} \sum_{i=1}^{n} [(\gamma + 1)i - n] v_i (v^\gamma)_{n-i}. \tag{15}$$

An ALTRAN program for computing v_n, $0 \leqq n \leqq$ nmax based on (14) and (15) is shown in Fig. 2. Since the procedure also computes $(v^\gamma)_n$, the c_n are easily obtained by setting $\gamma = -2\beta$.

4. Method 3. In Method 3, we use a technique similar to that used in Method 2 to derive a recurrence for the v_{pq} defined by (6). We first replace x and y in (3) by xv^β and yv^2, respectively, to arrive at

$$v(x, y) = \sum_{r=0}^{\infty} \sum_{s=0}^{\infty} f_{rs} x^s y^r v^{\beta s + 2r}. \tag{16}$$

Now let the γth power of v be represented by

$$v^\gamma = \sum_{p=0}^{\infty} \sum_{q=0}^{\infty} (v^\gamma)_{pq} x^q y^p, \tag{17}$$

where $(v^\gamma)_{pq}$ is used to denote the coefficient of $x^q y^p$ in the expansion of v^γ. Substituting (17) into (16) and equating this to (6), we have

$$\sum_{p=0}^{\infty} \sum_{q=0}^{\infty} v_{pq} x^q y^p = \sum_{r=0}^{\infty} \sum_{s=0}^{\infty} f_{rs} x^s y^r \left[\sum_{i=0}^{\infty} \sum_{j=0}^{\infty} (v^{\beta s + 2r})_{ij} x^j y^i \right]. \tag{18}$$

```
        procedure main

        integer nmax = 4
        integer i, k, n, s
        integer array altran maxexp

  # Declare the indeterminates and their maximum exponents.

        algebraic(x:nmax,y:nmax,f(0:nmax,0:nmax):maxexp(nmax,nmax),
                b:nmax,g:nmax)

        long algebraic array (0:nmax) c, v = 0, vg = 0

  # Step 1.   Initialize.

        v(0) = 1; vg(0) = 1

        do n = 1, nmax

  # Step 2.   Using (14), compute v(n).

          do k = 1, n
            do s = 0, k
              v(n) = v(n) + f(k-s,s) * x**s * y**(k - s) *
              vg(n-k) (g = b*s+2* (k-s) )
            doend
          doend

  # Step 3.   Using (15), compute vg(n). Note that v(0) = 1 and that
  #           vg(n) is initially 0.

          do i = 1, n
            vg(n) = vg(n) + ((g+1)*i-n) * v(i) * vg(n-i)
          doend
          vg(n) = vg(n)/n

  # Step 4.   Compute c(n), write it out, and recover the space.

          c(n) = vg(n) (g = -2*b)
          write c(n); c(n)=.null.

        doend

        end
```

FIG. 2. *Method* 2

Equating the coefficients of $x^q y^p$, we have

$$v_{pq} = \sum_{r=0}^{p} \sum_{s=0}^{q} f_{rs}(v^{\beta s+2r})_{p-r,q-s}. \tag{19}$$

Note that the term with $r = s = 0$ does not contribute to the sum when $p > 0$ or $q > 0$, because $\beta s + 2r = 0$ and $(1)_{pq} = 0$. Hence (19) expresses v_{pq} in terms of $(v^\gamma)_{ij}$ with $i \leqq p$, $j \leqq q$ and $i + j < p + q$.

Now for $p > 0$ or $q > 0$ and arbitrary γ, we can express $(v^\gamma)_{pq}$ in terms of the

```
        procedure main

        integer pmax = 4, qmax = 4
        integer i, j, p, q, r, s
        integer array altran maxexp

# Declare the indeterminates and their maximum exponents.

        algebraic (f(0:pmax,0:qmax):maxexp(pmax,qmax), b:pmax+qmax,
                g:pmax+qmax)

        long algebraic array (0:pmax, 0:qmax) c, v = 0, vg = 0

# Step 1.   Initialize.

        v(0,0) = 1; vg(0,0) = 1

        do p = 0, pmax
          do q = 0, qmax

            if (p+q .eq. 0) go to skip

# Step 2.   Using (19), compute v(p,q).

            do r = 0, p
              do s = 0, q
                v(p,q) = v(p,q) + f(r,s) * vg(p-r,q-s) (g=b*s+2*r)
              doend
            doend

# Step 3.   Using (20), compute vg(p,q). Note that v(0,0) = 1 and that
#           because vg(p,q) is initially 0, the term with i = j = 0 does
#           not contribute to the sum

            do i = 0, p
              do j =0, q
                vg(p,q) = vg(p,q) + (g+1)*(i+j) - (p+q))*v(i,j)*
                vg(p-i,q-j)
              doend
            doend
            vg(p,q) = vg(p,q)/(p+q)

# Step 4.   Compute c(p,q), write it out, and recover the space.

skip:       c(p,q) =vg(p,q) (g = -2*b)
            write c(p,q); c(p,q) = .null.

          doend
        doend

        end
```

FIG. 3. *Method* 3

lower order coefficients of v^γ and the v_{ij}, $i \leqq p$ and $j \leqq q$, by the powering formula (see Appendix C)

$$(20) \qquad (v^\gamma)_{pq} = \frac{1}{(p+q)v_{00}} \sum_{\substack{i=0 \\ i+j\neq 0}}^{p} \sum_{j=0}^{q} [(\gamma+1)(i+j) - (p+q)]v_{ij}(v^\gamma)_{p-i,q-j}.$$

An ALTRAN program for computing v_{pq}, $p \leqq$ pmax, $q \leqq$ qmax, is shown in Fig. 3. Since the procedure also computes $(v^\gamma)_{pq}$, c_{pq} can be obtained by setting $\gamma = -2\beta$.

5. Results. Methods 1 and 2 would normally be used to compute c_{pq} for all p, q such that $p + q \leqq n$, whereas Method 3 would be used to compute c_{pq} for $p \leqq$ pmax, $q \leqq$ qmax. To compare the speeds of these techniques, Method 3 was modified slightly so that it also computes c_{pq} for $p + q \leqq n$.

TABLE 1
Total processor time in seconds required for the computation of all c_{pq} for $p + q \leqq n$

n	*Method* 1	*Method* 2	*Method* 3
1	4.6	1.3	1.9
2	8.6	3.1	5.2
3	16.0	6.8	13.1
4	26.6	13.3	29.8

The processor time in seconds required for computing all c_{pq} with $p + q \leqq n$ for $n = 1, \cdots, 4$ is shown in Table 1. The ALTRAN system used was installed on the Honeywell 6070 computer (36 bit words, 1 μs cycle time) at Bell Telephone Laboratories, Murray Hill, New Jersey.

In each case, no more than 10,000 words of workspace were used. Using Method 2 with 24,000 words of workspace, we were able to compute all c_{pq} for $p + q \leqq 6$ in 53.1 seconds.

Part of the output resulting from Method 3 (modified) is shown in Fig. 4.

To put the c_{pq}, $p + q \leqq 6$, in a form similar to that given in [1], an attempt was made to devise a heuristic program to collect terms of the same degree in the f_{rs} and factor the resulting coefficient which is a polynomial in β. For example, c_{21} can be written

$$(21) \qquad C_{21} = -2\beta[f_{21} + (f_{10}f_{11} + f_{20}f_{01})(3-\beta) + \tfrac{1}{2}f_{10}^2 f_{01}(3-\beta)(2-\beta)].$$

This effort led to the discovery of the general form of the $(v^\gamma)_{pq}$ appearing in (6). Derivation of this solution will be deferred to a subsequent paper [3]. The formula is as follows:

$$(22) \qquad (v^\gamma)_{pq} = \frac{\gamma}{2p + \beta q + \gamma} \sum_{t=0}^{p+q} F_t(p,q)(2p + \beta q + \gamma)_t,$$

```
# C(2,0)
        2*F(1,0)**2*B**2 - 3*F(1,0)**2*B - 2*F(2,0)*B
# C(0,3)
        ( - F(0,1)**3*B**3 + 3*F(0,1)**3*B**2 - 2*F(0,1)**3*
        B - 6*F(0,1)*F(0,2)*B**2 + 6*F(0,1)*F(0,2)*B -
        6*F(0,3)*B) / 3
# C(1,2)
        - 2*F(0,1)*F(1,1)*B - 2*F(0,2)*F(1,0)*B - 2*F(1,2)*B
# C(2,1)
        - F(0,1)*F(1,0)**2*B**3 + 5*F(0,1)*F(1,0)**2*B**2 -
        6*F(0,1)*F(1,0)**2*B + 2*F(0,1)*F(2,0)*B**2 -
        6*F(0,1)*F(2,0)*B + 2*F(1,0)*F(1,1)*B**2 - 6*F(1,0)*
        F(1,1)*B - 2*F(2,1)*B
# C(3,0)
        ( - 4*F(1,0)**3*B**3 + 18*F(1,0)**3*B**2 -
        20*F(1,0)**3*B + 12*F(1,0)*F(2,0)*B**2 - 30*F(1,0)*
        F(2,0)*B - 6*F(3,0)*B ) / 3
```

FIG. 4. *Output from Method* 3

where the $F_t(r, s)$ are defined by

$$\sum_{r=0}^{\infty}\sum_{s=0}^{\infty}\sum_{t=0}^{\infty} F_t(r, s)x^s y^r z^t = e^{(f(x,y)-1)z} \tag{23}$$

and the notation $(\delta)_t$ is the falling factorial

$$(\delta)_0 = 1, \qquad (\delta)_t = \delta(\delta - 1)\cdots(\delta - t + 1), \qquad t > 0.$$

From (23) and the fact that $f_{00} = 1$, it is easy to derive the following identities which permit the computation of $F_t(r, s)$ for all t, r and s.

$$F_0(0, 0) = 1,$$
$$F_0(r, s) = 0, \qquad r + s > 0,$$
$$F_t(r, s) = 0, \qquad t > r + s,$$
$$F_{t+1}(r, s) = \frac{1}{r + s}\sum_{i=0}^{r}\sum_{j=0}^{s}(i + j)f_{ij}F_t(r - i, s - j), \qquad r + s > 0.$$

For example, we find that

$$F_0(2,1) = 0, \qquad F_2(2,1) = f_{10}f_{11} + f_{20}f_{01},$$
$$F_1(2,1) = f_{21}, \qquad F_3(2,1) = \tfrac{1}{2}f_{10}^2 f_{01}.$$

Thus

$$c_{21} = (v^{-2\beta})_{21} = -2\beta[f_{21} + (f_{10}f_{11} + f_{20}f_{01})(3-\beta) + \tfrac{1}{2}f_{10}^2 f_{01}(3-\beta)(2-\beta)],$$

in accordance with (21).

6. Conclusion. Although the need for a program to solve this particular problem has been obviated by the discovery of the general solution, it is clear that algebraic manipulation systems—and in particular, procedures for manipulating truncated power series—can greatly simplify the computation of solutions to seemingly difficult problems. Of the three methods described for solving this particular problem, Method 1, which uses a package specifically designed for the manipulation of truncated power series, was by far the easiest to derive and program. Methods 2 and 3 required considerably more work for only a small improvement in performance. This illustrates the fallacy of using processor time as the sole measure of the value or quality of an algebraic manipulation system. More often, the relevant comparison is the ease with which a given problem is solved, provided the cost is not exorbitant.

Appendix A. Maximum exponents. In order to write ALTRAN programs that make reasonably economical use of storage, it is sometimes necessary to obtain tight bounds for the exponents of the indeterminates. This is particularly true when a large number of indeterminates are present.

It is easy to show by induction from (19) and (20) that the $(v^\gamma)_{pq}$ are "homogeneous" in the sense each term satisfies the identities

$$\sum_{r=0}^{p}\sum_{s=0}^{q} r\cdot\varepsilon_{rs} = p, \qquad \varepsilon_{rs} \leqq p,$$
$$\sum_{r=0}^{p}\sum_{s=0}^{q} s\cdot\varepsilon_{rs} = q, \qquad \varepsilon_{rs} \leqq q,$$

where ε_{rs} is the exponent of f_{rs}.

It follows immediately that in $(v^\gamma)_{pq}$,

$$\varepsilon_{rs} \leqq \min\left(\left\lfloor\frac{p}{\max(1,r)}\right\rfloor, \left\lfloor\frac{q}{\max(1,s)}\right\rfloor\right).$$

Since $f_{00} = 1$, it does not appear formally in any of our computations, and we take $\varepsilon_{00} = 1$ (ALTRAN does not allow maximum exponent declarations to be 0). The procedure maxexp used in each of our programs is shown in Fig. 5.

It is also easy to show from (19) and (20) that for $p + q > 0$, the exponent of β in v_{pq} cannot exceed $p + q - 1$. Taking $\gamma = -2\beta$ in (20), we can therefore show that the maximum exponent of β in c_{pq} cannot exceed $p + q$.

```
procedure maxexp ( pmax, qmax )

integer pmax, qmax, r, s

integer array ( 0:pmax, 0:qmax ) exp

do r = 0, pmax
   do s = 0, qmax
      exp (r,s) = imin ( iquo(pmax,imax(1,r)), iquo(qmax,imax(1,s)))
   doend
doend

exp (0,0) = 1

return (exp)

end
```

FIG. 5. *The procedure* maxexp

Appendix B. Powering of one-dimensional series. We derive here the formula attributed to J. C. P. Miller in [8] for powering a one-dimensional series. Using the previous notation, let

$$v = \sum_{k=0}^{\infty} v_k \lambda^k \tag{B.1}$$

and

$$w = v^{\gamma} = \sum_{k=0}^{\infty} (v^{\gamma})_k \lambda^k. \tag{B.2}$$

Taking the derivative of w with respect to λ, we have

$$w' = \gamma v^{\gamma-1} v', \tag{B.3}$$

$$v w' = \gamma v' w. \tag{B.4}$$

Replacing v, w, v' and w' by their power series representations and equating coefficients of λ^n, we have

$$\sum_{i=0}^{n} v_i (n-i)(v^{\gamma})_{n-i} = \gamma \sum_{i=0}^{n} i v_i (v^{\gamma})_{n-i}, \tag{B.5}$$

and finally,

$$0 = \sum_{i=0}^{n} [(\gamma+1)i - n] v_i (v^{\gamma})_{n-i}. \tag{B.6}$$

For $n > 0$ and $v_0 \neq 0$, we can solve (B.6) for $(v^{\gamma})_n$. Thus

$$(v^{\gamma})_n = \frac{1}{n v_0} \sum_{i=1}^{n} [(\gamma+1)i - n] v_i (v^{\gamma})_{n-i}, \tag{B.7}$$

in accordance with (15). For $n = 0$, we obviously have $(v^{\gamma})_0 = v_0^{\gamma}$.

Appendix C. Powering of two-dimensional series. In a manner similar to that used in Appendix B, we derive a formula for the powering of two-dimensional series. Let

$$v = \sum_{p=0}^{\infty} \sum_{q=0}^{\infty} v_{pq} x^q y^p \tag{C.1}$$

and

$$w = v^\gamma = \sum_{p=0}^{\infty} \sum_{q=0}^{\infty} (v^\gamma)_{pq} x^q y^p. \tag{C.2}$$

Taking the derivative of w with respect to x, we have

$$w' = \gamma v^{\gamma-1} v', \tag{C.3}$$

$$vw' = \gamma w v'. \tag{C.4}$$

Replacing v, w, v' and w' by their power series representations and equating the coefficients of $x^q y^p$, we have

$$\sum_{i=0}^{p} \sum_{j=0}^{q} v_{ij}(q-j)(v^\gamma)_{p-i,q-j} = \gamma \sum_{i=0}^{p} \sum_{j=0}^{q} j v_{ij} (v^\gamma)_{p-i,q-j}, \tag{C.5}$$

or

$$0 = \sum_{i=0}^{p} \sum_{j=0}^{q} [(\gamma+1)j - q] v_{ij} (v^\gamma)_{p-i,q-j}. \tag{C.6}$$

Unfortunately, (C.6) is trivial for $q = 0$. However, by repeating the above process using derivatives with respect to y, we obtain

$$0 = \sum_{i=0}^{p} \sum_{j=0}^{q} [(\gamma+1)i - p] v_{ij} (v^\gamma)_{p-i,q-j}. \tag{C.7}$$

Adding (C.6) and (C.7), we get

$$0 = \sum_{i=0}^{p} \sum_{j=0}^{q} [(\gamma+1)(i+j) - (p+q)] v_{ij} (v^\gamma)_{p-i,q-j}. \tag{C.8}$$

Now for $p + q > 0$ and $v_{00} \neq 0$, we can solve (C.8) for $(v^\gamma)_{pq}$. Thus

$$(v^\gamma)_{pq} = \frac{1}{(p+q)v_{00}} \sum_{\substack{i=0 \\ i+j\neq 0}}^{p} \sum_{j=0}^{q} [(\gamma+1)(i+j) - (p+q)] v_{ij} (v^\gamma)_{p-i,q-j}, \tag{C.9}$$

in accordance with (20). For $p + q = 0$, we obviously have $(v^\gamma)_{00} = v_{00}^\gamma$.

REFERENCES

[1] R. A. Handelsman and J. S. Lew, *Analytical evaluation of energy eigenvalues for a class of anharmonic oscillators*, J. Chem. Phys., 50 (1969), pp. 3342–3354.

[2] J. S. Lew, *Problem 3—Reversion of a double series*, SIGSAM Bull. 23, 1972, pp. 6–7.

[3] A. J. Goldstein and A. D. Hall, *Solutions to a problem in power series reversion*, SIAM J. Math. Anal., 6 (1975), pp. 192–198.

[4] W. S. Brown, *ALTRAN User's Manual*, Bell Telephone Laboratories, Murray Hill, N.J., 1971; 2nd ed., 1972.

[5] A. D. HALL, *ALTRAN Installation and Maintenance*, Bell Telephone Laboratories, Murray Hill, N.J., 1971; 2nd ed., 1972.

[6] R. LOOS, *A user's solution of Problem #3 with REDUCE 2*, SIGSAM Bull. 26, 1973, pp. 12–14.

[7] J. FITCH, *A solution to Problem #3*, SIGSAM Bull. 26, 1973, pp. 24–27.

[8] P. HENRICI, *Automatic computation with power series*, J. Assoc. Comput. Mach., 3 (1956), pp. 10–15.

SIAM J. Comput.
Vol. 4, No. 2, June 1975

MATRIX FACTORIZATION OVER $GF(2)$ AND TRACE-ORTHOGONAL BASES OF $GF(2^n)$*

ABRAHAM LEMPEL†

Abstract. The main result of this paper is a theorem showing that every binary, symmetric matrix A can be factored over $GF(2)$ into $A = BB'$, where the number of columns of B is bounded from below by either the rank $\rho(A)$ of A, or by $\rho(A) + 1$, depending on whether at least one, or none, of the main-diagonal entries of A is nonzero. An algorithm for a minimal factorization of a given matrix A and an application of this result for finding a trace-orthogonal basis of $GF(2^n)$ are presented.

Key words. matrix factorization, trace, trace-orthogonal basis, finite fields

1. Statement of the main result.

1. Statement of the main result. Throughout this paper, all matrix operations and concepts such as rank, linear dependence, etc., are taken over the finite field of two elements $GF(2)$.

Let $A = (A_{ij})$ be a symmetric matrix of rank $\rho(A)$ and let

$$\delta(A) = \begin{cases} 1 & \text{if } A_{ii} = 0 \text{ for all } i, \\ 0 & \text{otherwise.} \end{cases}$$

A matrix B is called a *factor* of A if $A = BB'$, where B' is the transpose of B; B is called a *minimal factor* of A if no factor of A has fewer columns than B. The number of columns of a minimal factor of A will be denoted by $\mu(A)$. The main result can now be stated as follows.

Theorem 1. *Every binary, symmetric matrix A has a factor over $GF(2)$, and*

$$\mu(A) = \rho(A) + \delta(A). \tag{1}$$

Theorem 1 consists of two parts: an existence statement and a statement regarding the number of columns of a minimal factor. The existence part is proved in § 2 by exhibiting a simple construction which always results in a so-called elementary factorization. In § 3 we derive an algorithm for reducing an elementary factor of a nonsingular matrix A into a minimal one and thereby validate (1). In § 4 we extend the results of § 3 to the case of singular matrices A. A special case of the main result is discussed in § 5, where it is shown that every finite extension of $GF(2)$ contains a trace-orthogonal basis.

2. Elementary factorization.

2. Elementary factorization. Consider a symmetric matrix A of order n, and let $N = \{1, 2, \cdots, n\}$. We define a subset N_1 of N and a set N_2 of ordered pairs (i, j), $i < j$, from N as follows:

$$N_1 = \left\{ k \in N \,\middle|\, \sum_{j=1}^{n} A_{kj} = 1 \right\},$$

$$N_2 = \{(i, j) | i, j \in N, i < j, \text{and } A_{ij} = 1\}.$$

A *k-column*, $k \in N_1$, is a binary column of n rows with a 1 in row k and zeros

* Received by the editors July 20, 1973, and in revised form July 8, 1974.

† Sperry Rand Research Center, Sudbury, Massachusetts. Now at Department of Electrical Engineering, Technion–Israel Institute of Technology, Haifa, Israel.

elsewhere. An *(i, j)-column*, $(i, j) \in N_2$, is a binary column of n rows with a 1 in rows i and j and zeros elsewhere. The existence part of Theorem 1 is established by the following lemma.

LEMMA 1. *Let E be a matrix of n rows and $|N_1| + |N_2|$ columns such that E contains one k-column for each $k \in N_1$ and one (i, j)-column for each $(i, j) \in N_2$. Then E is a factor of A.*

The validity of this lemma is a straightforward consequence of the definitions of the sets N_1 and N_2, and hardly needs a formal proof. Before going through whatever proof is necessary, we offer the following example.

Example 1. Let

$$A = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}.$$

We have $N_1 = \{1, 3\}$ and $N_2 = \{(1,2), (1,3), (2,4), (3,4)\}$. Hence

$$E = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix},$$

and the reader can easily verify that $A = EE'$.

To verify the lemma, one readily observes that for $i \neq j$,

$$(EE')_{ij} = \sum_k E_{ik}E_{jk} = \begin{cases} 1 & \text{if } E \text{ contains an } (i, j)\text{-column}, \\ 0 & \text{otherwise}. \end{cases}$$

Hence for $i \neq j$, $A_{ij} = (EE')_{ij}$.

The main-diagonal entries of EE' are given by

$$(EE')_{kk} = \sum_j E_{kj}^2 = \sum_j E_{kj}.$$

Now, $E_{kj} = 1$ iff one of the following three alternatives holds:

(i) the jth column of E is a k-column;

(ii) the jth column of E is an (i, k)-column, $i < k$;

(iii) the jth column of E is a (k, j)-column, $k < j$.

Alternative (i) holds iff $\sum_j A_{kj} = 1$. The contribution (if any) of (i, k)-columns to the sum of row k is equal to $\sum_{i<k} A_{ik}$, and that of the (k, j)-columns is equal to $\sum_{j>k} A_{kj}$. Since $A_{ik} = A_{ki}$, we have

$$(EE')_{kk} = \sum_j A_{kj} + \sum_{i<k} A_{ki} + \sum_{j>k} A_{kj} = A_{kk},$$

which completes the proof.

In the following section we derive a procedure for reducing an elementary factor E of A to a minimal one.

3. Minimal factorization. Let $r(B)$ and $c(B)$ denote, respectively, the number of rows and columns of a matrix B. Let u denote an all-1 column whose number of rows $r(u)$ is implicitly specified by the context in which it is used; e.g., in Bu, $r(u) = c(B)$.

LEMMA 2. *If B is a factor of A, then*

$$c(B) \geqq \rho(A) + \delta(A). \tag{2}$$

Proof. Clearly, $c(B) \geqq \rho(B)$. Moreover, since $A_{kk} = \sum_j B_{kj}$, and, by the definition of $\delta(A)$, $Bu = 0$ iff $\delta(A) = 1$, it follows that $c(B) - \delta(A) \geqq \rho(B)$. Since $\rho(B) \geqq \rho(A)$, we obtain $c(B) - \delta(A) \geqq \rho(A)$. Q.E.D.

In Lemma 2, we have shown that $\delta(A) + \rho(A)$ is a lower bound on $\mu(A)$, the number of columns in a minimal factor of A. The following lemmas are needed to establish a procedure for achieving this bound.

LEMMA 3. *Let Z be a binary matrix such that $Zu = 0$ and $c(Z)$ is even. Let $\hat{Z} = Z + xu'$, where x is an arbitrary binary vector with $r(x) = r(Z)$. Then*

$$\hat{Z}\hat{Z}' = ZZ'. \tag{3}$$

Proof. We have

$$\hat{Z}\hat{Z}' = ZZ' + Zux' + xu'Z' + xu'ux'.$$

Since $Zu = 0$, also $u'Z' = 0$. Since $r(u) = c(Z)$ is even, $u'u = 0$ and, hence, (3) holds. Q.E.D.

LEMMA 4. *If A is nonsingular, if B is a factor of A, and $c(B) > \rho(A) + \delta(A)$, then B contains a* proper *subset of columns whose sum is zero.*

Proof. It is clear that if $A = BB'$ and A is nonsingular, then $\rho(A) = \rho(B)$. With $c(B) > \rho(A) + \delta(A)$, we obtain $\rho(B) + \delta(A) < c(B)$. Hence, independently of the value of $\delta(A)$, the columns of B are linearly dependent. Furthermore, if $\delta(A) = 1$, any subset of $\rho(B) + 1$ columns of B is proper and linearly dependent. Since over $GF(2)$ every set of dependent columns contains a subset which sums to zero, the lemma is valid when $\delta(A) = 1$. Now, if $\delta(A) = 0$, then $Bu \neq 0$, but, since the columns of B are still dependent, there must be a proper subset of columns of B whose sum is zero. Q.E.D.

It follows, that under the conditions of Lemma 4, B can be partitioned (possibly, after an appropriate reordering of columns, which does not affect the product BB') as $B = [F \quad G]$, where

$$c(F) \geqq 1, \tag{4}$$

$$c(G) \geqq 1 \tag{5}$$

and

$$Gu = 0. \tag{6}$$

Let

$$Z = \begin{cases} G & \text{if } c(G) \text{ is even,} \\ [G \quad 0] & \text{if } c(G) \text{ is odd,} \end{cases} \tag{7}$$

where $[G \quad 0]$ is the matrix obtained by adjoining an all-zero column to G. Clearly,

$ZZ' = GG'$, and hence

$$B^* = [F \quad Z] \tag{8}$$

is a factor of A, with

$$Zu = 0, \tag{9}$$

and $c(Z)$ is even.

Now, let F_1 and Z_1 be the first columns of F and Z, respectively; let

$$x = F_1 + Z_1 \tag{10}$$

and let

$$\hat{Z} = Z + xu'. \tag{11}$$

By Lemma 3, $\hat{Z}\hat{Z}' = ZZ'$, and therefore

$$\hat{B} = [F \quad \hat{Z}] \tag{12}$$

is a factor of A.

Observing now that for $\hat{Z}_1$, the first column of $\hat{Z}$,

$$\hat{Z}_1 = Z_1 + x = Z_1 + F_1 + Z_1 = F_1,$$

it follows that the joint contribution of F_1 and $\hat{Z}_1$ to the product $\hat{B}\hat{B}'$ is null, and hence the matrix $\tilde{B}$ obtained by deleting F_1 and $\hat{Z}_1$ from $\hat{B}$ is also a factor of A. Reviewing the transformation of the given factor B into $\tilde{B}$, as described above, we observe that $\tilde{B}$ has one or two columns fewer than B, depending on whether $c(G)$ is odd or even, respectively. Thus the net effect of the transformation is a strict reduction in the number of columns, and we have just proved the following lemma.

LEMMA 5. *If A is nonsingular, if B is a factor of A, and $c(B) > \rho(A) + \delta(A)$, then there exists a factor $\tilde{B}$ of A such that $c(\tilde{B}) < c(B)$.*

This concludes the proof of Theorem 1 for the case when A is nonsingular. The existence part of the theorem is covered by Lemma 1, and the minimality part is covered by Lemma 2 and Lemma 5. Before proceeding to prove the singular case, we summarize the main steps of the minimal factorization procedure for a nonsingular matrix A.

Step 1. Find the elementary factor E of A according to Lemma 1. If $c(E) = \mu(A) = \rho(A) + \delta(A)$, stop; otherwise, set $B = E$ and proceed to Step 2.

Step 2. Find a proper subset of columns of B whose sum is zero, call the submatrix formed by these columns G, and partition B as $B = [F \quad G]$. Go to Step 3.

Step 3. Substitute Z for G, according to (7), to obtain $B^* = [F \quad Z]$. Set $x = F_1 + Z_1$ and replace each column Z_j of Z by $\hat{Z}_j = Z_j + x$, to obtain $\hat{B} = [F \quad \hat{Z}]$. Go to Step 4.

Step 4. Delete F_1 and $\hat{Z}_1$ from $\hat{B}$ to obtain the matrix $\tilde{B}$. If $c(\tilde{B}) = \mu(A)$, stop; otherwise, set $B = \tilde{B}$ and go to Step 2.

Step 1 of this procedure was illustrated in Example 1. In the following example we carry out the rest of the procedure to obtain a minimal factorization.

Example 2. For the matrix A of Example 1, we have $\rho(A) = 4$ and $\delta(A) = 0$. Since $c(E) = 6 > \mu(A) = 4$, we apply Step 2. It is easy to see that the last four columns of E sum to zero. Hence

$$F = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \quad \text{and} \quad G = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix},$$

since $c(G)$ is even, $Z = G$ and

$$x = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}.$$

Thus

$$\hat{Z} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix},$$

with $\hat{Z}_1 = F_1$. Deleting F_1 and $\hat{Z}_1$, we obtain

$$\tilde{B} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix},$$

with $c(\tilde{B}) = 4 = \mu(A)$. The reader can easily verify that $\tilde{B}$ is indeed a factor of the given matrix A.

4. Minimal factorization of singular matrices. Consider a singular, symmetric matrix A of order n and rank $\rho(A)$. There exists a permutation matrix P such that PA is of rank $\rho(A)$ and the first $\rho(A)$ rows of PA are independent. Postmultiplying PA by P' yields a symmetric matrix A^* which can be partitioned as

$$A^* = \begin{bmatrix} L & M \\ M' & K \end{bmatrix} = PAP' \tag{13}$$

where $L = L'$, $K = K'$ and

$$r[L \quad M] = \rho[L \quad M] = \rho(A^*) = \rho(A). \tag{14}$$

Hence there exists a nonsingular transformation matrix

(15) $$T = \begin{bmatrix} I & 0 \\ S & I \end{bmatrix}$$

such that

(16) $$TA^* = \begin{bmatrix} L & M \\ 0 & 0 \end{bmatrix}.$$

Thus

(17) $$SL + M' = 0$$

and

(18) $$SM + K = 0.$$

From (17), since L is symmetric, we obtain

(19) $$LS' + M = 0.$$

Equations (16) and (19) imply

(20) $$TA^*T' = \begin{bmatrix} L & 0 \\ 0 & 0 \end{bmatrix}$$

and, since T is nonsingular,

(21) $$\rho(A) = \rho(A^*) = \rho(TA^*T') = \rho(L).$$

Thus L is a nonsingular symmetric matrix and, by Lemmas 1, 2 and 5, L has a factor H such that

(22) $$c(H) = \rho(L) + \delta(L).$$

Observing that

(23) $$T^{-1} = T$$

and

(24) $$P^{-1} = P',$$

we obtain from (13), (20), (23) and (24)

(25) $$A = P'A^*P = P'T\begin{bmatrix} L & 0 \\ 0 & 0 \end{bmatrix}T'P.$$

Substituting HH' for L in (25), we have

(26) $$A = P'T\begin{bmatrix} H \\ 0 \end{bmatrix}[H'0]T'P.$$

Hence

(27) $$B = P'T\begin{bmatrix} H \\ 0 \end{bmatrix}$$

is a factor of A, and

$$c(B) = c(H) = \rho(L) + \delta(L). \tag{28}$$

Since $\rho(L) = \rho(A)$, to complete the proof of Theorem 1 for the singular case, it remains to show that

$$\delta(L) = \delta(A). \tag{29}$$

To this end, we first observe that since $\delta(A) = \delta(A^*)$, (29) is valid iff $\delta(L) = 1$ implies $\delta(K) = 1$ (see (13)). Because, if $\delta(L) = 0$, then L has a nonzero entry on its main diagonal and so does A, whence $\delta(A) = 0$. If $\delta(L) = \delta(K) = 1$, then both L and A have an all-zero main diagonal, and also $\delta(A) = 1$.

$$S = M'L^{-1} \tag{30}$$

and

$$K = SM = M'L^{-1}M. \tag{31}$$

Equation (31) can be rewritten as

$$K = RLR', \tag{32}$$

where $R = M'L^{-1}$. From (32) we have

$$K_{ii} = \sum_j \sum_k R_{ij}R_{ik}L_{jk}. \tag{33}$$

Since $L_{jk} = L_{kj}$, and summation is mod 2, the double sum of (33) reduces to

$$K_{ii} = \sum_j R_{ij}^2 L_{jj} = \sum_j R_{ij}L_{jj}. \tag{34}$$

It is clear from (34) that if $L_{jj} = 0$ for all j, then $K_{ii} = 0$ for all i, and hence $\delta(L) = 1$ implies $\delta(K) = 1$. This validates (29) and thus completes the proof of Theorem 1.

In summary, to find a minimal factor of a singular, symmetric matrix A, we have to find first a permutation matrix P which transforms A into A^* according to (13). Then we find a minimal factor H of L, according to the procedure of § 3, in terms of which a minimal factor of A is given by (27), or more explicitly, by

$$B = P'\begin{bmatrix} I & 0 \\ M'L^{-1} & I \end{bmatrix}\begin{bmatrix} H \\ 0 \end{bmatrix} = P'\begin{bmatrix} H \\ M'L^{-1}H \end{bmatrix} = P'\begin{bmatrix} H \\ M'(H^{-1})' \end{bmatrix}.$$

For instance, if

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix},$$

then

$$A^* = A = \left[\begin{array}{cc|c} 1 & 1 & 0 \\ 1 & 0 & 1 \\ \hline 0 & 1 & 1 \end{array}\right]$$

and

$$L = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}, \qquad M' = [0 \quad 1].$$

Here

$$H = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}, \qquad H^{-1} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}, \qquad M'(H^{-1})' = [1 \quad 0]$$

and

$$B = \begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 1 & 0 \end{bmatrix}.$$

5. Trace-orthogonal bases of $GF(2^n)$. An interesting application of Theorem 1 and the minimal factorization procedure is a method of finding a trace-orthogonal basis for the elements of the field $GF(2^n)$. The *trace* of $\alpha \in GF(2^n)$ is defined by

(35) $$T(\alpha) = \sum_{i=0}^{n-1} \alpha^{2^i}.$$

A basis $\Omega = \{\omega_1, \omega_2, \cdots, \omega_n\}$ of $GF(2^n)$ over $GF(2)$ is called a *trace-orthogonal basis* (in short, TOB) if

(36) $$T(\omega_i) = 1 \quad \forall \omega_i \in \Omega$$

and

(37) $$T(\omega_i \omega_j) = 0 \quad \forall \omega_i, \omega_j \in \Omega, \quad i \neq j.$$

Theorem 1, as subsequently shown, guarantees the existence of a TOB in $GF(2^n)$ for every positive integer n. Before we show this, it might be helpful to present first a brief review of some well-known [1], [2], [3] properties of the trace operator T and its connection with maximal-length sequences (in short, M-sequences) from a linear shift register. Recalling that [4] $\alpha \in GF(q)$ iff $\alpha^q = \alpha$ and that over $GF(2)$ $(\alpha + \beta)^2 = \alpha^2 + \beta^2$, one can readily verify that for all α, $\beta \in GF(2^n)$ and $a, b \in GF(2)$, the following properties hold:

(38) $$T(\alpha) \in GF(2),$$

(39) $$T(\alpha^2) = T(\alpha),$$

(40) $$T(a\alpha + b\beta) = aT(\alpha) + bT(\beta).$$

Let γ be a primitive element of $GF(2^n)$, i.e., the multiplicative order of γ is $p = 2^n - 1$, and let $g(x) = \sum_{i=0}^{n} g_i x^i$ be the minimal polynomial of γ^{-1} (the inverse of γ, which is also primitive). If

(41) $$a_k = T(\gamma^k), \qquad 0 \leqq k < p,$$

it is well known [2], [3] that the sequence $\{a_k\}$ is a binary M-sequence of period

p, satisfying the linear recursion

$$\sum_{i=0}^{n} g_i a_{k-i} = 0. \tag{42}$$

Let m be an integer such that $n \leqq m \leqq p$, and let

$$A_k = (a_k, a_{k+1}, \cdots, a_{k+m-1}), \qquad 0 \leqq k < p, \tag{43}$$

where the index of a_i is taken mod p.

It has been established [5] that the p m-tuples A_k, $0 \leqq k < p$, and the *all-zero* m-tuple Z form a field, isomorphic to $GF(2^n)$, under componentwise mod 2 addition and multiplication defined by

$$\begin{aligned} A_k Z &= Z A_k = Z, \qquad 0 \leqq k < p,\\ ZZ &= Z,\\ A_i A_j &= A_{i+j}, \qquad 0 \leqq i, j < p, \end{aligned}$$

where, again, all indices are taken mod p. The one-to-one correspondence between m-tuples, $n \leqq m \leqq p$, and the elements of $GF(2^n)$ expressed as powers of the primitive element γ is

$$\begin{aligned} A_k &\leftrightarrow \gamma^k, \qquad 0 \leqq k < p,\\ Z &\leftrightarrow 0. \end{aligned} \tag{44}$$

Indeed, applying recursion (42) to the m-tuples A_k, we obtain

$$\sum_{i=0}^{n} g_i A_{k-i} = 0. \tag{45}$$

Letting $A_{k-i} = \gamma^{k-i}$, we have

$$0 = \sum_{i=0}^{n} g_i \gamma^{k-i} = \gamma^k \sum_{i=0}^{n} g_i \gamma^{-i} = \gamma^k \sum_{i=0}^{n} g_i (\gamma^{-1})^i,$$

which is consistent with the assumption that $g(x)$ is the minimal polynomial for γ^{-1}.

Now, let $m = p$ and let $M = (M_{ij})$ be the square matrix of order p defined by

$$M_{ij} = a_{i+j}, \qquad 0 \leqq i, j < p. \tag{46}$$

It is easy to see that the kth row of M is the p-tuple A_k, $0 \leqq k < p$, and that M is symmetric. Since the rows of M represent the nonzero elements of $GF(2^n)$, its rank is

$$\rho(M) = n \tag{47}$$

and, by (45), any n successive rows of M span the rest of its rows. By (39) and (41),

$$a_k = a_{2k}, \qquad 0 \leqq k < p, \tag{48}$$

and since $a_{2k} = M_{kk}$, the sequence of entries on the main diagonal of M is identical with $\{a_k\}$. Thus

$$\delta(M) = 0, \tag{49}$$

and, by Theorem 1, there exists a factor B of M, i.e.,

$$BB' = M \tag{50}$$

with

$$c(B) = n. \tag{51}$$

It is clear that the n rows of B' form a basis for $GF(2^n)$ since they span every row of M. We proceed to show now that this basis is actually a TOB.

Let $\Omega = (\omega_0, \omega_1, \cdots, \omega_{n-1})'$ be the vector of n elements of $GF(2^n)$ corresponding to the rows of B'. Since each ω_i is expressible as a power of the primitive element γ, we may write

$$\omega_i = \gamma^{e_i}, \qquad 0 \leqq i \leqq n-1, \quad 0 \leqq e_i < p. \tag{52}$$

Substituting γ^k for the kth row of M, we can rewrite (50) as

$$B\begin{bmatrix} \gamma^{e_0} \\ \gamma^{e_1} \\ \cdot \\ \cdot \\ \gamma^{e_{n-1}} \end{bmatrix} = \begin{bmatrix} \gamma^0 \\ \gamma^1 \\ \gamma^2 \\ \cdot \\ \cdot \\ \gamma^{p-1} \end{bmatrix}. \tag{53}$$

Since for every nth order permutation matrix P, $BB' = BPP'B'$, there is no loss of generality in assuming

$$e_0 < e_1 < \cdots < e_{n-1}. \tag{54}$$

Since the kth row of B' corresponds to γ^{e_k}, which, in turn, corresponds to A_{e_k}, we have

$$B = (A'_{e_0}A'_{e_1} \cdots A'_{e_{n-1}}). \tag{55}$$

Thus if $B = (B_{ij})$, it follows from (43) and (55) that

$$B_{ij} = a_{i+e_j}, \qquad 0 \leqq i < p, \quad 0 \leqq j < n. \tag{56}$$

From (53) it is clear that for each i such that $0 \leqq i < n$,

$$B_{e_i j} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise}. \end{cases} \tag{57}$$

Combining (56) and (57), we obtain

$$a_{e_i+e_j} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise} \end{cases} \tag{58}$$

for all $i, j = 0, 1, \cdots, n-1$.

Finally, from (41) and (58), we have

$$T(\gamma^{e_i+e_j}) = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise}, \end{cases} \qquad 0 \leqq i,j < n. \tag{59}$$

Since

$$T(\gamma^{e_i+e_j}) = T(\gamma^{e_i}\gamma^{e_j})$$

and since for $i = j$,

$$T(\gamma^{2e_i}) = T(\gamma^{e_i}),$$

we have just proved that the n elements

$$\gamma^{e_0}, \gamma^{e_1}, \cdots, \gamma^{e_{n-1}}$$

form a TOB of $GF(2^n)$. We summarize this result in the following theorem.

THEOREM 2. *For every positive integer n, $GF(2^n)$ has a trace-orthogonal basis. The elements forming such a basis correspond to the n columns of a minimal factor B of M according to* (50).

For practical reasons, it is desirable to have a representation of a TOB of $GF(2^n)$ in terms of n binary n-tuples, rather than vectors of length $m > n$. Since the one-to-one correspondence (44) is valid for each m in the range $n \leqq m < p$, it follows that the n-tuples formed by the first n entries of each row of B' provide the desired representation. Thus if D' is the square matrix of order n formed by the first n columns of B', then the rows of D' form a TOB of $GF(2^n)$.

Let $L = (L_{ij})$ be the square submatrix of order n occupying the upper left corner of M. It is clear that

$$L = DD' \tag{60}$$

and that

$$L_{ij} = T(\gamma^{i+j}), \qquad 0 \leqq i,j < n. \tag{61}$$

Thus, given a primitive polynomial $g(x)$, or the M-sequence $\{a_k = T(\gamma^k)\}$, where $g(\gamma^{-1}) = 0$, it is easy to construct the matrix L according to (61). Applying the minimal factorization procedure of § 3 to L, we obtain D' according to (60) whose rows correspond to a TOB of $GF(2^n)$.

The practical significance of a TOB for $GF(2^n)$ is due to the following theorem.

THEOREM 3. *If $\Omega = \{\omega_0, \omega_1, \cdots, \omega_{n-1}\}$ is a* TOB *of $GF(2^n)$ and if $\gamma^i \in GF(2^n)$ is represented by the n-tuple $B_i = (B_{i,0}, B_{i,1}, \cdots, B_{i,n-1})$, where*

$$\gamma^i = \sum_{j=0}^{n-1} B_{ij}\omega_j, \qquad 0 \leqq i < p, \tag{62}$$

then

$$T(\gamma^i\gamma^k) = \sum_{j=0}^{n-1} B_{ij}B_{kj}. \tag{63}$$

Proof. We have

$$T(\gamma^i\gamma^k) = T\left[\sum_{j=0}^{n-1} B_{ij}\omega_j \sum_{m=0}^{n-1} B_{km}\omega_m\right] = T\left[\sum_{j=0}^{n-1}\sum_{m=0}^{n-1} B_{ij}B_{km}\omega_j\omega_m\right],$$

which, by (40), can be written as

$$T(\gamma^i\gamma^k) = \sum_{j=0}^{n-1}\sum_{m=0}^{n-1} B_{ij}B_{km}T(\omega_j\omega_m). \tag{64}$$

Since $T(\omega_j\omega_m) = 0$ if $j \neq m$ and $T(\omega_j^2) = T(\omega_j) = 1$, (64) reduces to (63). Q.E.D.

Theorems 2 and 3 may find further applications in the analysis and synthesis of M-sequences, where trace computations are quite prevalent.

REFERENCES

[1] S. W. Golomb, *Shift Register Sequences*, Holden-Day, San Francisco, 1967.

[2] R. Gold, *Characteristic linear sequences and their cost functions*, SIAM J. Appl. Math., 14 (1966), pp. 980–985.

[3] A. Lempel, *Analysis and synthesis of polynomials and sequencers over GF(2)*, IEEE Trans. Information Theory, IT-17 (1971), pp. 297–303.

[4] B. L. Van der Waerden, *Modern Algebra*, Frederick Ungar, New York, 1953.

[5] T. C. Bartee and P. E. Wood, *Coding for tracking radar ranging*, Tech. Rep. 318, MIT Lincoln Lab., Lexington, Mass., 1963.

SIAM J. COMPUT.
Vol. 4, No. 2, June 1975

BOUNDS FOR MULTIPROCESSOR SCHEDULING WITH RESOURCE CONSTRAINTS*

M. R. GAREY AND R. L. GRAHAM†

Abstract. One well-studied model of a multiprocessing system involves a fixed number n of identical abstract processors, a finite set of tasks to be executed, each requiring a specified amount of computation time, and a partial ordering on the tasks which requires certain tasks to be completed before certain others can be initiated. The nonpreemptive operation of the system is guided by an ordered list L of the tasks, according to the rule that whenever a processor becomes idle, it selects for processing the first unexecuted task on L which may validly be executed. We introduce an additional element of realism into this model by postulating the existence of a set of "resources" with the property that for each resource, the total usage of that resource at any instant of time may not exceed its total availability. For this augmented model, we determine upper bounds on the ratio of finishing times achieved using two different lists, L and L', and exhibit constructions to show that the bounds are best possible.

Key words. scheduling models, graph theory, worst-case analysis, performance bounds

1. Introduction. A number of authors (cf. [12], [16], [7], [3], [11], [4], [5], [9]) have recently been concerned with scheduling problems associated with a certain model of an abstract multiprocessing system (to be described in the next section) and, in particular, with bounds on the worst-case behavior of this system as a function of the way in which the inputs are allowed to vary. In this paper, we introduce an additional element of realism into the model by postulating the existence of a set of "resources" with the property that at no time may the system use more than some predetermined amount of each resource. With this extra constraint taken into consideration, we derive a number of rather close bounds on the behavior of this augmented system. It will be seen that this investigation also leads to several interesting results in graph theory and analysis.

2. The standard model. We consider a system composed of (usually n) abstract identical processors. The function of the system is to execute some given set $\mathscr{T} = \{T_1, \cdots, T_r\}$ of tasks. However, $\mathscr{T}$ is partially ordered by some relation[1] $\prec$ which must be respected in the execution of $\mathscr{T}$ as follows: if $T_i \prec T_j$, then the execution of T_i must be completed before the execution of T_j can begin. To each task T_i is associated a positive real number τ_i which represents the amount of time T_i requires for its execution. The operation of the system is assumed to be *non-preemptive*, which means that once a processor begins to execute a task T_i, it must continue to execute it to completion, τ_i time units later. Finally, the order in which the tasks are chosen is determined as follows: a permutation (or *list*) $L = \{T_{i_1}, \cdots, T_{i_r}\}$ of $\mathscr{T}$ is given initially. At any time a processor is idle, it instantaneously scans L from the beginning and selects the first task T_k (if any) which may validly be executed (i.e., all $T_i \prec T_k$ have been completed) and which is not currently being executed by another processor. Ties by two or more processors for the same task may be broken arbitrarily (since the processors are assumed to be identical).

* Received by the editors June 14, 1974, and in revised form August 9, 1974.

† Bell Laboratories, Murray Hill, New Jersey 07974.

[1] Thus, $\prec$ is transitive, antisymmetric and irreflexive.

The system begins at time $t = 0$ and starts executing $\mathcal{T}$. The *finishing time* ω is defined to be the least time at which all tasks have been completed. Of course, ω is a function of $L, \prec$, n and the τ_i. It is known [7] that if $\mathcal{T}' = \{T'_1, \cdots, T'_r\}$ with $T'_i \prec T'_j \Rightarrow T_i \prec T_j$ and $\tau_i \leqq \tau'_i$ for all i and j, and $\mathcal{T}'$ is executed by the system using a list L', then the corresponding finishing time ω' satisfies

$$\omega'/\omega \leqq 2 - 1/n. \tag{1}$$

Furthermore, this bound is best possible. Efficient procedures are known [3], [4], [9] for generating optimal lists when all the τ_i are 1 and *either* $\prec$ (viewed as a directed graph in the obvious way) is a tree *or* $n = 2$. However, Ullman [12] has recently shown that even the case of $n = 2$ and $\tau_i \in \{1, 2\}$ for all i is polynomial complete[2] and therefore probably has no efficient solution in general.

3. The augmented model. Before proceeding to a description of the new model we first introduce some notation which will make the ensuing discussion mathematically more convenient.

For a given list L, let $F: \mathcal{T} \to 2^{[0,\omega)}$ be defined by $F(T_i) = [\sigma_i, \sigma_i + \tau_i)$, where σ_i is the time at which the execution of T_i was started. Let $f: [0, \omega) \to 2^{\mathcal{T}}$ be defined by $f(t) = \{T_i \in \mathcal{T} : t \in F(T_i)\}$. Thus $f(t)$ is just the set of tasks which are being executed at time t. The restriction that we have at most n processors can be expressed by requiring $|f(t)| \leqq n$ for all $t \in [0, \omega)$.

Assume now that we are also given a set of resources $\mathcal{R} = \{\mathcal{R}_1, \cdots, \mathcal{R}_s\}$ and that these resources have the following properties. The total amount of resource $\mathcal{R}_i$ available at any time is (normalized without loss of generality to) 1. For each j, the task T_j requires the use of $\mathcal{R}_i(T_j)$ units of resource $\mathcal{R}_i$ at all times during its execution, where $0 \leqq \mathcal{R}_i(T_j) \leqq 1$. For each $t \in [0, \omega)$, let $r_i(t)$ denote the total amount of resource $\mathcal{R}_i$ which is being used at time t. Thus

$$r_i(t) = \sum_{T_j \in f(t)} \mathcal{R}_i(T_j).$$

In this augmented model, the fundamental constraint is simply this:

$$r_i(t) \leqq 1 \quad \text{for all } t \in [0, \omega).$$

In other words, at no time can we use more of any resource than is currently available.

The basic problem we shall consider is to what extent the use of different lists for this model can affect the finishing time ω.

4. Summary of results. There are essentially three results which will be proved in this paper. They all are derived from the following situation. We assume we are given a set of tasks $\mathcal{T} = \{T_1, \cdots, T_r\}$, execution times τ_i, a partial order $\prec$ on $\mathcal{T}$, a set of resources $\mathcal{R} = \{\mathcal{R}_1, \cdots, \mathcal{R}_s\}$, task resource requirements[3] $\mathcal{R}_i(T_j)$ and a positive integer n. For an arbitrary list L, let $\omega = \omega(L)$ be the finishing time for the (augmented) system of n processors executing $\mathcal{T}$ according to list L. Let $\omega^* = \omega(L^*)$ denote the minimum of $\omega(L)$ over all lists L. (Note that the use of $n \geqq r$ processors is equivalent to having an unlimited number of processors

[2] See [10] for a definition of this term.

[3] These are as described in the preceding section.

available, since clearly there can never be more than r processors active at any time.)

THEOREM 1. *For* $\mathscr{R} = \{\mathscr{R}_1\}$,

$$\omega/\omega^* \leqq n. \tag{2}$$

THEOREM 2. *For* $\mathscr{R} = \{\mathscr{R}_1, \mathscr{R}_2, \cdots, \mathscr{R}_s\}, \prec$ *empty, and* $n \geqq r$,

$$\omega/\omega^* \leqq s + 1. \tag{3}$$

THEOREM 3. *For* $\mathscr{R} = \{\mathscr{R}_1, \mathscr{R}_2, \cdots, \mathscr{R}_s\}, \prec$ *empty, and* $n \geqq 2$,

$$\frac{\omega}{\omega^*} \leqq \min\left\{\frac{n+1}{2}, s + 2 - \frac{2s+1}{n}\right\}. \tag{4}$$

By way of comparison, the following result (now a special case of Theorem 3) is proved in [7].

THEOREM 0. *For* $\mathscr{R} = \varnothing$,

$$\omega/\omega^* \leqq 2 - 1/n.$$

Furthermore, as in the case of Theorem 0, examples will be given to show that each of these results is essentially best possible.

Thus the addition of limited resources into the standard model causes an increase in the worst-case behavior bounds, as might be expected. What is somewhat surprising, however, is the significant effect the partial order $\prec$ can have on these bounds. This is in contrast to the previous case of $\mathscr{R} = \varnothing$ in which the upper bound $\omega/\omega^* \leqq 2 - 1/n$ which holds for arbitrary $\prec$ could, in fact, be achieved by examples with $\prec$ empty. Also significant is the apparent need for somewhat more sophisticated mathematical techniques than were required previously.

Proof of Theorem 1. The proof of (2) is immediate. We merely need to observe that

$$\omega \leqq \sum_{i=1}^{r} \tau_i \leqq n\omega^*,$$

since at no time before time ω are all processors idle when using list L, and the number of processors busy at any time never exceeds n.

More interesting is the following example, which shows that (2) is best possible.

Example 1.

$$T = \{T_1, \cdots, T_n, \hat{T}_1, \cdots, \hat{T}_n\}, \qquad \mathscr{R} = \{\mathscr{R}_1\}$$

$$\tau_i = 1, \qquad \hat{\tau}_i = \varepsilon > 0,$$

$$\mathscr{R}_1(T_i) = \frac{1}{n}, \qquad \mathscr{R}_1(\hat{T}_i) = 1, 1 \leqq i \leqq n.$$

$\prec$ is defined by

$$\hat{T}_i \prec T_i \quad \text{for } 1 \leqq i \leqq n,$$

$$L = (T_1, \cdots, T_n, \hat{T}_1, \cdots, \hat{T}_n), \qquad L' = (\hat{T}_1, \cdots, \hat{T}_n, T_1, \cdots, T_n).$$

A simple calculation[4] shows that

$$\omega = n + n\varepsilon, \qquad \omega^* = \omega' = 1 + n\varepsilon.$$

Thus

$$\frac{\omega}{\omega^*} = \frac{n + n\varepsilon}{1 + n\varepsilon} \to n \quad \text{as } \varepsilon \to 0.$$

Proof of Theorem 2. In this case, we assume $\mathscr{R} = \{\mathscr{R}_1, \mathscr{R}_2, \cdots, \mathscr{R}_s\}$, $\prec$ is empty and $n \geqq r$. The proof will require several preliminary results. The meaning of undefined terminology in graph theory may be found in [8].

Let G denote a graph with vertex set $V = V(G)$ and edge set $E = E(G)$. By a *valid labeling* L of G we mean a function $L: V \to [0, \infty)$ which satisfies

$$\text{(5)} \qquad \text{for all } e = \{a, b\} \in E, \quad L(a) + L(b) \geqq 1.$$

Define the *score of* G, denoted by $S(G)$, by

$$S(G) = \inf_L \left\{ \sum_{v \in V} L(v) \right\},$$

where the inf is taken over all valid labelings L of G.

LEMMA 1. *For any graph G, there exists a valid labeling $L: V \to \{0, \frac{1}{2}, 1\}$ such that*

$$S(G) = \sum_{v \in V} L(v).$$

Proof. For the case of a bipartite graph, König's theorem [8] states that the number of edges in a maximum matching equals the point covering number.[5] Thus for any bipartite graph G, there exists a valid labeling $L: V \to \{0, 1\}$ such that $S(G) = \sum_{v \in V} L(v)$.

For an arbitrary graph G, we construct a bipartite graph G_B as follows: for each vertex $v \in V(G)$ we have two vertices $v_1, v_2 \in V(G_B)$; for each edge $\{u, v\} \in E(G)$ we have two edges $\{u_1, v_2\}, \{u_2, v_1\} \in E(G_B)$. It is not difficult to verify that $S(G_B) = 2S(G)$ and furthermore, if $L_B: V(G_B) \to \{0, 1\}$ is a valid labeling of G_B, then $L: V(G) \to \{0, \frac{1}{2}, 1\}$ by $L(v) = \frac{1}{2}(L(v_1) + L(v_2))$ is a valid labeling of G. $\square$

For positive integers m and s, let $G(m, s)$ denote the graph with vertex set $\{0, 1, \cdots, (s + 1)m - 1\}$ and edge set consisting of all pairs $\{a, b\}$ for which $|a - b| \geqq m$.

LEMMA 2. *Suppose $G(m, s)$ is partitioned into s spanning subgraphs H_i, $1 \leqq i \leqq s$. Then*

$$\text{(6)} \qquad \max_{1 \leqq i \leqq s} \{S(H_i)\} \geqq m.$$

Proof. Assume the lemma is false, i.e., there exists a partition of $G(m, s)$ into H_i, $1 \leqq i \leqq s$, such that $S(H_i) < m$ for $1 \leqq i \leqq s$. Thus, by Lemma 1, for each i there exists a valid labeling $L_i: V(H_i) \to \{0, \frac{1}{2}, 1\}$ such that

$$\text{(7)} \qquad \sum_{v \in V(H_i)} L_i(v) = S(H_i) < m.$$

[4] The reader will probably find it helpful to construct a timing diagram to understand the behavior of this (and succeeding) examples.

[5] That is, the cardinality of the smallest set of vertices of G incident to every edge of G.

Let $A = \{a_1 < \cdots < a_p : L_i(a_j) \leqq \frac{1}{2}$ for all i, $1 \leqq i \leqq s\}$, and let S^* denote $\sum_{i=1}^{s} S(H_i)$. There are three cases.

(i) $p \leqq m$. In this case we have $S^* \geqq m(s+1) - p \geqq m(s+1) - m = ms$, which contradicts (7).

(ii) $m < p \leqq 2m + 1$. For each edge $\{a_j, a_{m+j}\}$, $1 \leqq j \leqq p - m$, there must exist an i such that $L_i(a_j) + L_i(a_{m+j}) \geqq 1$. Thus $S^* \geqq m(s+1) - p + (p - m) = ms$, again contradicting (7).

(iii) $p > 2m + 1$. We first note that for each vertex $v \in V(G(m, s))$, there exists an i such that $L_i(v) \geqq \frac{1}{2}$. For suppose $L_i(v) = 0$ for $1 \leqq i \leqq s$. There must be some a_j such that $|a_j - v| \geqq m$. But since $L_i(a_j) \leqq \frac{1}{2}$ for all i, then $L_i(a_j) + L_i(v) \leqq \frac{1}{2}$ for all i, which is a contradiction.

For each i, let n_i denote the number of vertices v such that $L_i(v) = 1$. Then

$$|\{v : L_i(v) > 0\}| \leqq 2m - 1 - n_i,$$

since otherwise

$$\sum_{v \in V(H_i)} L_i(v) \geqq n_i \cdot 1 + (2m - 2n_i) \cdot \tfrac{1}{2} = m,$$

which contradicts (7). Therefore

$$(8) \qquad \sum_{i=1}^{s} |\{v : L_i(v) > 0\}| \leqq (2m - 1)s - \sum_{i=1}^{s} n_i.$$

Let q denote the number of vertices v such that there is exactly one i for which $L_i(v) > 0$. Then

$$(9) \qquad \sum_{i=1}^{s} |\{v : L_i(v) > 0\}| \geqq 2(m(s+1) - q) + q.$$

Combining (8) and (9), we have

$$(10) \qquad q \geqq 2m + s + \sum_{i=1}^{s} n_i.$$

Of course, we may assume without loss of generality that if $L_i(v) = 1$, then $L_j(v) = 0$ for all $j \neq i$. Hence, by the definition of n_i, there must be at least $2m + s$ vertices, say $b_1 < \cdots < b_{2m+s}$, such that $\sum_{i=1}^{s} L_i(b_j) = \frac{1}{2}$, i.e., for each b_j there is a unique L_i such that $L_i(b_j) = \frac{1}{2}$ and $L_k(b_j) = 0$ for all $k \neq i$. Thus, if $|b_j - b_k| \geqq m$, then for some i, $L_i(b_j) = L_i(b_k) = \frac{1}{2}$. Since $|b_1 - b_{2m+s}| \geqq m$, let i_0 be such that $L_{i_0}(b_1) = L_{i_0}(b_{2m+s}) = \frac{1}{2}$. But, by the same reasoning we must also have $L_{i_0}(b_{m+j}) = L_{i_0}(b_1) = \frac{1}{2}$ and $L_{i_0}(b_{2m+s}) = L_{i_0}(b_j) = \frac{1}{2}$ for $1 \leqq j \leqq m + s$. Therefore

$$S(H_{i_0}) = \sum_{v \in V(H_{i_0})} L_{i_0}(v) \geqq (2m + s) \cdot \tfrac{1}{2} \geqq m,$$

which is a contradiction. This completes the proof of Lemma 2. □

Recall that when $\mathcal{T}$ is executed using the list L, $F(T_i)$ is defined to be the interval $[\sigma_i, \sigma_i + \tau_i)$, where σ_i is the time at which T_i starts to be executed and

$\sigma_i + \tau_i$ is the time at which T_i is finished. Note that because of the way in which the operation of the system is defined, each σ_i is a sum of a subset of the τ_j's.

We may assume without loss of generality that $\omega^* = 1$. Assume now that $\omega > s + 1$. Furthermore, suppose each τ_i can be written as $\tau_i = k_i/m$, where k_i is a positive integer. Thus $k_i \leqq m$, since $\tau_i \leqq \omega^* = 1$. Also, for $1 \leqq i \leqq s$, each $r_i(t)$ is constant on each interval $[k/m, (k+1)/m)$, this value being $r_i(k/m)$. An important fact to note is that since $\prec$ is empty and $n \geqq r$, then, for $t_1, t_2 \in [0, \omega)$ with $t_2 - t_1 \geqq 1$, we must have

$$\max_{1 \leqq i \leqq s} \{r_i(t_1) + r_i(t_2)\} > 1.$$

For otherwise, any task being executed at time t_2 should have been executed at time t_1 or sooner. Thus, for each i, $1 \leqq i \leqq s$, we can construct a graph H_i as follows:

$$\begin{gathered} V(H_i) = \{0, 1, \cdots, (s+1)m - 1\}; \\ \{a, b\} \text{ is an edge of } H_i \quad \text{iff} \quad r_i\left(\frac{a}{m}\right) + r_i\left(\frac{b}{m}\right) > 1. \end{gathered} \tag{11}$$

Note that if $|a - b| \geqq m$, then $\{a, b\}$ is an edge of at least one H_i, $1 \leqq i \leqq s$. Hence it is not difficult to see that $G(m, s) \subseteq \bigcup_i H_i$. Note that by (11), the mapping $L_i : V(H_i) \to [0, \infty)$ defined by $L_i(a) = r_i(a/m)$ is a valid labeling of H_i. Since $G \subseteq G'$ implies $S(G) \leqq S(G')$ and the condition on the r_i in (11) is a strict inequality, then by Lemma 2 it follows that

$$\max_i \left\{ \sum_{k=0}^{(s+1)m-1} r_i\left(\frac{k}{m}\right) \right\} = \max_i \left\{ \sum_{v \in V(H_i)} L_i(v) \right\} > \max_i \{S(H_i)\} \geqq m. \tag{12}$$

But we must have

$$\frac{1}{m} \sum_{k=0}^{(s+1)m-1} r_i\left(\frac{k}{m}\right) \leqq \int_0^\infty r_i(t)\, dt \leqq 1, \qquad 1 \leqq i \leqq s, \tag{13}$$

i.e.,

$$\sum_{k=0}^{(s+1)m-1} r_i\left(\frac{k}{m}\right) \leqq m, \qquad 1 \leqq i \leqq s.$$

This is a *contradiction*, and Theorem 2 is proved in the case that $\tau_i = k_i/m$, where k_i is a positive integer for $1 \leqq i \leqq r$. Of course, it follows immediately that Theorem 2 holds when all the τ_i are *rational*. The proof of Theorem 2 will be completed by establishing the following lemma.

LEMMA 3. *Let $\tau = (\tau_1, \cdots, \tau_r)$ be a sequence of positive real numbers. Then for any $\varepsilon > 0$, there exists $\tau' = (\tau'_1, \cdots, \tau'_r)$ such that*

(i) $|\tau'_i - \tau_i| < \varepsilon$ *for* $1 \leqq i \leqq r$;

(ii) *for all* $S, T \subseteq \{1, \cdots, r\}$,

$$\sum_{s \in S} \tau_s \leqq \sum_{t \in T} \tau_t \quad \textit{iff} \quad \sum_{s \in S} \tau'_s \leqq \sum_{t \in T} \tau'_t;$$

(iii) *all τ'_i are positive rational numbers.*

Remark. The importance of (ii) is that it guarantees that the *order* of execution of the T_i using the list L is the same for τ and τ'. Thus if L is used to execute $\mathcal{T}$, once using execution times τ_i and once using execution times τ_i', then the corresponding finishing times ω and ω' satisfy $|\omega - \omega'| \leqq r\varepsilon$. Hence if there were an example $\mathcal{T}$ with $\omega/\omega^* > s + 1$ and some of the τ_i irrational, then we could construct another example $\mathcal{T}'$ by slightly changing the τ_i to rational τ_i' so that the corresponding new finishing times ω' and ω^* satisfy $|\omega - \omega'| \leqq r\varepsilon$ and $|\omega^* - \omega^{*\prime}| \leqq r\varepsilon$, and, therefore if ε is sufficiently small, we still have $\omega'/\omega^{*\prime} > s + 1$. However, this would contradict what has already been proved. Lemma 3 is implied by the following slightly more general result. The proof we give here is due to V. Chvátal (personal communication).

LEMMA 3'. *Let S denote a finite system of inequalities of the form*

$$\sum_{i=1}^{r} a_i x_i \geqq a_0 \quad \textit{or} \quad \sum_{i=1}^{r} a_i x_i > a_0,$$

where the a_i are rational. Then, for any $\varepsilon > 0$, if S has a real solution $(x_1, \cdots, x_r)$, then S has a rational *solution $(x', \cdots, x_r')$ with $|x_i - x_i'| < \varepsilon$ for all i.*

Proof. We proceed by induction on r. For $r = 1$ the result is immediate. Now, let S be a system of inequalities in $r > 1$ variables which is solvable in reals. S splits into two classes: S_0, the subset of inequalities not involving x_r, and $S_1 = S - S_0$. Each inequality in S_1 can be written in one of the following four ways:

$$\text{(a)} \qquad \alpha_0 + \sum_{i=1}^{r-1} \alpha_i x_i \leqq x_r,$$

$$\text{(b)} \qquad \alpha_0 + \sum_{i=1}^{r-1} \alpha_i x_i < x_r,$$

$$\text{(c)} \qquad \beta_0 + \sum_{i=1}^{r-1} \beta_i x_i \geqq x_r,$$

$$\text{(d)} \qquad \beta_0 + \sum_{i=1}^{r-1} \beta_i x_i > x_r.$$

For each pair of inequalities, one of type (a) and one of type (c), we shall consider the inequality

$$\text{(e)} \qquad \alpha_0 + \sum_{i=1}^{r-1} \alpha_i x_i \leqq \beta_0 + \sum_{i=1}^{r-1} \beta_i x_i.$$

Similarly, the pairs of types {(a), (d)}, {(b), (c)} and {(b), (d)} give rise to inequalities

$$\text{(f)} \qquad \alpha_0 + \sum_{i=1}^{r-1} \alpha_i x_i < \beta_0 + \sum_{i=1}^{r-1} \beta_i x_i.$$

Let S^* be the set of all inequalities of type (e) and (f) that we obtain from S_1. Since by hypothesis, $S = S_0 \cup S_1$ has a real solution $(x_1, \cdots, x_r)$, then $S_0 \cup S^*$ has the real solution $(x_1, \cdots, x_{r-1})$. But $S_0 \cup S^*$ only involves $r - 1$ variables, so that, by the induction hypothesis, $S_0 \cup S^*$ has a rational solution $(x_1', \cdots, x_{r-1}')$ with $|x_i - x_i'| < \varepsilon'$ for all i and any preassigned $\varepsilon' > 0$. Substituting the x_i' into (a), (b), (c) and (d), we obtain a set of inequalities

$$\text{(g)} \qquad a' \leqq x_r, \quad b' < x_r, \quad c' \geqq x_r, \quad d' > x_r,$$

where the a', b', c' and d' are rational. Since the x_i satisfy (e) and (f), we have $a' \leqq c'$, $b' < c'$, $a' < d'$, $b' < d'$. Thus for any $\varepsilon > 0$, if ε' is chosen to be suitably small, then there is a rational x'_r satisfying (g) and with $|x_r - x'_r| < \varepsilon$, completing the proof of Lemma 3′. This proves Lemma 3, and hence, Theorem 2. □

The following example shows that the bound in Theorem 2 cannot be improved.

Example 2.

$$\mathcal{T} = \{T_1, T_2, \cdots, T_{s+1}, T'_1, T'_2, \cdots, T'_{sN}\};$$

$$\prec\ = \varnothing; \qquad n \geqq s(N+1) + 1 = r;$$

$$\tau_i = 1 \quad \text{for} \quad 1 \leqq i \leqq s+1; \qquad \tau'_i = \frac{1}{N} \quad \text{for} \quad 1 \leqq i \leqq sN;$$

$$\mathcal{R}_i(T_i) = 1 - \frac{1}{N}, \qquad \mathcal{R}_i(T_j) = \frac{1}{sN}, \qquad j \neq i, \quad i \leqq i \leqq s;$$

$$\mathcal{R}_i(T'_j) = \frac{1}{N}, \qquad 1 \leqq j \leqq sN, \qquad 1 \leqq i \leqq s;$$

$$L = (T_1, T'_1, \cdots, T'_N, T_2, T'_{N+1}, \cdots, T_{k+1}, T'_{kN+1}, T'_{kN+2}, \cdots, T'_{(k+1)N}, T_{k+2}, \cdots, T'_{sN}, T_{s+1});$$

$$L' = (T'_1, T'_2, \cdots, T'_{sN}, T_1, T_2, \cdots, T_{s+1}).$$

It is easily checked that for this case, $\omega = s + 1$ and $\omega' = 1 + s/N$, so that ω/ω' (and hence ω/ω^*) is arbitrarily close to $s + 1$ for N sufficiently large.

Proof of Theorem 3. The proof of Theorem 3 consists primarily of two main lemmas, each of which gives a bound on ω/ω^* which is best possible for certain values of s and n. We let λ denote ordinary Lebesgue measure[6] on the real line.

LEMMA 4. *For* $\mathcal{R} = \{\mathcal{R}_1, \mathcal{R}_2, \cdots, \mathcal{R}_s\}$ *and* $\prec$ *empty,*

$$\omega = \frac{n+1}{2}, \qquad \omega^* = \omega' = 1,$$

Proof. Let $I = \{t : |f(t)| = 1\}$. We first show

$$\text{(14)} \qquad \lambda(I) \leqq \omega^*.$$

Consider the set T of tasks defined by $T = \bigcup_{t \in I} f(t)$. For any pair of tasks T_i, T_j belonging to T, there must exist some k, $1 \leqq k \leqq s$, such that $\mathcal{R}_k(T_i) + \mathcal{R}_k(T_j) > 1$, for otherwise, one of those tasks should have been started earlier (unless $n = 1$, in which case the lemma is trivial). But this implies that in the optimal schedule

[6] Since in all of our applications, the subsets X of $[0, \omega)$ under consideration are finite unions of disjoint half-open intervals, then $\lambda(X)$ is just the sum of the lengths of these intervals.

no two members of T can be executed simultaneously. Therefore we have

$$\omega^* \geqq \sum_{T_i \in T} \tau_i \geqq \lambda(I),$$

which proves (14).

To complete the proof of Lemma 4, observe that at least two processors must be active at each time $t \in \bar{I} = [0, \omega) - I$.
Thus

$$n\omega^* \geqq \sum_{i=1}^{r} \tau_i \geqq 2\lambda(\bar{I}) + \lambda(I) = 2\omega - \lambda(I) \geqq 2\omega - \omega^*,$$

and therefore $(n + 1)\omega^* \geqq 2\omega$. □

The bound given by Lemma 4 is best possible whenever $n \leqq s + 1$, as shown by the following examples.

Example 3.

$$\mathscr{T} = \{T_0, T_1, T'_1, T_2, T'_2, \cdots, T_{n-1}, T'_{n-1}\};$$
$$\mathscr{R} = \{\mathscr{R}_1, \mathscr{R}_2, \cdots, \mathscr{R}_s\}, \qquad 2 \leqq n \leqq s + 1; \qquad \prec = \varnothing;$$
$$\tau_0 = 1; \qquad \tau_j = \tau'_j = \tfrac{1}{2}, \qquad 1 \leqq j \leqq n - 1;$$
$$\mathscr{R}_i(T_0) = \frac{1}{2n}, \qquad 1 \leqq i \leqq s;$$
$$\mathscr{R}_i(T_i) = \mathscr{R}_i(T'_i) = \tfrac{1}{2}, \qquad 1 \leqq i \leqq n - 1;$$
$$\mathscr{R}_i(T_j) = \mathscr{R}_i(T'_j) = \frac{1}{2n}, \qquad i \neq j, \quad 1 \leqq i \leqq s, \quad 1 \leqq j \leqq n - 1;$$
$$L = (T_1, T'_1, T_2, T'_2, \cdots, T_{n-1}, T'_{n-1}, T_0);$$
$$L = (T_0, T_1, T_2, \cdots, T_{n-1}, T'_1, T'_2, \cdots, T'_{n-1}).$$

It is easily checked that for this case,

$$\omega = \frac{n+1}{2}, \qquad \omega^* = \omega' = 1,$$

showing that the bound of Lemma 4 is best possible whenever $n \leqq s + 1$. □

The following example, for the case $s + 1 < n \leqq 2s + 1$, is somewhat more complicated.

Example 4. For suitably small $\varepsilon > 0$ and a positive integer k, define

$$\varepsilon_i = \varepsilon(n - 1)^{i-2k}, \qquad 1 \leqq i \leqq 2k;$$
$$\mathscr{T} = \{T_0\} \cup \{T_{ij}: 1 \leqq i \leqq n - 1, 1 \leqq j \leqq k\} \cup \{T'_{ij}: 1 \leqq i \leqq n - 1, 1 \leqq j \leqq k\};$$
$$\mathscr{R} = \{\mathscr{R}_1, \mathscr{R}_2, \cdots, \mathscr{R}_s\}; \quad s + 1 < n \leqq 2s + 1; \quad \prec = \varnothing;$$
$$\tau_0 = 2k; \quad \tau_{ij} = \tau'_{ij} = 1, \qquad 1 \leqq i \leqq n - 1, \quad 1 \leqq j \leqq k;$$
$$\mathscr{R}_i(T_0) = \varepsilon_1, \qquad 1 \leqq i \leqq s;$$

$$\mathcal{R}_i(T_{ij}) = 1 - (n-1)\varepsilon_{2j-1}, \quad 1 \leqq i \leqq s, \quad 1 \leqq j \leqq k;$$

$$\mathcal{R}_l(T_{ij}) = \varepsilon_{2j-1}, \quad l \neq i, \quad 1 \leqq l \leqq s, \quad 1 \leqq i \leqq n-1, \quad 1 \leqq j \leqq k;$$

$$\mathcal{R}_i(T'_{s+i,j}) = 1 - (n-1)\varepsilon_{2j}, \quad 1 \leqq i \leqq n-s-1, \quad 1 \leqq j \leqq k;$$

$$\mathcal{R}_l(T'_{ij}) = \varepsilon_{2j}, \quad l \neq i-s, \quad 1 \leqq l \leqq s, \quad 1 \leqq i \leqq n-1, \quad 1 \leqq j \leqq k.$$

To illuminate the structure of the two lists, L and L', we describe them in block form.

$$L = (A_1, A_2, \cdots, A_k, A'_1, A'_2, \cdots, A'_{k-1}, A_0),$$

where

$$A_i = (B_{1i}, B_{2i}, \cdots, B_{si}), \quad 1 \leqq i \leqq k;$$

$$B_{ji} = (T_{ji}, T'_{ji}), \quad 1 \leqq i \leqq k, \quad 1 \leqq j \leqq s;$$

$$A'_i = (B'_{1i}, B'_{2i}, \cdots, B'_{n-1,i}), \quad 1 \leqq i \leqq k-1;$$

$$B'_{ji} = (T'_{s+j,i}, T_{s+j,i+1}), \quad 1 \leqq i \leqq k-1, \quad 1 \leqq j \leqq n-1;$$

$$A_0 = (T_0, T_{s+1,1}, T_{s+2,1}, \cdots, T_{n-1,1}, T'_{s+1,k}, T'_{s+2,k}, \cdots, T'_{n-1,k}).$$

Also

$$L' = (C_0, C_1, C'_1, C_2, C'_2, \cdots, C_k, C'_k),$$

where

$$C_0 = (T_0);$$

$$C_i = (T_{1i}, T_{2i}, \cdots, T_{n-1,i}), \quad 1 \leqq i \leqq k;$$

$$C'_i = (T'_{1i}, T'_{2i}, \cdots, T'_{n-1,i}), \quad 1 \leqq i \leqq k.$$

It is not difficult to check that when the list L is used, each of the pairs of tasks given in the sublists B_{ji} and B'_{ji} will be executed simultaneously on the first two processors, with the other $n-2$ processors remaining inactive during that time. After all such pairs have been executed, the tasks on sublist A_0 will be started. This results in

$$\omega = k(n-1) - (n-s-1) + 2k = k(n+1) - (n-s-1).$$

When the list L' is used, each of the sets of $n-1$ tasks given in the sublists C_i and C'_i, will be executed simultaneously on processors 2 through n, with processor 1 executing T_0. Thus $\omega^* = \omega' = 2k$. We then have

$$\frac{\omega}{\omega^*} = \frac{n+1}{2} - \frac{(n-s-1)}{2k},$$

which is arbitrarily close to $(n+1)/2$ for k sufficiently large. □

We now prove an upper bound for ω/ω^* which is best possible whenever $n > 2s + 1$.

LEMMA 5. *For* $\mathcal{R} = \{\mathcal{R}_1, \mathcal{R}_2, \cdots, \mathcal{R}_s\}$, $\prec = \varnothing$, *and* $n \geqq 3$,

$$\frac{\omega}{\omega^*} \leqq s + 2 - \frac{2s+1}{n}.$$

Proof. Suppose that we have a counterexample to the lemma. By Lemma 3, we may assume all the τ_i are rational, i.e., there exists a positive integer m such that for each i, $1 \leqq i \leqq r$, there exists an integer k_i satisfying $\tau_i = k_i/m$. Without loss of generality, we may also assume that $\omega^* = 1$. Thus each k_i satisfies $1 \leqq k_i \leqq m$ and $\omega = \omega(L) > s + 2 - (2s + 1)/n$.

Consider the operation of the system using the list L. Let $I = \{t \in [0, \omega) : |f(t)| = 1\}$, $I' = \{t \in [0, \omega) : |f(t)| = n\}$ and let $\bar{I} = [0, \omega) - I'$. By the proof of Lemma 4, $\lambda(I) \leqq 1$. Since at least two processors are active at each time $t \in \bar{I}$,

$$n \geqq \sum_{i=1}^{r} \tau_i \geqq n \cdot \lambda(I') + \lambda(I) + 2(\omega - \lambda(I) - \lambda(I'))$$
$$\geqq (n - 2)\lambda(I') + 2\omega - 1,$$

or

$$\lambda(I') \leqq \frac{n + 1 - 2\omega}{n - 2}. \tag{15}$$

Since $\omega > s + 2 - (2s + 1)/n$, we then have

$$\lambda(\bar{I}) = \omega - \lambda(I') \geqq \omega - \frac{n + 1 - 2\omega}{n - 2}$$
$$> s + 2 - \frac{2s + 1}{n} - \frac{n + 1 - 2\left(s + 2 - \frac{2s + 1}{n}\right)}{n - 2} \tag{16}$$
$$= s + 1.$$

Now observe that for any $t_1, t_2 \in \bar{I}$ satisfying $t_2 - t_1 \geqq 1$, there must exist an i, $1 \leqq i \leqq s$, such that

$$r_i(t_1) + r_i(t_2) > 1, \tag{17}$$

for otherwise, some task being executed at time t_2 should have been started at time t_1 or earlier. Recalling that $\bar{I}$ is a collection of intervals, each having the form $[k/m, (k + 1)/m)$ for some integer k, let $a_0 < a_1 < \cdots < a_p$ be integers such that

$$\bar{I} = \left\{\left[\frac{a_i}{m}, \frac{a_i + 1}{m}\right) : 0 \leqq i \leqq p\right\}.$$

Notice that (16) implies that $p \geqq (s + 1)m$. For each i, $1 \leqq i \leqq s$, we construct a graph H_i as follows:

$$V(H_i) = \{0, 1, 2, \cdots, (s + 1)m - 1\};$$

$$\{u, v\} \text{ is an edge of } H_i \quad \text{iff} \quad r_i\left(\frac{a_u}{m}\right) + r_i\left(\frac{a_v}{m}\right) > 1.$$

Note that $|u - v| \geqq m$ implies $|a_u - a_v| \geqq m$, which, by (17), implies that $\{u, v\}$ is an edge of at least one H_i, $1 \leqq i \leqq s$. Hence it is not difficult to see that $G(m, s) \subseteq \bigcup_i H_i$. The same reasoning used in the proof of Theorem 2 can be used now to show that, for some i, $1 \leqq i \leqq s$, $\int_0^\infty r_i(t)\,dt > 1$, which contradicts the assumption that $\omega^* = 1$. This completes the proof of Lemma 5. □

Combining Lemmas 4 and 5, we obtain Theorem 3. It remains to be shown that Lemma 5 is best possible whenever $n > 2s + 1$. This is done by the following example.

Example 5. For suitably small $\varepsilon > 0$ and a positive integer k', let $k = k'n$, and define

$$\varepsilon_i = \varepsilon(n-1)^{i-k}, \qquad 1 \leqq i \leqq k;$$

$$\mathcal{T} = \{T_0\} \cup \{T_{ij}: 1 \leqq i \leqq n-1, 1 \leqq j \leqq k\};$$

$$\mathcal{R} = \{\mathcal{R}_1, \mathcal{R}_2, \cdots, \mathcal{R}_s\}; \quad n > 2s+1; \quad \prec\, = \varnothing;$$

$$\tau_0 = k; \qquad \tau_{ij} = 1, \qquad 1 \leqq i \leqq n-1, \quad 1 \leqq j \leqq k;$$

$$\mathcal{R}_i(T_0) = \varepsilon_1, \qquad 1 \leqq i \leqq s;$$

$$\mathcal{R}_i(T_{ij}) = 1 - (n-1)\varepsilon_j, \qquad 1 \leqq i \leqq s, \quad 1 \leqq j \leqq k;$$

$$\mathcal{R}_l(T_{ij}) = \varepsilon_j, \qquad l \neq i, \quad 1 \leqq l \leqq s, \quad 1 \leqq i \leqq n-1, \quad 1 \leqq j \leqq k.$$

As in Example 4, we again describe the lists L and L' in block form.

$$L = (A_1, A_2, \cdots, A_{n-2s-1}, B_1, B_2, \cdots, B_s, C),$$

where

$$A_i = (T_{2s+i,1}, T_{2s+i,2}, \cdots, T_{2s+i,k}), \qquad 1 \leqq i \leqq n - 2s - 1;$$

$$B_i = (T_{i1}, T_{s+i,2}, T_{i2}, T_{s+i,3}, \cdots, T_{i,k-1}, T_{s+i,k}), \qquad 1 < i < s;$$

$$C = (T_0, T_{s+1,1}, T_{s+2,1}, \cdots, T_{2s,1}, T_{1k}, T_{2k}, \cdots, T_{sk}).$$

Also

$$L' = (T_0, D_1, D_2, \cdots, D_k),$$

where

$$D_i = (T_{1i}, T_{2i}, \cdots, T_{n-1,i}), \qquad 1 \leqq i \leqq k.$$

It is not difficult to check that

$$\omega = k'(n - 2s - 1) + (k-1)s + k = (s+2)k'n - (2s+1)k' - s$$

and $\omega^* = \omega' = k = k'n$. Thus

$$\frac{\omega}{\omega^*} = s + 2 - \frac{2s+1}{n} - \frac{s}{k'n},$$

which is arbitrarily close to the bound of Lemma 5 for k' sufficiently large.

5. Concluding remarks. The results which have been discussed in this paper lead naturally to a number of possible extensions, several of which we mention here.

We first note that for the case $\mathcal{R} = \{\mathcal{R}_1\}$, $n \geqq r$, and general $\prec$, Example 1 may be used to show that ω/ω^* can be arbitrarily large.

Regarding Lemma 1, an algorithm can be given which determines $S(G)$ (and a corresponding valid labeling as well) in at most

$$O(|E|\sqrt{|V|})$$

operations. A similar algorithm may be used for the following dual problem: given a graph G, determine

$$\max_{L^*} \sum_{e\in E} L^*(e),$$

where the max ranges over all functions $L^*: E \to [0, \infty)$ such that for all $v \in V$,

$$\sum_{e' \in E(v)} L^*(e') \leqq 1,$$

where $E(v)$ is the set of all edges incident to v. It would be interesting to investigate the analogous questions for hypergraphs.

The following result follows more or less directly from Lemma 2.

COROLLARY. *For a positive integer n, let* $f_i:[0, n + 1) \to [0, \infty)$, $1 \leqq i \leqq n$, *be (Lebesgue) measurable functions satisfying the following condition:*

If $t_1, t_2 \in [0, n + 1)$ *with* $|t_1 - t_2| \geqq 1$, *then*

$$\max_{1\leqq i\leqq n} \{f_i(t_1) + f_i(t_2)\} \geqq 1.$$

Then

$$\max_{1\leqq i\leqq n} \int_{[0,n+1]} f_i \, d\lambda \geqq 1.$$

It is interesting to note that, at present, no purely analytical proof of the Corollary is known.

The techniques of Lemma 2 may also be used to derive several new results in graph theory. In particular, it follows that if m is a positive integer and G_m denotes the graph with vertex set

$$V_m = \{0, 1, \cdots, 3m - 1\}$$

and edge set

$$E_m = \{\{a, b\} \subseteq V_m : \min\{a - b, 3m - a + b\} \geqq m\},$$

then *any* 2-coloring of E_m contains m disjoint edges having the same color.

The corresponding general conjecture is that for a fixed $s \geqq 1$, if we take

$$V_m = \{0, 1, \cdots, (s + 1)m - 1\}$$

and

$$E_m = \{\{a, b\} \subseteq V_m : \min\{a - b, (s + 1)m - a + b\} \geqq m\},$$

then any s-coloring of E_m contains m disjoint edges having the same color. At present, this conjecture is still open. If true, it is close to being best possible, since there exist m-colorings of the edges of the *complete* graph on $(s + 1)m - s$ vertices which have no set of m disjoint edges having a single color (cf. [1], [2]).

Finally, it is natural to inquire under what restrictions do there exist efficient algorithms for determining optimal schedules for problems of the type considered herein (cf. e.g., [6], [12]).

Acknowledgments. The authors take pleasure in acknowledging the resourceful suggestions of S. A. Burr, J. C. Cheng, V. Chvátal, D. S. Johnson, Shen Lin, and H. S. Witsenhausen. The introduction of limited resources into scheduling problems of this type was first suggested to the authors by E. Arthurs.

REFERENCES

[1] S. A. BURR, *Generalized Ramsey theory for graphs—A survey of recent results*, Graphs and Combinatorics, R. Bari and F. Harary, eds., Lecture Notes in Mathematics No. 406, Springer-Verlag, Berlin, 1974, pp. 52–75.

[2] E. J. COCKAYNE AND P. J. LORIMER, *The Ramsey graph numbers for stripes*, Univ. of Auckland Math. Dept. Rep. Ser. No. 37, Auckland, New Zealand 1973.

[3] E. G. COFFMAN AND R. L. GRAHAM, *Optimal scheduling for two-processor systems*, Acta Informatica 1 (1972), pp. 200–213.

[4] M. FUJII, T. KASAMI AND K. NINOMIYA, *Optimal sequencing of two equivalent processors*, SIAM J. Appl. Math., 17 (1969), pp. 784–789.

[5] ———, *Erratum*: *Optimal sequencing of two equivalent processors*, Ibid., 20 (1971), p. 141.

[6] R. L. GRAHAM, *Bounds on multiprocessing anomalies and related packing algorithms*, AFIPS Conf. Proc., 40 (1972), pp. 205–217.

[7] ———, *Bounds on multiprocessing timing anomalies*, SIAM J., Appl. Math., 17 (1969), pp. 263–269.

[8] F. HARARY, *Graph Theory*, Addison-Wesley, Reading, Mass., 1969.

[9] T. C. HU, *Parallel scheduling and assembly line problems*, Operations Res., 9 (1961), pp. 841–848.

[10] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972.

[11] M. T. KAUFMAN, *Anomalies in scheduling unit-time tasks*, Stanford Electronics Lab. Tech. Rep. 34, 1972.

[12] J. D. ULLMAN, *Polynomial complete scheduling problems*, Computer Science Tech. Rep. 9, Univ. of Calif., Berkeley, 1973.

SIAM J. COMPUT.
Vol. 4, No. 2, June 1975

THE COMPUTATION OF POWERS OF SYMBOLIC POLYNOMIALS*

ELLIS HOROWITZ† AND SARTAJ SAHNI‡

Abstract. Recent results on the computation of powers of symbolic polynomials are reviewed in perspective. Then a new algorithm is given which computes the nth power of a completely sparse polynomial using a linear number of multiplications. This is followed by experimental results comparing the new algorithm to iteration using both completely sparse and completely dense polynomials as data.

Key words. polynomial powers, symbolic powers, sparse polynomial powers

1. Introduction. Let $P(x_1, \cdots, x_v)$ be a polynomial in v variables with integral coefficients. Suppose that $d =$ degree (P) in x_i, $1 \leqq i \leqq v$, and that all possible terms of P are present. Then P has $(d+1)^v$ terms and is said to be *completely dense.* If P^i has $(id+1)^v$ terms $1 \leqq i \leqq n$, then P^i remains *completely dense to power n.* Using this worst case assumption of polynomial growth, and the classical polynomial multiplication algorithm [4, p. 362], Heindel in [2] showed that computing P^n by iteration was faster than using the binary method (binary expansion of the exponent, see [3, p. 399]). Briefly reviewing that result, we see that it follows from the completely dense assumption that the cost for iteration is asymptotically

$$\sum_{1 \leqq i \leqq n-1} (d+1)^v(id+1)^v < ((n-1)d+1)^v(d+1)^v(n-1) < n^{v+1}(d+1)^{2v},$$

while the cost for the binary method is bounded by

$$\sum_{1 \leqq i \leqq \log_2 n} (2^i d+1)^{2v} < (n(d+1))^{2v}.$$

Thus the ratio of these methods, iteration/binary $= n^{v+1}/n^{2v} = 1/n^{v-1}$, and so for $v > 1$ variables, iteration becomes asymptotically superior. This was a somewhat nonintuitive result in the sense that the binary method requires only $O(\log_2 n)$ polynomial multiplications, whereas iteration requires $O(n-1)$, and therefore one might naturally conclude (e.g., see Knuth [4]) that binary would be better.

The binary method and iteration have one thing in common; namely, they are whole polynomial methods. This is an intuitive idea by which we mean that at every step where a multiplication is done, it is done with polynomials. There are however, other methods for computing powers which do not rely on this whole polynomial property. One such approach, based upon evaluation and subsequent interpolation, was presented by Horowitz [3]. Using the previous assumptions, that method will compute P^n in time proportional to $(n(d+1))^{v+1}$. At the heart of this algorithm is a routine which computes the nth power of an integer using the binary method. Hence this algorithm, in addition to having a

* Received by the editors October 1, 1973, and in revised form June 4, 1974. This work was supported by the National Science Foundation under Grant GJ-33169.

† Computer Science Program, University of Southern California, at Los Angeles, Los Angeles, California 90007.

‡ Department of Computer, Information and Control Sciences, University of Minnesota, Minneapolis, Minnesota 55455.

better asymptotic time, showed that one could operate on multivariate polynomials via some transformational technique and return the problem of computing polynomial powers to computing powers of single precision numbers.

At present, the method which has the best asymptotic computing time is obtained by using the fast Fourier transform and its convolution property; see Pollard [5].

Table 1 gives the asymptotic computing times for four "polynomial power" algorithms applied to dense polynomials. The work factor gives the amount of work per term in the answer that each method requires. A work factor of 1 would be optimal; however, the best known is $\log(n(d + 1))$. The asymptotic computing times for the first 3 methods were obtained assuming the classical multiplication method is used. These could be reduced by using faster polynomial multiplication methods, though the direct use of the fast Fourier transform (FFT) would still yield the lowest upper bound.

TABLE 1
Asymptotic times, P^n completely dense

Method	Time	= Terms	*	Work factor
Binary	$n^{2v}(d + 1)^{2v}$	$= n^v(d + 1)^v$	*	$n^v(d + 1)^v$
Iteration	$n^{v+1}(d + 1)^{2v}$	$= n^v(d + 1)^v$	*	$n(d + 1)^v$
Eval-Interp	$n^{v+1}(d + 1)^{v+1}$	$= n^v(d + 1)^v$	*	$n(d + 1)$
FFT	$n^v(d + 1)^v \log(n(d + 1))$	$= n^v(d + 1)^v$	*	$\log(n(d + 1))$

The reliance on the completely dense model alone is somewhat limited because of the exponential growth of the number of terms in the answer. Practical computation dictates that dense polynomials in 3 or more variables can only be raised to quite small powers, e.g. see [3], before either core or time become excessive.

Existing algebra systems need to handle multivariate problems, but often these problems are of a sparse nature. In [1], Gentleman suggested the definition for a totally sparse polynomial, the intuitive opposite of the completely dense case. If P initially has t terms and P^i has $\binom{t+i-1}{t-1}$ terms, $1 \leqq i \leqq n$, then P is said to be *completely sparse to power n*. The motivation for this definition is simply that for P^i to grow exactly as $\binom{t+i-1}{t-1}$, it must be the case that the fewest possible number of terms combine as we compute each new iterate. An example of such a polynomial is

$$P(x_1, \cdots, x_t) = x_1 + \cdots + x_t,$$

which is completely sparse for all i. Now in [1], Gentleman gives a result similar in spirit to Heindel's: for completely sparse polynomials, computing P^n by iteration is faster than using the binary method. The computing time for iteration is

$$\sum_{1 \leqq i \leqq n-1} t\binom{t+i-1}{t-1} = t\binom{t+n-1}{t} - t.$$

No closed formula for the time using the binary method has been obtained, but in [1] it is shown that the ratio of the costs of binary to iteration needed to compute P^{2^n}, where P is completely sparse and initially has t terms, is given by

$$\frac{1}{2n}\binom{2n}{n}\left(1 - \frac{n^2}{t}\right) + O(t^{-2}).$$

This implies that the binary method is more costly by at least a binomial factor.

In this paper, we will present a new algorithm for computing a power of an arbitrary polynomial. Its motivation comes from the definition of a completely sparse polynomial, and its computing time has a logarithmic work factor. Thus, this new method corresponds in complexity to the use of the FFT for computing powers of completely dense polynomials. We then present empirical results comparing this new algorithm to iteration.

2. The algorithm. First let us consider a specific instance of a completely sparse polynomial, namely,

$$P(x_1, \cdots, x_t) = x_1 + \cdots + x_t.$$

By the multinomial expansion theorem (e.g., see [6, p. 64]), it follows that

$$(2.1) \qquad (x_1 + \cdots + x_t)^n = \sum_{n_1 + \cdots + n_t = n} \binom{n}{n_1, \cdots, n_t} x_1^{n_1} \cdots x_t^{n_t},$$

where the n_i are integers in the range $0 \leqq n_i \leqq n$. The number of distinct t-tuples which sum to n is precisely $\binom{t+n-1}{t-1}$, corresponding to the definition of a completely sparse polynomial. The definition of the multinomial coefficient is

$$\binom{n}{n_1, \cdots, n_t} = \frac{n!}{n_1! \cdots n_t!}.$$

Moreover, we emphasize that each time the n_i change, the next multinomial coefficient may be obtained from the previous one using one multiplication and one division, i.e.,

$$(2.2) \quad \binom{n}{n_1, \cdots, n_i - 1, n_{i+1} + 1, \cdots, n_t} = \frac{n_i}{n_{i+1} + 1}\binom{n}{n_1, \cdots, n_i, n_{i+1}, \cdots, n_t}.$$

Thus, if we generate the t-tuples in lexicographic order, it requires $2\binom{t+n-1}{t-1}$ coefficient multiplications to compute the nth power of $P(x_1, \cdots, x_t)$. Unfortunately, for general sparse polynomials, it becomes necessary to sort the terms thus adding a log factor to the computing time.

The general algorithm begins with an arbitrary polynomial, say

$$P(y_1, \cdots, y_v) = \sum_{1 \leqq i \leqq t} a_i y_1^{e_{i1}} \cdots y_v^{e_{iv}}$$

in v variables with t nonzero terms. Conceptually, the method then proceeds by setting

$$x_i = a_i y_1^{e_{i1}} \cdots y_v^{e_{iv}}, \qquad 1 \leqq i \leqq t,$$

producing the new polynomial

$$\bar{P}(x_1, \cdots, x_t) = x_1 + \cdots + x_t.$$

The nth power of $\bar{P}$ is computed in linear time and the substitutions back to the y_i followed by a sort increase the bound by a log factor. This algorithm is now given in complete detail.

The input polynomial with t terms is assumed to be stored term by term in the array TERM(1:t). The array N(1:t) contains the exponent vector and is initialized to:

$$\mathrm{N}(1) \leftarrow n, \quad \mathrm{N}(2) \leftarrow \cdots \mathrm{N}(t) \leftarrow 0;$$

and the global variable POW is set to (TERM(1))^n. t is a global variable whose value is the number of terms in the input polynomial. Then the following routine is called using

(2.3) MULT(TERM(1)^n, 1, 1).

```
ALGORITHM MULT(POL, COEF, $i$).
  Input:  POL, a multivariate polynomial
          COEF, an integer
          POW, a global variable initialized to (TERM(1))$^n$
          $i$, a nonnegative integer
  Output: the global variable POW is set to: (TERM(1) + ··· + TERM($t$))$^n$
 1. if $i \neg = t$ then /* move forward */
 2.    do;  do while (N($i$)$\neg$ = 0);
 3.            N($i$) ← N($i$) − 1;
 4.            N($i$ + 1) ← N($i$ + 1) + 1;
 5.            COEF ← COEF * (N($i$) + 1)/N($i$ + 1);
 6.            POL ← (POL/TERM($i$)) * TERM($i$ + 1);
 7.            POW ← POW + POL * COEF;
 8.            CALL MULT (POL, COEF, $i$ + 1):
 9.          end;
10.          if $i$ = 1  then return
11.                      else do/* backtrack */
12.                        N($i$) ← N($i$ + 1);
13.                        N($i$ + 1) ← 0;
14.                        return;
15.                        end;
16.    end;
17. end POWER;
```

We now show that algorithm MULT when called as in (2.3) with POW $= (\mathrm{TERM}(1))^n$ and $\mathrm{N}(1) = n$, $\mathrm{N}(2) = 0$ results in the desired solution POW $= (\sum_{1 \leqq i \leqq t} \mathrm{TERM}(i))^n$. It is clear that if all the terms of the sum in (2.1) are generated and then $\mathrm{TERM}(i)$ is substituted for x_i, we obtain the desired result. Associated with each term in the sum for (2.1) is a power sequence $(n_1, n_2, \cdots, n_t)$ and a coefficient $\binom{n}{n_1, \cdots, n_t}$. For any power sequence $(n_1, n_2, \cdots, n_t)$ and $2 \leqq i \leqq t$, define the i-prefix to be $(n_1, \cdots, n_{i-1})$ and, for $i = 1$, the 1-prefix is ( ). To see that only correct power sequences are generated and that each such sequence is generated exactly once, we note that:

(i) Steps 3 and 4, 12 and 13 are the only ones that alter the power sequence. Both pairs of steps preserve the value of $\sum \mathrm{N}(i)$ and maintain $\mathrm{N}(i) \geqq 0$ (note that the conditional of step 2 ensures that $\mathrm{N}(i) = 0$ when steps 12 and 13 are executed). Hence only valid power sequences are generated.

(ii) Each time a call to MULT is made, either initially or from step 8, the i- or $(i + 1)$-prefix, respectively, is different from all other calls with the same i-value. Hence each power sequence is generated only once.

(iii) For any i-prefix, a call to MULT results in the generation of all power sequences with the same i-prefix.

From (2.2) and steps 3, 4 and 5 of MULT, it follows that at all times the value of COEF is $\binom{n}{\mathrm{N}(1) \cdots \mathrm{N}(t)}$. Steps 3, 4 and 6 imply that POL at any time has the value $\prod (\mathrm{TERM}(i))^{\mathrm{N}(i)}$. Hence it follows that the routine MULT, when called as described above, results in the computation of $(\sum_{1 \leqq i \leqq t} \mathrm{TERM}(i))^n$.

To get an estimate of the computing time, we note that each call to MULT from step 8 results in 2 multiplication/divisions (abbreviated 2 M/D) in step 5, 2 M/D in step 6 and another call to MULT. However, each such call results in the generation of a new term. There are exactly $\binom{t + n - 1}{t - 1}$ such terms. To compute $(\mathrm{TERM}(1))^n$, $\log n$ multiplications are needed. Hence MULT requires

$$\log n + 4\binom{t + n - 1}{t - 1} \text{ M/D} \sim O\left(\binom{t + n - 1}{t - 1}\right) \text{ M/D}.$$

The only other cost to be considered is that of the addition in step 7. The best way to do this appears to be to just generate all $\binom{t + n - 1}{t - 1}$ terms, then sort them adding together terms with identical power sequences (this will be required only if the original polynomial $P = \sum \mathrm{TERM}(i)$ is not sparse to power n). This sort-add step can be done in

$$O\left(\binom{t + n - 1}{t - 1} \log \binom{t + n - 1}{t - 1}\right),$$

resulting in an overall computing time of $O(T \log T)$, where T is the number of terms in the answer. This is the same as for FFT over dense polynomials.

For comparison, let us consider computing P^n by computing the sequence $P, P^2, \cdots, P^n$ (i.e., iteration). Then the number of multiplications is

$$\sum_{i=1}^{n-1} t\binom{t+n-1}{t-1} = O\left(n\binom{t+n-1}{t-1}\right).$$

Here, too, a sort-add step is needed, thus adding a log factor to the computing time. The total computing time is then bounded by the sort-add time, which is

$$O\left(n\binom{t+n-1}{t-1}\log t\right).$$

Hence we see that as far as an M/D count is concerned, MULT is optimal to within a constant factor. It requires about $O(n)$ times fewer multiplications than iteration.

3. Empirical results. In this section we present the results of several tests that were made to determine the global efficiency of these 2 algorithms. Though asymptotic analyses are important, the value of practical testing should not be underestimated. This is especially true when dealing with symbolic problems, since the domain of actual computation is often moderately small, thus placing added importance on constants and less on asymptotic results. All tests were carried out on an IBM 360/65 using the SAC-1 System which provides, in part, for arithmetic operations on multivariate polynomials.

Both completely dense and completely sparse polynomials were used as test data for these algorithms. For completely sparse polynomials in v variables the polynomials used were

$$P(x_1, \cdots, x_v) = x_1 + \cdots + x_v,$$

except when $v = 1$, in which case $P(x_1) = x_1 + 1$. The completely dense polynomial in 1 variable had degree = 7, while the corresponding polynomial in 2 variables had maximum degree = 2 in each variable. Completely dense polynomials in 3 and 4 variables each have maximum degree = 1 in each variable. All coefficients of P were one. Table 2 gives the results in milliseconds for completely sparse powers, while Table 3 contains the completely dense results. The addition of step 7 was done using a standard polynomial add routine rather than a sort-add at the end as described in the analysis of MULT. Finally, a nonrecursive version of MULT was programmed so as to reduce the overhead of repeated procedure calls.

Similarly the additions required by iteration were not carried out by a sort-add. Considering that only relatively small problems were tested, it is unlikely that the advantage of using the asymptotically superior sort-add would have been reflected in the computing times of Tables 2 and 3.

4. Conclusion. We have seen that there are two basic complementary models for which one does an analysis of powering algorithms: completely dense and completely sparse polynomials. The main result here has been to exhibit an

TABLE 2
Completely sparse P^n

No. of variables	1		2		3		4	
n	Iter	Mult	Iter	Mult	Iter	Mult	Iter	Mult
2	16.6	16.6	16.6	16.6	33.2	133.1	133.1	266.2
4	16.6	49.9	83.2	66.5	316.1	366.0	898.5	1064.9
6	83.2	83.2	183.0	99.8	931.8	665.6	3095.0	2579.2
8	116.4	83.2	332.8	149.7	1896.9	1098.2	7937.2	5308.1
10	216.3	133.1	499.2	199.6	3577.6	1580.8	16872.9	9434.8
12	282.8	299.8	748.8	266.2	5990.4	2312.9	**	
14	432.6	166.4	1064.9	299.5	9085.4	2995.2		
16	449.2	232.9	1248.0	316.1	13295.3	3966.3		

** Power could not be computed with 23k words of work space.

TABLE 3
Completely dense P^n

No. of variables	1		2		3		4	
max degree	7		2		1		1	
n	Iter	Mult	Iter	Mult	Iter	Mult	Iter	Mult
2	116.4	332.8	116.4	399.3	116.4	915.2	515.8	2362.8
4	715.5	3927.0	848.6	3810.5	1913.6	8636.1	9368.3	63763.2
6	1597.4	22763.5	3461.1	17555.2	8253.4	46475.5	**	
8	2812.1	99274.2	7987.2	63015.6	**			
10	4509.4	>400 sec	14809.6	172689.9				

** Power could not be computed with 23k words of work space.

algorithm which requires $O(T)$ multiplications and $O(T \log T)$ exponent comparisons (T is the number of terms in the nth power of a completely sparse polynomial). From a complexity point of view, this means the best methods we know of for computing powers take $O(T \log T)$ operations (T is the number of terms in the result) for both the completely dense and completely sparse models.

In practice, a specific problem may have characteristics that give an advantage to any one of the other known methods; e.g., see [7], [8]. In addition to the number of arithmetic operations, one may have to consider other relevant factors such as the efficiency/inefficiency of recursion, procedure calls, etc., in the source language. We have shown that for completely sparse polynomials using a FORTRAN-based system, our new algorithm is better than iteration. But for any symbol manipulation system which wants to provide only a single powering routine, iteration seems the best choice (i) because of its simplicity, (ii) because it yields all intermediate powers which may be useful, e.g., in substituting a polynomial for x in $P(x)$, and (iii) because it is uniformly good for both polynomial models. The best known methods for either model are, on the average, better than iteration by a factor of 2. Unfortunately, these specialized algorithms, FFT for dense and MULT for sparse polynomials, perform very poorly on sparse and dense polynomials, respectively.

REFERENCES

[1] W. M. GENTLEMAN, *Optimal multiplication chains for computing a power of a symbolic polynomial*, Math. Comp., 26 (1972), pp. 935–939.

[2] L. HEINDEL, *Computation of powers of multivariate polynomials over the integers*, J. Comput. System Sci., 6 (1972), pp. 1–8.

[3] E. HOROWITZ, *The efficient calculation of powers of polynomials*, Ibid., 7 (1973), pp. 469–480.

[4] D. KNUTH, *The Art of Computer Programming. Vol. II: Seminumerical Algorithms*, Addison-Wesley, Reading, Mass., 1968.

[5] J. M. POLLARD, *The fast Fourier transform in a finite field*, Math. Comp., 25 (1971), pp. 365–374.

[6] D. KNUTH, *The Art of Computer Programming. Vol. I: Fundamental Algorithms*, Addison-Wesley, Reading, Mass., 1969.

[7] R. FATEMAN, *Polynomial multiplication, powers and asymptotic analysis: Some comments*, this Journal, 3 (1974), pp. 196–213.

[8] ———, *On the computation of powers of sparse polynomials*, Studies in Appl. Math., 52 (1974), pp. 145–155.

SIAM J. COMPUT.
Vol. 4, No. 3, September 1975

COMPUTATIONAL ALGORITHMS FOR THE ENUMERATION OF GROUP INVARIANT PARTITIONS*

D. E. WHITE AND S. G. WILLIAMSON†

Abstract. Let G be a finite group acting on a finite set S and hence on $\Pi(S)$, the lattice of partitions of S. Computational methods are developed for enumerating the invariants of this action.

Key words. partitions of a set, group actions

1. Introduction. In this paper, computational algorithms are developed for enumerating structures on G-invariant partitions. These algorithms are based on identities derived in a previous paper [6, Thms. 1 and 2]. Using [5, Thm. 2], we show how these results may be extended to the enumeration of partitions whose stabilizer is conjugate to a given subgroup of G.

2. Statement of results. We recall the basic definitions and results of [6].

Let G be a finite group acting on a finite set S (notation $G:S$). This action induces a natural action on the partitions $\Pi(S)$ of S. A partition $\pi \in \Pi(S)$ is G-*invariant* if the stabilizer subgroup $G_\pi = G$.

Let H, K be subgroups of G. We define

$$M_H(K) = \frac{1}{|H|} \sum_{\sigma \in G} \chi(\sigma K \sigma^{-1} \subset H),$$

where

$$\chi\,(\text{statement}) = \begin{cases} 1 & \text{if statement is true,} \\ 0 & \text{if statement is false,} \end{cases}$$

and $|H|$ denotes the cardinality of H. $M_H(K)$ is called the *mark* in G of H at K [1].

Let Δ be a system of orbit representatives for the action of G on S. Let $\Pi(\Delta)$ be the partitions of Δ. Let $\mathscr{H}$ be a complete set of nonconjugate subgroups of G. Let G_t be the stabilizer subgroup of $t \in S$. We have the following theorem shown in [6].

THEOREM 1. *The number of G-invariant partitions of S is given by*

$$\sum_{\delta \in \Pi(\Delta)} \prod_{A \in \delta} \sum_{H \in \mathscr{H}} \frac{1}{M_H(H)} \prod_{t \in A} M_H(G_t).$$

The computational results derived below from Theorem 1 extend immediately to [6, Thm. 2].

* Received by the editors August 3, 1974.

† Department of Mathematics, University of California at San Diego, La Jolla, California 92037. This research was sponsored by the Air Force Office of Scientific Research under Contract/Grant AFOSR 71-2089.

Observe that the identity in Theorem 1 depends only on the conjugate classes of the stability subgroups of the elements of Δ. Let H_i, $i = 1, \cdots, g$, denote the elements of $\mathscr{H}$ which occur as conjugates of some subgroup $G_t, t \in \Delta$. Let L_i, $i = 1, \cdots, g$, denote the set of all $t \in \Delta$ such that G_t is conjugate to H_i. Then $\gamma = \{L_i : i = 1, \cdots, g\}$ defines a partition of Δ. We denote $|L_i|$ by l_i. Let $n = |\Delta|$. An *integral partition* of n is a set of integers $\{k_1, \cdots, k_d\}$ such that $k_1 + \cdots + k_d = n$. Without loss of generality, we may assume that $0 < k_1 \leqq \cdots \leqq k_d$. Given an integral partition $\rho = \{k_1, \cdots, k_d\}$ of n, we construct a class M_ρ of nonnegative, integral $g \times d$ matrices. A matrix $B = (b_{ij})$ is in M_ρ if

(1) $$\sum_{j=1}^{d} b_{ij} = l_i, \qquad i = 1, \cdots, g;$$

(2) $$\sum_{i=1}^{g} b_{ij} = k_j, \qquad j = 1, \cdots, d;$$

(3) if $\mathbf{b}_j$ denotes the jth column vector of B and if $p < q$ and $k_p = k_q$, then $\mathbf{b}_p \leqq \mathbf{b}_q$ lexicographically.

Using (1), (2) and (3), any matrix B in M_ρ may be specified by notation $\mathbf{v}_1^{j_1}, \mathbf{v}_2^{j_2}, \cdots$, where the vectors $\mathbf{v}_t, t = 1, 2, \cdots$, are the distinct column vectors from B and $j_t, t = 1, 2, \cdots$, are their multiplicities in B.

For any vector $\mathbf{v} = (a_1, \cdots, a_g)$ define $\mathbf{v}! = (a_1!)(a_2!) \cdots (a_g!)$. Let $\mathbf{l} = (l_1, \cdots, l_g)$ where $l_i = |L_i|$ as defined above. Define

$$N(B) = \frac{\mathbf{l}!}{\prod_t (\mathbf{v}_t!)^{j_t} j_t!}.$$

THEOREM 2. *The number of G-invariant partitions is*

$$\sum_\rho \sum_{B \in M_\rho} N(B) \prod_{j=1}^{d} \sum_{H \in \mathscr{H}} \frac{1}{M_H(H)} \prod_{i=1}^{g} (M_H(H_i))^{b_{ij}}.$$

We note that the extension of this result to [6, Thm. 2] is obvious and will be illustrated in the example below (§ 3).

We also observe that this result may be extended to count the number of orbits of partitions whose stabilizer subgroups are conjugate to a given subgroup of G. We do this as follows: from [5, Thm. 2], we note that if T is a finite set, $G:T$ and $\bar{\Delta}$ a system of orbit representatives for this action, then for H a subgroup of G,

(4) $$\sum_{K \in \mathscr{H}} M_K(H) \sum_{t \in \bar{\Delta}} \chi(G_t \sim K) = \sum_{t \in T} \chi(H \subset G_t),$$

where $G_t \sim K$ means G_t is conjugate to K.

Thus by merely setting $T = \Pi(S)$ and applying (4), we obtain our desired result. In Theorem 11.3 of [2], M. J. Klass dealt with a slightly different problem, that of counting orbits of given cardinality. We remark that this number may be computed from (4). Furthermore, Theorem 11.3 of [2], which involves the Möbius function on the lattice of subgroups of G, follows easily from [4, Thm. 13].

3. Proofs and examples. We prove Theorem 2. Let $\gamma = \{L_i : i = 1, \cdots, g\}$ be defined as above. We represent $\delta = \{K_1, \cdots, K_d\} \in \Pi(\Delta)$ as a set

$$\mathscr{B} = \{\mathbf{B}_j : j = 1, \cdots, d\},$$

where

$$\mathbf{B}_j = (L_1 \cap K_j, \cdots, L_g \cap K_j).$$

With each such δ we associate a matrix $B = (b_{ij})$, where $b_{ij} = |L_i \cap K_j|$, and we adopt the convention that the blocks of δ are listed in increasing order of cardinality and that if $|K_p| = k_p = k_q$ for $p < q$, then the column vector $\mathbf{b}_p \leqq \mathbf{b}_q$ lexicographically. Observe that with this convention B satisfies (1), (2) and (3) of § 2 with $\rho = \{|K_1|, \cdots, |K_d|\}$. Thus we rewrite Theorem 1 in the form of Theorem 2, where $N(B)$ represents the cardinality of the set $\eta(B)$ of partitions δ corresponding to the matrix B.

We now compute $N(B)$. Let $\mathscr{G} = S_{L_1} \times \cdots \times S_{L_g}$ act on $\eta(B)$ as follows (S_Q denotes the symmetric group on Q): let $\mathscr{B}$ correspond to δ as above, $(\sigma_1, \cdots, \sigma_g) \in \mathscr{G}$. Define $(\sigma_1, \cdots, \sigma_g)\mathscr{B} = \mathscr{B}'$ where $\mathbf{B}'_j = (\sigma_1(L_1 \cap K_j), \cdots, \sigma_g(L_g \cap K_j))$. If $\delta' = \{K'_1, \cdots, K'_d\}$, where $|L_i \cap K'_j| = |L_i \cap K_j|$ for all i, j (that is, the associated matrices B and B' are equal, then for each i one may choose $\sigma_i \in S_{L_i}$ such that $\sigma_i(L_i \cap K'_j) = (L_i \cap K_j)$ for all j. Thus $\mathscr{G}$ acts transitively on $\eta(B)$ and $N(B) = |\mathscr{G}|/|\mathscr{G}_\delta|$ when $\mathscr{G}_\delta$ is the stabilizer of $\mathscr{G}$ at δ. As a conceptual aid, we display the set $\mathscr{B}$ associated with δ as the following array:

$$(B_{ij}) = \begin{bmatrix} B_{11} & B_{12} & \cdots & B_{1d} \\ B_{21} & B_{22} & \cdots & B_{2d} \\ \vdots & & & \\ B_{g1} & B_{g2} & \cdots & B_{gd} \end{bmatrix},$$

where $B_{ij} = L_i \cap K_j$. $\mathscr{B}$ is the set of column vectors of (B_{ij}). Observe that $\mathscr{G}_\delta$ consists of all permutations in $\mathscr{G}$ which permute those columns of (B_{ij}) yielding identical column vectors in $B = (b_{ij}) = (|B_{ij}|)$, or which permute elements within the sets B_{ij}. The number of such permutations is clearly $\prod_t (v_t!)^{j_t} j_t!$. Theorem 2 follows. □

We illustrate Theorem 2 with the following example. Let S be the squares of an 8×8 chessboard. Let G be the dihedral group of order 8 acting on S. We list a complete set of nonconjugate subgroups of G:

$$G_1 = G = \{e, \sigma, \sigma^2, \sigma^3, \tau, \sigma\tau, \sigma^2\tau, \sigma^3\tau\},$$

where e is the identity, σ is a 90° clockwise rotation and τ is a reflection through a line through opposite corners,

$$G_2 = \{e, \sigma, \sigma^2, \sigma^3\}, \quad G_3 = \{e, \sigma^2, \sigma\tau, \sigma^3\tau\}, \quad G_4 = \{e, \sigma^2, \tau, \sigma^2\tau\},$$
$$G_5 = \{e, \sigma^2\}, \quad G_6 = \{e, \sigma\tau\}, \quad G_7 = \{e, \tau\} \quad \text{and} \quad G_8 = \{e\}.$$

The table of marks of G is Table 1.

TABLE 1

	G_1	G_2	G_3	G_4	G_5	G_6	G_7	G_8
G_1	1							
G_2	1	2						
G_3	1	0	2					
G_4	1	0	0	2				
G_5	1	2	2	2	4			
G_6	1	0	2	0	0	2		
G_7	1	0	0	2	0	0	2	
G_8	1	2	2	2	4	4	4	8

where the (i, j)th entry is $M_{G_j}(G_i)$. We list a system of orbit representatives for $G:S$ on the chessboard below:

4	7	9	10				
	3	6	8				
		2	5				
			1				

;

TABLE 2.

r	$x(r)$	r	$x(r)$
1	1	18	65, 519, 488, 045
2	23, 828, 736	20	66, 268, 850, 337
3	336, 152, 832	22	66, 271, 980, 705
4	10, 675, 543, 925	24	66, 476, 118, 945
5	15, 217, 251, 125	26	66, 476, 253, 345
6	26, 284, 502, 917	28	66, 525, 347, 877
7	28, 249, 780, 357	30	66, 525, 350, 949
8	50, 568, 389, 776	32	66, 536, 146, 286
9	51, 074, 368, 656	36	66, 538, 303, 906
10	54, 278, 741, 392	40	66, 538, 696, 310
11	54, 341, 139, 856	44	66, 538, 760, 174
12	62, 293, 956, 800	48	66, 538, 769, 770
13	62, 298, 921, 152	52	66, 538, 770, 982
14	62, 780, 111, 040	56	66, 538, 771, 126
15	62, 780, 307, 648	60	66, 538, 771, 138
16	65, 472, 118, 765	64	66, 538, 771, 139

$\Delta = \{1, \cdots, 10\}$. We see that $\gamma = \{\{1, 2, 3, 4\}, \{5, 6, 7, 8, 9, 10\}\}$, $g = 2$, $l_1 = 4$, $l_2 = 6$, $H_1 = G_7$ and $H_2 = G_8$. We now ask: how many G-invariant partitions, $x(r)$, of the squares of the chessboard are there such that no block has cardinality greater than r? We answer this question in the Table 2, by using the extension of the present Theorem 2 to Theorem 2 of [6]. We remark that as was developed in [6], we may apply modifications of Theorem 2 to count rooted trees, full cycle permutations, etc., on the blocks of G-invariant partitions. We also observe that the classification of the elements of Δ by stabilizer subgroup may be performed using (4) and need not be stated explicitly.

REFERENCES

[1] W. Burnside, *Theory of Groups of Finite Order*, 2nd ed., Cambridge University Press, Cambridge, 1961; Dover, New York, 1955.

[2] M. J. Klass, *Enumeration of partition classes induced by permutation groups*, Ph.D. dissertation, Dept. of Math., Univ. of Calif. at Los Angeles, 1972.

[3] P. K. Stockmeyer, *Enumeration of graphs with prescribed automorphism group*, Ph.D. dissertation, Univ. of Mich., Ann Arbor, 1971.

[4] D. E. White, *Classifying patterns by automorphism group: An operator theoretic approach*, this Journal, submitted.

[5] ———, *Counting patterns with a given automorphism group*, Proc. Amer. Math. Soc., to appear.

[6] D. E. White and S. G. Williamson, *Combinatorial structures and group invariant partitions*, Ibid., submitted.

SIAM J. COMPUT.
Vol. 4, No. 3, September 1975

EVERY PRIME HAS A SUCCINCT CERTIFICATE*

VAUGHAN R. PRATT†

Abstract. To prove that a number n is composite, it suffices to exhibit the working for the multiplication of a pair of factors. This working, represented as a string, is of length bounded by a polynomial in $\log_2 n$. We show that the same property holds for the primes. It is noteworthy that almost no other set is known to have the property that short proofs for membership or nonmembership exist for all candidates without being known to have the property that such proofs are easy to come by. It remains an open problem whether a prime n can be recognized in only $\log_2^\alpha n$ operations of a Turing machine for any fixed α.

The proof system used for certifying primes is as follows.

AXIOM. $(x, y, 1)$.

INFERENCE RULES.

$$\mathrm{R}_1: \quad (p, x, a), q \vdash (p, x, qa) \quad \text{provided } x^{(p-1)/q} \not\equiv 1 \pmod p \text{ and } q|(p-1).$$

$$\mathrm{R}_2: \quad (p, x, p-1) \vdash p \quad \text{provided } x^{p-1} \equiv 1 \pmod p.$$

THEOREM 1. *p is a theorem $\equiv$ p is a prime.*

THEOREM 2. *p is a theorem $\supset$ p has a proof of $\lceil 4 \log_2 p \rceil$ lines.*

Key words. primes, membership, nondeterministic, proof, NP-complete, computational complexity

1. Proofs.

We know of no efficient method that will reliably tell whether a given number is prime or composite. By "efficient", we mean a method for which the time is at most a polynomial in the length of the number written in positional notation. Thus the cost of *testing* primes and composites is very high. In contrast, the cost of *selling* composites (persuading a potential customer that you have one) is very low—in every case, one multiplication suffices. The only catch is that the salesman may need to work overtime to prepare his short sales pitch; the effort is nevertheless rewarded when there are many customers.[1]

At a meeting of the American Mathematical Society in 1903, Frank Cole used this property of composites to add dramatic impact to the presentation of his paper. His result was that $2^{67} - 1$ was composite, contradicting a two-centuries-old conjecture of Mersenne. Although it had taken Cole "three years of Sundays" to find the factors, once he had done so he could, in a few minutes and without uttering a word, convince a large audience of his result simply by writing down the arithmetic for evaluating $2^{67} - 1$ and $193707721 \times 761838257287$.

We now show that the primes are to a lesser extent similarly blessed; one may certify p with a proof of at most $\lceil 4 \log_2 p \rceil$ lines, in a system each of whose inference rules are readily applied in time $O(\log^3 p)$. The method is based on the Lucas–Lehmer heuristic (Lehmer (1927)) for testing primeness.

In the system to be described, theorems take one of two forms:

(i) "p", asserting that p is prime, or

(ii) "(p, x, a)", asserting that we are making progress towards establishing that p is a prime and that x is a primitive root (mod p); a is a progress indicator

* Received by the editors May 24, 1974.

† Project MAC Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. This research was supported by the National Science Foundation under Grant GJ-34671.

[1] Edmonds (1965) discusses a similar situation with a "supervisor and his hard-working assistant".

such that when it reaches $p - 1$, we may establish these properties for p and x in one more step.

The system is as follows.

AXIOM. $(x, y, 1)$.

INFERENCE RULES.

R_1: $(p, x, a), q \vdash (p, x, qa)$ provided $x^{(p-1)/q} \not\equiv 1 \pmod{p}$ and $q | (p-1)$;

R_2: $(p, x, p-1) \vdash p$ provided $x^{p-1} \equiv 1 \pmod{p}$.

A certificate of p is then a proof in this system with last line p.

Some familiar primes are given by the following proofs.

(1)	$(2, 1, 1)$	Axiom;
(2)	2	(1), R_2, $1^1 \equiv 1 \pmod{2}$;
(3)	$(3, 2, 1)$	Axiom;
(4)	$(3, 2, 2)$	(3), (2), R_1, $2^1 \equiv 2 \pmod{3}$;
(5)	3	(4), R_2, $2^2 \equiv 1 \pmod{3}$.

No proof for 4 is possible because we would need to prove $(4, x, 3)$ for some $x \equiv 1 \pmod{4}$ (by the condition in R_2), which would contradict the condition in R_1.

(6)	$(5, 2, 1)$	Axiom;
(7)	$(5, 2, 2)$	(6), (2), R_1, $2^2 \equiv 4 \pmod{5}$;
(8)	$(5, 2, 4)$	(7), (2), R_1, $2^2 \equiv 4 \pmod{5}$;
(9)	5	(8), R_2, $2^4 \equiv 1 \pmod{5}$.

No proof for 6 is possible because $x^5 \not\equiv 1 \pmod{6}$ for all $x \not\equiv 1 \pmod{6}$.

(10)	$(11, 2, 1)$	Axiom;
(11)	$(11, 2, 2)$	(10), (2), R_1, $2^5 \equiv 10 \pmod{11}$;
(12)	$(11, 2, 10)$	(11), (9), R_1, $2^2 \equiv 4 \pmod{11}$;
(13)	11	(12), R_2, $2^{10} \equiv 1 \pmod{11}$;
(14)	$(23, 5, 1)$	Axiom;
(15)	$(23, 5, 2)$	(14), (2), R_1, $5^{11} \equiv 22 \pmod{23}$;
(16)	$(23, 5, 22)$	(15), (13), R_1, $5^2 \equiv 2 \pmod{23}$;
(17)	23	(16), R_2, $23^{22} \equiv 1 \pmod{23}$;
(18)	$(47, 5, 1)$	Axiom;
(19)	$(47, 5, 2)$	(18), (2), R_1, $5^{23} \equiv 46 \pmod{47}$;
(20)	$(47, 5, 46)$	(19), (17), R_1, $5^2 \equiv 25 \pmod{47}$;
(21)	47	(20), R_2, $5^{46} \equiv 1 \pmod{47}$.

Not counting the proof for 3, this (shortest) proof of 47 took 18 steps, not too far from the promised bound of $\lceil 4 \log_2 47 \rceil = 22$. The gap is mostly due to the proof of 47 not using the proof of 3 that is counted in the bound $\lceil 4 \log_2 p \rceil$. A much larger gap is exhibited by the proof of 474397531, which is 23 lines long; here, $\lceil 4 \log_2 p \rceil = 116$. This prime was constructed to show that our bound on proof length is not always tight. Steps (1) to (9) are as above.

(10)	(251, 6, 1)	Axiom;
(11)	(251, 6, 2)	(10), (2), R_1;
(12)	(251, 6, 10)	(11), (9), R_1;
(13)	(251, 6, 50)	(12), (9), R_1;
(14)	(251, 6, 250)	(13), (9), R_1;
(15)	251	(14), R_2;
(16)	(474397531, 2, 1)	Axiom;
(17)	(474397531, 2, 2)	(16), (2), R_1;
(18)	(474397531, 2, 6)	(17), (5), R_1;
(19)	(474397531, 2, 30)	(18), (9), R_1;
(20)	(474397531, 2, 7530)	(19), (15), R_1;
(21)	(474397531, 2, 1890030)	(20), (15), R_1;
(22)	(474397531, 2, 474397530)	(21), (15), R_1;
(23)	474397531	(22), R_2.

2. Metaproofs.

2. Metaproofs. We now prove soundness and completeness of our system.

THEOREM 1. *p is a prime if and only if p is a theorem.*

Proof. If. No number has multiplicative order $p - 1 \pmod p$ when p is not a prime. If such a p is proved, it must be by application of R_2 to $(p, x, p - 1)$ where $x^{p-1} \equiv 1 \pmod p$. Hence $x^j \equiv 1 \pmod p$ for some $j < p - 1$. Now $j | p - 1$, so $x^{(p-1)/q} \equiv 1 \pmod p$ for some prime q. But to prove $(p, x, p - 1)$, we had to build up $p - 1$ as the product of primes q which satisfied $x^{(p-1)/q} \not\equiv 1 \pmod p$. Applying the fundamental theorem of arithmetic then leads to a contradiction.

Only if. This part proceeds by induction on p. If p is prime, then p has a primitive root $(\bmod\ p)$, that is, a number whose multiplicative order $(\bmod\ p)$ is $p - 1$. A proof of p may start with the axiom $(p, x, 1)$ for such a primitive root x. By the induction hypothesis, each of the prime factors of $p - 1$ is a theorem. Moreover, for each such prime factor q, $x^{(p-1)/q} \not\equiv 1 \pmod p$; otherwise the order of x would be less than $p - 1$. Hence the proof system permits the inference of any theorem (p, x, a), where a is a product of prime factors of $p - 1$. In particular, $(p, x, p - 1)$ may be inferred, and since $x^{p-1} \equiv 1 \pmod p$, we may infer p. □

We now establish the efficiency of our method.

THEOREM 2. *If p is a theorem, then p has a proof of at most $\lceil 4 \log_2 p \rceil$ lines.*

Proof. The construction given in the proof of Theorem 1 yields such a proof.

First prove 2 and 3 in five lines. (These primes p are special because $p - 1$ is not composite.) We now assume as our induction hypothesis that by not counting the proofs of 2 and 3, each prime p can be proved in at most $\lfloor 4 \log_2 p \rfloor - 4$ lines.

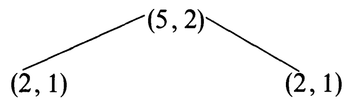
For $p = 2$ or 3, this follows directly from the identities $\lfloor 4 \log_2 2 \rfloor - 4 = 0$ and $\lfloor 4 \log_2 3 \rfloor - 4 = 2$. For $p > 3$, let $p - 1 = p_1 p_2 \cdots p_k$, $k \geqq 2$. Then the cost of proving p is bounded above by

$$\begin{aligned} &2 + k + \sum_{1 \leqq i \leqq k} (\lfloor 4 \log_2 p_i \rfloor - 4) && \text{(by the induction hypothesis)} \\ &\leqq \lfloor 4 \log_2 p_1 p_2 \cdots p_k) \rfloor - 4 && \text{(since } k \geqq 2) \\ &\leqq \lfloor 4 \log_2 p \rfloor - 4, && \text{(the desired answer).} \end{aligned}$$

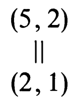
If we now count the 5 lines required to prove 2 and 3, the cost rises to $\lfloor 4 \log_2 p \rfloor + 1$ lines. For $p > 2$, $\log_2 p$ will not be an integer, and so the cost is bounded by $\lceil 4 \log_2 p \rceil$, a bound that is 4 when $p = 2$ and is therefore applicable to all p. □

Almost identical proofs may be used to show that no more than $\lfloor 3 \log_2 p \rfloor$ lines involve an exponentiation and $\lfloor 2 \log_2 p \rfloor$ a multiplication, facts which we will use in the next section.

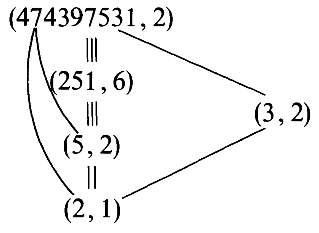
3. Picturesque proofs. The reader should have little difficulty in seeing that all the information in the proof that 5 is prime is contained in the following tree, whose vertices are primes together with their primitive roots.



or, collapsing repeated vertices for brevity:



The proof tree for 474397531, when collapsed, becomes



It is straightforward to check a proof tree without reconstructing the proof. The "VELP" test (vertices, edges, leaves, products) is

(i) For each vertex (p, x), $x^{p-1} \equiv 1 \pmod p$.

(ii) For each edge (p, x) down to (q, y), $x^{(p-1)/q} \not\equiv 1 \pmod p$ and $q | p - 1$.

(iii) Each leaf is $(2, 1)$.

(iv) For each vertex (p, x) with immediate descendants $(p_1, x_1), \cdots, (p_k, x_k)$, $p = p_1 p_2 \cdots p_k + 1$.

The proof tree approach is more picturesque than the proof system, whose raison d'être is that it is more formally and compactly presented.

4. Computations. Returning to our customer, we find him dissatisfied with the exponentiation he must carry out to check a line. He protests that the evaluation of x^b requires $b - 1$ multiplications, and also that the numbers produced along the way have $O(b)$ digits, which he has neither time nor paper to write down for large b.

The first protest is dealt with by the well-known trick of exponentiating by repeated squaring, which yields x^b with at most $2\lfloor \log_2 b \rfloor$ multiplications. This method is an essential feature of the Lucas–Lehmer heuristic. The method may be described recursively as:

$$\begin{aligned} x^b\colon\ & b = 0 \to 1, \\ & b \text{ odd} \to xx^{b-1}, \\ & b \text{ even} \to (x^2)^{b/2}. \end{aligned}$$

To eliminate the recursion and attendant waste of space, we translate this algorithm into a "deterministic" system whose rules are

$$\begin{aligned} (u, v, w) &\vdash (u^2, v/2, w) && \text{if } v \text{ is even}, \\ (u, v, w) &\vdash (u, v - 1, uw) && \text{if } v \text{ is odd}. \end{aligned}$$

Now wu^v is an invariant of these rules, each of which reduces either the number of significant bits (provided $v \neq 0$) or the number of 1's in v (expressed in binary notation), but not both. Hence $(x, b, 1) \overset{*}{\vdash} (y, 0, x^b)$ in a number of steps exactly one less than the number of bits plus the number of 1's in b, which is at most $2\lceil \log_2 (b + 1)\rceil - 1$. By skipping the multiplication the first time w is multiplied by u, and beginning with (x, b, x), only $2\lfloor \log_2 b \rfloor$ multiplications are required.

The second protest is disposed of by performing each multiplication modulo p in the above algorithms when testing $x^b \equiv 1 \pmod{p}$.

In any proof of p, each multiplication is performed modulo q for some prime $q \leqq p$. Moreover, in testing $x^b, b < p$. Hence each exponentiation requires at most $2\lfloor \log_2 p \rfloor$ multiplications of numbers smaller than p. At most $\lfloor 3 \log_2 p \rfloor$ exponentiations are required, whence no more than $6 \log_2^2 p$ multiplications plus the $\lfloor 2 \log_2 p \rfloor$ multiplications from R_1 are needed. Each multiplication may be carried out in $O(\log p \log \log p)$ steps on a random access machine (RAM) (Schönhage and Strassen, (1971)), and so $O(\log^3 p \log \log p)$ steps suffice to check a proof of p on a RAM. (A factor of $\log \log \log p$ creeps in for those who do the arithmetic on paper (or on a Turing machine) due to time spent scanning and shuffling the sheets!)

An item that might find a market among consumers of prime numbers would be a pocket calculator with a predicate (x, b, p) that evaluates $x^{(p-1)/b} \equiv 1 \pmod{p}$. Only one bit of output is required, only integer arithmetic (multiple-precision) is used, and so the unit should cost about \$100 in quantity at today's prices, assuming that it handles integers of up to several hundred bits. Users of the Hewlett Packard HP-65 pocket computer with the appropriate program may find it suitable but expensive. A proof using our method of, say, the smallest Mersenne prime yet undiscovered would require a considerably more expensive unit, with perhaps 30,000-bit integers and sufficient parallelism to make the computation time acceptably low.

5. Complexity. The families NP (P) of sets of strings accepted (recognized) in time some polynomial function of their length by some nondeterministic

(deterministic) Turing machine[2] have recently engaged the attention of computational complexity theorists. The family P is of interest in that it includes all sets that can be recognized reasonably quickly, a property that has become identified to some extent with membership in P. The family NP is of interest (Cook (1971), Karp (1972)) because it includes thirty or more operations-research-related sets each with the astonishing property that if it belonged to P, then NP = P, implying that all of its fellow O.R. sets would be in P, along with other sets in NP (such as {primes} as we showed above) not known at present to belong to P. In view of the effort that has been expended in the past twenty years or so on trying to show that any one of these sets is in P, it is widely conjectured that none is, that is, NP $\neq$ P. These peculiar sets are called NP-complete.

A family of sets that has only very recently attracted any attention is coNP $= \{S | \bar{S} \in \mathrm{NP}\}$. Of course, coP = P, whence if NP = P, then coNP = NP. However, it is conceivable that NP $\neq$ P but NP = coNP. It is straightforward to show that NP = coNP if and only if some NP-complete set is in coNP, just as NP = P if and only if some NP-complete set is in P, and it is conjectured that NP $\neq$ coNP.

If true, this implies that NP $\cap$ coNP contains no NP-complete problems. One is tempted to speculate that NP $\cap$ coNP = P. After all, the families RE and R of recursively enumerable and recursive sets, whose relationship resembles the NP – P relationship, satisfy RE $\cap$ coRE = R; and until recently, every known member of NP $\cap$ coNP was known to be in P. Thus one could be forgiven for wanting to conjecture that NP $\cap$ coNP = P.

An immediate corollary of § 4 above is that the primes are in NP $\cap$ coNP. Provided NP $\neq$ coNP, this settles in the negative a question raised by Cook as to whether the composites are NP-complete. Conjecture aside, it gives us the first known member of NP $\cap$ coNP not known to be in P. Chvatal has recently exhibited another set with this property, namely the set of pairs (linear programming problem, optimal solution to it). No other such sets are known, although a plausible candidate is the set of irreducible univariate polynomials over the integers. Berlekamp (1967) has shown that over any finite field such a set is in P. A somewhat less plausible candidate is the set of pairs of isomorphic graphs.

If the primes or the optimal-lp-solutions are not in P, it will not be because they are NP-complete (still supposing NP $\neq$ coNP) which is the *usual* reason. One might therefore say that these problems were *anomalously* hard, although any term for this phenomenon lacks the all-or-nothing significance of "NP-completeness". The whole question of proving lower bounds on the complexity of sets in NP is completely open, and any information about the structure of hard problems would be welcome. In particular, the criterion that membership in NP $\cap$ coNP precludes NP-completeness, though based only on a conjecture, is nonetheless a useful guide considering how few tools we have in the area.

6. Conclusion. We exhibited a simple system whose theorems are exactly the set of all primes and whose proofs are very short. We inferred from this that the primes are in NP $\cap$ coNP, giving us our first example of a member of NP $\cap$ coNP not known to be in P. We advocated membership in NP $\cap$ coNP as a strong

[2] That is, for each such set there is a polynomial and a Turing machine.

reason for presuming non-NP-completeness, based on the plausible and moderately popular conjecture that NP $\neq$ coNP. We observed the striking paucity of sets that are candidates for lying between P and NP-complete sets. It is interesting to find the number theorists' most famous set occupying a special position in complexity theory.

REFERENCES

E. R. Berlekamp (1967), *Factoring polynomials over finite fields*, Bell System Tech. J., 46, pp. 1853–1860.

S. A. Cook (1971), *The complexity of theorem-proving procedures*, Conf. Rec. of 3rd Ann. ACM Symp. on Theory of Computing, pp. 151–158.

J. Edmonds (1965), *Minimum partition of a matroid into independent subsets*, J. Res. Nat. Bur. Standards Sect. B, 69B, pp. 67–72.

R. M. Karp (1972), *Reducibilities among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, pp. 85–103.

D. H. Lehmer (1927), Bull. Amer. Math. Soc., 33, pp. 327–340.

A. Schönhage and V. Strassen (1971), *Fast multiplication of large numbers*, Computing, 7, pp. 281–292. (In German.) English description: D. E. Knuth, The Art of Computer Programming, vol. 2, 2nd printing, pp. 270–275.

SIAM J. COMPUT.
Vol. 4, No. 3, September 1975

FINDING A MAXIMUM CUT OF A PLANAR GRAPH IN POLYNOMIAL TIME*

F. HADLOCK†

Abstract. The problem of finding a maximum cut of an arbitrary graph is one of a list of 21 combinatorial problems (Karp–Cook list). It is unknown whether or not there exist algorithms operating in polynomial bounded time for any of these problems. It has been shown that existence for one implies existence for all. In this paper we deal with a special case of the maximum cut problem. By requiring the graph to be planar, it is shown the problem can be translated into a maximum weighted matching problem for which there exists a polynomial bounded algorithm.

Key words. maximum cut, planar graph, geometric dual, polynomial time

1. Introduction. In this paper, it is shown that the maximum cut problem can be translated into the maximum weighted matching problem when the graph under consideration is planar. For an arbitrary graph, several algorithms exist for finding a maximum cut [4] and [5]. Both require exponential time in worst-case situations. Since the maximum weighted matching problem has a polynomial bounded algorithm [1], [2], a maximum cut of a planar graph can be found in polynomial time by using the translation process to be presented.

2. Maximum cuts and odd circuits. An edge set D whose removal leaves a subgraph free of odd circuits will be called an *odd-circuit cover*. The purpose of this section is to obtain an alternative formulation for the problem of finding a maximum cut. First the relationship between cuts and odd-circuit covers is established.

THEOREM 1. *An edge set is contained in a cut if and only if its complement is an odd-circuit cover.*

Proof. Let Q be an edge set contained in a cut C. The intersection of any circuit with C is even and so $\sim Q$ must contain an edge of any odd circuit. Hence $\sim Q$ is an odd-circuit cover.

Conversely, if $\sim Q$ is an odd-circuit cover, its removal leaves a graph free of odd circuits and hence bipartite. Thus Q is contained in a cut.

As a consequence of Theorem 1, an alternative to looking for maximum cuts is to look for minimum odd-circuit covers. This is justified by the following corollary, which follows immediately from Theorem 1.

COROLLARY 1. *An edge set is a maximum cut if and only if its complement is a minimum odd-circuit cover.*

The following fact means we can confine our attention to a circuit basis rather than looking at the entire space in constructing an odd-circuit cover. Since a graph is bipartite if and only if its circuit space has an even basis [6], an edge set D is an odd-circuit cover if and only if its removal leaves a subgraph with an even basis. The term even *basis* refers to a circuit basis in which every element is an even circuit.

3. Odd-circuit covers and odd-vertex pairings. To obtain a maximum cut of a planar graph G, we suppose some embedding and take as a basis the contours of the finite faces. It is more convenient to work with the geometric dual, G_D, of G, where

* Received by the editors November 26, 1973, and in revised form July 24, 1974.

† Department of Mathematics, Florida Atlantic University, Boca Raton, Florida 33432.

the odd basis elements (along with the contour of the infinite face, if odd) become precisely the set of odd vertices. An edge e in G corresponds to an edge e' in G_{D} if and only if the two faces separated by e in the embedding of G correspond to the endpoints of e' in G_{D}.

An edge set whose removal leaves a subgraph free of odd vertices will be called an *odd-vertex pairing.* Thus a subgraph with an *odd-vertex pairing* as edge set has an Euler subgraph as complement. The following theorem establishes a correspondence between odd-circuit covers and odd-vertex pairings.

THEOREM 2. *An edge set D is an odd-circuit cover of a planar graph G if and only if the corresponding edge set P is an odd-vertex pairing for the geometric dual G_{D} of G.*

Proof. Let G_{D} be the geometric dual of G for some embedding of G, with D and P corresponding edge sets of G and G_{D}, respectively. Let G' and G'_{D} be the subgraphs of G and G_{D} left by the removal of D and P. Circuits of G correspond by the 1–1 edge correspondence to cut-sets of G_{D}. This is also true for G' and G'_{D}. In particular, circuit basis elements of G' correspond to cut-set basis elements of G'_{D} as follows. A circuit basis element of G is the contour of a finite face. Its edges correspond in 1–1 fashion with the edges of G_{D} which are incident with the vertex representing that face. The set of edges incident with the vertex is a cut-set basis element.

If D is an odd-circuit cover, the circuit basis for G' is even. Since the edge correspondence is 1–1, the cut-set basis of G'_{D} is even. Consequently the degree must be even for any vertex of G'_{D} corresponding to a finite face of G'. The vertex corresponding to the infinite face cannot be the sole odd vertex. Hence P is an odd-vertex pairing.

The converse follows by a similar argument.

To find a maximum cut of a planar graph, it suffices to find a minimum odd-vertex pairing of its dual. The following theorem gives a useful characterization of odd-vertex pairings.

THEOREM 3. *For an edge set P of an arbitrary multigraph G, P is a minimum odd-vertex pairing if and only if P forms an edge disjoint collection of paths with odd vertices of G as endpoints, using each once as endpoint, and with minimum sum of path lengths.*

Proof. Let P be a minimum odd-vertex pairing for a multigraph G. The parity of a vertex refers to its degree: odd or even. If H is the subgraph left by the removal of P, since H is an Euler subgraph, a vertex has the same parity in P as in G. In any component of a graph, there must be an even number of odd vertices; hence any odd vertex in P is connected to another. Remove a path connecting a pair of odd vertices from both P and G to obtain subgraphs P_1 and G_1. P_1 is an odd-vertex pairing for G_1 since its removal leaves H. Any vertex has the same parity in P_1 as G_1. In going from P to P_1, the number of odd vertices has been reduced by two. Repeating the process eventually yields an odd-vertex pairing P_i with no odd vertices for a multigraph G_i. Since any vertex has the same parity in G_i as P_i, G_i is Euler. Since P was assumed to be minimal, $G_i = H$ and $P_i = (V, \varnothing)$ (i.e., no edges). Then P is the disjoint collection of paths with the odd vertices of G as endpoints, using each once as endpoint. The sum of the path lengths is minimum since P is minimum.

Now suppose P is a collection of edge-disjoint paths with odd vertices as endpoints, using each endpoint once as endpoint, and with minimum sum of path lengths. Remove P from G, one path at a time. Denote by H the subgraph remaining after P has been removed. The removal of each path leaves the endpoints even and does not alter the parity of intermediate vertices. Since any vertex odd in G appears once as an endpoint, it is even in H, and so P is an odd-vertex pairing. P is minimum since the sum of path lengths is minimum.

4. Odd-vertex pairings. The task of pairing odd vertices so as to minimize the sum of the lengths of the paths pairing them is easily posed as a maximum matching problem as observed in [3].

Given a multigraph G, a minimum odd-vertex pairing P for G is obtained as follows. Let G_c be the complete graph with vertices corresponding to the odd vertices of G. With each edge $e = (u, v)$, associate the weight $W - d(u, v)$ where $d(u, v)$ is the length of the minimum length path connecting u and v and $W = 1 + \max \{d(u, v) | u, v \text{ odd in } G\}$. Let M be a maximum matching of G_c. Then M defines a minimum odd vertex pairing as follows. For each edge $e = (u, v)$ in M, include in P the edges of *any* minimum length path connecting u and v in G.

The problem is now in a form for which there exists an algorithm [2] for its solution. It is an algorithm which is good in the sense that the amount of time it requires is a polynomial function of an input parameter (the number of vertices in this case).

5. An example. To illustrate the process of translating a solution to the maximum matching problem into a solution to the maximum cut problem, we use an example (Fig. 1) in which the matching problem is solved by inspection. A minimum odd-circuit cover may be found by determining a minimum odd-vertex pairing of the geometric dual G_D (Fig. 2). In turn, this may be found by determining a maximum matching for the complete graph on the odd vertices of G_D (Fig. 3). Since the weights here are 1 or 2, any complete matching with all edge weights 2 is a maximum matching (Fig. 4). A minimum odd-vertex pairing for G_D is obtained by taking a minimum path connecting u and v for each edge (u, v) in the maximum matching. In this case each minimum path is single edge (u, v).

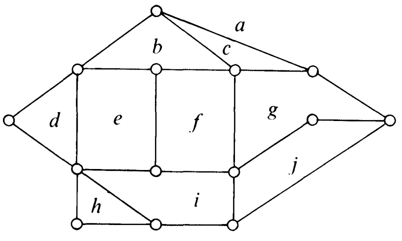


FIG. 1. *Planar graph G*

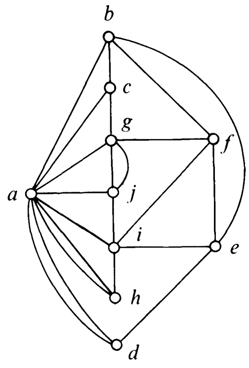


FIG. 2. *Dual of G, G_D*

$$\begin{array}{c|cccccc}
 & a & c & d & g & h & i \\
a & & 2 & 2 & 2 & 2 & 2 \\
c & & & 1 & 2 & 1 & 1 \\
d & & & & 1 & 1 & 1 \\
g & & & & & 1 & 1 \\
h & & & & & & 2 \\
i & & & & & &
\end{array}$$

FIG. 3. *Edge weights for complete graph G_c on odd vertices of G_D*

$$M = \{(a, d), (c, g), (h, i)\}$$

FIG. 4. *Maximum matching for G_c*

$$P = \{(a, d), (c, g), (h, i)\}$$

FIG. 5. *Minimum odd-vertex pairing for G_D*

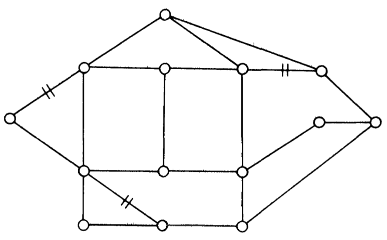


FIG. 6. *Minimum odd circuit cover for G*

Their collection forms an odd-vertex pairing (Fig. 5). The corresponding minimum odd-circuit cover for *G* consists of the marked edges (Fig. 6). Its complement is a maximum cut.

6. Conclusions. Finding a maximum cut of a planar graph is a special case, as remarked earlier, of a problem on a list [8] of combinatorial optimization problems, including the traveling salesman problem and the problem of vertex coloring a graph with the fewest number of colors. If any of these problems have a polynomial bounded algorithm, all do. In this paper, the existence of a polynomial bounded algorithm for a special case (planar graphs) of one of these problems may aid in defining special cases of the others for which polynomial bounded algorithms exist. At the same time, an attempt to extend this approach to the general case might lend insight as to why a polynomial bounded algorithm does *not* (as is widely believed) exist for the general case.

REFERENCES

[1] J. Edmonds, *Paths, trees, and flowers*, Canad. J. Math., 17 (1965), pp. 449–467.

[2] ———, *Maximum matching and a polyhedron with 0,1-vertices*, J. Res. Nat. Bur. Standards Sect. B, 69B (1965), pp. 125–130.

[3] S. Goodman and S. Hedetniemi, *Eulerian walks in graphs*, this Journal, 2 (1973), pp. 16–27.

[4] F. Hadlock, *The minimal 2-coloration problem*, Proc. 3rd Southeastern Conf. on Combinatorics, Graph Theory, and Computing, F. Hoffman, R. B. Levow, R. S. D. Thomas, eds., Utilitas Mathematica, Winnipeg, Canada, 1972, pp. 221–241.

[5] ———, *Optimal graph partitions*, Proc. 4th Southeastern Conf. on Combinatorics, Graph Theory, and Computing, F. Hoffman, R. B. Levow, R. S. D. Thomas, eds., Utilitas Mathematica, Winnipeg, Canada, 1973, pp. 309–328.

[6] F. Harary, *Graph Theory*, Addison-Wesley, Reading, Mass., 1969.

[7] S. Johnson, *Approximation algorithms for combinatorial problems*, Proc. 5th Annual ACM Symp. on Theory of Computing, Austin, Tex., 1973, pp. 38–49.

[8] R. Karp, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–103.

SIAM J. COMPUT.
Vol. 4, No. 3, September 1975

COMPLETE REGISTER ALLOCATION PROBLEMS*

RAVI SETHI†

Abstract. The search for efficient register allocation algorithms dates back to the time of the first FORTRAN compiler for the IBM 704. The model we use in this paper is a single processor with an arbitrarily large number of general registers. The objective is to use as few registers as possible, under the constraint that no stores into memory are permitted. The programs under consideration are sequences of assignment instructions. We show that, given a program and an integer k, determining if the program can be computed using k registers is polynomial complete. It should be noted that k can be any integer.

Key words. register allocation, program optimization, polynomial complete, straight line program, basic block, dag

1. The machine model. Despite the fact that register allocation has been of interest to compiler writers since the time of the first FORTRAN compiler for the IBM 704 [2], there are few algorithms available for producing optimal allocations. In this paper we will demonstrate that register allocation is a member of a class of problems for which there is no known nonenumerative solution.

For the purposes of introduction, the machine model in this paper consists of a single processor, a memory and an arbitrarily large number of registers. Instructions are of two types:

(i) Load register j—take a value from a specified location in memory and place it in register j; all other registers are unchanged.
(ii) reg $j \leftarrow \theta(\text{reg } i_1, \text{reg } i_2)$—the result of applying the operator θ to the contents of registers i_1 and i_2 is placed in register j; all other registers are unchanged. Registers i_1, i_2 and j need not necessarily be distinct.

The absence of input-output instructions leads to the assumption that all initial values are already in memory. All registers are initially empty.

The absence of control flow and test instructions means that the programs that can be computed using the above model are sequences of assignment instructions. Such programs have been referred to as *straight line programs* [1] or *basic blocks* [2].

As an aside, we note that single expressions are a special class of straight line programs. Algorithms for determining the minimal number of registers needed to compute expressions which have no common subexpressions, may be found in [3], [8], [9], [10]. The algorithms operate in linear time and can handle algebraic properties like associativity and commutativity.

Another restriction on the instruction set in the model is the absence of *stores*, which transfer a value from a register into memory. Straight line sections encountered during compilation tend to require a small number of registers. Hence the objective, as it will be in this paper, is to limit the number of registers used, while not permitting stores.

2. Graphical representation of programs. Graphical representations of programs are intuitive, and perhaps best introduced by an example.

* Received by the editors February 1, 1973, and in revised form May 13, 1974.

† Pennsylvania State University, University Park, Pennsylvania 16802. This work was supported in part at Princeton University by the National Science Foundation under Grant GJ-1052.

Example 2.1. Consider the evaluation of the polynomial $a + bx + cx^2$, using the expression $a + (b + c * x) * x$. The "dag" corresponding to this expression is given by Fig. 1.

A *directed graph* G is a pair (V, B) where V is a set of *nodes*, or *vertices*. Elements of B are called *branches*, or *edges*, and are pairs of nodes. Informally, nodes represent values relevant to a program, and a branch (x, y) indicates that node y must be computed before node x. Given a branch (x, y), node x is called a *direct ancestor* of node y, and y is called a *direct descendant* of x. A sequence $x_0 x_1 \cdots x_k$, $k \geqq 0$, where for all i, $1 \leqq i \leqq k$, (x_{i-1}, x_i) is a branch, is called a *path of length k*. S is said to be *from* x_0 *to* x_k. If S is a path from x to y, then x is said to be an *ancestor* of y, and y is said to be a *descendant* of x.

In keeping with intuitive understanding that a branch (x, y) implies that y must be computed before x, it follows that a path from x to y means that y must be computed before x.

A path of length greater than 0 from a node x to x is called a *cycle*. A directed acyclic graph (abbreviated dag), is a directed graph with no cycles.

The graph in Fig. 1 is a dag. An algorithm to construct the dag for a straight line program may be found in [1]. For our purposes, we will assume that a straight line program is specified by giving its dag representation.

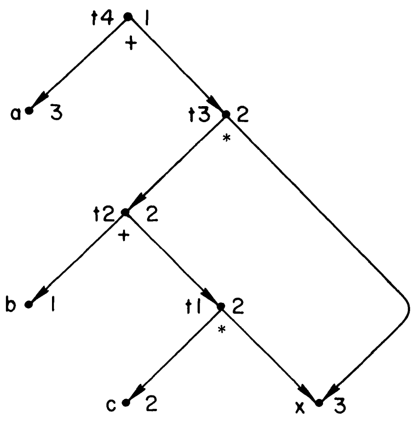


FIG. 1. *A dag for the expression $a + (b + c*x)*x$. The integers give the registers into which nodes are computed*

We need to identify nodes that correspond to initial values in a straight line program : a node with no descendants is called a *leaf*. A node with no ancestors is called a *root*. Register allocation for a special kind of dag has been considered in [3], [8], [9], [10] : a *tree* is a dag with a single root, in which each node except the root has a unique direct ancestor.

Since we assume that a straight line program is specified as a dag, we will modify our notion of "computation" so that it is defined in terms of dags, rather than in terms of a machine model. Instead of computing a value into a register, we will think of placing a "stone" that identifies a register onto the node representing the value. Computing a straight line program then corresponds to placing and moving "stones" on the appropriate dag.

Game 1. Let D be a dag. Let there be an infinite supply of *stones*, where stones may be thought of as registers. A *move in Game* 1 is one of the following:

1. place a stone on a leaf in D, or
2. pick up a stone from a node in D, or

   if there are stones on every direct descendant of a node x in D; then
3. place a new stone on x, or
4. move a stone to x from one of the direct descendants of x.

The definition above has been adapted from one in [11], [12]. In terms of the machine model, step 1 in the definition of Game 1 corresponds to loading an initial value into a register. Step 2 does not have an instruction counterpart. It can be viewed as declaring that the register in question can now be used to hold a new value. Steps 3 and 4 correspond to the operation reg $j \leftarrow \theta(\text{reg } i_1, \text{reg } i_2)$. In step 3, j is not in the set $\{i_1, i_2\}$. In step 4, j is either i_1 or i_2.

An important difference between the instruction in the machine model and steps 3 and 4 is that steps 3 and 4 refer to all the possibly many direct descendants of a given node x. While the instruction in the machine model is in terms of binary operations, for pedagogical reasons it will be convenient to allow a node in a dag to have a finite but arbitrarily large number of direct descendants. Restricting the number of direct descendants to two does not change the complexity of the register allocation problem.

Example 2.2. Consider the dag in Fig. 1. A sequence of moves in Game 1 using 3 stones is given in Table 1. In order to relate the moves to the machine model, the corresponding machine instructions have also been given. In Example

TABLE 1

Node that stone is placed on	Stone	Move	Instruction
c	2	1	load, reg 2 $\leftarrow c$
x	3	1	load, reg 3 $\leftarrow x$
$t1$	2	4	reg 2 $\leftarrow$ reg 2 * reg 3
b	1	1	load, reg 1 $\leftarrow b$
$t2$	2	4	reg 2 $\leftarrow$ reg 1 + reg 2
$t3$	2	4	reg 2 $\leftarrow$ reg 2 * reg 3
a	3	2, 1	load, reg 3 $\leftarrow a$
$t4$	1	3	reg 1 $\leftarrow$ reg 3 + reg 2

2.2, initially there were no stones on nodes in the dag. Moreover, a stone was placed on each node exactly once. In § 4, a "computation" will be a sequence of moves in Game 1 that place a stone exactly once on each node, starting with no stones on any node, and ending with a stone on each root in the dag. This notion of "computation" arises during code generation while compiling.

It will be shown in § 4 that given a dag and an integer k, determining if there is a "computation" of the dag that uses no more than k stones is "polynomial complete". (The term "polynomial complete" will be defined in § 3.)

Walker and Strong [11], [12] consider register allocation within the context of flowchartable recursions. They permit recomputation of values as necessary. The following examples shows that permitting recomputation of values may reduce the number of stones used in a "computation".

Example 2.3. Consider the expression $b + c - \theta(a, (b + c)/d, e)$, represented by the dag in Fig. 2. Table 2 is a computation of the dag using 3 stones.

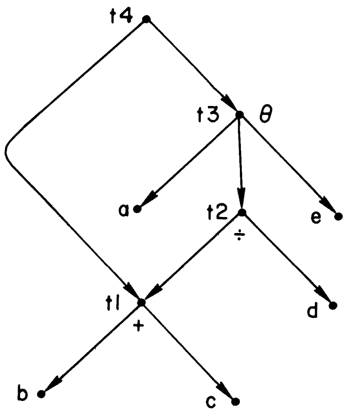


FIG. 2. *A dag for the expression* $b + c - \theta(a, (b + c)/d, e)$

TABLE 2

Node that stone is placed on	Stone	Move	Instruction
b	2	1	load, reg 2 ← b
c	3	1	load, reg 3 ← c
$t1$	2	4	reg 2 ← reg 2 + reg 3
d	3	2, 1	load, reg 3 ← d
$t2$	2	4	reg 2 ← reg 2/reg 3
a	1	1	load, reg 1 ← a
e	3	2, 1	load, reg 3 ← e
$t3$	3	4	reg 3 ← θ(reg 1, reg 2, reg 3)
b	2	2, 1	load, reg 2 ← b
c	1	2, 1	load, reg 1 ← c
$t1$	2	4	reg 2 ← reg 2 + reg 1
$t4$	2	4	reg 2 ← reg 2 − reg 3

Note that $b + c$ is computed twice. If the recomputation of $b + c$ is not permitted, then at least 4 stones will be used.

A "computation" in § 5 may recompute values as necessary, i.e., a node may have a stone placed on it more than once. A result similar to that in § 4 will be shown for the model of computation in § 5.

When no node in a dag is recomputed, an allocation for the dag may be viewed as a function from nodes to registers. It should be clear that not all functions from nodes to registers are allocations. For example, it would not do to assign all nodes to the same register. A somewhat less trivial example is given by Fig. 3. It will be shown in § 6 that the problem of determining if a function from nodes to registers is an allocation is "polynomial complete". Thus, not only is it difficult to find a good allocation, it is difficult to verify that a function is an allocation.

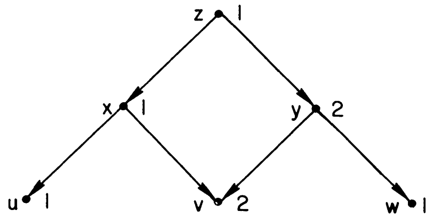


FIG. 3. *Suppose each node has to be computed into the register given by the integer at the node. Then there is no viable order of computation for the values in the dag. If x is computed first into register* 1, *then w cannot be brought into the same register. If y is computed first, the value of v is lost, so x cannot be computed*

In order to appreciate why the last result mentioned is interesting, consider the coloring problem, which may be stated as follows: given an undirected graph G, a *coloring* of G is a function from nodes in G to colors, such that no two nodes joined by an edge in G may have the same color. Given an integer k, determine a coloring of G that uses no more than k colors.

While it is known [7], that the coloring problem is polynomial complete, given a function from nodes to colors, it is easy to check if the function is a coloring of the graph. All that needs to be done is to check that no two nodes joined by an edge are assigned the same color.

3. Polynomial completeness. Let Σ be some alphabet. Let P be the class of languages accepted by polynomial time bounded deterministic Turing machines, and let NP be the class of languages accepted by polynomial time bounded non-deterministic Turing machines. P is clearly a subset of NP. It is not known if $P = NP$.

Just as languages accepted by Turing machines are defined, it is possible to define the "function computed" by a Turing machine. See [6], [7], for instance.

Let Π be the class of functions from Σ^* into Σ^* computed by polynomial time bounded deterministic Turing machines. Let L and M be languages. L is said to be *reducible* to M, if there exists a function $f \in \Pi$, such that $f(x)$ is in M if and only if x is in L. L is called [7] "*(polynomial) complete* if L is in NP, and every language in NP is reducible to L. Either all complete languages are in P, or none of them is. The former alternative holds if and only if $P = NP$."

Demonstrating that all languages in NP are reducible to a given language L is facilitated by a theorem due to Cook [4]. Informally, Cook showed that acceptance of any language in NP is reducible to determining if a formula in the propositional calculus is satisfiable. We will have occasion to deal with this problem in some detail.

DEFINITION. There is a set $\{x_1, x_2, \cdots x_n\}$ of *variables*. If x is a variable, then the symbols x and $\bar{x}$ are called *literals*. x is called a *complement* of $\bar{x}$, and $\bar{x}$ is called a *complement* of x. A *clause* is a subset of the set of literals. A clause $C = \{y_1, y_2, \cdots, y_m\}$ will often be represented by $C = y_1 \vee y_2 \vee \cdots \vee y_m$.

If $C_1, C_2, \cdots, C_m$ are clauses, then C, the *conjunction* of the clauses, will be represented by $C_1 \wedge C_2 \wedge \cdots \wedge C_m$. C will also be referred to as an *m-clause satisfiability problem over n variables.*

C is said to be *satisfiable* if there exists a set S which is a subset of the set of literals, such that:

1. S does not contain a pair of complementary literals,
2. $S \cap C_i \neq \emptyset$ for $i = 1, 2, \cdots, m$.

If the set S exists, then a literal y in S will be said to be *true*, or have *value* 1, and the complement of the literal will be said to be *false*, or have *value* 0. If a literal in a clause is true, the clause will be said to be true.

It is easy to associate a language with the set of satisfiability problems. Following Cook [4], each variable can be represented by some element in Σ, followed by a number in binary notation. Note that there may be an arbitrarily large number of variables. The complement of a variable can be represented, say, by the symbol $-$ followed by the representation of the variable. The other connectives are $\vee$ and $\wedge$. When no confusion can occur, the term "satisfiability problem" will be used to refer to the corresponding string, generated as outlined in this paragraph.

THEOREM (Cook). *If a language L is in NP, then L is reducible to the set of satisfiability problems.*

Proof. See [4]. $\square$

Just as satisfiability problems were defined, it is possible to define satisfiability problems in which each clause has exactly three literals.

THEOREM (Cook). *If a language L is in NP, then L is reducible to the set of satisfiability problems with exactly three literals per clause.*

Proof. The proof is immediate from the result for satisfiability problems with at most three literals per clause [4]. $\square$

The approach in the following section will be to show that the problem on hand can be associated with a language L in NP, and that the set of satisfiability problems with exactly three literals per clause is reducible to L.

4. Reduction to register allocation.

DEFINITION 4.1. A *computation of a dag D* is a sequence of moves in Game 1 that starts with no stones on any node, places a stone on every node exactly once, and ends with stones on all the roots of the dag D. A node is said to be *computed* when a stone is placed on the node. A computation of D is said to *use k stones* if during some move in the computation there are k stones on nodes in D, and during every other move there are no more than k stones on nodes in D.

When this notion of computation has to be identified, we will use the term *4-computation* (the 4 refers to § 4).

DEFINITION. Let C be an m clause satisfiability problem over n variables with exactly 3 literals per clause. Then C will be referred to as a $(3, m, n)$ *satisfiability problem.*

Consider a $(3, m, n)$ satisfiability problem C. A solution of C may be found nondeterministically as follows. Consider variable x_1. Select a value for x_1 from the set {true, false}. Alternatively, select one of x_1 and $\bar{x}_1$ to be assigned the value true. Then consider the variable x_2, etc. Once truth values have been assigned to all n variables, consider clause 1 consisting of literals y_{11}, y_{12} and y_{13}. If all of y_{11}, y_{12} and y_{13} are false, then stop. Otherwise consider clause 2, etc.

We can construct a dag D such that a computation of D simulates the above solution of C. D will consist of $n + m$ *stages*, where stages 1 through n assign truth values to the variables, and stages $n + 1$ through $n + m$ test to see if each of the m clauses is satisfiable.

Let us consider stage i, where $1 \leqq i \leqq n$. This stage refers to the variable x_i. The flowchart in Fig. 4 summarizes the actions performed in stage i; the dag in Fig. 5 gives the nodes and edges relevant to the stage. Stage i has a node z_i such that all nonleaf nodes in the stage are ancestors of node z_i. Thus none of the nonleaf nodes in the stage can be computed until node z_i is computed. Stage i also has two nodes x_i and $\bar{x}_i$, which represent the variable x_i and its complement, respectively.

It will turn out that when node z_i is computed, there are exactly $n - i + 1$ stones in hand. Now both x_i and $\bar{x}_i$ require $n - i + 1$ stones to be computed. Moreover, both x_i and $\bar{x}_i$ are direct descendants of the very last node to be computed. Thus, a stone placed on either x_i or $\bar{x}_i$ will remain on the node in question until the very last step. It follows that computing either x_i or $\bar{x}_i$ ties up one stone, leaving $n - i$ stones—too few to compute the other one of the pair, x_i, $\bar{x}_i$. We want to pass exactly $n - i$ stones to the next stage, which is stage $i + 1$. If one of x_i and $\bar{x}_i$ is indeed computed, there is no problem. We have to take care of the case in which neither x_i nor $\bar{x}_i$ is computed.

Consider the nodes u_i and $\bar{u}_i$ in Fig. 5. Both u_i and $\bar{u}_i$ are direct descendants of the initial node. As its name implies, the initial node will be the very first nonleaf node to be computed, since all other nonleaf nodes will be ancestors of the initial node. Thus before stage i is reached, stones will have been placed on nodes u_i and $\bar{u}_i$. As long as nodes u_i and $\bar{u}_i$ have an uncomputed direct ancestor, the stones on u_i and $\bar{u}_i$ must remain on them. Recall from Definition 4.1 that a stone cannot be placed on a node more than once.

If neither x_i nor $\bar{x}_i$ is computed, then the stones on u_i and $\bar{u}_i$ cannot be moved, and we have to place a new stone on w_i, leaving $n - i$ stones in hand.

Suppose x_i has been computed. Then w_i is the only direct ancestor of u_i that has yet to be computed. So we can move the stone from u_i to w_i, and never need to place a stone on u_i again, for all the direct ancestors of u_i would then have been computed. Once again we have $n - i$ stones in hand. The case when $\bar{x}_i$ is computed instead of x_i is very similar.

Later in this section we will show that when w_i is computed, there are exactly $n - i$ stones that can be picked up. And when w_i is computed, at most one of x_i and $\bar{x}_i$ has been computed.

In case the reader is concerned about the fact that one of x_i and $\bar{x}_i$ has not been computed, at a later stage (when all the clauses have been shown to be true), the uncomputed nodes will be taken care of.

Now let us see how we can verify that the values "assigned" to the variables by stages 1 through n lead to all the clauses being true. Consider clause j given by

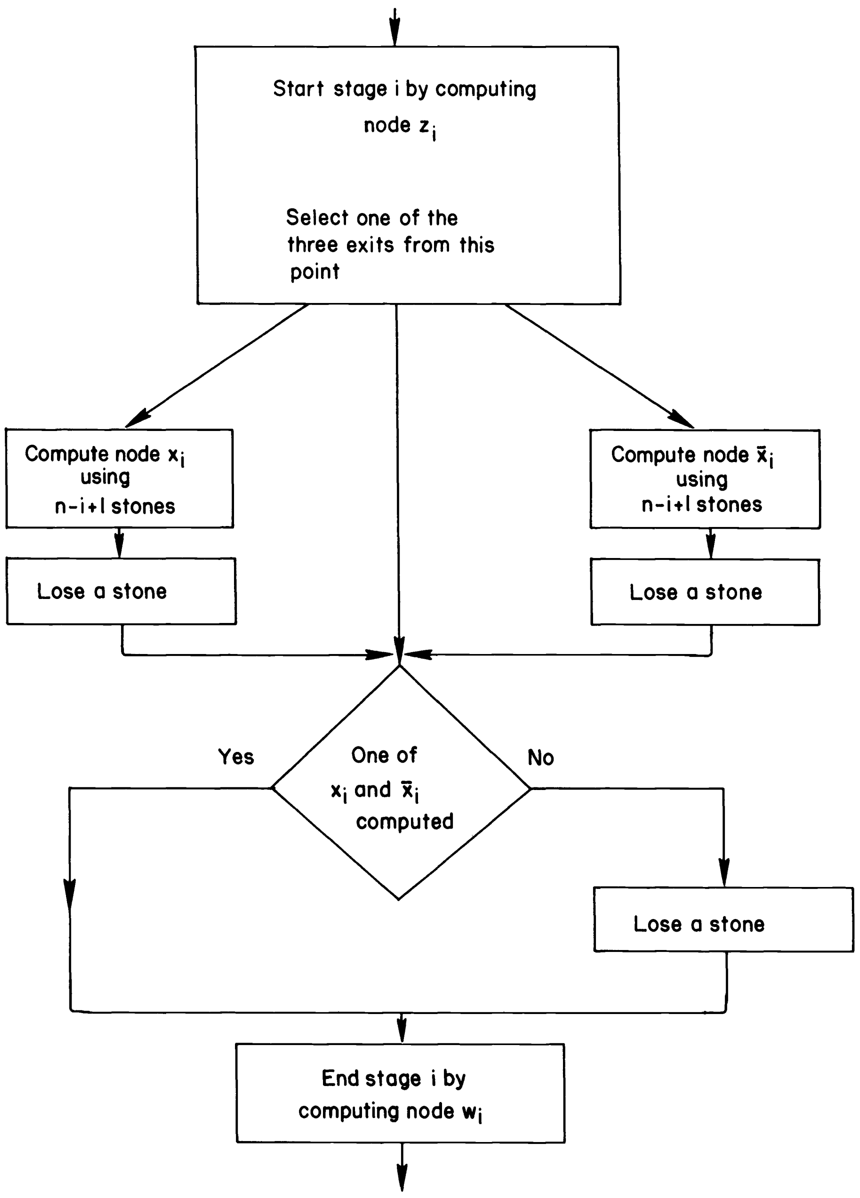

Start stage i by computing node z_i
Select one of the three exits from this point
Compute node x_i using n-i+l stones
Lose a stone
Compute node $\bar{x}_i$ using n-i+l stones
Lose a stone
Yes
One of x_i and $\bar{x}_i$ computed
No
Lose a stone
End stage i by computing node w_i

FIG. 4. *Flowchart depicting the possibilities while computing the part of the dag that assigns truth values to variables. Stage* 1 *is started with n stones in hand*

$y_{j1} \vee y_{j2} \vee y_{j3}$. It is easy to check that the expression $y_{j1} \vee \bar{y}_{j1}y_{j2} \vee \bar{y}_{j1}\bar{y}_{j2}y_{j3}$ has the same truth value as $y_{j1} \vee y_{j2} \vee y_{j3}$. In addition, if $y_{j1} \vee \bar{y}_{j1}y_{j2} \vee \bar{y}_{j1}\bar{y}_{j2}y_{j3}$ is true, then exactly one of the three terms is true. Figure 6 represents the portion of the dag D that checks to see if clause j is true. If the clause is true, then from the above discussion, exactly one of f_{j1}, f_{j2} and f_{j3} will be left with c_j as the only uncomputed direct ancestor. The stone at the appropriate f can then be moved to

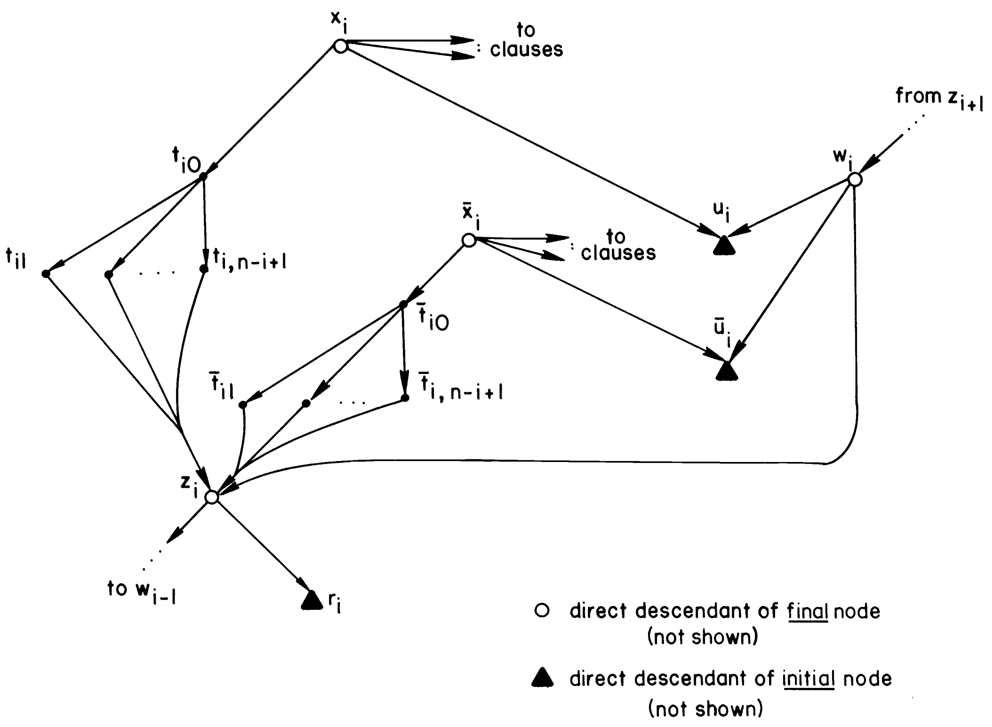


FIG. 5. *The subdag that assigns truth values to x_i and $\bar{x}_i$ by computing at most one of them. If a stone is ever placed on the direct descendant of a final node, it remains there until the last step. The computation starts by placing stones on all direct descendants of the initial node*

c_j. If the clause is false, the computation will be unable to proceed for want of an extra stone.

Finally we need to show how the remaining nodes in stages 1 through n can be computed if all clauses are true. Node d indicated in Fig. 7 is a direct ancestor of node c_m. From the discussion above on clauses, node c_m will be computed without need for an extra stone only when all clauses are true. Node d has $n + 1$ direct descendants $b_0, b_1, \cdots, b_n$, all with stones on them. As soon as c_m is computed a stone can be moved to d from say b_0, and the other n stones can be picked up. These n stones will be enough to compute any nodes in the dag that remain to be computed.

REDUCTION 1. Let C be a $(3, m, n)$ satisfiability problem over the variables $x_1, x_2, \cdots, x_n$, where clause i, for all i, $1 \leqq i \leqq m$, has literals y_{i1}, y_{i2} and y_{i3}. The reduction constructs a dag D and determines an integer k. The value of k depends only on n and m, and will be equal to $5n + 3m + 1$.

The dag D will be divided into a *prologue*, $n + m$ *stages* and an *epilogue*. We first give the nodes and edges within the stages.

STAGE i, $i = 1, 2, \cdots, n$ (see Fig. 5).

Nodes: $r_i, z_i, x_i, \bar{x}_i, u_i, \bar{u}_i, w_i$, and
$t_{ij}, \bar{t}_{ij}, j = 0, 1, \cdots, n - i + 1.$

Edges: (z_i, r_i),
(w_i, z_i),
$(t_{ij}, z_i), (\bar{t}_{ij}, z_i), j = 1, 2, \cdots, n - i + 1$,
$(t_{i0}, t_{ij}), (\bar{t}_{i0}, \bar{t}_{ij}), j = 1, 2, \cdots, n - i + 1$,
$(x_i, t_{i0}), (x_i, u_i), (\bar{x}_i, \bar{t}_{i0}), (\bar{x}_i, \bar{u}_i)$,
$(w_i, u_i), (w_i, \bar{u}_i)$.

STAGE $n + i$, $i = 1, 2, \cdots, m$ (see Fig. 6).
Nodes: $c_i, f_{i1}, f_{i2}, f_{i3}$.
Edges: (c_i, f_{ij}), $j = 1, 2, 3$.
Edges between stages:
$(z_i, w_{i-1}), i = 2, 3, \cdots, n$,
(c_1, w_n),
$(c_i, c_{i-1}), i = 2, 3, \cdots, m$.

Recall that each clause has exactly three literals, and that each literal y_{ij}, $1 \leqq i \leqq m$ and $1 \leqq j \leqq 3$, is either the variable x_l or $\bar{x}_l$ for some l, $1 \leqq l \leqq n$. For the definition of the next set of edges, if $y_{ij} = x_l$, then we use the symbol y_{ij} in the definition of the edges to refer to node x_l in stage l, and the symbol $\bar{y}_{ij}$ to refer to the node $\bar{x}_l$. Otherwise if $y_{ij} = \bar{x}_l$, we use the symbol y_{ij} to refer to the node $\bar{x}_l$, and the symbol $\bar{y}_{ij}$ to refer to x_l.

See Fig. 6 for the following edges:
$(y_{ij}, f_{ij}), (\bar{y}_{ij}, f_{il}), i = 1, 2, \cdots, m, j = 1, 2, 3, l = j + 1, \cdots, 3.$

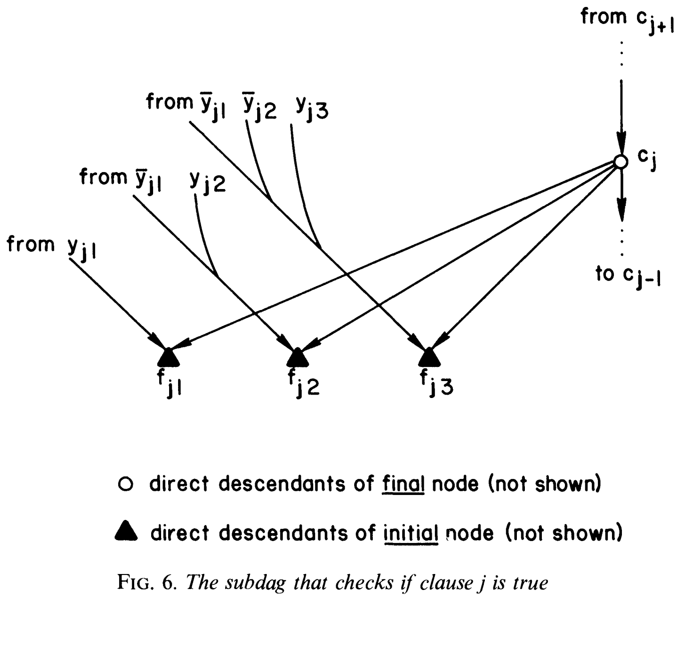

from c_{j+1}
from $\bar{y}_{j1}$ $\bar{y}_{j2}$ y_{j3}
from $\bar{y}_{j1}$ y_{j2}
from y_{j1}
c_j
to c_{j-1}
f_{j1}
f_{j2}
f_{j3}
○ direct descendants of final node (not shown)
▲ direct descendants of initial node (not shown)

FIG. 6. *The subdag that checks if clause j is true*

Once node c_m in stage $n + m$ is computed, we need to ensure that there are enough stones to compute any nodes that remain uncomputed. Once node d with descendants $b_0, b_1, \cdots, b_n$ is computed, n stones can be picked up (see Fig. 7).

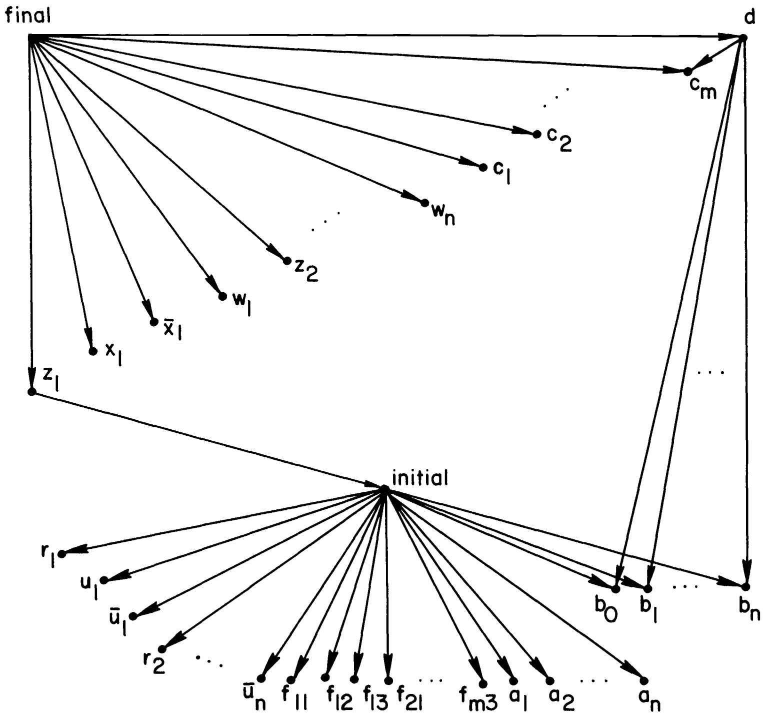


FIG. 7. *Nodes and edges in the prologue and epilogue. All other nodes and edges have not been shown*

EPILOGUE.

Nodes: $d, b_0, b_1, \cdots, b_n$, final.

Edges: (d, c_m),

$(d, b_i), i = 0, 1, \cdots, n$.

The final node will be an ancestor of all the other nodes in the dag. Its direct descendants are shown in Fig. 7.

(final, z_i), (final, x_i), (final, $\bar{x}_i$), (final, w_i), $i = 1, 2, \cdots, n$,

(final, c_i), $i = 1, 2, \cdots, m$,

(final, d).

The prologue has been kept to the last since all leaves will be descendants of the initial node. The purpose of the initial node is to force stones to be placed on all leaves before any nonleaf nodes are computed. With the addition of leaves $a_1, a_2, \cdots, a_n$, the initial node has $5n + 3m + 1$ direct descendants, making it easy to show that at least $5n + 3m + 1$ stones are required to compute D.

PROLOGUE.

Nodes: initial, $a_1, a_2, \cdots, a_n$.

Edges (see Fig. 7):

(initial, r_i), (initial, u_i), (initial, $\bar{u}_i$), $i = 1, 2, \cdots, n$,

(initial, f_{ij}), $i = 1, 2, \cdots, m, j = 1, 2, 3$,

(initial, b_i), $i = 0, 1, \cdots, n$,

(initial, a_i), $i = 1, 2, \cdots, n$,

(z_1, initial).

DEFINITION. A stone is said to be *available* or *free*, if it is not on any node, or it is on a node x, and all the direct ancestors of x have been computed.

LEMMA 4.2. *Let C be a $(3, m, n)$ satisfiability problem, and let D be the dag constructed by Reduction 1 with input C. If C is satisfiable, then D can be computed using $5n + 3m + 1$ stones.*

Proof. From the prologue in Reduction 1, the initial node has $5n + 3m + 1$ direct descendants. Thus at least $5n + 3m + 1$ stones are required to compute D.

Given $5n + 3m + 1$ stones, place them on the direct descendants of the initial node. Such a step is possible since all descendants of the initial node are leaves. From the prologue, n of the leaves, viz., $a_1, a_2, \cdots, a_n$, have the initial node as their only direct ancestor. Thus a stone can be moved to the initial node from a_1. Once the initial node is computed, node z_1 can be computed in the next move, since z_1 has only two direct descendants—the initial node and r_1, both with stones on them.

The moment z_1 is computed, n stones become available. These are the stones that were initially on $a_1, a_2, \cdots, a_n$. Let us now refer to Fig. 5. With n stones available, we can compute one of x_1 and $\bar{x}_1$, as called for by the solution of the satisfiability problem. In fact, as Fig. 5 indicates, we can progress through stages 1 through n computing x_i or $\bar{x}_i$ as appropriate. When w_n is computed, there are no more stones available.

At this point, there are stones on all direct descendants of node c_1. Refer to Fig. 6; since the satisfiability problem is satisfiable, each clause is true, so it is possible to move the stone from one of f_{11}, f_{12} and f_{13} to c_1. We can now progress through the stages corresponding to the clauses, computing c_i just as c_1 was computed.

When c_m is computed, node d (Fig. 7) becomes ready to be computed. Computing node d makes n stones on nodes $b_1, b_2, \cdots, b_n$ available. These n stones can be used to compute any nodes that have yet to be computed.

The lemma follows. □

LEMMA 4.3. *Let D be the dag constructed by Reduction 1 for a $(3, m, n)$ satisfiability problem C. Let D be computed using $5n + 3m + 1$ stones. Then for all j, $1 \leqq j \leqq n$, just after w_j is computed,*

(a) *for all i, $1 \leqq i \leqq j$, at most one of x_i and $\bar{x}_i$ has been computed, and*

(b) *$n - j$ stones are available.*

Proof. From the construction of the dag D, the first moves in any computation will be to compute the initial node, and then node z_1. As in Lemma 4.2, when node z_1 is computed, n stones become available. We will now prove the lemma by induction on j.

Basis: $j = 1$. From Fig. 5, with n stones available it is not possible to compute both x_1 and $\bar{x}_1$. A stone placed on either x_1 or $\bar{x}_1$ must remain on the node until the last step. If $x_1(\bar{x}_1)$ is computed, then the stone at $u_1(\bar{u}_1)$ can be moved to w_1, and given the fact that a stone has been left at $x_1(\bar{x}_1)$, there are $n - 1$ stones available. If neither x_1 nor $\bar{x}_1$ is computed, the stones at u_1 and $\bar{u}_1$ cannot be moved, and a new stone must be placed on w_1, again leaving $n - 1$ stones available when w_1 is computed.

For the inductive step, note that the only node that can be computed just after w_{j-1} is computed is z_j. With $n - j + 1$ stones available, none of the elements of the set $\{x_1, \bar{x}_1, \cdots, x_{j-1}, \bar{x}_{j-1}\}$ that remain to be computed can be touched. Thus we can only compute nodes in stage j with $n - j + 1$ stones available. As in the basis, there will be $n - j$ stones available when w_j is computed. □

LEMMA 4.4. *Let D be the dag constructed by Reduction* 1 *for a* $(3, m, n)$ *satisfiability problem. If D is computed using* $5n + 3m + 1$ *stones, then, at all moves between (and not including) the moves at which w_n and d are computed, there are no stones available.*

Proof. From Lemma 4.3, there are no stones available just after w_n is computed. We will prove by induction on the number of moves that take place after w_n is computed that the only nodes computed between w_n and d are elements of the set $\{c_1, c_2, \cdots, c_m\}$, and that there are no stones available at any move.

Since no stones are available initially, none of the elements of the set $\{x_1, \bar{x}_1, \cdots, x_n, \bar{x}_n\}$ that remain to be computed can be touched. All of them require one or more stones. Thus the only node that can be computed is c_1. Note that in the expression $y_{11} \vee \bar{y}_{11}y_{12} \vee \bar{y}_{11}\bar{y}_{12}y_{13}$, regardless of the values assigned to y_{11}, y_{12} and y_{13}, only one of the terms can be true. Thus, as in Fig. 6, just one of f_{11}, f_{12} and f_{13} can be left with c_1 as the only remaining uncomputed direct ancestor. Thus the stone at the node f_{1j}, can only be moved to c_1. Since c_1 is a direct descendant of the final node, the stone at c_1 does not become available.

The inductive step is very similar to the basis. □

LEMMA 4.5. *Let D be the dag constructed by Reduction* 1 *from a* $(3, m, n)$ *satisfiability problem C. If D can be computed using* $5n + 3m + 1$ *stones, then C is satisfiable.*

Proof. From Lemma 4.3, for all j, $1 \leqq j \leqq n$, at most one of x_j and $\bar{x}_j$ has been computed just after w_n is computed. If x_j has been computed, assign the value true to the variable x_j in C. Otherwise, assign the value false.

Since $c_1, c_2, \cdots, c_m$ are computed without any extra stones, for all c_j, just after w_n is computed, one of f_{j1}, f_{j2} and f_{j3} is left with c_j as its only remaining uncomputed direct ancestor. As in Fig. 6, at least one literal in clause j must be true. C is therefore satisfiable. □

Problem 1. Given a dag D and an integer k, does there exist a 4-computation of D that uses no more than k stones. Note that the integer k is part of the input.

LEMMA 4.6. *Problem* 1 *is in NP.*

Proof. If the integer k is greater than or equal to n, the number of nodes in the dag D, then the dag can be computed by placing a distinct stone on each node. Therefore, suppose $k < n$.

Given the dag D, and the integer k, let T be a nondeterministic Turing machine that generates a sequence of n pairs, $(i_1, x_1), (i_2, x_2), \cdots, (i_n, x_n)$, $1 \leqq i_j \leqq k$, and x_j is a node in D. The integers can be represented in binary notation, and the nodes by the symbol x followed by an integer in binary notation. The length of the sequence will be $O(n \log n)$.

Intuitively, the pair (i, x) may be thought of as specifying that the stone i is placed on node x.

The Turing machine then scans the sequence generated to see if there is a computation of D that corresponds to the sequence. The time taken by T is clearly polynomial in n, and independent of k. □

THEOREM 4.7. *Problem* 1 *is polynomial complete.*

Proof. Given a $(3, m, n)$ satisfiability problem, Reduction 1 constructs a dag with $O(n^2 + m)$ nodes. It is easy to see that dag D can be constructed in polynomial time. The theorem follows from Lemmas 4.2–4.6. □

Reduction 1 constructs a dag in which some of the nodes have large numbers of direct descendants. We will informally indicate how the dag constructed by Reduction 1 can be modified so that each node has at most two descendants.

In the prologue, the only node we have to be concerned about is the initial node. The purpose of the initial node is to force stones to be placed on certain leaves. A tree of the sort in Fig. 8 would serve a similar purpose. Node z_1 would then become the direct ancestor of the root of the tree replacing the initial node.

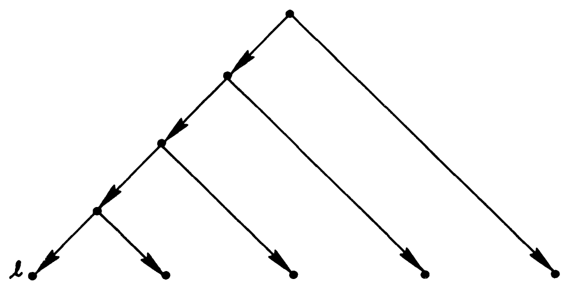


FIG. 8. *Construction used to replace some nodes of high out-degree like the initial node, by a binary subdag. The high degrees of these nodes is used to force stones to be placed on certain leaves*

In stage i, for $1 \leqq i \leqq n$, the reason for providing nodes t_{i0} and $\bar{t}_{i0}$ with $n - i + 1$ direct descendants is to force a computation of nodes t_{i0} and $\bar{t}_{i0}$ to require $n - i + 1$ nodes. A dag of the form of the "pyramid" represented in Fig. 9 has been shown in [5] to require as many stones as it has leaves. Nodes t_{i0} and $\bar{t}_{i0}$ can be replaced by "pyramids" with $n - i + 1$ leaves each.

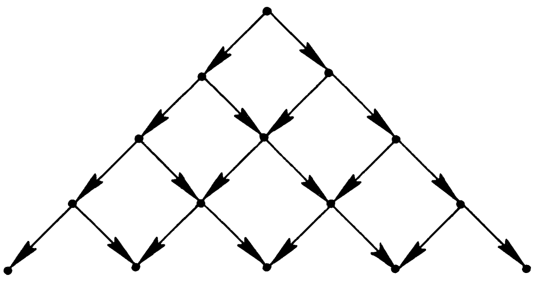


FIG. 9. *A "pyramid" subdag that requires as many stones for computation as it has leaves. Used to replace high out-degree nodes where the high degree forces a high register requirement*

Nodes x_i and $\bar{x}_i$ have a number of direct descendants, forcing stones to be held at certain leaves, until x_i or $\bar{x}_i$ is computed. Both nodes x_i and $\bar{x}_i$ can be replaced by trees like the one in Fig. 8. As long as the leaf marked l in Fig. 8 becomes a direct ancestor of t_{i0} or $\bar{t}_{i0}$, as appropriate, the other leaves can represent the other direct descendants of x_i and $\bar{x}_i$.

Nodes d and the final node would be implemented much like nodes x_i and $\bar{x}_i$.

It remains to specify how the stages representing the clauses would be implemented. The answer may be found in Fig. 10. Since the modifications above add a polynomial number of nodes in the dag constructed by Reduction 1, restricting Problem 1 to dags in which each node has at most two direct descendants can be shown to be polynomial complete.

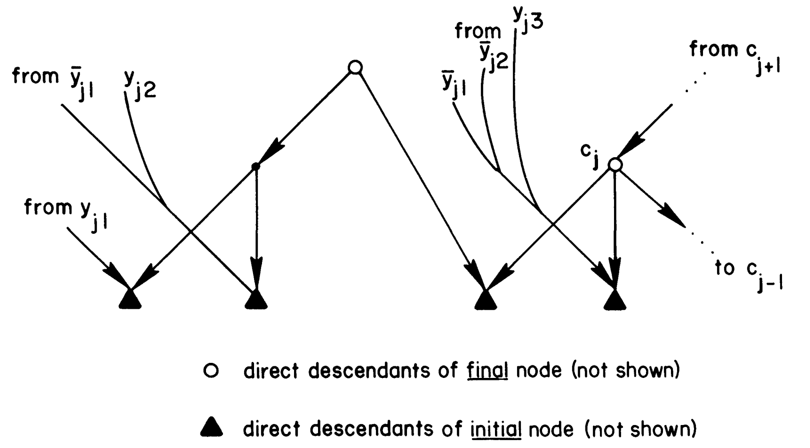

from $\bar{y}_{j1}$ y_{j2}
from y_{j1}
from y_{j3} $\bar{y}_{j2}$ $\bar{y}_{j1}$
from c_{j+1}
c_j
to c_{j-1}
○ direct descendants of final node (not shown)
▲ direct descendants of initial node (not shown)

FIG. 10. *Clause-checking using binary operators*

5. Permitting recomputation. The reduction in the last section relied heavily on the ability to hold stones at designated nodes. If recomputation is permitted, the reductions of the last section have to be suitably modified.

DEFINITION 5.1. A *computation of a dag D* is a sequence of moves in Game 1, that starts with no stones on any node, places a stone one or more times on every node and ends with stones on all the roots of the dag D. A node is *computed* when a stone is placed on the node. A computation of D is said to *use k stones* if during some move there are k stones on nodes in D, and during every other move there are no more than k stones on nodes in D.

When this notion of computation has to be identified, we will use the term *5-computation.*

Example 5.2. Consider the dag in Fig. 11. In order to compute node b, m stones must be placed on the nodes $a_1, a_2, \cdots, a_m$. In order to compute node d, stones must be placed on nodes in the set $C = \{c_1, c_2, \cdots, c_m\}$. If there are exactly m stones that may be used, once stones are placed on all nodes in the set C, *none* of the nodes in C may be recomputed. In order to recompute a node in C, there must be a stone on b. Since it takes m stones to compute b, recomputing a node in b is tantamount to starting anew.

Clearly, treating d as the "initial" node, and elements of C as "leaves", we can ensure that no "leaves" are recomputed.

Example 5.2 showed how the high "stone requirements" of some nodes ensured that these nodes would be computed exactly once. We can extend the idea so that, as in Fig. 12, we can force stones to be held at nodes z_i, x_i, $\bar{x}_i$ and w_i just as in the last reduction.

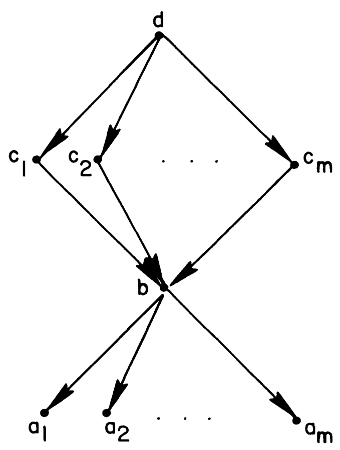


FIG. 11. *If m stones are to be used, recomputing $c_1, c_2, \cdots$, or c_m is tantamount to starting anew*

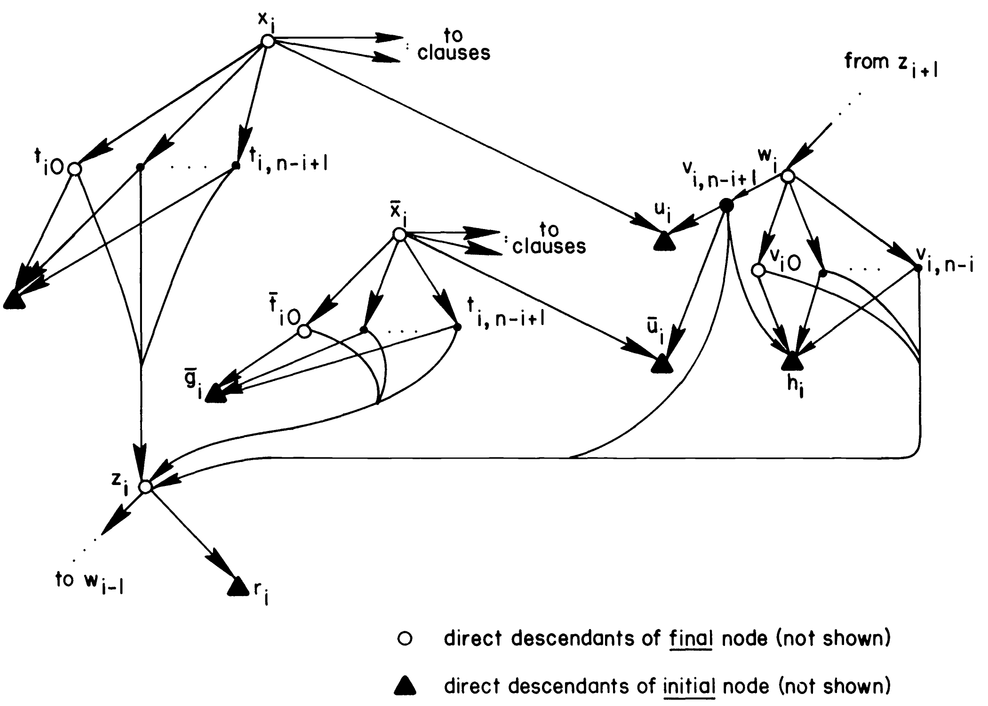


FIG. 12. *A modification of the dag in Fig. 5. A construction (not shown) as in Fig. 11 forces the direct descendants of the initial node to be computed exactly once. The moment the stone at a ▲ node is moved, none of the direct ancestors of the ▲ node may be recomputed*

REDUCTION 2. Let C be a $(3, m, n)$ satisfiability problem over the variables $x_1, x_2, \cdots, x_n$, where clause i, for all i, $1 \leqq i \leqq m$, has literals y_{i1}, y_{i2} and y_{i3}. The reduction constructs a dag D and determines an integer k. The value of k depends only on n and m, and will be equal to $8n + 3m + 1$.

The dag D will be divided into a *prologue*, $n + m$ *stages* and an *epilogue*. We first give the nodes and edges within the stages. The major changes from reduction 1 are in stages $1, \cdots, n$ and the prologue.

STAGE i, $i = 1, 2, \cdots, n$ (see Fig. 12).

Nodes: $r_i, z_i, g_i, \bar{g}_i, x_i, \bar{x}_i, u_i, \bar{u}_i, h_i, w_i$, and
$t_{ij}, \bar{t}_{ij}, v_{ij}, j = 0, 1, \cdots, n - i + 1$.

Edges: (z_i, r_i),
$(t_{ij}, z_i), (\bar{t}_{ij}, z_i), (v_{ij}, z_i), j = 0, 1, \cdots, n - i + 1$,
$(t_{ij}, g_i), (\bar{t}_{ij}, \bar{g}_i), (v_{ij}, h_i), j = 0, 1, \cdots, n - i + 1$,
$(x_i, t_{ij}), (\bar{x}_i, \bar{t}_{ij}), (w_i, v_{ij}), j = 0, 1, \cdots, n - i + 1$,
$(x_i, u_i), (\bar{x}_i, \bar{u}_i)$,
$(v_{i,n-i+1}, u_i), (v_{i,n-i+1}, \bar{u}_i)$.

STAGE $n + i$, $i = 1, 2, \cdots, m$ (see Fig. 6).

Nodes: $c_i, f_{i1}, f_{i2}, f_{i3}$.

Edges: $(c_i, f_{ij}), j = 1, 2, 3$.

Edges between stages:
$(z_i, w_{i-1}), i = 2, 3, \cdots, n$,
(c_1, w_n),
$(c_i, c_{i-1}), i = 2, 3, \cdots, m$.

Recall that each clause has exactly three literals, and that each literal y_{ij}, $1 \leqq i \leqq m$ and $1 \leqq j \leqq 3$, is either the variable x_l or $\bar{x}_l$ for some l, $1 \leqq l \leqq n$. For the definition of the next set of edges, if $y_{ij} = x_l$ then we use the symbol y_{ij} in the definition of the edges to refer to node x_l in stage l, and the symbol $\bar{y}_{ij}$ to refer to the node $\bar{x}_l$. Otherwise if $y_{ij} = \bar{x}_l$, we use the symbol y_{ij} to refer to the node $\bar{x}_l$, and the symbol $\bar{y}_{ij}$ to refer to x_l.

See Fig. 6 for the following edges:
$(y_{ij}, f_{ij}), (\bar{y}_{ij}, f_{il}), i = 1, 2, \cdots, m, j = 1, 2, 3, l = j + 1, \cdots, 3$.

Once node c_m in stage $n + m$ is computed, we need to ensure that there are enough stones to compute any nodes that remain uncomputed. Once node d with descendants $b_0, b_1, \cdots, b_n$ is computed, n stones can be picked up.

EPILOGUE.

Nodes: $d, b_0, b_1, \cdots, b_n$, final.

Edges: (d, c_m),
$(d, b_i), i = 0, 1, \cdots, n$,
The final node will be an ancestor of all the other nodes in the dag.
(final, z_i), (final, x_i), (final, $\bar{x}_i$), (final, w_i), $i = 1, 2, \cdots, n$,
(final, t_{i0}), (final, $\bar{t}_{i0}$), (final, v_{i0}), $i = 1, 2, \cdots, n$,
(final, c_i), $i = 1, 2, \cdots, m$,
(final, d).

The prologue has been kept to the last since all leaves will be descendants of the initial node. The purpose of the initial node is to force stones to be placed on all leaves before any nonleaf nodes are computed. With the addition of leaves $a_1, a_2, \cdots, a_n$ the initial node has $8n + 3m + 1$ direct descendants making it easy to show that at least $8n + 3m + 1$ stones are required to compute D.

PROLOGUE.

Nodes: initial, $a_1, a_2, \cdots, a_n$, pivot, $e_1, e_2, \cdots, e_{8n+3m+1}$.

Edges: $(z_1, \text{initial})$,
$(\text{initial}, g_i), (\text{initial}, \bar{g}_i), (\text{initial}, h_i), i = 1, 2, \cdots, n$,
$(g_i, \text{pivot}), (\bar{g}_i, \text{pivot}), (h_i, \text{pivot}), i = 1, 2, \cdots, n$,
$(\text{initial}, r_i), (\text{initial}, u_i), (\text{initial}, \bar{u}_i), i = 1, 2, \cdots, n$,
$(r_i, \text{pivot}), (u_i, \text{pivot}), (\bar{u}_i, \text{pivot}), i = 1, 2, \cdots, n$,
$(\text{initial}, f_{ij}), i = 1, 2, \cdots, m, j = 1, 2, 3$,
$(f_{ij}, \text{pivot}), i = 1, 2, \cdots, m, j = 1, 2, 3$,
$(\text{initial}, b_i), i = 0, 1, \cdots, n$,
$(b_i, \text{pivot}), i = 0, 1, \cdots, n$,
$(\text{initial}, a_i), i = 1, 2, \cdots, n$,
$(a_i, \text{pivot}), i = 1, 2, \cdots, n$,
$(\text{pivot}, e_i), i = 1, 2, \cdots, 8n + 3m + 1$.

LEMMA 5.3. *Let C be a $(3, m, n)$ satisfiability problem, and let D be the dag constructed by Reduction 2 with input C. If C is satisfiable, then D can be computed using $8n + 3m + 1$ stones.*

Proof. The proof is similar to the proof of Lemma 4.2. □

LEMMA 5.4. *Let D be the dag constructed by Reduction 2 for a $(3, m, n)$ satisfiability problem C. Let D be computed using $8n + 5m + 1$ stones. Then for all j $1 \leqq j \leqq n$, just after w_j is computed,*

(a) *for all i, $1 \leqq i \leqq j$, at most one of x_i and $\bar{x}_i$ has been computed, and*

(b) *$n - j$ stones can be picked up without forcing the computation to start anew.*

Proof. We will show that nodes on which stones were held in Reduction 1 also hold stones in this reduction.

First note that all $8n + 3m + 1$ stones must first be placed on the nodes $e_1, e_2, \cdots, e_{8n+3m+1}$. Their direct ancestor, the pivot node, can then be computed. The pivot node has $8n + 3m + 1$ direct ancestors, all of which must have stones on them for the initial node to be computed. As in Example 5.2, a stone cannot be left on the pivot node. Thus none of the direct ancestors of the pivot node may be recomputed. Which just says that none of the nodes that were leaves in Reduction 1 may be recomputed.

When node z_1 is computed, as in Lemma 4.3, n stones become available. We will now prove the lemma by induction on j.

Basis: $j = 1$. The first node to be computed is z_1. Notice what happens when one of x_1, $\bar{x}_1$ or w_1 is computed. (See Fig. 12).

Suppose the next node to be computed is x_1. Since x_1 has $n + 1$ direct descendants, there must be $n + 1$ stones on $t_{i0}, \cdots, t_{1,n}$. Only n stones were available before z_1 was computed. We want to be able to say that only n stones can be picked up after z_1 is computed. Note that r_1 cannot be recomputed. Thus if the stone at r_1 is moved to z_1, then z_1 cannot be recomputed. Since z_1 has uncomputed direct ancestors, a stone will be held on one of r_1 and z_1, leaving n stones in hand.

The $(n + 1)$st stone for $t_{10}, \cdots, t_{1n}$ must therefore have come from g_1. But then $t_{10}, \cdots, t_{1n}$ cannot be recomputed. Therefore once a stone is moved to x_1, x_1 cannot be recomputed. Since t_{10} is also a direct descendant of the final node, two stones must remain on t_{10} and x_1 until the final node is computed.

Now note that we can no longer compute $\bar{x}_1$ with the $n - 1$ stones that remain.

In order to compute w_1, move the stone from u_1 to $v_{1,n}$, the stone at h_1 to v_{10} and place the $n - 1$ stones on $v_1, \cdots, v_{1,n-1}$. No other situation is possible (stones cannot be distinguished). Once w_1 is computed, stones must remain on v_{10} and w_1, leaving $n - 1$ stones.

The rest of the proof follows that of Lemma 4.3. □

LEMMA 5.5. *Let D be the dag constructed by Reduction* 2 *from a* $(3, m, n)$ *satisfiability problem C. If D can be computed using* $8n + 3m + 1$ *stones, then C is satisfiable.*

Proof. From Lemma 5.4, when node w_n is computed there are no stones free. Stages $n + 1, \cdots, n + m$, which check if each clause is satisfiable, are therefore just as in Reduction 1. The point here is that the moment a stone is moved from some node f_{ij} to c_i, c_i cannot be recomputed since f_{ij} cannot, and the stone therefore remains on c_i. The lemma follows. □

Problem 2. Given a dag D and an integer k, does there exist a 5-computation of D that uses no more than k stones? Note that the integer k is part of the input.

THEOREM 5.6. *The satisfiability problem with exactly* 3 *literals per clause reduces to Problem* 2 *in polynomial time.*

Proof. The dag D constructed by Reduction 2 from a $(3, m, n)$ satisfiability problem has $O(n^2 + m)$ nodes. It is easy to see that dag D can be constructed in polynomial time. The theorem follows from Lemmas 5.4 and 5.5. □

6. Validating register allocations. In this section we consider a seemingly simpler problem in register allocation. Suppose no value is computed more than once. Then, for any computation, a register can be associated with each node (see Example 2.2). In other words, each computation defines a function from nodes to registers. From Fig. 3, the converse—for each function from nodes to registers, there exists a computation—is not true. We are interested in determining if given a function from nodes to registers there exists a computation that computes nodes into those registers.

DEFINITION. Let $Q = x_1, x_2, \cdots, x_n$ be a sequence of nodes in a dag. Node u is said to *appear before* node v in Q, if for some i, j, $1 \leqq i < j \leqq n$, u is x_i and v is x_j. Node v is said to *appear after* node u in Q. If a node u appears before node v, and v appears before node w in Q, then v is said to *appear between* u and w in Q. The term *occur* may sometimes be substituted for "appear".

DEFINITION. Let $Q = x_1, x_2, \cdots, x_n$ be a sequence of nodes in a dag D. Q is called a *complete* sequence of nodes in D if every node in D appears exactly once in Q.

DEFINITION. Given a dag D, let L be a function from nodes in D into the set of *names*. L will be called an *allocation* for D. The pair (D, L) will often be referred to as a *program dag*, or just *program*.

Suppose a value is in a specified register. It should be retained in the register, at least as long as it is needed. It will be needed until all its direct ancestors have been computed.

DEFINITION. Given a program (D, L), let Q be a sequence of nodes in D. (Q, L) is said to be *consistent* if for all nodes u, v and w in D, if

(i) u is a direct descendant of w, and

(ii) v appears between u and w,

then $L(u) \neq L(v)$.

DEFINITION. Let Q be a complete sequence of nodes in a dag D. Q is called a *realization* of a program (D, L) if (Q, L) is consistent, and for all nodes u and v in D, if u appears before v in D, then v is not a descendant of u.

Completeness of a sequence ensures that an attempt will be made to compute every node. A realization also forces descendants to be computed before their ancestors. Consistency ensures that the value of a node will be retained in the appropriate register, as long as it is needed.

Example 6.1. Consider the dag D in Fig. 1. A computation for this dag is given in Example 2.2. The example implicitly specifies a function L from nodes into stone labels. For convenience, L is given below.

node	c	x	$t1$	b	$t2$	$t3$	a	$t4$
stone	2	3	2	1	2	2	3	1

(a)

The sequence of nodes in (a) is a realization of (D, L), that corresponds to the program in Example 2.2.

$$\begin{array}{cccccccc} c & \multicolumn{4}{c}{\overgroup{x \quad b \quad a \quad t1}} & t2 & t3 & t4 \\ 2 & 3 & 1 & 3 & 2 & 2 & 2 & 1 \end{array}$$

(b)

Consider the sequence Q in (b). Q is complete, since it contains all the nodes in D. Moreover, descendants appear before their ancestors in Q. However, Q is not a realization of D, since (Q, L) is not consistent—the value of node x is needed to compute $t1$, but placing a into register 3 destroys the value of x before it can be used.

Problem 3. Given a program dag (D, L), does (D, L) have a realization?

REDUCTION 3. Given an m-clause satisfiability problem over n variables $x_1, x_2, \cdots, x_n$, where for all i, $1 \leqq i \leqq m$, clause i has exactly 3 literals, y_{i1}, y_{i2} and y_{i3}, construct a program dag (D, L) as follows:

1. For all k, $1 \leqq k \leqq n$, construct two leaves s_k and $\bar{s}_k$, corresponding to the literals x_k and $\bar{x}_k$. Let $L(s_k) = L(\bar{s}_k) = S_k$.
2. For all i, j, $1 \leqq i \leqq m$, $1 \leqq j \leqq 3$, construct nodes p_{ij}, q_{ij}, r_{ij} and $\bar{r}_{ij}$, as in Fig. 13, and edges (q_{ij}, p_{ij}), (p_{ij}, r_{ij}). Let $L(p_{ij}) = P_{ij}$, $L(q_{ij}) = Q_{ij}$ and $L(r_{ij}) = L(\bar{r}_{ij}) = R_{ij}$. Also construct the edges $(q_{i1}, \bar{r}_{i2})$, $(q_{i2}, \bar{r}_{i3})$ and $(q_{i3}, \bar{r}_{i1})$.

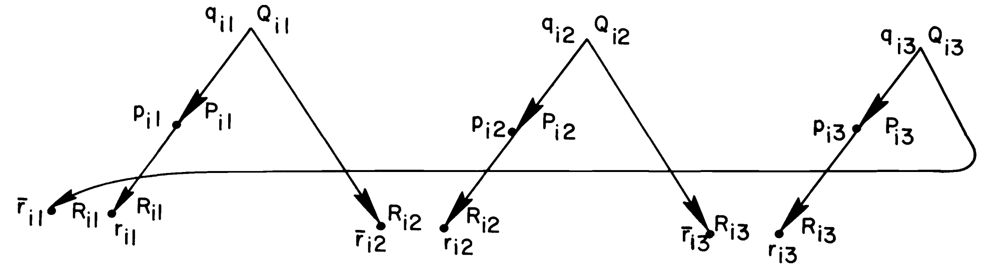


FIG. 13. *The subdag corresponding to a clause*

The subdag created in Fig. 13 corresponds to clause i. The nodes r_{ij} and $\bar{r}_{ij}$ correspond to the literals y_{ij} and $\bar{y}_{ij}$.

3. For all i, $1 \leqq i \leqq m$, clause i consists of the literals y_{i1}, y_{i2} and y_{i3}. For all j, $1 \leqq j \leqq 3$, since y_{ij} is a literal, there exists a k, $1 \leqq k \leqq n$, such that y_{ij} is either x_k or $\bar{x}_k$. If $y_{ij} = x_k$, we use the symbol y_{ij} to refer to node s_k, and the symbol $\bar{y}_{ij}$ to refer to node $\bar{s}_k$. Otherwise, if $y_{ij} = \bar{x}_k$, we use the symbol y_{ij} to refer to node $\bar{s}_k$ and $\bar{y}_{ij}$ to refer to node s_k.

For all i, j, $1 \leqq i \leqq m$, $1 \leqq j \leqq 3$, construct the edges (r_{ij}, y_{ij}) and $(\bar{r}_{ij}, \bar{y}_{ij})$.

LEMMA 6.2. *Let (D, L) be the program constructed by Reduction 3 for a $(3, m, n)$ satisfiability problem. If the conjunction of clauses is satisfiable, then (D, L) has a realization.*

Proof. We will construct a realization for (D, L).

1. Initially the sequence Q is empty. As a convention, nodes may be added to Q on the right only. Do step 2 as follows for $k = 1, 2, \cdots, n$.

2. If the literal x_k is true for the conjunction of clauses to be satisfiable, then add s_k, all direct ancestors of s_k, and $\bar{s}_k$ to Q. Otherwise, add $\bar{s}_k$, all direct ancestors of $\bar{s}_k$, and s_k to Q.

Note that all direct ancestors of s_k and $\bar{s}_k$ are elements of the set $\{r_{ij}, \bar{r}_{ij} | 1 \leqq i \leqq m, 1 \leqq j \leqq 3\}$. Each element of this set has only one direct descendant. Moreover, by construction, for all i, j, $1 \leqq i \leqq m$, $1 \leqq j \leqq 3$, both r_{ij} and $\bar{r}_{ij}$ cannot both be direct ancestors of the same node. Therefore only one of r_{ij} and $\bar{r}_{ij}$ has been added to Q. Since $L(r_{ij})$ is equal only to $L(\bar{r}_{ij})$, (Q, L) must so far be consistent.

3. For all i, j, $1 \leqq i \leqq m$, $1 \leqq j \leqq 3$, if r_{ij} has been added to Q, then add p_{ij} and $\bar{r}_{ij}$ to Q. Note that p_{ij} is the unique direct ancestor of r_{ij}, and that for all nodes x in D, if $x \neq p_{ij}$, then $L(x) \neq L(p_{ij})$.

4. At this stage, note that for all i, j, $1 \leqq i \leqq m$, $1 \leqq j \leqq 3$, $\bar{r}_{ij}$ has been added to the list, but that r_{ij} may not have. For all i, $1 \leqq i \leqq m$, do step 5 below.

5. Consider clause i, given by $y_{i1} \vee y_{i2} \vee y_{i3}$. Since the conjunction of clauses is satisfiable, clause i must be true. Without loss of generality, let y_{i1} be true. We will show that r_{i1} appears in Q before $\bar{r}_{i1}$.

Since y_{i1} is a literal, for some k, $1 \leqq k \leqq n$, y_{i1} is either x_k or $\bar{x}_k$. If y_{i1} is x_k, then, by construction, r_{i1} is a direct ancestor of s_k. Since y_{i1} is true, x_k must be true, so from step 2, above, s_k and r_{i1} are added to Q.

If, on the other hand, y_{i1} is $\bar{x}_k$, then, by construction, r_{i1} is a direct ancestor of $\bar{s}_k$. Since y_{i1} is true, $\bar{x}_k$ is true, and $\bar{s}_k$ and r_{i1} are added to Q by step 2.

Since both p_{i1} and $\bar{r}_{i2}$ have been added to Q, node q_{i1} can now be added to the sequence Q. Once q_{i1} is added to Q, $L(\bar{r}_{i2})$ (which $\bar{r}_{i2}$ shares with r_{i2}) can be used for r_{i2}, unless, of course, r_{i2} is already in Q. In either case, q_{i2} can now be computed, and similarly q_{i3}. $\square$

LEMMA 6.3. *Let u_1, u_2, v_1 and v_2 be nodes in a dag D such that for $i = 1, 2$, u_i is a direct ancestor of v_i. Let $L(u_1) = L(u_2)$ and $L(v_1) = L(v_2)$. Let Q be a realization of (D, L). Then v_1 appears before v_2 in Q if and only if u_1 appears before u_2* (see Fig. 14).

Proof. Suppose u_1 appears before u_2, but that v_2 appears before v_1. We will show that a contradiction must occur.

Since Q is a realization of (D, L), descendants must appear before their ancestors. Since v_1 is a descendant of u_1, it follows that the nodes appear in the

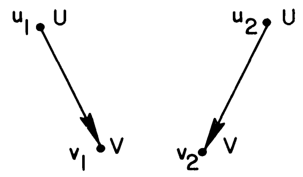


FIG. 14. *u_1 is computed before u_2 if and only if v_1 is computed before v_2*

order v_2, v_1, u_1, u_2. Since u_2 is a direct ancestor of v_2, and $L(v_1) = L(v_2)$, (Q, L) cannot be consistent. Hence Q cannot be a realization of (D, L), a contradiction.

The converse follows similarly. □

LEMMA 6.4. *Let (D, L) be the program constructed by Reduction 3 for a $(3, m, n)$ satisfiability problem. Let Q be a realization of (D, L). Then for all i, $1 \leqq i \leqq m$ there exists a j, $1 \leqq j \leqq 3$, such that r_{ij} appears before $\bar{r}_{ij}$ in Q.*

Proof. Suppose the lemma is false. Then there exists an i, $1 \leqq i \leqq m$, such that for all j, $1 \leqq j \leqq 3$, $\bar{r}_{ij}$ appears before r_{ij} in Q. We will show that a contradiction must occur.

Note (in Fig. 13) that $L(\bar{r}_{ij}) = L(r_{ij})$. Thus, for (Q, L) to be consistent, any direct ancestors of $\bar{r}_{ij}$ must appear before r_{ij} in Q. Note also that r_{ij}, being a descendant of q_{ij}, must appear before q_{ij} in Q. We therefore conclude that:

$$\begin{aligned} q_{i1} &\text{ before } r_{i2}, \\ r_{i2} &\text{ before } q_{i2}, \\ q_{i2} &\text{ before } r_{i3}, \\ r_{i3} &\text{ before } q_{i3}, \\ q_{i3} &\text{ before } r_{i1}, \\ r_{i1} &\text{ before } q_{i1}. \end{aligned}$$

Thus Q cannot be a realization of (D, L), a contradiction. □

LEMMA 6.5. *Let (D, L) be the program constructed by Reduction 3 for an $(3, m, n)$ satisfiability problem. (D, L) has a realization if and only if the conjunction of clauses is satisfiable.*

Proof. The "if" part is provided by Lemma 6.2. So we only need to show that if (D, L) has a realization, then the conjunction of clauses is satisfiable.

Let Q be a realization for (D, L). For all k, $1 \leqq k \leqq n$, if s_k is computed before $\bar{s}_k$, assign the value true to x_k; otherwise assign the value false to x_k.

Suppose this assignment of values is such that the conjunction of clauses is not satisfied. Then we will show that a contradiction must occur.

Since the conjunction of clauses is not satisfied, there must be at least one clause such that all the literals in the clause are false. Let clause i be such a clause.

From Lemma 6.4, there exists a j, $1 \leqq j \leqq 3$, such that r_{ij} appears before $\bar{r}_{ij}$. Without loss of generality, let j be 1. For some k, $1 \leqq k \leqq n$, r_{i1} either has s_k or $\bar{s}_k$ as direct descendant.

Case 1. s_k is a direct descendant of r_{i1}. Since r_{i1} appears before $\bar{r}_{i1}$, from Lemma 6.3, s_k appears before $\bar{s}_k$. Thus x_k is assigned the value true. By construction, the literal y_{i1} must be x_k. Hence y_{i1} must be true, a contradiction.

The other case follows similarly. □

LEMMA 6.6. *Problem 3 is in NP.*

Proof. The proof is straightforward. □

THEOREM 6.7. *Given a program (D, L), the problem of determining if (D, L) has a realization is polynomial complete.*

Proof. The construction of Reduction 3 can clearly be performed in polynomial time. The theorem follows from Lemmas 6.2–6.6. □

7. Practical significance. Fortunately, the dags used in the reductions in this paper tend not to occur in practice. In most programs, straight line sections tend to be fairly small. The practical significance of the results that have been presented is twofold: (i) since the dags that occur in practice tend to be simple, it would be worthwhile to study register allocation for restricted classes of dags; (ii) if the dags are small enough, then efficient enumerative techniques might be worth considering.

Acknowledgments. Interest, suggestions and criticism by John Bruno, Neil Jones, Ray Strong and Jeff Ullman are greatly appreciated.

REFERENCES

[1] A. V. AHO AND J. D. ULLMAN, *Optimization of straight line programs*, this Journal, 1 (1972), pp. 1–19.

[2] J. W. BACKUS, ET AL., *The Fortran automatic coding system*, Proc. Western Joint Computer Conf., vol. 11, 1957, pp. 188–198; also in Programming Systems and Languages, S. Rosen, ed., McGraw-Hill, New York, 1967, pp. 29–47.

[3] J. C. BEATTY, *An axiomatic approach to code optimization for expressions*, J. Assoc. Comput. Mach., 19 (1972), pp. 615–640.

[4] S. A. COOK, *The complexity of theorem-proving procedures*, 3rd Ann. ACM Symp. on Theory of Computing, Shaker Heights, Ohio, May 1971, pp. 151–158.

[5] ———, *An observation on time-storage tradeoff*, 5th Ann. ACM Symp. on Theory of Computing, Austin, Texas, May 1973, pp. 29–33.

[6] J. E. HOPCROFT AND J. D. ULLMAN, *Formal Languages and their Relation to Automata*, Addison-Wesley, Reading, Mass., 1969.

[7] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.

[8] I. NAKATA, *On compiling algorithms for arithmetic expressions*, Comm. ACM, 10 (1967), pp. 492–494.

[9] R. SETHI AND J. D. ULLMAN, *The generation of optimal code for arithmetic expressions*, J. Assoc. Comput. Mach., 17 (1970), pp. 715–728.

[10] P. F. STOCKHAUSEN, *Adapting optimal code generation for arithmetic expressions to the instruction sets available on present-day computers*, Comm. ACM, 16 (1973), pp. 353–354.

[11] S. A. WALKER, *Some graph games related to the efficient calculation of expressions*, IBM Res. Rep. RC 3628, Yorktown Heights, N.Y., Nov. 1971.

[12] S. A. WALKER AND H. R. STRONG, *Characterization of flowchartable recursions*, short version, 4th Ann. ACM Symp. on Theory of Computing, Denver, Colo., May 1972, pp. 18–34.

SIAM J. Comput.
Vol. 4, No. 3, September 1975

WORST-CASE ANALYSIS OF A PLACEMENT ALGORITHM RELATED TO STORAGE ALLOCATION*

ASHOK K. CHANDRA and C. K. WONG†

Abstract. In this paper, a discrete minimization problem arising from storage allocation considerations is studied. Owing to the complexity of finding an optimum solution, a heuristic is proposed and its performance is analyzed. The worst-case ratio of the cost by this algorithm to that by the optimum algorithm is shown to lie between 1.03 and 1.04, implying that this algorithm produces a solution within 4 per cent of the optimum. A generalization of this problem to a class of cost functions is also considered. The worst-case ratios for these functions tend, in the limit, to that of the cost function studied by Graham in his classical paper [1].

Key words. approximate solution, discrete minimization, storage allocation on disk packs, arm contention

1. Introduction. In a time-sharing environment, different programs can simultaneously attempt to access different data sets on the same disk pack, causing arm contention. In Appendix A, an abstract model is proposed which leads to the following discrete minimization problem.

Given a sequence of positive real numbers $x_1 \geqq x_2 \geqq \cdots \geqq x_n$, partition them into m parts, $m \geqq 2$. The parts will henceforth be called *rows*. Let q_j be the sum of all numbers in row j. Find a scheme for assigning the x_i's to rows so as to minimize the cost function

$$C^{(m)} = \sum_{j=1}^{m} q_j^2. \tag{1}$$

Variations of this problem also arise in other situations. For example, Cody and Coffman [2] studied the problem of placing a set of records on the sectors of a drum to minimize the average latency time and were confronted with essentially the same problem. And Garey, Hwang and Johnson [3] in their study of packing circuits on cards attacked another variation of this problem by means of dynamic programming.

Graham in his classical paper [1] studied the same partitioning problem but with a different cost function:

$$C_{\infty}^{(m)} = \max(q_1, \cdots, q_m). \tag{2}$$

He proposed a simple algorithm S and discovered that the ratio of the cost of this algorithm to that of the optimal algorithm is bounded by $4/3 - 1/(3m)$.

Like Graham's problem, our problem, too, does not seem to have any easily computable optimal solution—both problems are NP-complete[1] in the sense of Karp [4]. It is reasonable, therefore, to analyze the performance of nonoptimal heuristic algorithms. A natural candidate is the algorithm S (§ 2). For our cost

* Received by the editors April 26, 1974, and in revised form August 20, 1974.

† Computer Sciences Department, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.

[1] This is sometimes called "P-complete" in the literature. We are using, here, the terminology suggested by Knuth [5]. The proof of NP-completeness involves a straightforward reduction from Karp's PARTITION problem, and is omitted.

function (1), it has the significance that it goes through the sequence $x_1, x_2, \cdots, x_n$ assigning each x_i to a row (without backing up) so as to minimize the increase in cost at each step.

As it turns out, the algorithm S performs even better for the cost function (1) than for (2). In fact, we show in § 3 that S never costs over about 4 % more than the optimal algorithm. And there exist examples where S costs about 3 % more than optimal.

In § 4 we analyze the case where there are just two rows.

Section 5 extends these results to the more general class of cost functions of the form

$$C_\alpha^{(m)} = \left(\sum_{i=1}^{m} q_i^\alpha\right)^{1/\alpha}, \tag{3}$$

where α is a real number, $\alpha > 1$. For each such cost function we consider the least upper bound of the ratio of the cost of algorithm S to that of an optimal algorithm, and find that in the limit as $\alpha \to \infty$, the least upper bound is the same as that for the cost function (2).

2. Description of the algorithm *S*. In this section we describe the algorithm S and present Graham's result [1].

Given a sequence of positive real numbers $\mathbf{x} = x_1, x_2, \cdots, x_n$, where $x_1 \geqq x_2 \geqq \cdots \geqq x_n$, to partition them into m rows, the algorithm S is the following:

(i) *Initialization.* Set $p_1, p_2, \cdots, p_m$ to 0, $i \leftarrow 1$.

(ii) *Placement step.* If $i > n$, the partitioning is complete. Otherwise, let j be the index of a row for which p_j is the minimum of all p's. Put x_i in row j, and set $p_j \leftarrow p_j + x_i$.

(iii) *Repeat.* $i \leftarrow i + 1$, go to step (ii).

Note that the algorithm S is nondeterministic since in step (ii) there may be several values of j for which p_j is minimal. For our cost functions the choice of j does not matter, and we will, unless implied otherwise, resolve ties by choosing the smallest value for j.

Let P be any partitioning of $\mathbf{x}$ into m rows, and let $C_\infty^{(m)}(\mathbf{x}, S)$ and $C_\infty^{(m)}(\mathbf{x}, P)$ be the costs (2) of S and P. Graham showed that

$$\frac{C_\infty^{(m)}(\mathbf{x}, S)}{C_\infty^{(m)}(\mathbf{x}, P)} \leqq \frac{4}{3} - \frac{1}{3m}. \tag{4}$$

The bound can, in fact, be achieved if $n = 2m + 1$, and

$$\begin{aligned} x_{2i-1} = x_{2i} &= 2m - i, \qquad i = 1, \cdots, m, \\ x_{2m+1} &= m. \end{aligned} \tag{5}$$

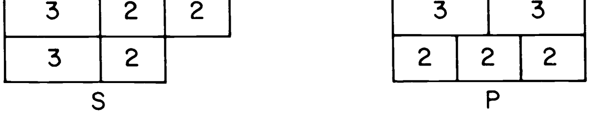


FIG. 1

For example, when $m = 2$, the optimal partition P is shown in Fig. 1, and

$$\frac{C_\infty^{(2)}(\mathbf{x}, S)}{C_\infty^{(2)}(\mathbf{x}, P)} = \frac{7}{6}.$$

3. The cost function $C^{(m)}$. In this section, we consider the cost function (1) and obtain lower and upper bounds for the least upper bound of the ratio of the cost of S to that of an optimal partition.

DEFINITION. Given any sequence $\mathbf{x}$ of numbers, and partition P of $\mathbf{x}$ into m rows, the sum of numbers in row i, $1 \leqq i \leqq m$, is denoted by

$$q_i(\mathbf{x}, P) \tag{6}$$

and the cost is

$$C^{(m)}(\mathbf{x}, P) = \sum_{i=1}^{m} (q_i(\mathbf{x}, P))^2. \tag{7}$$

DEFINITION. Let P be any partition of $\mathbf{x}$ into m rows; let

$$\tau^{(m)}(\mathbf{x}, P) = \frac{C^{(m)}(\mathbf{x}, S)}{C^{(m)}(\mathbf{x}, P)}, \tag{8a}$$

$$\tau^{(m)} = \operatorname*{l.u.b.}_{\mathbf{x},P} \tau^{(m)}(\mathbf{x}, P). \tag{8b}$$

The main result of this section can be stated as follows.

THEOREM 1. *For all* $m > 1$,

$$\tau^{(m)} \geqq \begin{cases} \dfrac{37}{36} & \textit{for } m \textit{ even}, \\[2ex] \dfrac{83}{81} & \textit{for } m = 3, \\[2ex] \dfrac{37}{36} - \dfrac{1}{36m}, & \textit{for } m \textit{ odd and } m \geqq 5, \end{cases} \tag{9}$$

and

$$\tau^{(m)} \leqq \frac{25}{24}. \tag{10}$$

In particular, for large m, $\tau^{(m)}$ lies between about 1.03 and 1.04. The upper bound implies that the algorithm S is at most 4% off optimum.

In order to obtain the lower bound, we have only to modify the examples given in § 2 for $m = 2$ and 3. Thus, for m even, we take $m/2$ copies of the sequence 3, 3, 2, 2, 2. The resulting ratio will be 37/36. For $m = 3$, we just take the sequence 5, 5, 4, 4, 3, 3, 3, whose ratio will be 83/81. For other odd m, $m \geqq 5$, take one element of size 6 and $(m - 1)/2$ copies of 3, 3, 2, 2, 2, with resulting ratio $37/36 - 1/(36m)$.

The lower bound can be tightened slightly by considering more elaborate examples and by using the worst case for $m = 2$, as will be discussed in § 4. For example, the lower bound for even values of m is improved from $(37/36) = 1.0278$ to 1.0285. The present examples, however, have the virtue of simplicity.

The following trivial lemma is useful later.

LEMMA 1. *Given nonnegative numbers* $q_1, \cdots, q_m, q'_i, q'_j$ *such that* $q_i + q_j = q'_i + q'_j$ *and* $|q_i - q_j| \geqq |q'_i - q'_j|$, *we have*

$$q_1^2 + \cdots + q_i^2 + \cdots + q_j^2 + \cdots + q_m^2 \geqq q_1^2 + \cdots + (q'_i)^2 + \cdots + (q'_j)^2 + \cdots + q_m^2.$$

If no confusion is likely, we will not distinguish a partitioning algorithm from the resulting partition.

We now consider the following auxiliary double minimization problem M:

Given an integer $l \geqq 0$, a sequence $\mathbf{x} = x_1, \cdots, x_l$ of positive numbers and a nonnegative number c, let P be any partition of $\mathbf{x}$ into m rows yielding row sums $r_1, \cdots, r_m$. Let $\mathbf{b} = b_1, \cdots, b_m$ be any sequence of nonnegative numbers such that $\sum_{i=1}^m b_i = c$. Define the cost as

$$C(\mathbf{x}, P, c, \mathbf{b}) = \sum_{i=1}^{m} (r_i + b_i)^2. \tag{11}$$

(12) M': Let $C(\mathbf{x}, P, c)$ be the minimal $C(\mathbf{x}, P, c, \mathbf{b})$ for all such $\mathbf{b}$.

(13) M'': Let $C^{(m)}(\mathbf{x}, c)$ be the minimal $C(\mathbf{x}, P, c)$ for all partitions P of $\mathbf{x}$ into m rows. In the sequel the superscript (m) will be omitted.

The word "minimal" above is in the sense of "greatest lower bound".

The minimization problem M is to find the value $C(\mathbf{x}, c)$ for a given $\mathbf{x}$ and c.

In the minimization M' we can imagine that c corresponds to some plastic sheet of area c, which can be continuously stretched, shaped or broken up, and put in the right of the solid blocks of size r_i in the rows (see Fig. 2).

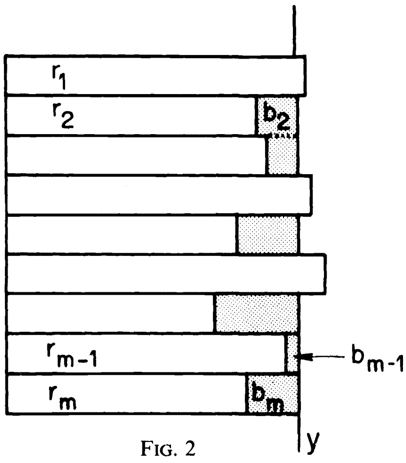


FIG. 2

From Lemma 1 it is easy to prove the following.

LEMMA 2. *Given* $\mathbf{x}$, *a partition* P *of* $\mathbf{x}$ *into* m *rows and a positive*[2] *number* c, *there exists a sequence* $\mathbf{b}^* = b_1^*, \cdots, b_m^*$ *for* $\mathbf{b}$, *satisfying the constraints in* M, *that minimizes* $C(\mathbf{x}, P, c, \mathbf{b})$ *and has the following properties*:

(i) $r_i + b_i^* = r_j + b_j^*$ *for all* $b_i^*, b_j^* > 0$. *Let this common value be* y.

(ii) *If* $r_i < y$, *then* $b_i^* > 0$.

[2] When $c = 0$ the minimization of $C(\mathbf{x}, P, c, \mathbf{b})$ in M' is trivial.

The value y will also be referred to as the *boundary value of P* (see Fig. 2). Note that conditions (i), (ii) define $\mathbf{b}^*$ uniquely. We extend this definition to $c = 0$, then $\mathbf{b}^*$ is the sequence of m zeros.

LEMMA 3. *Given* $\mathbf{x}$, P, c *where* $c \geqq 0$, *and let* $i, j \leqq m$. *Suppose the numbers in row i (in partition P of* $\mathbf{x}$*) are further partitioned into two (possibly empty) parts whose sums total u and v, and let w, x be similar sums for row j and suppose* $u \geqq w$ *and* $v \geqq x$. *Then let P′ be the partition obtained from P by transferring all numbers corresponding to v from row i to row j, and all numbers corresponding to x from row j to row i. Then* $C(\mathbf{x}, P, c) \geqq C(\mathbf{x}, P', c)$.

Proof. Let $\mathbf{b}^*$ be the sequence that minimizes $C(\mathbf{x}, P, c, \mathbf{b})$ as in Lemma 2. We have the following cases (see Fig. 3):

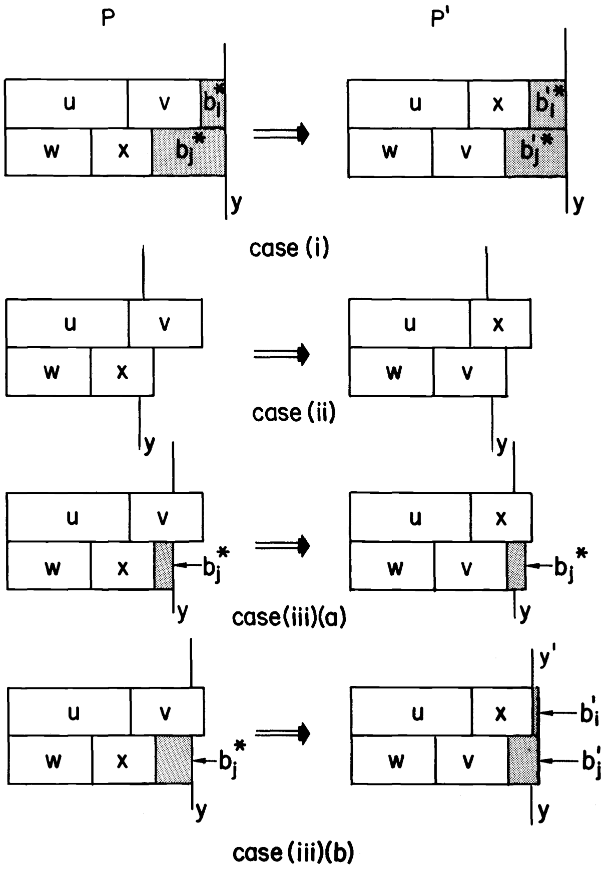


FIG. 3

(i) $b_i^* > 0, b_j^* > 0$. Clearly the boundary values of P, P' are the same and $C(\mathbf{x}, P, c) = C(\mathbf{x}, P', c)$.

(ii) $b_i^* = 0, b_j^* = 0$. Again the boundary values are the same, and as $|(u + v) - (w + x)| \geqq |(u + x) - (w + v)|$, by Lemma 1, $C(\mathbf{x}, P, c) \geqq C(\mathbf{x}, P', c)$.

(iii) $b_i^* = 0, b_j^* \leqq (u + v) - (w + x)$. Without loss of generality, we assume that $u + x \geqq w + v$. There are two subcases:

(a) $b_j^* \leqq (u + x) - (w + v)$. Again by Lemma 1, $C(\mathbf{x}, P, c, \mathbf{b}^*) \geqq C(\mathbf{x}, P', c, \mathbf{b}^*)$ and hence $C(\mathbf{x}, P, c) \geqq C(\mathbf{x}, P', c)$.

(b) $b_j^* > (u + x) - (w + v)$. Let $\mathbf{b}'$ be the same as $\mathbf{b}^*$ except that $b_i' = \frac{1}{2}(w + v + b_j^* - (u + x))$, $b_j' = b_j^* - b_i'$; then by Lemma 1, $C(\mathbf{x}, P, c, \mathbf{b}^*) \geqq C(\mathbf{x}, P', c, \mathbf{b}')$ and hence $C(\mathbf{x}, P, c) \geqq C(\mathbf{x}, P', c)$.

We now return to the question of finding an upper bound for $\tau^{(m)}$. Clearly the algorithm S is optimal when $n \leqq m$. Given a sequence of positive numbers $x_1 \geqq x_2 \geqq \cdots \geqq x_n$, $n \geqq m$, we will apply the algorithm S to part of the sequence in the following way:

(i) Place x_i in row i for $1 \leqq i \leqq m$.

(ii) Place x_{m+k} in row $m - k + 1$, for $k = 1, 2, \cdots$, as long as $k \leqq m$ and $x_{m+k} \geqq \frac{1}{2}x_{m-k+1}$. Otherwise stop.

Suppose the subsequence thus placed is $\mathbf{x}' = x_1, \cdots, x_{m+k_0}$, $k_0 \geqq 0$. We have the following property:

(*) The sum of any two numbers in the subsequence $x_{m-k_0+1}, x_{m-k_0+2}, \cdots, x_{m+k_0}$ is not smaller than any one number in the subsequence.

Let S' denote the partition of $\mathbf{x}'$ as above (see also Fig. 4). Let $c = \sum_{i=m+k_0+1}^{n} x_i$.

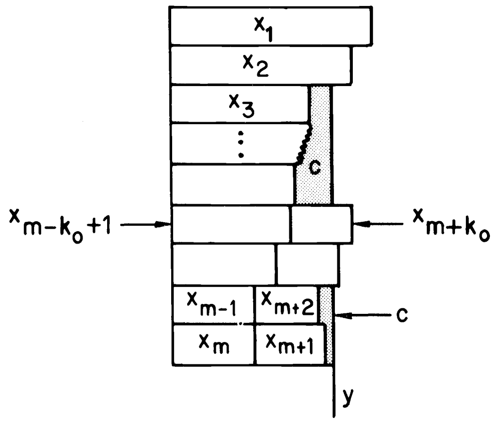


FIG. 4

Then the minimization problem M for $\mathbf{x}', c$ has a solution that agrees with S'. This can be restated as follows.

LEMMA 4.

$$(14) \qquad C(\mathbf{x}', c) = C(\mathbf{x}', S', c).$$

Proof. Suppose P is a partition of $\mathbf{x}'$ such that $C(\mathbf{x}', c) = C(\mathbf{x}', P, c)$. We will show that by a sequence of cost-preserving transformations similar to those in Lemma 3, P can be transformed into S' one row at a time.

First transform P into P_0 such that in P_0 no row has three or more elements: if any row, say row i, has three or more elements, there must be a row, say row j, which has either no element, or just one element x_l where $m - k_0 + 1 \leqq l \leqq m + k_0$. Then by property (*) above and Lemma 3, all but two elements can be transferred from row i to row j. This process must eventually stop to give P_0.

For any i, $0 \leqq i < m$, P_i is now transformed as follows into P_{i+1} such that: (i) $C(\mathbf{x}', P_i, c) \geqq C(\mathbf{x}', P_{i+1}, c)$; (ii) in P_{i+1} the first $i + 1$ rows are identical with those in S'; and (iii) every row in P_{i+1} has at most two elements. Without loss of generality, we can assume that x_{i+1} is in row $i + 1$ of P_i (by permuting the rows).

Case (i). $i \leqq m - k_0 - 1$. If row $i + 1$ contains only x_{i+1}, P_i is the desired partition P_{i+1}; otherwise row $i + 1$ contains another element x_k, and there must exist a row, say row j, $j > i + 1$, with at most one element. Then x_k can be moved to row j (Lemma 3) to give the desired P_{i+1}.

Case (ii). $i > m - k_0 - 1$. All rows numbered $i + 1$ and greater must contain exactly two elements. If row $i + 1$ contains x_{i+1} and x_{2m-i}, we have the desired P_{i+1}; otherwise row $i + 1$ contains x_{i+1} and some x_k, and some row j ($j > i + 1$) contains x_l and x_{2m-i}. But $x_{i+1} \geqq x_l$, $x_k \geqq x_{2m-i}$, so x_k and x_{2m-i} can be interchanged (Lemma 3) to give P_{i+1}. This completes the proof of Lemma 4.

Define

$$K^{(m)} = \operatorname*{l.u.b.}_{\substack{p_1,\cdots,p_m\\ p'_1,\cdots,p'_m}} \left(\sum_{i=1}^{m} (p_i)^2 \Big/ \sum_{i=1}^{m} (p'_i)^2 \right), \tag{15}$$

where $\sum_{i=1}^m p_i = \sum_{i=1}^m p'_i$, each $p_i, p'_i \geqq 0$, and $\max\{p_i\} \leqq \frac{3}{2}\min\{p_i\}$.

LEMMA 5. *For any* $\mathbf{x}$, *let* $\mathbf{x}'$, c *be defined as above. Then*

$$\operatorname*{l.u.b.}_{\mathbf{x}} \frac{C^{(m)}(\mathbf{x}, S)}{C(\mathbf{x}', c)} \leqq K^{(m)}. \tag{16}$$

Proof. Assume (16) is false, and let m be the smallest number of rows for which this is so. Let $\mathbf{x} = x_1, \cdots, x_n$ be a vector such that

$$\frac{C^{(m)}(\mathbf{x}, S)}{C(\mathbf{x}', c)} > K^{(m)}. \tag{17}$$

Clearly $c > 0$. Let k_0, y, S', $\mathbf{b}^*$ also be defined as above. Let $r_1, \cdots, r_m$ be the sums of numbers in the rows for partition S' of $\mathbf{x}'$, and $q_1, \cdots, q_m$ the sums for partition S of $\mathbf{x}$.

Case (i). *For some* j, $r_j > y$. We have a contradiction as follows. S agrees with S' for $x_1, \cdots, x_{m+k_0}$. Further, $\min\{q_i\} \leqq y$, so that S will not place any additional element in row j, and hence $r_j = q_j$. If we now consider the vector $\mathbf{x}_1$ which is the same as $\mathbf{x}$ except that the elements in row j are removed, and we consider $m - 1$ rows, then all rows in S, S' remain the same, but with row j removed, so that

$$\frac{C^{(m-1)}(\mathbf{x}_1, S)}{C^{(m-1)}(\mathbf{x}'_1, c)} = \frac{C^{(m)}(\mathbf{x}, S) - r_j^2}{C^{(m-1)}(\mathbf{x}', c) - r_j^2} > K^{(m)},$$

i.e., we have a smaller counterexample.

Case (ii). *For every* j, $r_j \leqq y$. Now $\max\{q_i\} \geqq y$, and for any row, once the row sum exceeds or equals y, the algorithm S will place no additional x_i in this row. Thus

$$\max\{q_i\} - \min\{q_i\} \leqq x' \leqq x_{m+k_0+1},$$

where x' is the last number placed in the row with final sum $\max\{q_i\}$. By construction,

$$x_{m+k_0+1} \leqq \tfrac{1}{2}\min\{r_i\} \leqq \tfrac{1}{2}\min\{q_i\}.$$

Therefore, $\max\{q_i\} \leqq \frac{3}{2}\min\{q_i\}$ and (17) cannot be true—again a contradiction.

LEMMA 6. $\tau^{(m)} \leqq 25/24$.

Proof. Let $\mathbf{x}'$, c be defined for $\mathbf{x}$ as above. Then

$$\tau^{(m)} = \underset{\mathbf{x},P}{\text{l.u.b.}} \frac{C^{(m)}(\mathbf{x}, S)}{C^{(m)}(\mathbf{x}, P)} \leqq \underset{\mathbf{x}}{\text{l.u.b.}} \frac{C^{(m)}(\mathbf{x}, S)}{C(\mathbf{x}', c)}$$

since $C^{(m)}(\mathbf{x}, P) \geqq C(\mathbf{x}', c)$ for all $\mathbf{x}, P$. Thus $\tau^{(m)} \leqq K^{(m)}$ (by Lemma 5), and it remains only to show that $K^{(m)} \leqq 25/24$. Consider the continuous minimization problem M^*:

Find a piecewise continuous and monotonically nonincreasing function $q(t), 0 \leqq t \leqq m$ such that

(i) $q(0) \leqq \frac{3}{2}q(m)$,

(ii) $\int_0^m q(t)\,dt = s$ (s is some constant),

(iii) $\int_0^m q^2(t)\,dt$ is maximized.

If $q_*(t)$ is a solution, then

$$K^{(m)} \leqq \frac{\int_0^m q_*^2(t)\,dt}{s^2/m}. \tag{18}$$

It is easy to show that $q_*(t)$ must satisfy

$$q_*(t) = \begin{cases} 3x, & 0 \leqq t < k, \\ 2x, & k < t \leqq m, \end{cases}$$

where x, k are parameters to be determined (Fig. 5).

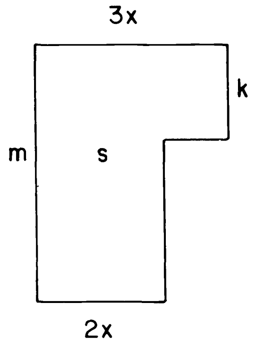


FIG. 5

By condition (ii), $k = (s - 2mx)/x$ and $\int_0^m q_*^2(t)\,dt = 5sx - 6mx^2$, which has maximum value

$$\frac{25}{24}\frac{s^2}{m} \quad \text{when } x = \frac{5s}{12m}.$$

$K^{(m)} \leqq 25/24$ follows by using (18).

The upper bound on $\tau^{(m)}$ can be sharpened by computing $K^{(m)}$ for specific values of m. Thus $\tau^{(3)} \leqq K^{(3)} = 51/49$ and $\tau^{(4)} \leqq K^{(4)} = 26/25$.

4. The case of $m = 2$. Unlike the general case of $\tau^{(m)}$, the value of $\tau^{(2)}$ has been determined.

THEOREM 2. *$\tau^{(2)} \approx 1.0285$, and is achieved by a sequence of the form y, y, z, z, z, where $z = \frac{1}{27}(1 + 5\sqrt{13})y$.*

Proof. In order to maximize the ratio $C^{(2)}(\mathbf{x}, S)/C^{(2)}(\mathbf{x}, P)$ we can assume that the elements of $\mathbf{x}$ sum to 1. We first show that in order to maximize $C^{(2)}(\mathbf{x}, S)/C^{(2)}(\mathbf{x}, P)$ we need only consider those $\mathbf{x}, P$ where (i) $C^{(2)}(\mathbf{x}, P) \leqq C^{(2)}(\mathbf{x}, S)$, and (ii) $x_n \geqq |q_1(\mathbf{x}, S) - q_2(\mathbf{x}, S)|$.

The former is trivial. Also as $C^{(2)}(\mathbf{x}, P) = r^2 + (1 - r)^2$, where $r = \max\{q_1(\mathbf{x}, P), q_2(\mathbf{x}, P)\}$, is a monotonically increasing function of r in $\frac{1}{2} \leqq r \leqq 1$, from (i) we see that

$$\max\{q_1(\mathbf{x}, P), q_2(\mathbf{x}, P)\} \leqq \max\{q_1(\mathbf{x}, S), q_2(\mathbf{x}, S)\}.$$

To prove (ii), suppose we are given $P, \mathbf{x} = x_1, \cdots, x_n$, satisfying (i) such that $x_n < |q_1(\mathbf{x}, S) - q_2(\mathbf{x}, S)|$. Suppose $q_1(\mathbf{x}, S) > q_2(\mathbf{x}, S)$ (the case $q_2(\mathbf{x}, S) > q_1(\mathbf{x}, S)$ is similar). Then x_n must be in row 2 (of partition S). Let $\mathbf{x}' = x_1, \cdots, x_{n-1}$. Clearly $q_1(\mathbf{x}', S) = q_1(\mathbf{x}, S), q_2(\mathbf{x}', S) = q_2(\mathbf{x}, S) - x_n$. Arbitrarily, let x_n be in row 2 of P, and let P' be the partition for $\mathbf{x}'$ that agrees with P over $\mathbf{x}'$. For notational convenience, let $u = q_2(\mathbf{x}, S)$, $v = q_2(\mathbf{x}, P)$. Then we claim that

$$\frac{C^{(2)}(\mathbf{x}', S)}{C^{(2)}(\mathbf{x}', P')} \geqq \frac{C^{(2)}(\mathbf{x}, S)}{C^{(2)}(\mathbf{x}, P)},$$

i.e., that

$$\frac{(1 - u)^2 + (u - x_n)^2}{(1 - v)^2 + (v - x_n)^2} \geqq \frac{(1 - u)^2 + u^2}{(1 - v)^2 + v^2}.$$

If we let

$$t(x) = \frac{C_1(x)}{C_2(x)} = \frac{(1 - u)^2 + (u - x)^2}{(1 - v)^2 + (v - x)^2},$$

then

$$\frac{dt}{dx} = \frac{2(C_1(x)(v - x) - C_2(x)(u - x))}{(C_2(x))^2}.$$

But as $v \geqq u \geqq x$, and $C_1(x) \geqq C_2(x)$, we have $dt/dx \geqq 0$. We can make the sum of all numbers in $\mathbf{x}'$ equal to 1 by scaling up all numbers by the same ratio. Hence, given any $\mathbf{x}, P$ satisfying (i) but not (ii), we can successively remove the smallest elements from $\mathbf{x}$ until (ii) is also satisfied, and we obtain an example for which the ratio of the cost of S to that of P is at least as large as for $\mathbf{x}$.

We now show that we need only consider those $\mathbf{x}, P$ where (iii) $\mathbf{x}$ contains exactly 5 elements.

The algorithm S is optimal for any $\mathbf{x}$ with no more than 4 elements. The cost $C^{(2)}(\mathbf{x}, P)$ is a monotonically increasing function of the difference $|q_1(\mathbf{x}, P) - q_2(\mathbf{x}, P)|$ of the row sums, henceforth called the *gap* in P, and there is an example, viz., $\mathbf{x} = \frac{1}{4}, \frac{1}{4}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}$, for which this difference is $\frac{1}{6}$ for S and zero for optimal P. Thus any $\mathbf{x}, P$ for which the ratio of the costs is larger, must have $|q_1(\mathbf{x}, S) - q_2(\mathbf{x}, S)| > \frac{1}{6}$, and by condition (ii), $x_n > \frac{1}{6}$; but this is possible only when $n \leqq 5$ since $x_n \leqq 1/n$.

Given any $\mathbf{x} = x_1, x_2, \cdots, x_5$, with $x_i \geqq x_{i+1}$ and $x_5 > \frac{1}{6}$, we have $x_1 < \frac{1}{3}$. It follows that the sum of any three of the x_i's is greater than that of the other two. The optimal partition P must have x_1, x_2 in one row and x_3, x_4, x_5 in the other. There are two cases (see Fig. 6) as follows.

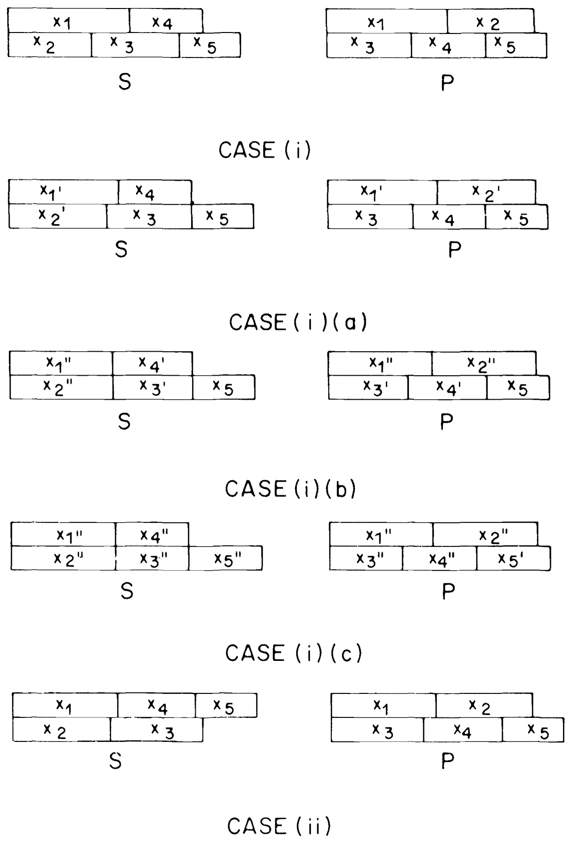


FIG. 6

Case (i). $x_1 + x_4 > x_2 + x_3$. Since $x_5 > \frac{1}{6}$, we have $x_2 + x_3 + x_5 > \frac{1}{2} > x_1 + x_4$. S is therefore as shown in Fig. 6. Let P be the optimal partition (with x_1, x_2 in one row and x_3, x_4, x_5 in the other), and let the ratio $C^{(2)}(\mathbf{x}, S)/C^{(2)}(\mathbf{x}, P)$ be τ.

(a) Let $x_1 - x_2 = g_1$, $(x_1 + x_4) - (x_2 + x_3) = g_2$. Then $g_1 \geqq g_2$ since $x_3 \geqq x_4$. For any ε, $0 < \varepsilon \leqq \frac{1}{2}g_2$, replacing x_1 by $x'_1 = x_1 - \varepsilon$, x_2 by $x'_2 = x_2 + \varepsilon$ will result in a larger gap in S but will have no effect on the gap in P. Thus it has a larger τ.

Increasing ε to $\frac{1}{2}g_2$, we have in the resulting configuration $x'_1 + x_4 = x'_2 + x_3$.

(b) If we now replace x'_1 by $x''_1 = x'_1 - \varepsilon$, x'_2 by $x''_2 = x'_2 + \varepsilon$, x_4 by $x'_4 = x_4 + \varepsilon$, and x_3 by $x'_3 = x_3 - \varepsilon$, where $\varepsilon > 0$ and $x''_1 \geqq x''_2$, τ will remain unchanged. Thus, we increase ε to $(x'_1 - x'_2)/2$, resulting in a sequence $x''_1, x''_2, x'_3, x'_4, x_5$ such that $x''_1 = x''_2, x'_3 = x'_4$.

(c) Finally, replacing x_5 by $x_5' = x_5 + 2\varepsilon$, x_4' by $x_4'' = x_4' - \varepsilon$, and x_3' by $x_3'' = x_3' - \varepsilon$, where $\varepsilon > 0$ and $x_5' \leqq x_3''$, will increase the gap in S by 2ε and leave the gap in P unchanged.

Thus, increasing ε to $(x_3' - x_5)/3$ will result in a sequence $x_1'', x_2'', x_4'', x_3'', x_5'$ such that $x_1'' = x_2'', x_4'' = x_3'' = x_5'$.

Case (ii). $x_1 + x_4 \leqq x_2 + x_3$. S and P are as shown in Fig. 6. Replacing x_3 by $x_3' = x_3 - \varepsilon$ and x_4 by $x_4' = x_4 + \varepsilon$, where $0 < \varepsilon \leqq ((x_2 + x_3) - (x_1 + x_4))/2$, will increase τ. Therefore, we increase ε to $((x_2 + x_3) - (x_1 + x_4))/2$, resulting in a sequence $x_1, x_2, x_3', x_4', x_5$, with $x_1 + x_4' = x_2 + x_3'$. This is similar to the Case (i)(a) above.

In conclusion, the maximum value of τ is achieved by a sequence of the form y, y, z, z, z, and the determination of this value is trivial.

5. Other cost functions. The Theorems 1, 2 stated above for the cost function $C^{(m)}$ also apply, with minor modifications, to the more general class of cost functions $C_\infty^{(m)}$ and $C_\alpha^{(m)}$, $\alpha > 1$:

$$C_\infty^{(m)}(\mathbf{x}, P) = \max_i q_i(\mathbf{x}, P), \tag{19a}$$

$$C_\alpha^{(m)}(\mathbf{x}, P) = \left(\sum_{i=1}^{m} (q_i(\mathbf{x}, P))^\alpha \right)^{1/\alpha}. \tag{19b}$$

For these cost functions, we define $\tau_\infty^{(m)}(\mathbf{x}, P)$, $\tau_\alpha^{(m)}(\mathbf{x}, P)$, $\tau_\infty^{(m)}$, $\tau_\alpha^{(m)}$ as in (8a), (8b)—see Appendix B. Graham [1] studied the case of $C_\infty^{(m)}$ and showed that $\tau_\infty^{(m)} = 4/3 - 1/(3m)$. In this section we consider the case of $C_\alpha^{(m)}$. It is clear that when $\alpha = 2$, we have essentially the case of $C^{(m)}$, and

$$\tau_2^{(m)} = (\tau^{(m)})^{1/2}. \tag{20}$$

For the class $C_\alpha^{(m)}$, the optimization problem is nontrivial in that the following is NP-complete (in the sense of Karp [4]) for every pair of integers $m, \alpha > 1$: to decide if, for a given sequence $\mathbf{x}$ of rational numbers and a rational c, there is a partition P such that $C_\alpha^{(m)}(\mathbf{x}, P) \leqq c$. The proof is easy and is omitted.

Below we use $(m \bmod m')$ to mean $m - m'\lfloor m/m' \rfloor$.

THEOREM 1′. *For any integer $m > 1$ and real number $\alpha > 1$,*

$$\tau_\alpha^{(m)} \geqq \max_{m'} \left(\frac{\left\lfloor \frac{m}{m'} \right\rfloor ((4m' - 1)^\alpha + (m' - 1)(3m' - 1)^\alpha) + (m \bmod m')(3m')^\alpha}{m(3m')^\alpha} \right)^{1/\alpha}, \tag{21}$$

$$\tau_\alpha^{(m)} \leqq \frac{3^\alpha - 2^\alpha}{\alpha} \left(\frac{\alpha - 1}{2 \cdot 3^\alpha - 3 \cdot 2^\alpha} \right)^{(\alpha - 1)/\alpha}. \tag{22}$$

When $\alpha = 2$ these simplify to (9), (10) after using (20).

Proof. The lower bound (21) is constructed by taking a sequence $\mathbf{x}$ consisting of $(m \bmod m')$ numbers of magnitude $3m'$, and $\lfloor m/m' \rfloor$ copies of the sequence $2m' - 1, 2m' - 1, 2m' - 2, 2m' - 2, \cdots, m', m', m'$.

The upper bound is obtained as in the proof of Theorem 2. We can show that

$$\tau_\alpha^{(m)} \leqq \max_q \left(\frac{\int_0^m (q(t))^\alpha \, dt}{m(s/m)^\alpha} \right)^{1/\alpha}, \tag{23}$$

where $q(t)$ is a piecewise continuous monotonically nonincreasing function in $0 \leqq t \leqq m$ such that $q(0) \leqq \frac{3}{2}q(m)$ and $\int_0^m q(t)\,dt = s$. The right-hand side of (23) gives the upper bound (22).

In the limit as $\alpha \to \infty$ and m is constant, the lower and upper bounds are $4/3 - 1/(3m)$ and $\frac{3}{2}$, respectively (for the lower bound the maximum is obtained for $m' = m$).

THEOREM 2′. *$\tau_\alpha^{(2)}$ is achieved by a sequence of the form y, y, z, z, z where $y/2 < z < y$.*

In the limit as $\alpha \to \infty$, $(z/y) \to \frac{2}{3}$ and $\tau_\alpha^{(2)} \to \frac{7}{6}$.

The proof of Theorem 2′ is almost identical to that of Theorem 2.

Table 1 lists the lower and upper bounds (21), (22) for some values of α, m, and the values of $\tau_\alpha^{(2)}$ obtained from Theorem 2′. The $\tau_\alpha^{(2)}$'s are given in percentages, obtained by $100(\tau_\alpha^{(2)} - 1)$; so are the bounds.

TABLE 1

α	Upper bound $\tau_\alpha^{(m)}(\%)$	Lower bound $\tau_\alpha^{(m)}(\%)$					$\tau_\alpha^{(2)}(\%)$
		$m = 2$	$m = 3$	$m = 5$	$m = 10$	$m = 25$	
1.0	0	0	0	0	0	0	0
1.5	1.03	.69	.60	.56	.69	.67	.71
2.0	2.06	1.38	1.23	1.11	1.38	1.32	1.42
2.5	3.10	2.05	1.86	1.65	2.05	1.97	2.11
3.0	4.13	2.70	2.50	2.18	2.70	2.60	2.77
4	6.17	3.95	3.78	3.19	3.95	3.80	4.04
5	8.17	5.09	5.05	4.15	5.09	4.90	5.19
6	10.10	6.12	6.27	5.19	6.12	6.05	6.23
7	11.94	7.05	7.42	6.36	7.05	7.18	7.16
8	13.69	7.86	8.49	7.52	7.86	8.23	7.98
9	15.33	8.59	9.48	8.64	8.79	9.21	8.70
10	16.87	9.22	10.38	9.71	9.71	10.10	9.33
15	23.08	11.45	13.72	14.08	14.08	14.08	11.52
20	27.35	12.70	15.71	16.92	16.92	16.92	12.76
25	30.38	13.48	16.97	18.78	18.78	18.78	13.53
50	37.89	15.06	19.57	22.65	24.15	24.12	15.08
100	42.82	15.86	20.89	24.64	27.04	27.82	15.87
∞	50.00	16.67	22.22	26.67	30.00	32.00	16.67

For any $\mathbf{x}$, P we have $\lim C_\alpha^{(m)}(\mathbf{x}, P) = C_\infty^{(m)}(\mathbf{x}, P)$. In view of this, and the similarity between $\tau_\alpha^{(m)}$ and $\tau_\infty^{(m)}$ as $\alpha \to \infty$, in Theorems 1′, 2′, the following may be anticipated.

THEOREM 3.

$$\lim_{\alpha \to \infty} \tau_\alpha^{(m)} = \tau_\infty^{(m)}. \tag{24}$$

Proof. Since $\tau_\alpha^{(m)}$ is bounded for all α, it must have lower and upper limits $\underline{l}$, $\bar{l}$, respectively:

$$\underline{l} = \varliminf_{\alpha \to \infty} \tau_\alpha^{(m)}, \tag{25}$$

$$\bar{l} = \varlimsup_{\alpha \to \infty} \tau_\alpha^{(m)}. \tag{26}$$

It suffices to show that

$$\underline{l} \geqq \tau_\infty^{(m)} \tag{27}$$

and

$$\bar{l} \leqq \tau_\infty^{(m)}. \tag{28}$$

To show (27), we use the fact, stated in § 2, that there exists an $\mathbf{x}$ and partition P such that $\tau_\infty^{(m)} = \tau_\infty^{(m)}(\mathbf{x}, P)$. For all α,

$$\tau_\alpha^{(m)} \geqq \tau_\alpha^{(m)}(\mathbf{x}, P) = \left(\frac{\sum_{i=1}^{m} (q_i(\mathbf{x}, S))^\alpha}{\sum_{i=1}^{m} (q_i(\mathbf{x}, P))^\alpha} \right)^{1/\alpha} \geqq \left(\frac{\max_i q_i(\mathbf{x}, S)^\alpha}{m \max_i q_i(\mathbf{x}, P)^\alpha} \right)^{1/\alpha}$$

$$= \left(\frac{1}{m}\right)^{1/\alpha} (\tau_\infty^{(m)\alpha})^{1/\alpha} = \left(\frac{1}{m}\right)^{1/\alpha} \tau_\infty^{(m)}.$$

As $\alpha \to \infty$, $(1/m)^{1/\alpha}\tau_\infty^{(m)} \to \tau_\infty^{(m)}$, proving (27).

Suppose (28) is false. By definition of $\bar{l}$, for every $\delta, \alpha_0 > 0$ there exists an $\alpha > \alpha_0$ such that $\tau_\alpha^{(m)} \geqq \bar{l} - \delta$. Choose δ such that $\bar{l} - \delta > \tau_\infty^{(m)} + \delta$. Then for all $\alpha \geqq \alpha_0$, $\tau_\alpha^{(m)} > \tau_\infty^{(m)} + \delta$. Choose a sequence of triples $(\alpha_j, \mathbf{x}_j, P_j)$, $j = 1, 2, \cdots$, such that $\alpha_j \to \infty$ and for all j, $\tau_{\alpha_j}^{(m)}(\mathbf{x}_j, P_j) > \tau_\infty^{(m)} + \delta$. Then

$$\tau_{\alpha_j}^{(m)}(\mathbf{x}_j, P_j) \leqq \left(\frac{m \max_i q_i(\mathbf{x}_j, S)^{\alpha_j}}{\max_i q_i(\mathbf{x}_j, P)^{\alpha_j}} \right)^{1/\alpha_j} = m^{1/\alpha_j}\tau_\infty^{(m)}(\mathbf{x}_j, P_j)$$

$$\leqq m^{1/\alpha_j}\tau_\infty^{(m)} < m^{1/\alpha_j}(\tau_{\alpha_j}^{(m)}(\mathbf{x}_j, P_j) - \delta).$$

But this is a contradiction for some large enough α_j since δ is independent of j. This concludes the proof of the theorem.

6. Conclusions. This paper adds one more example to the rapidly growing body of literature [6]–[14] dealing with near-optimal algorithms. For most cases, the computation to obtain solutions is prohibitively large and heuristics become a necessity. It is relevant, in such cases, to evaluate the performance of these heuristics. The heuristic studied in the present paper performs very well for our particular problem. One would conjecture that this algorithm with minor modifications might perform as well for the following variation of the present problem. Given a set of mk positive numbers, partition them into m parts of k each so that the corresponding cost functions (1), (2) and (3) are minimized. This problem corresponds to the physical situation of fully packed storage and bounds on the performance of the heuristic are still under investigation.

Appendix A. An application. We use the following abstract model: suppose we have m disks of large capacity; n data sets are to be allocated to these m disks. Assume that each data set is associated with a number x_i, $0 < x_i < 1$, which is its access probability. Therefore, we have n independent Bernoulli trials simultaneously in progress. The time axis is discretized, and at each point, the sequence $x_1, \cdots, x_n$ completely characterizes the access pattern of the entire collection of data sets. Simultaneous access to the same disk causes conflict. The problem is to formalize the notion of conflict and to find a placement of the n data sets such that conflict is minimized. Depending on the physical situation, there are many ways to define conflict, and we mention only one very simple way here.

Suppose data sets $D_{i_1}, \cdots, D_{i_k}$ are placed in disk i. Suppose their access probabilities are $x_{i_1}, \cdots, x_{i_k}$. We can imagine that there are k windows; with probability x_{i_j}, the jth window will display the number 1, and with probability $1 - x_{i_j}$, it will display 0. We define conflict in disk i, C_i, as the expected number of pairs of 1's. Therefore

$$C_i = \sum_{j,l} x_{i_j} x_{i_l} = \frac{1}{2}\left[\left(\sum_{j=1}^{k} x_{i_j}\right)^2 - \sum_{j=1}^{k} x_{i_j}^2\right].$$

If we define total conflict $C = \sum_{i=1}^{m} C_i$, then

$$C = \frac{1}{2}\sum_{i=1}^{m}\left(\sum_{j=1}^{k} x_{i_j}\right)^2 - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{k} x_{i_j}^2. \tag{29}$$

For a given sequence $\{x_i\}$, $1 \leqq i \leqq n$, the second term is a constant. Therefore, minimization of (29) reduces to minimization of the first term.

If the capacity of a disk is large relative to n, we can assume that we can place as many data sets as desired on one disk. Then the problem of minimizing total conflict is exactly the problem we mentioned in § 1.

If such an assumption is invalid, we then have to consider the bounded problem as mentioned in § 6.

Appendix B. Notation and definitions.

$\mathbf{x}$	ordered sequence of numbers $x_1, x_2, \cdots, x_n$ such that $x_1 \geqq x_2 \geqq \cdots \geqq x_n > 0$.
$\mathbf{y}$	ordered sequence of numbers (like $\mathbf{x}$).
m	number of rows.
P, Q	partition of an ordered sequence of numbers into m rows.
S	algorithm (§ 2). This symbol also denotes the partition resulting from this algorithm.
$q_i(\mathbf{x}, P)$	sum of numbers in row i of partition P of $\mathbf{x}$ (number of rows m is implicit) (§ 3; (6)).

$$C^{(m)}(\mathbf{x}, P) = \sum_{i=1}^{m} (q_i(\mathbf{x}, P))^2 \quad (\S\, 3; (7)).$$

$$C_\alpha^{(m)}(\mathbf{x}, P) = \left(\sum_{i=1}^{m} (q_i(\mathbf{x}, P))^\alpha\right)^{1/\alpha} \quad (\S\, 5).$$

$C_\infty^{(m)}(\mathbf{x}, P) = \max_i q_i(\mathbf{x}, P)$ (§ 5).

$\tau^{(m)}(\mathbf{x}, P) = C^{(m)}(\mathbf{x}, S)/C^{(m)}(\mathbf{x}, P)$ (§ 3; (8a)).

$\tau_\alpha^{(m)}(\mathbf{x}, P) = C_\alpha^{(m)}(\mathbf{x}, S)/C_\alpha^{(m)}(\mathbf{x}, P)$ (§ 5).

$\tau_\infty^{(m)}(\mathbf{x}, P) = C_\infty^{(m)}(\mathbf{x}, S)/C_\infty^{(m)}(\mathbf{x}, P)$ (§ 5).

$\tau^{(m)} = \operatorname{l.u.b.}_{\mathbf{x},P} \tau^{(m)}(\mathbf{x}, P)$ (§ 3; (8b)).

$\tau_\alpha^{(m)} = \operatorname{l.u.b.}_{\mathbf{x},P} \tau_\alpha(\mathbf{x}, P)$ (§ 5).

$\tau_\infty^{(m)} = \operatorname{l.u.b.}_{\mathbf{x},P} \tau_\infty(\mathbf{x}, P)$ (§ 5).

$C(\mathbf{x}, P, c, \mathbf{b})$ (§ 3; (11)) (m is implicit).
$C(\mathbf{x}, P, c)$ (§ 3; (12)) (m is implicit).
$C^{(m)}(\mathbf{x}, c)$ (§ 3; (13)).
$C(\mathbf{x}, c)$ (§ 3; (13)) (m is implicit).
$K^{(m)}$ (§ 3; (15)).

REFERENCES

[1] R. L. GRAHAM, *Bounds on multiprocessing timing anomalies*, SIAM J. Appl. Math., 17 (1969), pp. 416–429.

[2] R. CODY AND E. G. COFFMAN, *Record allocation for minimizing expected retrieval costs on drum type storage devices*, Tech. Rep. 147, Computer Science Department, Penna. State Univ., College Park, February, 1974.

[3] M. R. GAREY, K. F. HWANG AND D. S. JOHNSON, *Economic aspects of designing standard library for circuit cards*, Tech. Rep., Bell Labs., to appear.

[4] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–103.

[5] D. E. KNUTH, *A terminological proposal*, SIGACT News 1974, 6 (1974), pp. 12–18.

[6] R. L. GRAHAM, *Bounds on multiprocessing anomalies and related packing algorithms*, Proc. Spring Joint Computer Conf., Atlantic City, N.J., 1972, pp. 205–218.

[7] M. R. GAREY, R. L. GRAHAM AND J. D. ULLMAN, *Worst-case analysis of memory allocation algorithms*, Proc. 4th Ann. ACM Symp. on Theory of Computing, Denver, Colorado, 1972, pp. 143–150.

[8] D. S. JOHNSON, *Approximation algorithms for combinatorial problems*, Proc. 5th Ann. ACM Symp. on Theory of Computing, Austin, Texas, 1973, pp. 38–49.

[9] C. K. WONG, C. L. LIU AND J. APTER, *A drum scheduling algorithm*, Lecture Notes in Computer Science, vol. 2, K. Indermark, ed., Springer-Verlag, Berlin, 1973, pp. 267–275.

[10] C. L. LIU, *Optimal scheduling on multi-processor computing systems*, Proc. 13th Ann. IEEE Symp. on Switching and Automata Theory, College Park, Maryland, 1972, pp. 155–160.

[11] J. NIEVERGELT AND C. K. WONG, *On binary search trees*, Information Processing 71, North-Holland, Amsterdam, 1972, pp. 91–98.

[12] C. K. WONG AND D. COPPERSMITH, *A combinatorial problem related to multimodule memory organizations*, J. Assoc. Comput. Mach., 21 (1974), pp. 392–402.

[13] R. M. KARP, A. C. MCKELLAR AND C. K. WONG, *Near-optimal solutions to a 2-dimensional placement problem*, IBM Res. Rep. RC 4740, IBM T. J. Watson Res. Center, Yorktown Heights, N.Y., 1974; also in this Journal, to appear.

[14] P. C. YUE AND C. K. WONG, *Near-optimal heuristics for the two-dimensional storage assignment problem*, IBM Res. Rep. RC 4729, IBM T. J. Watson Res. Center, Yorktown Heights, N.Y., 1974; also in Internat. J. Comput. and Information Science, to appear.

SIAM J. Comput.
Vol. 4, No. 3, September 1975

IMPROVED DIVIDE/SORT/MERGE SORTING NETWORKS*

R. L. (SCOT) DRYSDALE III† AND FRANK H. YOUNG‡

Abstract. This paper develops sorting networks using the divide/sort/merge strategy. These networks require

$$(0.25)N(\log_2 N)^2 - (0.386)N(\log_2 N) + O(N)$$

comparison-interchanges to sort a list of length N. This is an improvement of order $N(\log_2 N)$ over the best networks previously reported. (Using different methods, Van Voorhis [8] has improved upon these results.)

Key words. networks, sorting networks, sorting algorithms, divide/sort/merge strategy

1. Introduction. In this paper we develop an improved divide/sort/merge algorithm for nonadaptive sorting. Equivalently, we develop improved sorting networks using the divide/sort/merge strategy. Our algorithm is an extension of procedures due to Batcher [1], Green [4] and Van Voorhis [6], [7]. We first discuss some necessary preliminaries.

We will adopt the convention that a sorting algorithm operates on the contents of a linear array (or list). The operation consists of a succession of "comparison-interchanges" at the conclusion of which the contents of the list should be ordered, with smaller numbers in lower numbered locations of the list; $c(b)$, $1 \leqq b \leqq N$, will denote the contents of position b of the list, and $[a:b]$ will denote the comparison-interchange which compares $c(a)$ with $c(b)$ and interchanges them if and only if $c(a) > c(b)$.

We will frequently make use of the "zero-one principle": if a sorting network sorts all possible sequences of 0's and 1's into nondecreasing order, then it will sort any arbitrary sequence of numbers into nondecreasing order. For a proof, see Knuth [5, p. 224].

To sort a list of length $2k$, (k even) by Batcher's "odd-even merge", one initially sorts the first and second halves of the list. Since the contents of the odd (and even) numbered locations in each half are sorted after this initial step, the process of sorting the contents of all the odd (or even) numbered locations can be done by merging two sorted lists. It will be helpful to visualize the list as a $k \times 2$ array where the first (second) column is the odd (even) numbered locations in ascending order. At the conclusion of the sorts and merges referred to above, the rows and columns of this $k \times 2$ array are sorted. However, the list is not completely sorted.

After the initial step of the sort, each half of the array has at most one row with both a zero and a one. After the merge step, there are at most two such rows. Thus the portion of the array which is (possibly) out of order is any square (2×2) subarray. Such an out-of-order square will be row and column sorted and, when it exists, it will be called the *square of uncertainty*.

* Received by the editors August 21, 1973 and in revised form March 27, 1974.

† Department of Computer Science, Stanford University, Stanford, California 94305. This research was performed while this author was a student in the Honors Program at Knox College, Galesburg, Illinois.

‡ Department of Mathematics and Computer Science, Knox College, Galesburg, Illinois 61401.

If the square of uncertainty starts with the ith row, then the comparison-interchange $[2i:2i+1]$ may be needed to order the list. Since the square of uncertainty could begin in any row except the last, Batcher's sort will be completed by the additional comparison-interchanges $[2i:2i+1]$, $1 \leqq i < k$. Note that Batcher's sort is completed by first finding a sort for the (row and column sorted) square of uncertainty and then performing this sort in all possible positions of the square of uncertainty.

Green [4] extended Batcher's algorithm by dividing the original list (of length $4k$, with k a multiple of 4) into 4 parts. First each of these parts is sorted. Then (viewing the list as a $k \times 4$ array) each column is ordered by merging its 4 sorted parts. Now the array has at most 4 rows of mixed 0's and 1's, i.e., there may be a 4×4 square of uncertainty somewhere in the array. Green developed a sort for this (row and column sorted) square which takes 21 steps. Green's sort is concluded by performing this 21-step sort in all possible positions of the square of uncertainty. Green's 21-step sort is constructed so as to insure a large amount of overlap when it is done in successive positions.

In this paper we develop methods for sorting $2^r \times 2^r$ squares which are row and column sorted. These sorts provide the necessary final step for a divide/sort/merge procedure where the original list is divided into 2^r lists. Our method reduces to Batcher's when $r = 1$ and to Green's when $r = 2$.

2. Description of the sort. Our sorting networks will operate on lists of length 2^n which we will view as if they are $2^{n-r} \times 2^r$ arrays, $2r \leqq n$. The element in position (s, t) of the array is at position $(s-1)2^r + t$ in the list; $c(s, t)$ will denote the *contents* of position (s, t). The ith *subarray of width* 2^a will refer to the positions in 2^a successive columns where the number of the last column is $i \cdot 2^a$. Two columns t_1 and t_2 are *corresponding columns* (of subarrays of width 2^a) if $t_1 \equiv t_2 \pmod{2^a}$. Two positions are *corresponding positions* if they are in corresponding columns and in the same row.

We now define sets of comparison-interchanges, $S_{w,k}$, where $1 \leqq w \leqq r$ and $1 \leqq k \leqq r$. When $w \geqq k$,

$$S_{w,k} = \{[(s,t):(s+2^{w-k}, t-2^{w-1})]\},$$

where t is a column in the right half of a subarray of width 2^w. When $w < k$,

$$S_{w,k} = \{[(s,t):(s+1, t-2^k+2^{w-1})]\},$$

where t is a column in the right half of the ith subarray of width 2^w where $i \equiv 0 \pmod{2^{k-w}}$ together with

$$\{[(s,t):(s, t+2^{w-1})]\},$$

where t is a column in the right half of the jth subarray of width 2^w where $j \not\equiv 0 \pmod{2^{k-w}}$.

The comparison-interchanges in $S_{w,k}$ form regular geometric patterns when applied to an array. Figures 1 and 2 depict these patterns when the original array is $2^r \times 2^r$. Figure 1 shows the action of $S_{w,k}$, $w \geqq k$, on a subarray of width 2^w. Figure 2 shows the action of $S_{w,k}$, $w < k$, on a subarray of width 2^k which has been rearranged so that it is an array of width 2^w. In each case the region labeled A is compared with the region labeled B.

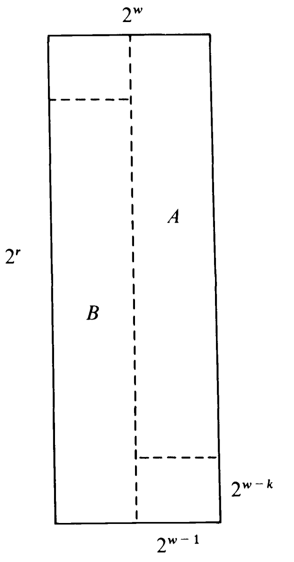


FIG. 1

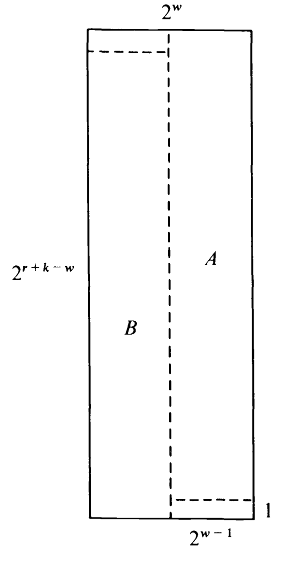


FIG. 2

We shall call a comparison-interchange in $S_{w,k}$ *special* when it is of the form $[(s,t):(s,t+2^{w-1})]$, where (s,t) is in the left half of a subarray of width 2^k and $(s,t+2^{w-1})$ is in the right half of a subarray of width 2^k.

We also define three types of comparison-interchanges which we later eliminate.

Type 1. The special comparison-interchanges in $S_{w,k}$ in the first $1 + 2(2^{r-k} - i)$ rows and the last $1 + 2(i-1)$ rows in the ith subarray of width 2^k.

Type 2. Comparison-interchanges in $S_{w,k}$ between two positions in the first (last) row which both lie in an odd (even) numbered subarray of width 2^{k-1}.

Type 3. When $k \leqq r - 1$, the comparison-interchanges in $S_{w,k}$ between two positions in the second (next to last) row which both lie in the first (last) subarray of width 2^{k-1}.

Note that each type of comparison-interchange defined above is between positions which are in the same row. Hence they only occur in $S_{w,k}$ when $w < k$.

We now define $\mathbf{S}_{w,k}$ to be $S_{w,k}$ minus any comparison-interchange of types 1, 2 or 3 which occur in it.

We let $G_k^r = S_{r,k}S_{r-1,k} \cdots S_{1,k}$ and $\mathbf{G}_k^r = \mathbf{S}_{r,k} \cdots \mathbf{S}_{1,k}$.

THEOREM. $\mathbf{G}_1^r\mathbf{G}_2^r \cdots \mathbf{G}_r^r$ *will sort a* $2^{n-r} \times 2^r$ *array which has been row and column sorted and whose square of uncertainty is of size* $2^r \times 2^r$.

The somewhat lengthy proof of this theorem comprises § 4 of this paper.

When $n \geqq 2r$, the sorts and merges that begin the divide/sort/merge strategy leave an array which satisfies the conditions of this theorem. For a discussion of methods which will adapt the divide/sort/merge strategy to arrays of arbitrary length, see Van Voorhis [6], [7].

3. Numerical results. We now count the number of comparison-interchanges required to sort a list of length 2^n, n a multiple of r, using a 2^r-way divide/sort/merge strategy.

The number of comparison-interchanges in $S_{w,k}$ is $2^{r-1}(2^{n-r} - 2^{w-k})$ when $w \geqq k$, and is $(2^n - 2^{r-k}2^w)/2$ when $w < k$. Thus the total number of comparison-interchanges in $G_1^r G_2^r \cdots G_r^r$ is

$$g_{n,r} = \sum_{w=1}^{r} \sum_{k=1}^{w} 2^{r-1}(2^{n-r} - 2^{w-k}) + \sum_{k=1}^{r} \sum_{w=1}^{k-1} \frac{2^n - 2^{r-k}2^w}{2}.$$

We have $g_{n,r} = r^2 2^{n-1} - 2^{2r} + 2^{r+1} - 1$.

Denoting the number of Type i comparison-interchanges by t_i, we have

$$t_1 = \sum_{k=1}^{r} \sum_{w=1}^{k-1} 2^{r-k+1} 2^{r-k} 2^{w-1},$$

$$t_2 = 2 \sum_{k=1}^{r} \sum_{w=1}^{k-1} 2^{r-k}(2^{k-w-1} - 1)2^{w-1},$$

$$t_3 = 2 \sum_{k=1}^{r-1} \sum_{w=1}^{k-1} (2^{k-w-1} - 1)2^{w-1}.$$

These yield $t_1 = (2^{2r} + 2)/3 - 2^r$, $t_2 = (r^2 - 5r + 8)2^{r-2} - 2$, $t_3 = (r = 5)2^{r-1} + 2(r + 1)$. Thus the total number of comparison-interchanges in $\mathbf{G}_1^r \cdots \mathbf{G}_r^r$ is $\mathbf{g}_{n,r} = g_{n,r} - t_1 - t_2 - t_3$, and we have

(1) $$\mathbf{g}_{n,r} = r^2 2^{n-1} - (2^{2r+2} + 5)/3 - (r^2 - 3r - 14)2^{r-2} - 2r.$$

The procedure described in the Introduction leads to a pair of recursion relations. Letting $p_{n,r}$ denote the number of comparison-interchanges required to merge 2^r sorted lists of length 2^{n-r} and $p_{n,r}^*$ denote the total number required to sort 2^n locations by the 2^r-way divide/sort/merge strategy, we find that

(2a) $$p_{n,r} = 2^r p_{n-r,r} + \mathbf{g}_{n,r},$$

(2b) $$p_{n,r}^* = 2^r p_{n-r,r}^* + p_{n,r}.$$

Solving (2a) and (2b) for fixed r gives

(3a) $$p_{n,r} = (n + a_r)r2^{n-1} + C_r(2^r - 1)^{-1},$$

(3b) $$p_{n,r}^* = (n^2 + (2a_r + r)n + b_r)2^{n-2} - C_r/(2^r - 1)^2,$$

where a_r and b_r are constants determined by the boundary conditions and

$$C_r = (2^{2r+2} + 5)/3 + (r^2 - 3r - 14)2^{r-2} + 2r.$$

Using $p_{0,r}^* = 0$ as a boundary condition, we can restate (3b) as

(4) $$p_{n,r}^* = [n^2 + (2a_r + r)n + 4C_r(2^r - 1)^{-2}]2^{n-2} - C_r(2^r - 1)^{-2}.$$

The value of a_r can be determined by setting $p_{r,r}^* = S(r)$, where $S(r)$ equals the number of comparison-interchanges required to sort 2^r locations by any strategy. This gives us

$$a_r = (S(r) - C_r(2^r - 1)^{-1})r^{-1}2^{1-r} - r.$$

From (4) we see that the number of comparisons required to sort N locations by the 2^r way divide/sort/merge strategy is

$$(.25)N(\log_2 N)^2 - Z_r N \log_2 N + O(N), \quad \text{where } Z_r = -(2a_r + r)/4.$$

For Batcher's odd-even merge, $S(1) = 1$ and we have $Z_1 = .25$. For Green's 4-merge, $S(2) = 5$ and $Z_2 = \frac{1}{3}$.

The value of Z_4 depends on which 16-sort is chosen. Green's 4-merge gives $S(4) = 61$ and $Z_4 = 89/240 = .3708\bar{3}$. But Green [4] also developed a special 16-sort requiring only 60 comparison-interchanges. Using this 16-sorter gives $S(4) = 60$ and $Z_4 = 371/960 = .386458\bar{3}$. The best results previously known are due to Van Voorhis [6], [7], whose values for Z_r as $r \to \infty$ approach $.357^-$ without Green's 16-sorter and $.372^+$ with it. To sort a 256 position list, Batcher's procedure requires 3,839 comparison-interchanges, Green's requires 3,725, Van Voorhis' requires 3,673 and ours requires 3,657.

[Subsequent to the submission of this paper, Van Voorhis [8] developed an improvement of his previous sorting networks. Van Voorhis' new networks require $(.25)N(\log_2 N)^2 - (.395)N \log_2 N + O(N)$ comparison-interchanges to sort a list of length N. These new networks require 3651 comparison-interchanges to sort a 256 position list.]

Van Voorhis notes that with his merge strategy the most efficient networks occur when r is a power of 2. It thus would seem likely that the size of Z_r for our networks would increase as r increases. However, $Z_8 \cong .384896$, and this slow decrease appears to continue. We conjecture that as r increases, there exist other comparison-interchanges which can be eliminated.

It is interesting to note that the number of comparison-interchanges in the 2^r-way divide/sort/merge 2^n-sorter network that uses $G_1^r \cdots G_r^r$ is identical to the number in Batcher's 2^n-sorter based on the odd-even merge. The improvement is due entirely to the eliminations.

The reader interested in comparing these results with the best known results for adaptive sorting should see Ford and Johnson [3].

4. Proof of theorem. We assume throughout this section that we are sorting a $2^{n-r} \times 2^r$ array, $n \geqq 2r$, which is already row and column sorted and which has a square of uncertainty of size $2^r \times 2^r$. Note that these conditions are satisfied if we perform the first steps of the 2^r-way divide/sort/merge strategy on the array. The theorem to be proven states that $\mathbf{G}_1^r \mathbf{G}_2^r \cdots \mathbf{G}_r^r$ is a sufficient final step for the divide/sort/merge strategy.

LEMMA 1. *Each $S_{w,k}$ preserves the order in the columns of the array.*

LEMMA 2. *If $w \geqq k$ and $w > \max(v, j)$, then $S_{w,k}$ and $S_{v,j}$ preserve the order of each other.*

LEMMA 3. *After $S_{r,1}$ is applied to the array, then corresponding positions in the two subarrays of width 2^{r-1} are ordered.*

Proof. Suppose there exists a pair of corresponding positions which is out of order. There are four ways this could arise. Three would contradict the ordering of the columns, and the other would require too many mixed rows.

LEMMA 4. *After $S_{r,1}S_{r,2} \cdots S_{r,k}$ is applied to the array, the left subarray of width 2^{r-1} has between 0 and 2^{2r-k-1} more 0's than the right subarray of width 2^{r-1}.*

Proof. The upper bound follows since $S_{r,k}$ compares all except $2^{r-1} \cdot 2^{r-k}$ positions in the left subarray with a position in the right subarray. The lower bound follows from Lemma 3.

LEMMA 5. *$G_1^r G_2^r \cdots G_r^r$ sorts the array.*

Proof. We use induction on r, noting that G_1^1 is the final part of Batcher's sort. We have

$$G_1^r G_2^r \cdots G_r^r = S_{r,1} G_1^{r-1} S_{r,2} G_2^{r-1} \cdots S_{r,r-1} G_{r-1}^{r-1} G_r^r.$$

$S_{r,1}$ leaves each subarray of width 2^{r-1} with a $2^{r-1} \times 2^{r-1}$ square of uncertainty. Thus by Lemma 2 and the induction hypothesis, each of these subarrays is sorted by $G_1^r G_2^r \cdots G_{r-1}^r$. In addition, Lemma 4 implies that there are between 0 and 2^r more 0's in the left subarray of width 2^{r-1} than in the right subarray after $G_1^r \cdots G_{r-1}^r$. Thus the number of rows with mixed 0's and 1's following $G_1^r \cdots G_{r-1}^r$ is no more than three. $S_{r,r}$ reduces this to no more than two, and the rest of G_r^r sorts these two rows in a Batcher-like manner.

COROLLARY 1. *After $G_1^r G_2^r \cdots G_k^r$, each subarray of width 2^k is sorted.*

LEMMA 6. *After $G_1^r G_2^r \cdots G_k^r$, each of the first (and last) 2^{r-k} rows is sorted.*

Proof. It suffices to show that if $c(s, 2^{r-1}+1) = 0$, then $c(s, 2^{r-1}) = 0$ for $s \leqq 2^{r-k}$. Suppose that this fact is not true for some r and k. Choose the smallest possible such r and the smallest possible k for that r ($k < r$). Then after applying $G_1^r G_2^r \cdots G_k^r$, we have a row s, $s \leqq 2^{r-k}$, such that $c(s, 2^{r-1}+1) = 0$ and $c(s, 2^{r-1}) = 1$. Because $s \leqq 2^{r-k}$, $c(s, 2^{r-1}) = 1$ before applying G_k^r. Therefore, since by assumption $G_1^r \cdots G_{k-1}^r$ leaves the first 2^{r-k+1} rows sorted, $c(s, 2^{r-1}+1) = 1$ before G_k^r, which implies $c(s + 2^{r-k}, 1) = 0$ before G_k^r. Choose the smallest m ($1 \leqq m < 2^{r-k}$) such that $c(s+m, 2^{r-1} - m2^{k-1}) = 0$ before G_k^r. (This m exists because, by Corollary 1, $c(s + 2^{r-k} - 1, 2^{k-1}) = 0$ before G_k^r.) If the contents of this position were still zero after $S_{r,k}$, then the rest of G_k^r would move a 0 into $(s, 2^{r-1})$. To avoid this contradiction, $(s + m - 2^{r-k}, 2^r - m2^{k-1})$ must exist and contain a 1 before G_k^r. Thus, since by assumption the first 2^{r-k+1} rows are sorted, $c(s + m - 2^{r-k}, 2^r - m2^{k-1} + 1) = 1$ before G_k^r. In addition, $c(s + m - 1, 2^{r-1} - (m-1)2^{k-1}) = 1$, which implies that $c(s+m, 2^{r-1} - m2^{k-1} + 1) = 1$ by Corollary 1. Therefore $c(s + m - 2^{r-k}, 2^r - m2^{k-1} + 1) = 1$ after $S_{r,k}$. But this implies that the rest of G_k^r will move the zero from $(s, 2^{r-1}+1)$, contradicting our assumptions.

LEMMA 7. *After $G_1^r \cdots G_{k-1}^r S_{r,k} \cdots S_{k,k}$, the first $1 + 2(2^{r-k} - i)$ and the last $1 + 2(i-1)$ rows of the i-th subarray of width 2^k are sorted.*

Proof. Lemma 6 shows that after $G_1^r \cdots G_{k-1}^r$, the first 2^{r-k+1} rows are sorted. After $G_1^r \cdots G_{k-1}^r S_{r,k} \cdots S_{m+1,k}$, where $r > m \geqq k$, the first $2^{r-k+1} - (i-1)2^{m-k+1}$ rows of the ith subarray of width 2^m are sorted since they are always compared with ordered rows. The proof will be completed if we show that $S_{k,k}$ cannot produce an unordered row in the first $2^{r-k+1} - (2i-1)$ rows. Suppose such an unordered row exists after $S_{k,k}$. Then there must exist three rows of mixed 0's and 1's within the first $2^{r-k+1} - 2(i-1)$ rows of the ith subarray of width 2^k before $S_{k,k}$ (i.e., after $S_{k+1,k}$). Similarly, suppose there exist $2^{m+1}+1$ mixed rows within the first $2^{r-k+1} - (i-1)2^{m+1}$ rows of the subarray of width 2^{k+m} before $S_{k+m,k}$. Then there must exist $2^{m+2}+1$ mixed rows within the first $2^{r-k+1} - (j-1)2^{m-k+2}$ rows of the jth subarray of width 2^{k+m+1} (where $j = [(i+1)/2]$)

before $S_{k+m+1,k}$. Clearly as m approaches $r - k$, a contradiction is inevitable. At some point there will not be enough rows available.

COROLLARY 2. *Type* 1 *and Type* 2 *comparison-interchanges are redundant and may be eliminated.*

Type 3 comparison-interchanges differ from the other types in that there exist configurations where a Type 3 comparison-interchange will actually perform an interchange. Suppose the contents of two positions in the second row of the first subarray of width 2^{k-1} are out of order after $\mathbf{G}_1^r \cdots \mathbf{G}_k^r$. Then Lemma 6 and Lemma 5 imply that before $\mathbf{G}_k^r$, $c(1, 2^{k-1}) = c(2, 1) = 0$ and $c(1, 2^k) = c(2, 2^{k-1}) = c(3, 1) = 1$. After $\mathbf{G}_k^r$, the first 2^k positions in the first row contain 0, and $c(3, 1) = 1$. Since by Lemma 6 the first row is sorted and by Lemma 1 all columns are sorted, we have 0's occurring only in the first two rows of this subarray. Because $k < r$, $\mathbf{G}_{k+1}^r$ exists and will complete the sort by first moving the first portion of the second row to the first row and then ordering it. Thus we have shown the following lemma.

LEMMA 8. *Type* 3 *comparison-interchanges are unnecessary and can be eliminated.*

Finally, the theorem follows immediately from Lemma 5, Corollary 2 and Lemma 8.

REFERENCES

[1] K. E. BATCHER, *Sorting networks and their applications*, Proc. AFIPS Spring Joint Comp. Conf., 32 (1968), pp. 307–314.

[2] R. L. DRYSDALE III, *Sorting networks which generalize Batcher's odd–even merge*, Senior Honors paper, Knox College, Galesburg, Ill., 1973.

[3] L. R. FORD AND S. M. JOHNSON, *A tournament problem*, Amer. Math. Monthly, 661 (1959), pp. 282–296.

[4] M. W. GREEN, *Some improvements in non-adaptive sorting algorithms*, Proc. 6th Princeton Conf. on Information Sciences and Systems, 1972, pp. 387–391.

[5] D. E. KNUTH, *Sorting and Searching, The Art of Computer Programming*, vol. 3, Addison-Wesley, Reading, Mass., 1973.

[6] D. C. VAN VOORHIS, *A generalization of the divide/sort/merge strategy for sorting networks*, Tech. Rep. 16, Digital Systems Lab., Stanford Univ., Stanford, Calif., 1971.

[7] ———, *Large* [*g*, *d*] *sorting networks*, Tech. Rep. 18, Digital Systems Lab., Stanford Univ., Stanford, Calif., 1971.

[8] ———, *An economical construction for sorting networks*, Working paper 16/A45 no. 1, IBM System Development Div., Los Gatos, Calif., 1974, also published in Proc. NCC, 1974.

SIAM J. COMPUT.
Vol. 4, No. 3, September 1975

NEAR-OPTIMAL SOLUTIONS TO A 2-DIMENSIONAL PLACEMENT PROBLEM*

R. M. KARP,† A. C. McKELLAR‡ AND C. K. WONG‡

Abstract. We consider the problem of placing records in a 2-dimensional storage array so that expected distance between consecutive references is minimized. A simple placement heuristic which uses only relative frequency of access for different records is shown to be within an additive constant of optimal when distance is measured by the Euclidean metric. For the rectilinear and maximum metrics, we show that there is no such heuristic. For the special case in which all access probabilities are equal, however, heuristics within an additive constant of optimal do exist, and their implementation requires solution of differential equations for which we give numerical solutions.

Key words. near-optimal algorithms, placement problems, heuristics, storage applications, expected distances, Euclidean metrics, rectilinear metrics, maximum metrics, L_p metrics

1. Introduction. The problem of positioning records in a linear storage medium in such a way that the expected access time is minimized has been thoroughly studied [1]–[5]. The solution is to place the most frequently accessed record and then repetitively to place the next most frequently accessed record alternating between the position immediately to the left of those already placed and the position immediately to the right.

In this paper, we consider a generalization of this problem in which the storage medium is an infinite 2-dimensional rectangular array of storage cells, and it is desired to minimize the expected Euclidean distance between consecutively referenced records. It is quite easy to construct examples to show that it is not sufficient to know only the ordering of records by frequency of access in order to construct the optimal solution. Thus there is no hope of finding as elegant an algorithm for this 2-dimensional case as for the 1-dimensional case.

The problem we consider is a special case of the quadratic assignment problem which arises for example in various circuit placement problems [6].

In this paper we consider an heuristic which operates only on the relative frequency with which records are accessed and show that the resulting placement is within an additive constant of optimal. This asymptotically optimal heuristic consists of placing the most frequently accessed record and then filling "shells" of storage cells which are equidistant from the center with a set of next most frequently accessed records.

We then consider the problem of replacing Euclidean distance with rectilinear distance, and with distance defined to be the maximum of the difference in the x-coordinates and the difference in the y-coordinates. In each case, we show that there is an analogue of the "shell" heuristic which is within an additive constant of optimal when the access probabilities are equal. However, a shell no longer

* Received by the editors March 7, 1974, and in revised form September 11, 1974.

† Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, California 94720. This work was performed while this author was visiting at IBM Thomas J. Watson Research Center.

‡ Computer Sciences Department, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.

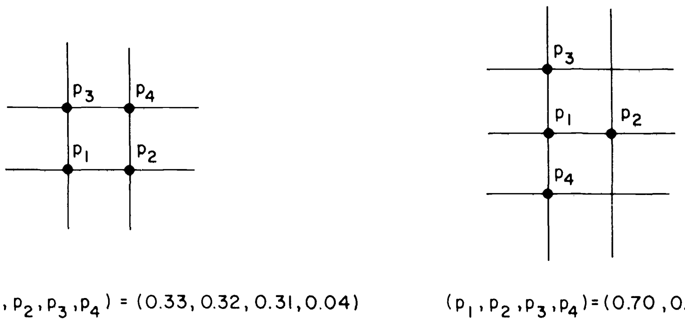
consists of the set of storage locations equidistant from the center, but rather consists of the set of cells on a contour given by the solution of a differential equation for which we have only been able to obtain numerical solutions. We use these results to show that, in general, there is no heuristic for the maximum and rectilinear metrics which operates only on the relative frequency of access and produces solutions within an additive constant of optimal.

2. Formulation of the problem. Consider a set of n records $x_1, \cdots, x_n$ which are referenced repetitively, where with probability p_i the reference is to x_i and consecutive references are independent. We adopt the convention that the records are numbered such that $p_1 \geqq p_2 \geqq \cdots \geqq p_n$. We wish to place these records into an infinite 2-dimensional rectangular array of storage cells such that the expected distance between consecutively referenced records is minimized, i.e., we wish to minimize

$$\bar{D} = \sum_{i=1}^{n} p_i \left(\sum_{j=1}^{n} p_j d_{ij} \right), \tag{1}$$

where d_{ij} is the distance between record i and record j. We will regard the storage cells as points with integral coordinates in the Euclidean plane, and adjacent cells are assumed to be at unit distance from each other.

Figures 1(a) and 1(b) give two examples of optimal placements. These examples show that it is not sufficient to know the relative frequency of access to minimize $\bar{D}$; one must have more detailed knowledge of the probabilities of access.



p_3
p_4
p_1
p_2
$(p_1, p_2, p_3, p_4) = (0.33, 0.32, 0.31, 0.04)$
p_3
p_1
p_2
p_4
$(p_1, p_2, p_3, p_4) = (0.70, 0.15, 0.10, 0.05)$

FIG. 1. *Optimal placement depends on probabilities rather than only on relative frequency*

However, in case optimal solutions are not absolutely essential, one may want to use the simple heuristic mentioned in the Introduction, namely, filling "shells" of storage cells which are equidistant from the center with a set of records with next largest probabilities. This algorithm depends only on the ordering of the probabilities of access and is referred to as the "shell" algorithm from now on.

In the next section, we will show that the expected distance between consecutively referenced records resulting from the "shell" algorithm is within an additive constant of that resulting from an optimal placement algorithm.

3. Analysis of the algorithm. Define

$$\Delta_i = p_i - p_{i+1}, \qquad 1 \leqq i < n,$$

and

$$\Delta_n = p_n,$$

so that

$$p_i = \sum_{r=i}^{n} \Delta_r \quad \text{and} \quad \sum_{r=1}^{n} r\Delta_r = \sum_{i=1}^{n} p_i = 1.$$

Hence $\bar{D}$ can be rewritten as

$$\bar{D} = \sum_{i=1}^{n} \sum_{r=i}^{n} \Delta_r \left(\sum_{j=1}^{n} \sum_{s=j}^{n} \Delta_s \right) d_{ij}.$$

Interchanging the orders of summation yields

$$\bar{D} = \sum_{r=1}^{n} \sum_{s=1}^{n} \Delta_r \Delta_s E_{rs},$$

where

$$E_{rs} = \sum_{i=1}^{r} \sum_{j=1}^{s} d_{ij}.$$

The effect of this transformation has been to replace the probabilities by a set of n variables among which there are no ordering constraints. Furthermore, the effect of the placement algorithm has been localized to a term, E_{rs}, which is independent of the access probabilities. For given r and s, there is a placement dependent only on the relative frequency of access which minimizes E_{rs}. However, that placement is incompatible with the placement for some other values of r and s. For example, the shape of Fig. 1(a) minimizes E_{rs} with $r = 3$, $s = 4$, whereas Fig. 1(b) is optimal for $r = 1$, $s = 4$. Thus, in general, it is not possible to enlarge an optimal solution for n points to an optimal solution for $n + 1$ points in a straightforward way, which explains why our problem is more difficult than the 1-dimensional case.

Let $\bar{D}(\text{opt})$, $E_{rs}(\text{opt})$ denote the values produced by an optimum placement algorithm (i.e., one which minimizes $\bar{D}$) and $\bar{D}(\text{shell})$, $E_{rs}(\text{shell})$ denote the values produced by the "shell" algorithm. We shall show that

(2) $$E_{rs}(\text{shell}) \leqq E_{rs}(\text{opt}) + crs$$

where c is a constant independent of r, s. As a consequence,

(3) $$\bar{D}(\text{shell}) \leqq \bar{D}(\text{opt}) + c \sum_{r=1}^{n} \sum_{s=1}^{n} \Delta_r \Delta_s rs = \bar{D}(\text{opt}) + c.$$

We were unable to find a straightforward proof of (2), and so we consider the continuous analogue of E_{rs} for which it is relatively easy to find the optimal solution.

Since $d_{ij} = d_{ji}$, without loss of generality, we can assume $r \leqq s$. The problem is then to find two regions ω_0 and ω_1 with areas r and s, respectively, and $\omega_0 \subset \omega_1$ such that the integral

(4) $$\int_{x \in \omega_1} \int_{y \in \omega_0} d(x, y)$$

is minimized, where x, y denote points in ω_1, ω_0, respectively, and $d(x, y)$ is the distance between points x and y. The following definition formalizes an obvious geometric property.

DEFINITION. Let ω be any region and L any straight line dividing the plane into A and B. ω is said to have the *covering property with respect to* L if either $I(A \cap \omega) \supset (B \cap \omega)$ or $I(B \cap \omega) \supset (A \cap \omega)$, where $I(A \cap \omega)$ means the mirror image of $A \cap \omega$ with respect to L. $I(B \cap \omega)$ is similarly defined. (See Fig. 2.)

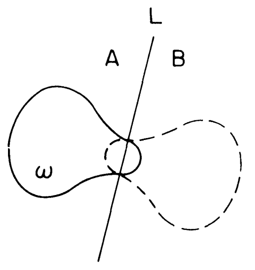


FIG. 2. *Region with covering property with respect to L*

LEMMA 1. *If* $\omega_0^*, \omega_1^*(\omega_0^* \subset \omega_1^*)$ *form a minimal solution for* (4), *then* ω_0^*, ω_1^* *have the covering property with respect to any straight line L. Furthermore, if L partitions the plane into A and B, and if* $I(A \cap \omega_1^*) \supset (B \cap \omega_1^*)$, *then* $I(A \cap \omega_0^*) \supset (B \cap \omega_0^*)$. *Similarly, if* $I(B \cap \omega_1^*) \supset (A \cap \omega_1^*)$, *then* $I(B \cap \omega_0^*) \supset (A \cap \omega_0^*)$.

The lemma can be obtained easily by applying the technique used by Bergmans [5]. The present Appendix A contains a proof.

LEMMA 2. *Any region which has the covering property with respect to any straight line must be a disk.*

Proof. Let C be the center of mass of the region. (See Fig. 3.) Suppose there exist points α, β on the boundary such that $d(\alpha, C) < d(\beta, C)$. Draw a straight line L through C, bisecting the angle $\alpha C \beta$ and meeting the boundary at γ.

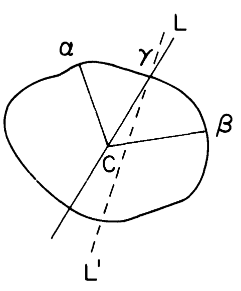


FIG. 3. *Illustration for proof of Lemma* 2

Let L' be a straight line through γ cutting the region into two parts with equal areas ω_1, ω_2. By the covering property, ω_1, ω_2 should be symmetric images of each other with respect to L'. In particular, L' should go through C, hence L, L' coincide. Thus α, β are symmetric with respect to L, a contradiction.

LEMMA 3. *The minimal solution to* (4) *consists of two concentric circles.*

Proof. Let ω_0^*, ω_1^* form an optimal solution and $\omega_0^* \subset \omega_1^*$. By Lemmas 1 and 2, they must be circles. It remains to show that they are concentric. Suppose it is not the case. Let C_0, C_1 be the centers of ω_0^*, ω_1^*. Let L be the perpendicular bisector of the line segment C_0C_1 (see Fig. 4). With respect to L, the second part of Lemma 1 is violated, hence a contradiction.

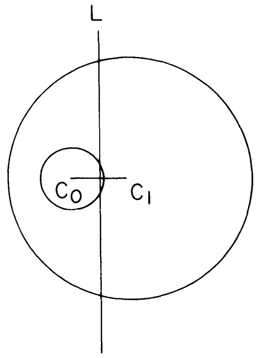


FIG. 4. *Illustration for proof of Lemma* 3

Let E_{rs}^{cont} denote the optimal value of (4), i.e.,

$$E_{rs}^{\text{cont}} = \int_{x\in\omega_1^*}\int_{y\in\omega_0^*} d(x, y),$$

where ω_1^*, ω_0^* are concentric circles with areas s, r respectively. We will compare this with the "shell" algorithm and the optimal placement algorithm.

If we look at the first r and s points ($r \leqq s$) in the configuration resulting from the "shell" algorithm, and compute the function $\sum_{i=1}^{r}\sum_{j=1}^{s} d_{ij}$, we obtain E_{rs}(shell). Suppose we replace each point by a unit square with center at this point; then we have two regions ω_0, ω_1 with areas r, s, respectively, and $\omega_0 \subset \omega_1$. Let

$$E_{rs}^{\text{cont}}(\text{shell}) = \int_{x\in\omega_1}\int_{y\in\omega_0} d(x, y).$$

We define the continuous analogue of the optimal placement in the same way and let $E_{rs}^{\text{cont}}(\text{opt})$ be the value of the integral in (4) evaluated over the corresponding regions.

We will show that

$$E_{rs}(\text{shell}) \leqq E_{rs}^{\text{cont}}(\text{shell}) + \sqrt{2}rs$$

by Lemma 4, that

$$E_{rs}^{\text{cont}}(\text{shell}) \leqq E_{rs}^{\text{cont}} + (2\sqrt{2} + 8\sqrt{\pi})rs$$

by Lemma 6, that

$$E_{rs}^{\text{cont}} \leqq E_{rs}^{\text{cont}}(\text{opt})$$

which is obvious, and that

$$E_{rs}^{\text{cont}}(\text{opt}) \leqq E_{rs}(\text{opt}) + \sqrt{2}\, rs$$

by Lemma 5. Combining these results yields the following theorem.

THEOREM. *$E_{rs}(\text{shell}) \leqq E_{rs}(\text{opt}) + (4\sqrt{2} + 8\sqrt{\pi})rs$. Consequently, $\bar{D}(\text{shell}) \leqq \bar{D}(\text{opt}) + (4\sqrt{2} + 8\sqrt{\pi})$.*

LEMMA 4. *$E_{rs}(\text{shell}) \leqq E_{rs}^{\text{cont}}(\text{shell}) + \sqrt{2}\, rs$.*

Proof. Only notice that for any x in the square of i and any y in the square of j (see Fig. 5),

$$d_{ij} \leqq d(x, y) + \frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2} = d(x, y) + \sqrt{2}.$$

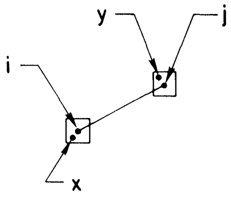


FIG. 5. *Illustration for proof of Lemma* 4

Although we are showing that the continuous case is bounded by the discrete case, similar argument shows that the following lemma holds.

LEMMA 5. *$E_{rs}^{\text{cont}}(\text{opt}) \leqq E_{rs}(\text{opt}) + \sqrt{2}\, rs$.*

Next we will prove the following result.

LEMMA 6. *$E_{rs}^{\text{cont}}(\text{shell}) \leqq E_{rs}^{\text{cont}} + (2\sqrt{2} + 8\sqrt{\pi})rs$.*

Proof. On the regions ω_0, ω_1 for the continuous version of the "shell" algorithm, we superimpose the two concentric circles ω_0^*, ω_1^* with areas r, s,

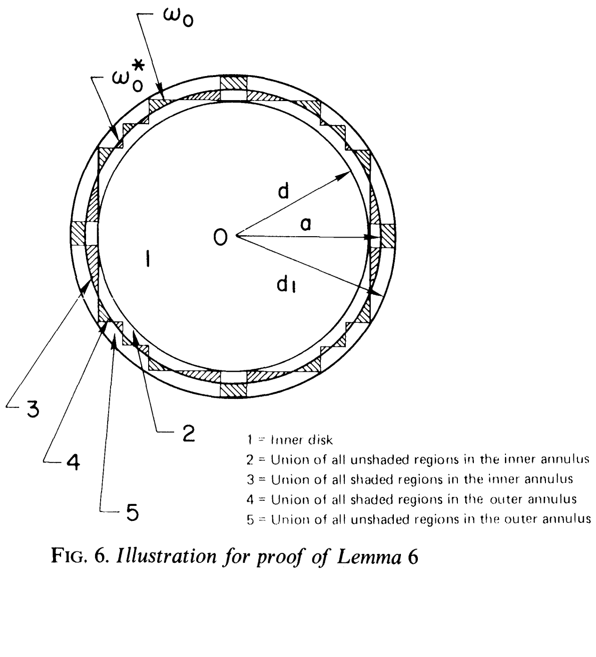

ω_0
ω_0^*
d
a
O
I
d_1
3
2
4
5
1 = Inner disk
2 = Union of all unshaded regions in the inner annulus
3 = Union of all shaded regions in the inner annulus
4 = Union of all shaded regions in the outer annulus
5 = Union of all unshaded regions in the outer annulus

FIG. 6. *Illustration for proof of Lemma* 6

respectively, such that their centers 0 coincide with the point where the record x_1 is located as shown in Fig. 6. (ω_1^* is not shown in the figure.)

Let us look at ω_0 and ω_0^*. In Fig. 6, the region $\omega_0 - \omega_0^*$ is shaded in one direction and the region $\omega_0^* - \omega_0$ is shaded in another. Let d denote the radius of the largest circle centered at 0 inside ω_0. Let $a = \sqrt{r/\pi}$ be the radius of ω_0^*. Let d_1 denote the radius of the smallest circle centered at 0 outside ω_0. Then

$$(5) \qquad d_1 \leqq d + \sqrt{2} \quad \text{and} \quad d \leqq a \leqq d_1 \leqq d + \sqrt{2}.$$

To see this, let c be the center of the square farthest away from 0. Let d_c be its distance from the center. Then $d_1 \leqq d_c + \sqrt{2}/2$ and $d \geqq d_c - \sqrt{2}/2$. The first is obvious. To show the second one, assume the contrary; it follows that there exists an empty square, the distance of whose center to 0 is less than d_c, a contradiction to the "shell" algorithm.

Let us classify the regions inside the circle with radius d_1 into 5 classes as denoted in Fig. 6. Therefore, regions 1, 2, 3 will form the region ω_0^*, regions 4, 5 will form the outer annulus A, and regions 2, 3 will form the inner annulus B. Also, area of region 3 equals that of region 4.

To obtain an upper bound on area 4 and hence on area 3 we note that

$$\text{area } 4 \leqq \min(\text{area } A, \text{area } B).$$

But min (area A, area B) is maximized when $d_1 - d = \sqrt{2}$ and area A = area B. This occurs when $d_1 - a = \frac{1}{2}(\sqrt{2} - 2a + \sqrt{4a^2 - 2})$. In this case, area A = area $B = \pi\sqrt{2a^2 - 1}$. Therefore

$$(6) \qquad \text{area } 3 = \text{area } 4 < \pi\sqrt{2a^2} = \sqrt{2\pi r}.$$

Let us do exactly the same thing for ω_1 and ω_1^*, and call the corresponding regions 1′, 2′, 3′, 4′, 5′, A' and B'. Also let the radius of ω_1^* be b. Therefore $b = \sqrt{s/\pi}$. Thus

$$E_{rs}^{\text{cont}}(\text{shell}) = \int_{124}\int_{1'2'4'},$$

$$E_{rs}^{\text{cont}} = \int_{123}\int_{1'2'3'},$$

$$\begin{aligned} E_{rs}^{\text{cont}}(\text{shell}) - E_{rs}^{\text{cont}} &= \int_{124}\int_{2'4'} + \int_{124}\int_{1'} - \int_{123}\int_{2'3'} - \int_{123}\int_{1'} \\ &= \int_{124}\int_{2'4'} - \int_{123}\int_{2'3'} + \int_{4}\int_{1'} - \int_{3}\int_{1'} \\ &\leqq \int_{124}\int_{2'4'} + \int_{4}\int_{1'} - \int_{3}\int_{1'}; \end{aligned}$$

$$\text{area } 2'4' = \text{area } 2'3' \leqq \pi b^2 - \pi(b - \sqrt{2})^2 = 2\sqrt{2\pi s} - 2\pi;$$

$$\text{area } 124 = r.$$

Hence

$$\int_{124}\int_{2'4'} \leqq r(2\sqrt{2\pi s} - 2\pi)(a + b + 2\sqrt{2})$$
$$= r(2\sqrt{2\pi s} - 2\pi)(\sqrt{r/\pi} + \sqrt{s/\pi} + 2\sqrt{2})$$
$$\leqq (2\sqrt{2} + 6\sqrt{\pi})rs.$$

To bound $\int_4\int_{1'} - \int_3\int_{1'}$, we use polar coordinates (ρ', θ') for points in $1'$ and (ρ, θ) for points in A. (Recall that A is an annulus with inner radius a and outer radius d_1).

Let $d(\rho, \theta, \rho', \theta')$ denote the distance from point (ρ, θ) to (ρ', θ'). Let the radius of $1'$ be d'.

For each point (ρ, θ) in A, let

$$I_{\rho,\theta} = \int_{\rho'=0}^{d'}\int_{\theta'=0}^{2\pi} d(\rho, \theta, \rho', \theta')\rho'\, d\rho'\, d\theta'.$$

It is easy to show that (i) $I_{\rho,\theta}$ depends on ρ only, and (ii) if $\rho < \hat{\rho}$, then $I_{\rho,\theta} < I_{\hat{\rho},\theta}$. Therefore

$$I_{\rho,\theta} \leqq \int_{\rho'=0}^{d'}\int_{\theta'=0}^{2\pi} d(d_1, 0, \rho', \theta')\rho'\, d\rho'\, d\theta',$$

and

$$\int_4\int_{1'} = \int_{\rho=a}^{d_1}\int_{\theta=0}^{2\pi} \rho\, d\rho\, d\theta\, I_{\rho,\theta}$$
$$\leqq (\text{area } 4)\int_{\rho'=0}^{d'}\int_{\theta'=0}^{2\pi} d(d_1, 0, \rho', \theta')\rho'\, d\rho'\, d\theta'.$$

Similarly,

$$\int_3\int_{1'} \geqq (\text{area } 3)\int_{\rho'=0}^{d'}\int_{\theta'=0}^{2\pi} d(d, 0, \rho', \theta')\rho'\, d\rho'\, d\theta'.$$

Thus

$$\int_4\int_{1'} - \int_3\int_{1'} \leqq (\text{area } 3)\sqrt{2}\,(\text{area } 1') \leqq \sqrt{2\pi r}\,\sqrt{2}\, s = 2\sqrt{\pi r}\, s.$$

Therefore

$$E_{rs}^{\text{cont}}(\text{shell}) - E_{rs}^{\text{cont}} \leqq (2\sqrt{2} + 8\sqrt{\pi})rs.$$

4. Other metrics with uniform distribution. The proof in the preceding section goes through practically unchanged for other metrics, although the constants change. The stumbling block is in finding the optimal solution to the continuous problem. In this section we consider the special case in which all access probabilities are equal. Thus we have

$$\Delta_n = \frac{1}{n} \quad \text{and} \quad \Delta_i = 0, \qquad 1 \leqq i < n.$$

Hence we are concerned only with E_{nn} in (2) and need only find the optimal continuous solution for the case $r = s = n$. In this section we state this continuous problem in a more general form and characterize its optimal solution for two metrics of practical interest.

Let $\xi = (x_1, y_1)$, $\eta = (x_2, y_2)$ be any two points in the Euclidean plane. We shall consider the following family of metrics:

$$m_p(\xi, \eta) = (|x_1 - x_2|^p + |y_1 - y_2|^p)^{1/p}, \qquad 1 \leqq p \leqq \infty.$$

By $p = \infty$ we mean $m_\infty(\xi, \eta) = \max(|x_1 - x_2|, |y_1 - y_2|)$. For a fixed p and a fixed R, we need to find the solution to the following minimization problem:

$$(7) \qquad \min_{\omega \in \Omega} \iint_{\xi, \eta \in \omega} m_p(\xi, \eta)$$

where Ω is the set of all closed regions in the Euclidean plane with area R.

The following is a necessary condition for an optimal region.

LEMMA 7. *Let ω_0 be an optimal region. Let α be a point on the boundary of ω_0. Define*

$$(8) \qquad P(\alpha) = \int_{\eta \in \omega_0} m_p(\alpha, \eta).$$

Then $P(\alpha)$ is a constant for all α on the boundary.

Proof. Let β be another point on the boundary (see Fig. 7). Let ε_α be a region at α inside ω_0 and ε_β a region at β outside ω_0. Assume that ε_α and ε_β both have area ε. Let the new region obtained by removing ε_α from ω_0 and adding ε_β to ω_0 be ω_ε. Then ω_ε has area R.

$$I = \iint_{\xi, \eta \in \omega_\varepsilon} m_p(\xi, \eta) - \iint_{\xi, \eta \in \omega_0} m_p(\xi, \eta)$$

$$= \int_{\omega_0}\int_{\omega_0} + \int_{\omega_0}\int_{\varepsilon_\beta} - \int_{\omega_0}\int_{\varepsilon_\alpha} + \int_{\varepsilon_\beta}\int_{\omega_0} + \int_{\varepsilon_\beta}\int_{\varepsilon_\beta} - \int_{\varepsilon_\beta}\int_{\varepsilon_\alpha}$$

$$- \int_{\varepsilon_\alpha}\int_{\omega_0} - \int_{\varepsilon_\alpha}\int_{\varepsilon_\beta} + \int_{\varepsilon_\alpha}\int_{\varepsilon_\alpha} - \int_{\omega_0}\int_{\omega_0}.$$

Noting that terms 5, 6, 8 and 9 are of order ε^2, we have

$$I = 2\left(\int_{\omega_0}\int_{\varepsilon_\beta} - \int_{\omega_0}\int_{\varepsilon_\alpha}\right) + O(\varepsilon^2)$$

$$= 2\left(\varepsilon \int_{\eta \in \omega_0} m_p(\alpha_\varepsilon, \eta) - \varepsilon \int_{\eta \in \omega_0} m_p(\beta_\varepsilon, \eta)\right) + O(\varepsilon^2),$$

where $\alpha_\varepsilon \in \varepsilon_\alpha$, $\beta_\varepsilon \in \varepsilon_\beta$ are determined by the mean value theorem and $\alpha_\varepsilon \to \alpha$, $\beta_\varepsilon \to \beta$ as $\varepsilon \to 0$.

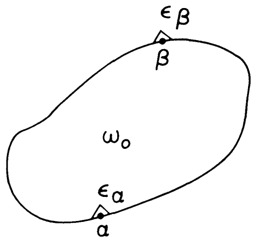


FIG. 7. *Illustration for proof of Lemma 7*

By optimality,

$$\lim_{\varepsilon\to 0}\frac{\int_{\omega_\varepsilon}\int_{\omega_\varepsilon}-\int_{\omega_0}\int_{\omega_0}}{\varepsilon}=0.$$

Therefore

$$\int_{\eta\in\omega_0} m_p(\alpha,\eta)-\int_{\eta\in\omega_0} m_p(\beta,\eta)=0,$$

as required.

The two cases of interest are $p = 1$, the rectilinear metric, and $p = \infty$, the maximum metric. For each of these metrics, we can use Bergmans' [5] methods of proof to show symmetry with respect to horizontal and perpendicular lines as well as lines at 45° and 135°.

We want to find the curve $f(x)$ valid in the first quadrant as shown in Fig. 8. Then, by symmetry, we can complete the figure. Because of symmetry about a line at 45°, $f(f(x)) = x$. Consider any point $(u, v = f(u))$ in the first quadrant, and, without loss of generality, let $u \leqq v$. Then (v, u) is also a point on the curve, as is $(-v, -u)$.

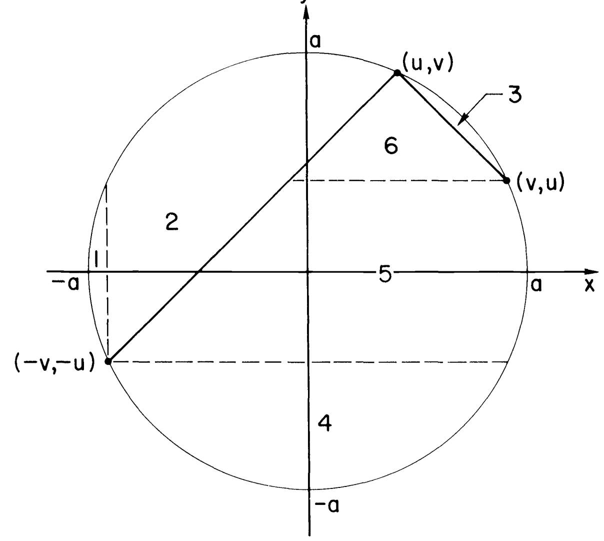


FIG. 8. *Computation of a shell for the maximum metric*

For the maximum metric, (8) can now be written as

$$\begin{aligned}P = &\int_{x=-a}^{-v}\int_{y=-f(x)}^{f(x)} (u-x)\,dy\,dx + \int_{x=-v}^{u}\int_{y=x-u+v}^{f(x)} (u-x)\,dy\,dx\\
(9)\qquad &+\int_{x=u}^{v}\int_{y=-x+v+u}^{f(x)} (x-u)\,dy\,dx + \int_{y=-a}^{-u}\int_{x=-f(y)}^{f(y)} (v-y)\,dx\,dy\\
&+\int_{y=-u}^{u}\int_{x=y+u-v}^{f(y)} (v-y)\,dx\,dy + \int_{y=u}^{v}\int_{x=y+u-v}^{x=-y+v+u} (v-y)\,dx\,dy,\end{aligned}$$

where the areas in Fig. 8 are numbered to correspond to the terms in (9). Performing all integrations not involving f and collecting terms yields

$$\begin{aligned}P = &\int_{x=-a}^{-v} 2(u-x)f(-x)\,dx + \int_{-v}^{u} (u-x)f(|x|)\,dx + \int_{u}^{v} (x-u)f(x)\,dx\\
&+\int_{y=-a}^{-u} 2(v-y)f(-y)\,dy + \int_{y=-u}^{u} (v-y)f(|y|)\,dy + \frac{1}{3}v^3 + vu^2.\end{aligned}$$

We apply the condition of Lemma 7 by requiring that $dP/du = 0$. Carrying out the differentiation and collecting terms again yields

$$\begin{aligned}&\int_{x=-a}^{-v} 2f(-x)\,dx + \int_{x=-v}^{u} f(|x|)\,dx - \int_{x=u}^{v} f(x)\,dx + 2f'(u)\int_{x=-a}^{-u} f(-x)\,dx\\
(10)\qquad &\qquad + f'(u)\int_{x=-u}^{u} f(|x|)\,dx + f'(u)(f^2(u) - u^2) = 0.\end{aligned}$$

Let the total area surrounded by the curve be R. Then the area in one quadrant is $R/4$. In view of the identity

$$R/4 + uf(u) = \int_0^u f(x)\,dx + \int_0^v f(x)\,dx,$$

(10) can be converted to

$$(11)\qquad (R + 2(f^2(u) - u^2))f'(u) + 8\int_0^u f(x)\,dx - 4uf(u) = 0.$$

Referring to Fig. 9, it is quite easy to obtain for the rectilinear metric

$$\begin{aligned}P = 2&\int_{x=u}^{a}\int_{y=-f(x)}^{f(x)} (x+y+v)\,dy\,dx + 2\int_{y=v}^{a}\int_{x=-f(y)}^{f(y)} (x+y+u)\,dx\,dy\\
&+\int_{x=-u}^{u}\int_{y=-v}^{v} (x+u+y+v)\,dy\,dx,\end{aligned}$$

which after simplification becomes

$$P = 4\int_{x=u}^{a} (x+v)f(x)\,dx + 4\int_{y=v}^{a} (y+u)f(y)\,dy + 4uv(u+v).$$

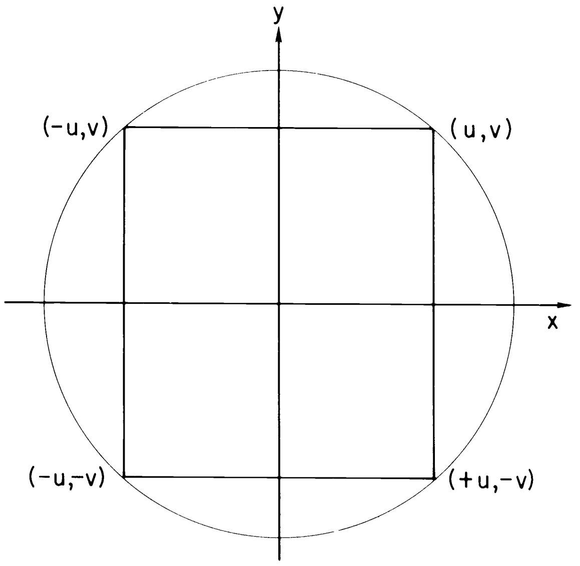


FIG. 9. *Computation of a shell for the rectilinear metric*

Differentiation with respect to u and applying Lemma 7 yields

$$f'(u)\int_{x=u}^{a} f(x)\,dx + \int_{x=f(u)}^{a} f(x)\,dx + uf(u)f'(u) + uf(u) = 0.$$

As before, let R be the total area. Noting that

$$\int_{x=u}^{a} f(x)\,dx = \frac{R}{4} - \int_{0}^{u} f(x)\,dx$$

and that

$$\frac{R}{4} = \int_{x=u}^{a} f(x)\,dx + \int_{x=v}^{a} f(x)\,dx + uf(u),$$

we have

$$f'(u)\left[\frac{R}{4} - \int_{0}^{u} f(x)\,dx + uf(u)\right] + \int_{0}^{u} f(x)\,dx = 0. \tag{12}$$

We have not been able to obtain closed form solutions for (11) or (12). Numerical solutions were obtained and are shown in Fig. 10 for the case $R = 4$, i.e., the area in each quadrant is 1. For other values of R, the shape is obtained by linear scaling.

A priori, one might have been tempted to guess that the square and diamond were optimal shapes for the maximum and rectilinear metrics, respectively, since each has the property that every point on the boundary is equidistant from the center. Instead, the shape has turned out to be quite close to a circle in each case, although it is true for the maximum metric that the circle is distorted toward the shape of a square, and for the rectilinear metric, distortion is toward a diamond.

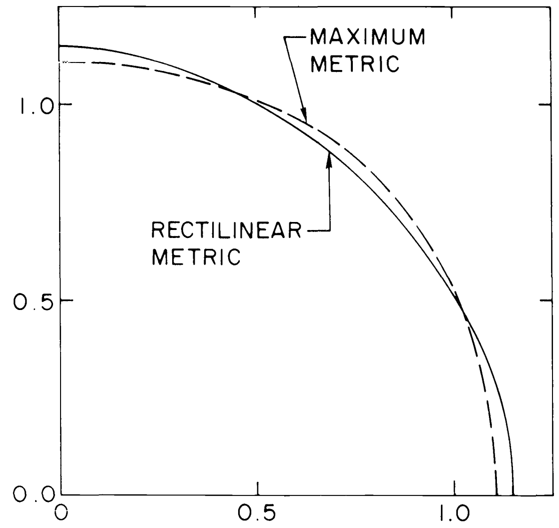


FIG. 10. *First quadrant of optimum shells compared for maximum and rectilinear metrics with a total area, $R = 4$*

5. Nonexistence of heuristics for the general case. In contrast to the Euclidean case, we will show that for the maximum and rectilinear metrics there is no heuristic which operates only on relative frequencies and which is within an additive constant of optimal.

It follows easily from the methods in § 3 that the optimal solution to the discrete problem is within an additive constant of the optimal solution for the corresponding continuous problem.

Given a continuous solution with total area n, we define a scaled solution with total area 1 by shrinking each area by a factor of $1/n$ and increasing the probability per unit area by a factor of n. Let $\bar{D}_n(\text{opt})$ be the expected distance for the solution with total area n. Then the expected distance $\bar{D}_1(\text{opt})$ for the scaled version is given by $\bar{D}_n(\text{opt}) = \sqrt{n}\,\bar{D}_1(\text{opt})$, and the scaled solution is also optimal.

Consider the case

$$p_1 = p \quad \text{and} \quad p_i = \frac{q}{n-1}, \qquad 2 \leqq i \leqq n,$$

where $p + q = 1$. Then for the corresponding continuous problem,

$$\bar{D}_1(\text{opt}) = 2pqa + q^2 b + O(1/\sqrt{n}),$$

where a is the average distance between a point with weight np and a point with weight $nq/(n-1)$ and b is the average distance between points with weight $nq/(n-1)$. Thus

$$\bar{D}_n(\text{opt}) = \sqrt{n}\,q(2pa + qb) + O(1).$$

The optimal solutions were obtained in § 4 for the case $p = 1/n$. Asymptotically for large n, the shapes of these optimal solutions are the shapes which minimize b.

On the other hand, the shape which minimizes a (again asymptotically for large n) is the diamond for the rectilinear metric and the square for the maximum metric. Thus there is no single shape which simultaneously minimizes a and b. Thus as p varies from $1/n$ to 1, the shape of the optimal solution varies from the shapes given in § 4 to the diamond or square.

Suppose that there was a heuristic operating only on relative frequencies for the discrete case. Then for any value of n, this heuristic essentially provides a template for position of the n records, and the shape of this template is independent of p. The continuous analogue of the solution provided by this template would have to be within an additive constant of optimal. But this is a contradiction, since as argued above, given any shape there is a value of p so that the solution deviates from optimal by a quantity proportional to $\sqrt{n}$.

For the Euclidean metric, of course, the circle simultaneously minimizes a and b.

6. Concluding remarks. This paper adds another example to the growing body of literature [7]–[18] which deals with near-optimal solutions which are computationally more efficient than any known algorithm for creating optimal solutions.

It was surprising to discover that of the three metrics studied, only the Euclidean metric permits a heuristic dependent only on relative frequencies which is within an additive constant of optimal. One wonders if there are other nontrivial metrics with this property.

Algorithms which compute exact solutions appear to be prohibitively exhaustive. Thus it would be interesting to explore other criteria for goodness of a heuristic, e.g., within a fixed percentage of optimal, in the hope that good heuristics would then exist for all metrics. It would also be interesting to study heuristics which use the exact access probabilities but in nonexhaustive ways.

Finally, we remark that the constant in the theorem of § 3 is probably much too large, and we do not have any guess as to the least upper bound.

Appendix A. Proof of Lemma 1. Suppose the lemma is not true. Without loss of generality, we can assume that there exists a straight line L such that one of the following cases would occur:

(A.1) $\quad I(A \cap \omega_1^*) \not\supset (B \cap \omega_1^*), \qquad I(A \cap \omega_0^*) \not\supset (B \cap \omega_0^*),$

(A.2) $\quad I(A \cap \omega_1^*) \not\supset (B \cap \omega_1^*), \qquad I(A \cap \omega_0^*) \supset (B \cap \omega_0^*),$

and

(A.3) $\quad I(A \cap \omega_1^*) \supset (B \cap \omega_1^*), \qquad I(A \cap \omega_0^*) \not\supset (B \cap \omega_0^*),$

where $\not\supset$ means "does not contain". We will show that (A.1) leads to a contradiction. The other two cases can be similarly dealt with. Note that case (A.1) does not rule out $I(B \cap \omega_1^*) \supset (A \cap \omega_1^*)$, $I(B \cap \omega_0^*) \supset (A \cap \omega_0^*)$. But the assumption that the lemma is not true takes care of the objection.

Figure 11 is an illustration of (A.1).

Denote the region HAF by $\mathscr{G}$, $AKGF$ by $\mathscr{A}$, KMG by $\mathscr{C}$, GNF by $\mathscr{C}'$, $BJCE$ by $\mathscr{D}$, JDC by $\mathscr{B}$, CPE by $\mathscr{B}'$, IQR by $\mathscr{F}$ and $QBER$ by $\mathscr{E}$. We will show that

$$\int_{\omega_1^*}\int_{\omega_0^*} > \int_{\mathscr{G}\mathscr{A}\mathscr{C}'}\int_{\mathscr{F}\mathscr{E}\mathscr{D}\mathscr{B}'},$$

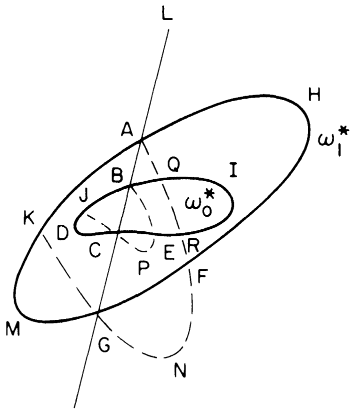


FIG. 11. *Figure for proof of Lemma* 1

hence a contradiction.

$$\int_{\omega_1^*}\int_{\omega_0^*} - \int_{\mathscr{G}\mathscr{A}\mathscr{C}'}\int_{\mathscr{F}\mathscr{E}\mathscr{D}\mathscr{B}'}$$

$$= \int_{\mathscr{G}\mathscr{A}\mathscr{C}}\int_{\mathscr{F}\mathscr{E}\mathscr{D}\mathscr{B}} - \int_{\mathscr{G}\mathscr{A}\mathscr{C}'}\int_{\mathscr{F}\mathscr{E}\mathscr{D}\mathscr{B}'}$$

$$= \int_{\mathscr{G}}\int_{\mathscr{F}} - \int_{\mathscr{G}}\int_{\mathscr{F}} + \int_{\mathscr{G}}\int_{\mathscr{E}\mathscr{D}\mathscr{B}} - \int_{\mathscr{G}}\int_{\mathscr{E}\mathscr{D}\mathscr{B}'} + \int_{\mathscr{A}\mathscr{C}}\int_{\mathscr{F}} - \int_{\mathscr{A}\mathscr{C}'}\int_{\mathscr{F}}$$
$$+ \int_{\mathscr{A}\mathscr{C}}\int_{\mathscr{E}\mathscr{D}\mathscr{B}} - \int_{\mathscr{A}\mathscr{C}'}\int_{\mathscr{E}\mathscr{D}\mathscr{B}'}$$

$$= \int_{\mathscr{G}}\int_{\mathscr{B}} - \int_{\mathscr{G}}\int_{\mathscr{B}'} + \int_{\mathscr{C}}\int_{\mathscr{F}} - \int_{\mathscr{C}'}\int_{\mathscr{F}} + \int_{\mathscr{A}}\int_{\mathscr{B}} - \int_{\mathscr{A}}\int_{\mathscr{B}'}$$
$$+ \int_{\mathscr{C}}\int_{\mathscr{B}} - \int_{\mathscr{C}'}\int_{\mathscr{B}'} + \int_{\mathscr{C}}\int_{\mathscr{E}} - \int_{\mathscr{C}'}\int_{\mathscr{E}} + \int_{\mathscr{C}}\int_{\mathscr{D}} - \int_{\mathscr{C}'}\int_{\mathscr{D}}$$

$$= \int_{\mathscr{G}}\int_{\mathscr{B}} - \int_{\mathscr{G}}\int_{\mathscr{B}'} + \int_{\mathscr{C}}\int_{\mathscr{F}} - \int_{\mathscr{C}'}\int_{\mathscr{F}} + \int_{\mathscr{C}}\int_{\mathscr{E}} - \int_{\mathscr{C}'}\int_{\mathscr{E}}.$$

For each point y in $\mathscr{B}$, let y' in $\mathscr{B}'$ be its image with respect to L. Let x be a point of $\mathscr{G}$. By properties of the Euclidean metric, $d(x, y) > d(x, y')$, for x, y not on L. Therefore, it follows that

$$\int_{\mathscr{G}}\int_{\mathscr{B}} - \int_{\mathscr{G}}\int_{\mathscr{B}'} > 0,$$

and similarly for other pairs of integrals. Thus the result follows.

REFERENCES

[1] G. H. HARDY, J. E. LITTLEWOOD, AND G. PÓLYA, *Inequalities*, Cambridge University Press, Cambridge, England, 1952.

[2] D. D. GROSSMAN AND H. F. SILVERMAN, *Placement of records on a secondary storage device to minimize access time*, J. Assoc. Comput. Mach., 20 (1973), pp. 429–438.

[3] V. R. PRATT, *An N log N algorithm to distribute N records optimally in a sequential access file*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 111–118.

[4] P. C. YUE and C. K. WONG, *On the optimality of the probability ranking scheme in storage applications*, J. Assoc. Comput. Mach., 20 (1973), pp. 624–633.

[5] P. P. BERGMANS, *Minimizing expected travel time on geometrical patterns by optimal probability rearrangements*, Information and Control, 20 (1972), pp. 331–350.

[6] M. HANAN AND J. M. KURTZBERG, *A review of the placement and quadratic assignment problems*, SIAM Rev., 14 (1972), pp. 324–342.

[7] R. L. GRAHAM, *Bounds on multiprocessing anomalies and related packing algorithms*, Proc. Spring Joint Computer Conf. 1972, pp. 205–218.

[8] M. R. GAREY, R. L. GRAHAM, AND J. D. ULLMAN, *Worst-case analysis of memory allocation algorithms*, Proc. 4th Ann. ACM Symp. on Theory of Computing, 1972, pp. 143–150.

[9] C. L. LIU, *Optimal scheduling on multi-processor computing systems*, Proc. 13th Ann. Symp. on Switching and Automata Theory, 1972, pp. 155–160.

[10] D. S. JOHNSON, *Approximation algorithms for combinatorial problems*, Proc. 5th Ann. ACM Symp. on Theory of Computing, 1973, pp. 38–49.

[11] J. NIEVERGELT AND C. K. WONG, *On binary search trees*, Information Processing 71 (Proc. of IFIP Congress), North-Holland Publishing Co., Amsterdam, 1972, pp. 91–98.

[12] C. K. WONG AND D. COPPERSMITH, *A combinatorial problem related to multimodule memory organizations*, J. Assoc. Comput. Mach., 21 (1974), pp. 392–402.

[13] C. K. WONG, C. L. LIU, AND J. APTER, *A drum scheduling algorithm*, Lecture Notes on Computer Science, vol. 2, Springer-Verlag, Berlin, 1973, pp. 267–275.

[14] H. S. STONE AND S. H. FULLER, *On the near-optimality of the shortest-latency-time-first drum scheduling discipline*, Comm. ACM, 16 (1973), pp. 352–353.

[15] J. RISSANEN, *Bounds for weight balanced trees*, IBM J. Res. Develop., 17 (1973), pp. 101–105.

[16] A. K. CHANDRA AND C. K. WONG, *Worst-case analysis of a placement algorithm related to storage allocation*, this Journal, 4 (1975), pp. 249–263.

[17] P. C. YUE AND C. K. WONG, *Near optimal heuristics for an assignment problem in mass storage*, Internat. J. Computer and Information Sciences, to appear.

[18] M. C. EASTON AND C. K. WONG, *The effect of a capacity constraint on the minimal cost of a partition*, J. Assoc. Comput. Mach., to appear.

SIAM J. COMPUT.
Vol. 4, No. 3, September 1975

MANAGING STORAGE FOR EXTENDIBLE ARRAYS*

ARNOLD L. ROSENBERG†

Abstract. Schemes which allocate storage for extendible arrays cannot utilize storage as efficiently as can their nonextendible counterparts. Relative to formal notions of array scheme and (extendible) array realization, a formal way of gauging efficiency of storage utilization by extendible array realizations is proposed; a lower bound of $O(p \cdot (\log p)^{d-1})$, where p is the array size and d the dimensionality, is derived for this measure; and an extendible allocation scheme which achieves this lower bound is exhibited. Certain seminorms on Euclidean spaces can be used to construct extendible array realizations. It is shown that for realizations so constructed, the lower bound on storage utilization efficiency is $O(p^d)$. In the opposite direction, certain restrictions on the patterns of expansions of arrays can be used to circumvent the lower bound: When arrays are constrained to expand according to some fixed finite set of patterns, then one can devise extendible realizations which (a) utilize storage very efficiently ($O(p)$) on arrays which conform to the patterns and (b) approach the general lower bound ($O(p \cdot (\log p)^{d-1})$) on arrays which do not conform. It is not known if this improvement is available for infinite sets of patterns.

Key words. array, array realization, extendible array, storage utilization, storage allocation for arrays

1. Introduction.

Conventional schemes for storing arrays do not admit easy dynamic extension of a stored array. In two dimensions, for instance, the familiar "store-by-row" scheme admits easy adjunction of new rows but only awkward adjunction of a new column. Such asymmetry in extendibility is not inevitable: it is not hard to devise computed-access schemes for storing arrays, which are readily extendible in all directions. (A scheme is said to use *computed access* if it computes the address assigned to a given position of the array as a displacement from the address of position $\langle 1, \cdots, 1 \rangle$.) As we showed in [4], arbitrary extendibility in array realizations does not come without cost. The present paper continues the study begun in [4] of the properties and limitations of extendible array realizations. The current research investigates the cost of extendibility in terms of efficiency of storage utilization.

1.1. Summary of main results.

For ease of exposition, we discuss only two-dimensional arrays in this summary. Any individual array can be stored without "gaps": one merely stores the array in a contiguous block of storage locations of just the right size. An extendible array realization (according to our worldview), however, is not storing an individual array; it is, rather, storing an array and all its potential extensions. One cost of this flexibility is that extendible allocation schemes must inevitably leave gaps when storing some arrays. How big must these gaps be? We show that any extendible array realization must, for each integer p, spread some array with p or fewer positions (an $m \times n$ array has $p = mn$ positions) over at least $\mathscr{U}(p) = \sum_{k=1}^{p} [p/k] > (p/4) \cdot [\log_2 p]$ storage locations. We establish two positive results which take some of the sting out of this lower bound. First, the lower bound is achievable: we present an extendible

* Received by the editors August 8, 1973, and in final revised form September 18, 1974.

† Mathematical Sciences Department, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598. This research was supported in part by the Office of Naval Research under Contract N00014-69-C-0023.

array realization which never spreads any p-position array over more than $\mathscr{U}(p)$ locations. Second, the lower bound can often be overcome! Say that one selects a finite set of *shapes* for arrays. (An *array shape* is an infinite set of arrays specified by a height function $\mathbf{h}$ and a width function $\mathbf{w}$. The "shape" is then all arrays of size $\mathbf{h}(n) \times \mathbf{w}(n)$ for some n. Thus, we have as a shape all square arrays ($\mathbf{h} = \mathbf{w} = \lambda n[n]$), or all $2n \times 3n$ arrays, or all $n \times n^2$ arrays, etc.) Then one can find an extendible array realization which spreads any p-position array of one of these favored shapes over at most $c \cdot p$ storage locations, where the constant c is (approximately) the number of shapes to be favored. In addition, this realization can be designed to be "gentle" on arrays of unfavored shapes, spreading such a p-position array over at most roughly $c \cdot p \cdot \log_2 p$ locations. This realization is, therefore, close to perfect where it counts most and close to minimax optimal everywhere else. We do not know of any realization which favors a nontrivial infinite family of shapes, in the sense that, for each shape $\mathbf{S}_i$ there is an integer c_i such that a p-position array of shape $\mathbf{S}_i$ is spread over at most $c_i \cdot p$ locations; it remains open whether or not such a realization exists.

1.2. Background and related work. Conventional schemes for storing arrays are discussed at length in [2, § 2.2.6]. Our treatment of extendibility in computed-access array storage schemes follows the development in [4]. Here, array realizations are rendered extendible in a particular direction by having them allocate storage (i.e., assign addresses for) an array which is infinite in that direction. Arbitrary extendibility is then modeled by realizations of an *orthant array*, an array whose set of positions is the set of *all* positive integer d-tuples (in d dimensions). The rationale behind this worldview is discussed in [4] and is summarized in § 2.3. This method of modeling extendibility makes the notion of *pairing function* (= a one-to-one function from $N \times N$ to N) very germane to our investigation. Brief discussions of pairing functions frequent the literature of mathematical logic and computability theory; a number of examples of such functions appear in [3, pp. 182, 288ff.].

Extendibility in array realizations can be attained also by abandoning computed-access allocation schemes in favor of either a linking strategy or a hashing scheme. Linked allocation schemes for arrays are described in [2, § 2.2.6]. We know of no systematic study of hashing-based schemes for storing arrays, but an interesting empirical study of such schemes is reported in [1]. The advantage of computed access in array realizations is that such realizations afford one both easy probing of the array (which is inevitably lacking in linked schemes) and easy traversal along, say, rows and columns of the array (which is not present with hashing schemes). The concomitants of easy traversal in extendible computed-access realizations form the subject of an interesting paper by Stockmeyer [7]. In that paper, he considers the effect of easy traversal in extendible realizations on other criteria for assessing the quality of array realizations, notably efficiency of storage utilization.

2. Arrays and their realizations.

2.1. A formal notion of array. We need a formal notion of "array" which emphasizes those aspects of arrays germane to the study of computed-access realizations. In this context, the key to the structure of arrays resides in the familiar coordinate system which pictures a d-dimensional array as being imbedded in the positive orthant of d-dimensional space, with array positions laid on the lattice points. (Consider the name of position a_{ij}.) The domain from which the contents of array positions are chosen is immaterial, providing that we assume—as we always shall—that only one memory location need be assigned to each array position. (The same scheme will allocate storage for an array of integers, of literals, etc.) Thus, our formal notion of array takes the following simple form.[1] Let d be a positive integer.

DEFINITION 2.1. A d-dimensional *array scheme* (*array*, for short) is a set $A = C_1 \times C_2 \times \cdots \times C_d$ of *positions*; each *coordinate set* C_i is either the set N of positive integers or the set $N_n = \{1, \cdots, n\}$ for some $n \in N$. When $C_1 = C_2 = \cdots = C_d = N$ (i.e., $A = N^d$), then A is called the d-dimensional *orthant array* and is denoted Ω_d. For any array scheme A, position $\langle 1, \cdots, 1\rangle \in A$ is called the *base position* of A and is (ambiguously) denoted ε. (Context will assure unambiguity.)

Note that, in accord with convention, we demand "rectangularity" in our arrays; that is, each A is the cross product of its coordinate sets. Anticipating our formal notion of extendibility, we do not constrain our arrays to be finite. In order to orient the reader for subsequent illustrations, we depict the array scheme $A = N_3 \times N_4$ in Fig. 1. Note that the base position of A is in the southwest rather than northwest corner of A in order to emphasize the imbedding in the orthant.

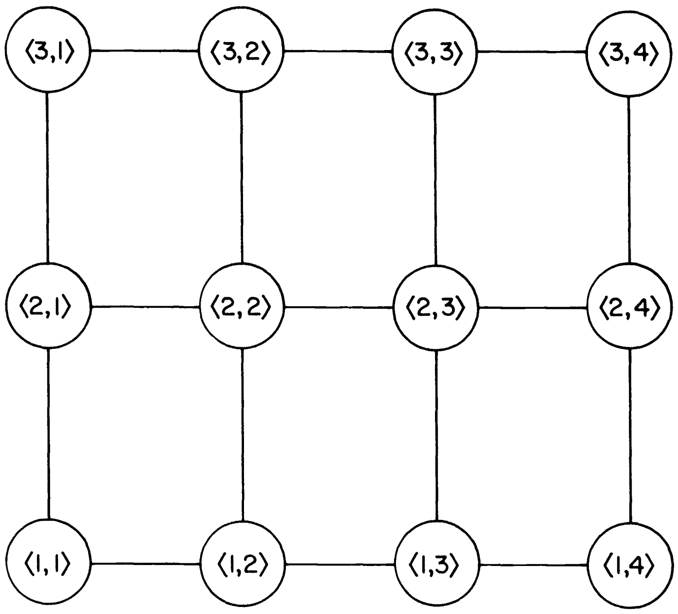


FIG. 1. *The array scheme* $A = N_3 \times N_4$

[1] Because of the questions studied in [4], we needed a more complicated notion of "array scheme" there; this simplified notion suffices here.

2.2. Realizations of array schemes. We seek a formal notion of "computed-access array realization". Informally, we wish to model those realizations which determine the memory location assigned to each position π of an array as a displacement from the location assigned to the base position ε, the displacement being computed from the coordinates of π. To simplify our task, we view the computer in which our array is to be stored as having an infinite random access memory with locations indexed (or "addressed") by natural numbers. Our formal notion of realization (or *allocation scheme*, or *storage map*) can, in this framework, have the following simple form.[2]

DEFINITION 2.2. A *realization* of the array scheme A is a total one-to-one function $\mathbf{r}: A \to N$ such that $\mathbf{r}(\varepsilon) = 1$.

The normalizing condition "$\mathbf{r}(\varepsilon) = 1$" is useful in the sequel but is not indispensable for our investigation. There seems however, to be some aesthetic merit in "beginning" all realizations with the base position.

Note. The existence of pairing functions guarantees that all array schemes can be realized.

2.3. Extendibility in array realizations. At an intuitive level, we adjudge an array realization to be *extendible* (in a given direction) if it can be "easily" converted to a realization of any extension of an array (in that direction), all the while "retaining its computational characteristics". While this statement can have no precise meaning in view of the undefined terms, it should, nonetheless, convey to the reader that we view extendibility as basically some kind of stability in the face of certain changes in the environment. We home in on a formal notion of extendibility by examining two sample realizations of the array $A = N_3 \times N_4$.

Realization 1 (Store by row). For $\langle i, j\rangle \in N_3 \times N_4$, $\mathbf{r}(i, j) = 4(i - 1) + j$.

This realization is easy to compute. It uses storage well, storing the 12 positions of A in "locations" 1 through 12. It is easy to extend along columns; that is, it is easily converted to a realization of any superarray of A of the form $N_k \times N_4$. The extended realization will remain easy to compute; in fact, it will be represented by the same linear form. Moreover, the new realization will also utilize storage well, assigning locations 1 through 12 to the original array A (as did $\mathbf{r}$) and locations 13 through $4k$ to the new positions. In contrast, the realization is not easy to extend along rows. Consider, for example, converting $\mathbf{r}$ to a realization of $N_3 \times N_5$, i.e., A with an additional column. One is faced with two undesirable alternatives (since $\mathbf{r}$ is not one-to-one on $N_3 \times N_5$). One could store the new column (positions $\langle 1, 5\rangle$, $\langle 2, 5\rangle$, $\langle 3, 5\rangle$) in some arbitrary manner, but then the "simplicity" inherent in $\mathbf{r}$ would be lost. Alternatively, one could retain $\mathbf{r}$'s simplicity by using the linear form $5(i - 1) + j$, a 5-column store-by-row scheme, to store the extended array. This latter alternative, though, is hard to implement, since it entails reallocating storage for all but the first row of A. It is thus clear, even at this intuitive level, that $\mathbf{r}$ possesses a certain stability relative to the adjunction of rows that it does not enjoy relative to the adjunction of columns.

[2] As with the notion of array scheme, our investigation in [4] demanded a more complicated notion of realization than that used here.

Realization 2 (Gödel numbering). For $\langle i, j\rangle \in N_3 \times N_4$, $\mathbf{r}_{gn}(i, j) = 2^{i-1}3^{j-1}$.

This realization is bad in almost every respect. It is hard to compute, requiring a number of multiplications which grows with max $\{i, j\}$. It uses storage abysmally, spreading A's 12 positions over 108 storage locations. But, it *is* easily extended along both rows and columns. If one extends A to *any* superarray $N_m \times N_n$, the corresponding extension of $\mathbf{r}_{gn}$ will retain $\mathbf{r}_{gn}$'s exponential form, will leave the positions of A unmoved, and will retain $\mathbf{r}_{gn}$'s pattern of storage utilization. In other words, $\mathbf{r}_{gn}$ enjoys the stability which we are equating with "easy extendibility".

Why is Realization 2 easily extendible? What makes Realization 1 easily extendible along columns but not along rows? Intuitively, it appears that the answer is the same in both cases: the realization in question is a restriction (qua functional restriction) of a realization of a superarray which is *infinite* in precisely the directions of easy extendibility. The alternative to this explanation is to envision some notion of "adaptive" realization which starts out small and grows on demand. This is certainly one view of extendibility; however, it is hard to see how such a growing realization can progress "uniformly" without some infinite model to line up with. Since we are compelled at this time to proceed by intuition, we shall adopt the "infinite superarray" model. While this choice may detract from the generality of our investigation, it cannot lead us astray: finite restrictions of infinite realizations are surely easy to extend along infinite directions. In particular, this view of extendibility leads us to the following definition: *a d-dimensional extendible array realization is a realization of the d-dimensional orthant array* Ω_d.

A strategy for constructing extendible array realizations is presented in [4]; we briefly describe this *shell strategy* since it is useful in the sequel. For $d \in N$, let $\mathbf{s}: N^d \to N$ be any total function which is monotonic in all variables (for all $\pi \in N^d$ and all $\delta \in (N \cup \{0\})^d$, $\mathbf{s}(\pi) \leq \mathbf{s}(\pi + \delta)$)[3] and which has finite preimages (for all $n \in N$, the set $\mathbf{s}^{-1}(n)$ is finite, maybe empty). Call each set $\mathbf{s}^{-1}(n)$ a *shell*, and call $\mathbf{s}$ a *shell index*. Shell indexes, which often arise naturally in computational situations, can be used to construct extendible realizations in the following simple way: design $\mathbf{r}: N^d \to N$ to linearize the partial order induced by $\mathbf{s}$. ($\mathbf{r}$ *linearizes* the shell index $\mathbf{s}$ if $\mathbf{r}(\pi) > \mathbf{r}(\pi')$ whenever $\mathbf{s}(\pi) > \mathbf{s}(\pi')$, for all $\pi, \pi' \in N^d$.) Thus, $\mathbf{r}$ assigns locations, in order, to the shells $\mathbf{s}^{-1}(1)$, $\mathbf{s}^{-1}(2)$, and so on. The following realizations, which are depicted in Figs. 2 and 3, respectively, illustrate the shell strategy.

Realization 3 (Diagonal shells). For $\langle i, j\rangle \in N \times N$, $\mathbf{r}_d(i, j) = \frac{1}{2}(i + j - 1) \cdot (i + j - 2) + j$.

This realization is constructed from the *diagonal* shell index $\mathbf{s}_d(i, j) = i + j$.

Realization 4 (Square shells). For $\langle i, j\rangle \in N \times N$, $\mathbf{r}_s(i, j) = (m - 1)^2 + m + j - i$; $m = \max(i, j)$.

This realization is based on the *square* shell index $\mathbf{s}_s(i, j) = \max(i, j)$.

[3] As we noted in [4], s's monotonicity removes one obstacle to efficient storage use since, for instance, gaps needn't be left while storing a row for later entries in that row.

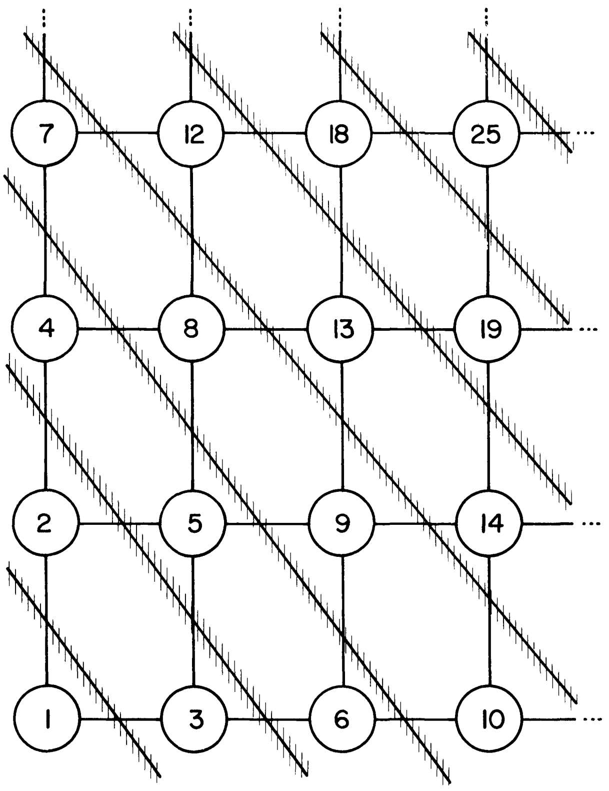


FIG. 2. *Realization* 3 : Ω_2 *stored by diagonal shells*

3. Efficiency of storage utilization.

3.1. A measure of efficiency and the lower bound. Few readers would dispute our contention that Realization 2 (Gödel numbering) utilizes storage very inefficiently. We would probably retain the readers' concurrence when we claim that Realization 3 (diagonal shells) is more efficient in its management of storage than is Realization 2. A comparison between Realization 3 and Realization 4 (square shells), however, is unlikely to lead to a clear-cut decision since they both have the same order of worst-case behavior (although $\mathbf{r}_s$ has certain advantages over $\mathbf{r}_d$ which are discussed in § 4). It is our purpose in this section to propose and study a formal measure of efficiency of storage utilization by extendible array realizations. The measure exposes the behavior of an extendible realization on finite array schemes. We then "diagonalize" over all realizations of Ω_d to obtain from our measure a minimax lower bound on efficiency of storage utilization by d-dimensional extendible array realizations.

DEFINITION 3.1. The *spread* function now defined associates an integer with each extendible array realization $\mathbf{r}$ and integer p. For each realization $\mathbf{r}$ of Ω_d, for each $p \in N$,

$$\mathcal{S}(p;\mathbf{r}) = \max\left\{\mathbf{r}(n_1, \cdots, n_d) | \prod_{i=1}^{d} n_i \leqq p\right\}.$$

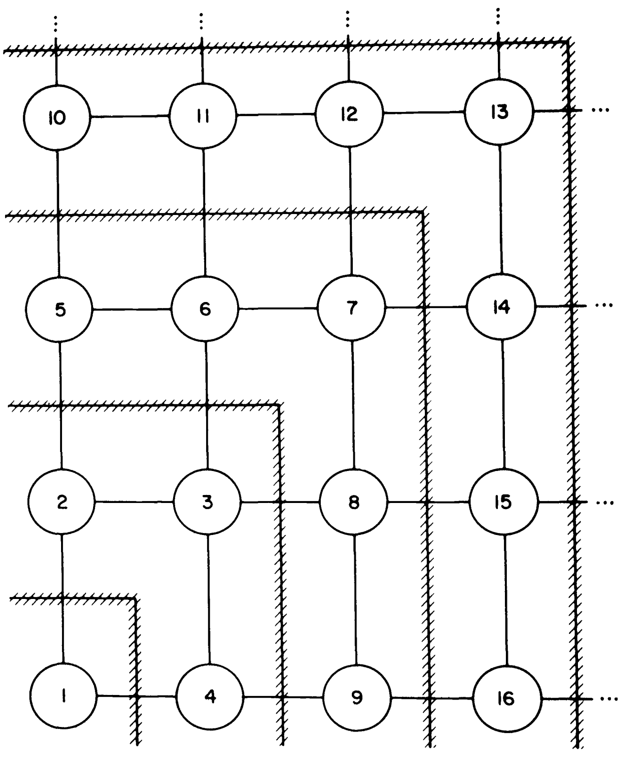


FIG. 3. *Realization 4: Ω_2 stored by square shells*

Informally, $\mathcal{S}(p; \mathbf{r})$ is the highest "address" that $\mathbf{r}$ assigns to any position of an array scheme having p or fewer positions. (Recall that an array is a set of positions, so that, e.g., $A = N_{n_1} \times \cdots \times N_{n_d}$ has $\prod n_i$ positions.) Since every array realization maps ε onto 1, the function $\mathcal{S}$ measures the extent to which $\mathbf{r}$ spreads out p- or fewer-position arrays in memory. Obviously $\mathcal{S}$ is a measure of worst-case behavior.

Returning to our informal discussion, one shows easily that, for all integers p: $\mathcal{S}(p; \mathbf{r}_{gn}) = 3^{p-1}$; $\mathcal{S}(p; \mathbf{r}_s) = p^2$; $\mathcal{S}(p; \mathbf{r}_d) = O(p^2)$.[4] (Look at $\mathbf{r}_{gn}$, $\mathbf{r}_d$ and $\mathbf{r}_s$ on row arrays, that is, arrays of the form $N_1 \times N_p$.) Minsky ([3, pp. 288ff.]) exhibits pairing functions $\mathbf{r}$ with $\mathcal{S}(p; \mathbf{r}) = O(p^{1+e})$ for arbitrarily small $e > 0$. How good can an extendible realization be with respect to this measure of spread?

DEFINITION 3.2. For each dimensionality $d \in N$, for all $p \in N$,

$$\mathcal{U}_d(p) = \min \{\mathcal{S}(p; \mathbf{r}) | \mathbf{r} \text{ realizes } \Omega_d\}.$$

The *storage utilization* function $\mathcal{U}_d$ represents a minimax lower bound on efficiency of storage utilization by d-dimensional extendible array realizations.

[4] By "$g(n) = O(f(n))$" we mean that there exist positive constants K_1 and K_2 such that $K_1 f(n) < g(n) < K_2 f(n)$.

THEOREM 3.1. *For all dimensionalities $d \in N$ and all $p \in N - \{1\}$,*[5]

$$\mathscr{U}_d(p) = \sum_{\substack{\langle k_1, \cdots, k_{d-1}\rangle \in N^{d-1} \\ \text{with } \prod k_i \leqq p}} [p/k_1 \cdots k_{d-1}] = O(p \cdot (\log p)^{d-1}).$$

In particular, $\mathscr{U}_2(p) > (p/4)[\log_2 p]$. Moreover, for each d, a realization $\mathbf{r}$ of Ω_d exists for which $\mathscr{S}(p; \mathbf{r}) \equiv \mathscr{U}_d(p)$.

Proof. Let $d \in N$ and $p \in N - \{1\}$ be arbitrary.

(a) $\mathscr{U}_d(p) \geqq \sum [p/\prod k_i]$. Let $\mathbf{r}$ be any realization of Ω_d. Since $\mathbf{r}$ is one-to-one, $\mathscr{S}(p; \mathbf{r})$ can be no smaller than the number of $\langle h_1, \cdots, h_d\rangle \in N^d$ which are positions of some d-dimensional array with p or fewer positions, i.e., which satisfy $\prod h_i \leqq p$. But $\prod h_i \leqq p$ iff $h_d \leqq [p/\prod_{i=1}^{d-1} h_i]$. Therefore, in the "column" $\{h_1\} \times \cdots \times \{h_{d-1}\} \times N$ of Ω_d, there are precisely $[p/\prod_{i \neq d} h_i]$ such positions. (Intuitively, this is saying that, due to the rectangularity of arrays, all of the columns $\{a_1\} \times \cdots \times \{a_{d-1}\} \times N$ with $a_i \leqq h_i$ must each contribute as many positions to an array as does the column in question.) It follows that $\mathscr{S}(p; \mathbf{r}) \geqq \sum_{k_1 \cdots k_{d-1} \leqq p} [p/\prod k_i]$; hence the same inequality holds for $\mathscr{U}_d(p)$, the smallest of the $\mathscr{S}(p; \mathbf{r})$.

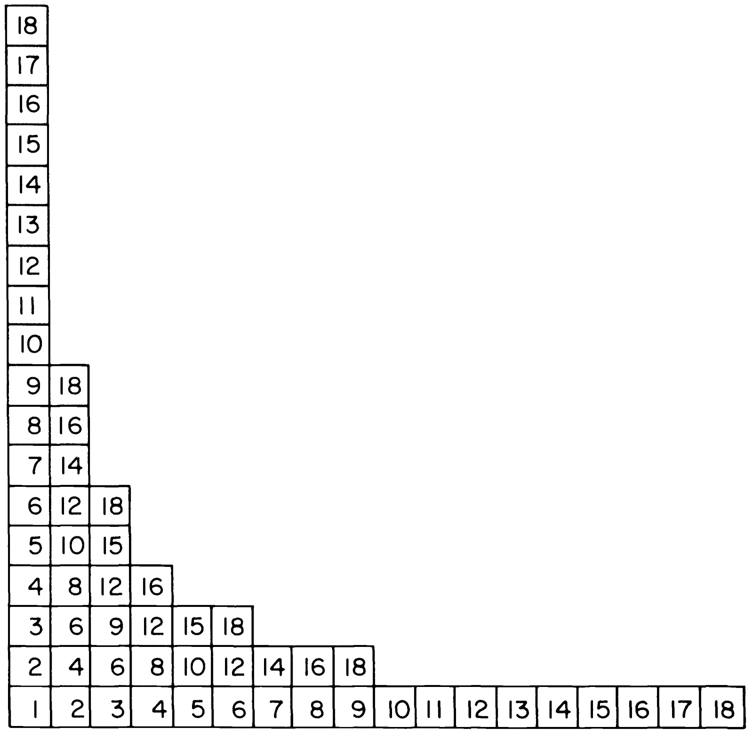


FIG. 4. *The first eighteen hyperbolic shells for Ω_2*

[5] $[x]$ denotes the integer part of real x.

(b) $\mathcal{U}_d(p) \leqq \sum [p/\prod k_i]$. We describe a family of d-dimensional extendible array realizations which assign storage in as conservative a manner as possible. We describe the two-dimensional case in some detail, being more sketchy in the general case. The realizations in question assign storage in shells, first for arrays having just one position (the array N_1^d), then for arrays having exactly two positions (the d arrays $S_1 \times \cdots \times S_d$ with some $S_i = N_2$ and all other $S_j = N_1$), then for those with three positions (d more arrays), then four positions, and so on; see Fig. 4. For each integer p, the shell that must be added to accommodate p-position arrays contains precisely $\delta_d(p)$ (= the number of $\langle n_1, \cdots, n_d\rangle \in N^d$ with $\prod n_i = p$, the number of divisors of p for $d = 2$) positions. To see this, consider any d-element factorization of p, say $p = m_1 \cdots m_d$. Now $N_{m_1} \times \cdots \times N_{m_d}$ is a p-position array; moreover,

$$N_{m_1} \times \cdots \times N_{m_d} = \{\langle m_1, \cdots, m_d\rangle\} \cup (\cup N_{a_1} \times \cdots \times N_{a_d})$$

where the second union is over all tuples $\langle a_1, \cdots, a_d\rangle$ with some $a_i = m_i - 1$ and all other $a_j = m_j$. Of these positions, only position $\langle m_1, \cdots, m_d\rangle$ has not been dealt with (i.e., assigned an address) in an earlier shell. Thus, the described shells arise from the d-dimensional *hyperbolic shell* index $s_h(m_1, \cdots, m_d) = \prod m_i$. One realization which linearizes this shell index is the following; see Fig. 5 for the two-dimensional version.

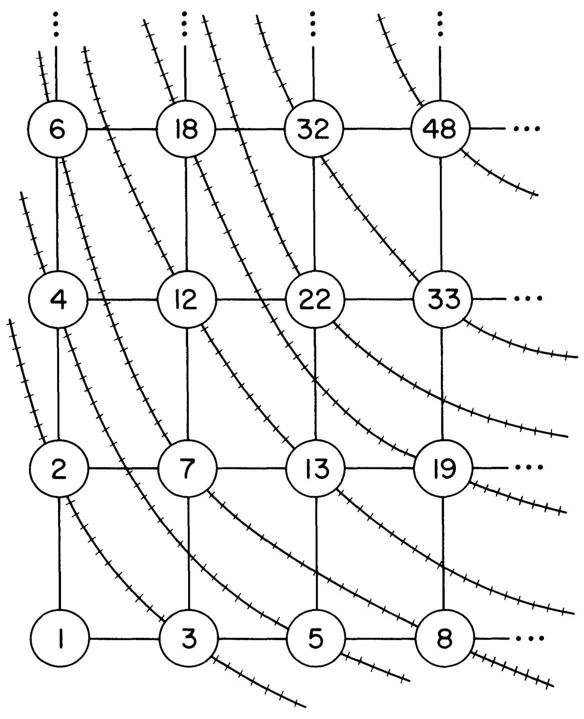


FIG. 5. *Realization* 5 : Ω_2 *stored by hyperbolic shells*

Realization 5 (Hyperbolic shells). For $\langle i_1, \cdots, i_d \rangle \in N^d$,

$$\mathbf{r}_h(i_1, \cdots, i_d) = \sum_{k=1}^{\prod i_j - 1} \delta_d(k)$$
+ (the position of $\langle i_1, \cdots, i_d \rangle$ among factorizations of $\prod i_j$ into d parts, in reverse lexicographic order).

The first term of $\mathbf{r}_h$ is the number of positions in shells lower than $\langle i_1, \cdots, i_d \rangle$'s; the second term (which when $d = 2$ is the number of divisors of $i_1 i_2$ which do not exceed i_2) determines $\langle i_1, \cdots, i_d \rangle$'s position in its shell. It is easy to verify that $\mathscr{S}(p; \mathbf{r}_h) = \mathbf{r}_h(1, 1, \cdots, p) = \sum_{k=1}^{p} \delta_d(k) =$ (the number of $\langle h_1, \cdots, h_d \rangle \in N^d$ with $\prod h_i \leqq p$). Let $k_1, \cdots, k_{d-1}$ be such that $\prod k_i \leqq p$. As in part (a), the number of points in question which reside in the "column" $\{k_1\} \times \cdots \times \{k_{d-1}\} \times N$ of Ω_d is precisely $[p/\prod k_i]$. Moreover, *every* position $\langle h_1, \cdots, h_d \rangle$ with $\prod h_i \leqq p$ must reside in one of these columns. It follows that $\mathscr{S}(p; \mathbf{r}_h) \leqq \sum_{k_1 \cdots k_{d-1} \leqq p} [p/\prod k_i]$; hence, the same inequality must hold for $\mathscr{U}_d(p)$, the smallest of the $\mathscr{S}(p; \mathbf{r})$. The expression given for $\mathbf{r}_h$ is rather cumbersome computationally. For the case $d = 2$, a computationally superior expression for $\mathbf{r}_h$ derives from the following lemma.

LEMMA 3.1. ([7, pp. 159ff.]). *For all* $p \in N$,

$$\sum_{k=1}^{p} \delta_2(k) = \sum_{k=1}^{p} [p/k] = 2 \cdot \sum_{k=1}^{[\sqrt{p}]} [p/k] - [\sqrt{p}]^2.$$

(c) $\mathscr{U}_d(p) = O(p \cdot (\log p)^{d-1})$. We establish the order of $\mathscr{U}_d(p)$ by means of the following lemma.

LEMMA 3.2. *For all* $p \in N - \{1\}$ *and all* $e \in N \cup \{0\}$,

$$\sum_{k_1 k_2 \cdots k_e \leqq p} [p/\prod k_i] = O(p \cdot (\log p)^e).$$

Proof. We estimate the sum above and below using integrals.

Below.

$$\begin{aligned}\sum_{\prod k_i \leqq p} [p/\prod k_i] &> \sum_{\prod k_i \leqq p} (p/\prod k_i - 1)\\ &> \sum_{k_1, \cdots, k_e = 1}^{[p^{1/e}]} (p/\prod k_i - 1)\\ &\geqq p \cdot \left(\sum_{k_1, \cdots, k_e = 1}^{[p^{1/e}]} 1/\prod k_i - 1 \right)\\ &> p \cdot \left(\int_{x_1, \cdots, x_e = 1}^{[p^{1/e}] + 1} \frac{dx_1 \cdots dx_e}{x_1 \cdots x_e} - 1 \right)\\ &= O(p \cdot (\log p)^e).\end{aligned}$$

Above.

$$\begin{aligned}\sum_{\prod k_i \leqq p} [p/\prod k_i] &\leqq \sum_{\prod k_i \leqq p} p/\prod k_i < \sum_{k_1, \cdots, k_e = 1}^{p} p/\prod k_i\\ &< K \cdot p \cdot \int_{x_1, \cdots, x_e = 1}^{p+1} \frac{dx_1 \cdots dx_e}{x_1 \cdots x_e} \quad \text{for some } K > 0\\ &= O(p \cdot (\log p)^e). \quad \square\end{aligned}$$

Our claim (c) follows from parts (a) and (b) and from Lemma 3.2 with $e = d - 1$.

(d) $\mathscr{U}_2(p) > (p/4)[\log_2 p]$. The inequality $\sum [p/k] > (p/4)[\log_2 p]$ can be verified using standard estimates. We shall not verify claim (d) this way, however, since the claim will emerge as Corollary 1 of Theorem 3.2.

This completes the proof of the Theorem. $\square$

The message of Theorem 3.1 can be encapsulated as follows.

COROLLARY. *Let* $\mathbf{r}$ *be any realization of* Ω_2. *For each integer* p, *there is an array* A *with* p *or fewer positions such that* $\max(\mathbf{r}(A)) \geqq \sum [p/k]$; *that is,* $\mathbf{r}$ *spreads* A*'s positions over at least roughly* $p \cdot \log_2 p$ *locations.*

3.2. Two special cases. We examine two special realization problems which are interesting for different reasons. First, we investigate the problem of storing only arrays whose sets of positions have cardinality a power of two. The storage problem for these arrays will lend some insight into the occurrence of the logarithmic gaps predicted by Theorem 3.1. Second, we consider efficiency of storage utilization by extendible realizations which linearize shell indexes arising from norms on Euclidean spaces. (These are discussed briefly in [4, § 4].) Many pairing functions, for instance $\mathbf{r}_s$, and especially the polynomial ones, such as $\mathbf{r}_d$ or the *excess-squares* function $\mathbf{r}_e(i, j) = (i + j)^2 + j - 4$, can be viewed as arising from norms. All of these pairing functions have a spread $\mathscr{S}(p; \mathbf{r})$ which is $O(p^2)$. We shall see in Theorem 3.3 that such a spread is inevitable with norm-based pairing functions.

3.2.1. Powers of two. We define analogues of the spread and utilization functions.

DEFINITIONS 3.3. For each realization $\mathbf{r}$ of Ω_2 and each $p \in N$, define

$$S(p; \mathbf{r}) = \max \{\mathbf{r}(i, j) | \langle i, j\rangle \leqq \text{some } \langle m, n\rangle \text{ with } m \cdot n = p\}^{6}$$

and

$$U(p) = \min \{S(p; \mathbf{r}) | \mathbf{r} \text{ realizes } \Omega_2\}.$$

Informally, $\langle i, j\rangle$ is in some p-position array precisely when $\langle i, j\rangle \leqq$ some $\langle m, n\rangle$ with $m \cdot n = p$; thus, $S(p; \mathbf{r})$ measures the extent to which $\mathbf{r}$ spreads out arrays having *exactly* p positions.

Remarks. (a) For all realizations $\mathbf{r}$ of Ω_2 and all $p \in N$, $S(p; \mathbf{r}) \leqq \mathscr{S}(p; \mathbf{r})$; (b) for all $p \in N$, $U(p) \leqq \mathscr{U}_2(p)$.

THEOREM 3.2. *For all* $k \in N$, $U(2^k) > k \cdot 2^{k-1}$. *Hence*

$$\limsup_{p \to \infty} \frac{U(p)}{p \cdot \log_2 p} > 0.$$

Proof. We employ the same counting technique as in part (a) of the proof of Theorem 3.1.

Let $\mathbf{r}$ be any realization of Ω_2. Since $\mathbf{r}$ is one-to-one, $S(2^k; \mathbf{r})$ must be at least as great as the number of $\langle i, j\rangle \in N^2$ which are positions of some 2^k-position array, i.e., which are $\leqq$ some $\langle 2^a, 2^b\rangle$ with $a + b = k$. In $N \times \{1\}$, there are 2^k such positions. For $b \in N_k$, there are 2^{k-b} such positions in "column" $N \times \{c\}$ for all c in the range $2^{b-1} < c \leqq 2^b$. It therefore follows that $S(2^k; \mathbf{r}) \geqq 2^k$

[6] $\langle a, b\rangle \leqq \langle c, d\rangle$ iff $a \leqq c$ and $b \leqq d$.

$+ \sum_{b=1}^{k} 2^{b-1}2^{k-b} = (k + 2) \cdot 2^{k-1}$. Since $\mathbf{r}$ was arbitrary, this lower bound must hold also for $U(2^k)$, the minimum of the $S(2^k; \mathbf{r})$. □

COROLLARY 1. *For all $p \in N$, $\mathcal{U}_2(p) > (p/4)[\log_2 p]$.*

Proof. The result is clear for $p < 4$. We find the following inequalities for $p \geqq 4$:

$$\begin{aligned} \mathcal{U}_2(p) &\geqq U(2^{[\log_2 p]}) && \text{(since } \mathcal{U}_2 \text{ is monotonic nondecreasing by definition)} \\ &> 2^{[\log_2 p]-1} \cdot [\log_2 p] && \text{(by Theorem 3.2)} \\ &> (p/4)[\log_2 p] && \text{(since } [\log_2 p] > \log_2 p - 1\text{).} \end{aligned}$$ □

Remark. $\mathcal{U}_2$ is a monotonic increasing function; U is not even nondecreasing. The former assertion follows from Theorem 3.1; the latter follows from Theorem 3.2 and the easily verified fact that $U(p) = 2p - 1$ whenever p is a prime.

Finally, we note that Theorem 3.2 ensures that the logarithmic gaps of Theorem 3.1 appear even if one restricts attention to arrays whose height and width are both powers of two.

COROLLARY 2. *Let $\mathbf{r}$ be any realization of Ω_2. For each integer k, there is an array $A = N_{2^a} \times N_{2^b}$ with $a + b = k$ such that* $\max(\mathbf{r}(A)) > k \cdot 2^{k-1}$.

3.2.2. Norm-induced shell indexes. Natural shell indexes are often dictated by external considerations (as was the hyperbolic index, and as will be the indexes discussed in § 4); however, one often encounters problem situations which offer no hints about how to construct a realization. In [4], it was suggested that one could often look for inspiration to those functions on Euclidean spaces which, in the continuous case, are used to determine the shapes of neighborhoods, namely norms. We now show that realizations based on norm-induced shells must exhibit materially inferior worst-case storage characteristics than that of Theorem 3.1. The results reported now were obtained jointly with Larry Stockmeyer.

DEFINITION 3.4. The shell index $\mathbf{s}: N^d \to N$ is *norm-induced* (*is an* **N**-*index*) if

(a) for all $\pi, \rho \in N^d$, $\mathbf{s}(\pi + \rho) \leqq \mathbf{s}(\pi) + \mathbf{s}(\rho)$; and

(b) for all $\pi \in N^d$ and $n \in N$, $\mathbf{s}(n \cdot \pi) = n \cdot \mathbf{s}(\pi)$.

Thus, $\mathbf{s}$ comes from a seminorm which is integer-valued at the lattice points.

THEOREM 3.3. *Let $\mathbf{r}$ be any realization of Ω_d which linearizes an* **N**-*index* $\mathbf{s}$. *For all integers $p \in N$, $\mathcal{S}(p; \mathbf{r}) > [(p - 1)/d]^d$.*

Proof. Let $p \in N$ be arbitrary, and consider the position $(p + d - 1) \cdot \varepsilon \in N^d$. Since $\mathbf{s}$ is an N-index, we have (by Definition 3.4(a))

$$\mathbf{s}((p + d - 1) \cdot \varepsilon) \leqq \mathbf{s}(1, \cdots, 1, p) + \mathbf{s}(1, \cdots, p, 1) + \cdots + \mathbf{s}(p, 1, \cdots, 1).$$

Hence for at least one of the positions π on the right,

$$\mathbf{s}(\pi) \geqq (1/d) \cdot \mathbf{s}((p + d - 1) \cdot \varepsilon) \geqq \mathbf{s}(([(p - 1)/d] + 1) \cdot \varepsilon). \tag{1}$$

This last inequality follows from Definition 3.4(b). Next, letting $q = [(p - 1)/d]$, note that, for all $\rho \in (N_q)^d$, we have

$$\mathbf{s}(\rho) \leqq \mathbf{s}(q \cdot \varepsilon) < \mathbf{s}((q + 1) \cdot \varepsilon) \leqq \mathbf{s}(\pi).$$

These inequalities follow, respectively, from the monotonicity of $\mathbf{s}$, from Definition

3.4(b), and from the inequalities (1). Thus, at least $q^d = [(p-1)/d]^d$ positions of Ω_d reside in shells lower than that of π. Since $\mathbf{r}$ linearizes $\mathbf{s}$, we have $\mathbf{r}(\pi) > [(p-1)/d]^d$. Since π is in some p-position array, the theorem follows. □

The diagonal and square shell indexes are both norm-induced; hence, their linearizations' spread behaviors are as good as possible in terms of growth rate, among norm-induced realizations. We infer from the multi-dimensional versions of these realizations that $\mathscr{U}_d(p) = O(p^d)$ when only norm-induced realizations are considered.

4. On favoring arrays of specified shapes. The results in § 3 have a predominantly negative tone: sizable (and growing gaps are inevitable concomitants of extendible array realizations. Yet the square shell realization $\mathbf{r}_s$ (Realization 4) gives some reason for optimism. Despite $\mathbf{r}_s$'s poor worst-case storage utilization—which, by Theorem 3.1, is materially worse than even the pessimistic lower bound—$\mathbf{r}_s$ manages storage perfectly for square arrays; that is, for all $n \in N$, $\mathbf{r}_s$ maps $(N_n)^2$ one-to-one onto N_{n^2}. Thus, when confronted with an algorithm which uses successively bigger *square* arrays, one has access to an extendible array realization which manages storage perfectly. This section is devoted to showing that analogues to $\mathbf{r}_s$ can be devised for *any* fixed array "shape". Moreover, any fixed number of such perfect-storage-managing shape-favorers can be combined into an extendible realization which favors all of the shapes its constituents do—and also approaches the minimax optimality of the hyperbolic shell realization on arrays of unfavored shapes. This combined realization is not free of gaps even on the favored arrays, but the size of the gaps is bounded (multiplicatively, by approximately the number of shapes to be favored). We close the section with an open question about the existence of realizations which favor infinitely many shapes.

4.1. Realizations which favor a specific shape. We begin by formalizing the notions of "array shape" and "storing an array compactly".

DEFINITIONS 4.1. (a) A d-dimensional *array shape* is a d-tuple of functions $\mathbf{S} = \langle \mathbf{h}_1, \cdots, \mathbf{h}_d \rangle$, where each $\mathbf{h}_i$ is an unbounded nondecreasing total function from N into N, such that for no $n \in N$ do we simultaneously have all $\mathbf{h}_i(n) = \mathbf{h}_i(n+1)$.[7]

(b) The array $N_{n_1} \times \cdots \times N_{n_d}$ *has shape* $\mathbf{S}$ if, for some $k \in N$, $n_i = \mathbf{h}_i(k)$ for all i.

The intention of Definitions 4.1 is that the formal analogue of "array shape" is the specification of the infinite family of arrays having that "shape". The specification is by means of the functions $\mathbf{h}_1, \cdots, \mathbf{h}_d$, where $\mathbf{h}_i(k)$ specifies the *height* along axis i of the kth array having that shape. Thus, the shape $\langle \mathbf{h}_1, \cdots, \mathbf{h}_d \rangle$ specifies the infinite indexed family of arrays $A_1, A_2, \cdots$, where, for each $n \in N$, $A_n = N_{\mathbf{h}_1(n)} \times \cdots \times N_{\mathbf{h}_d(n)}$. The examples in Table 1 should aid the reader's intuition. (Functions in the table are specified using λ-notation.)

DEFINITIONS 4.2. Let $\mathbf{r}$ be a realization of Ω_d.

(a) $\mathbf{r}$ stores the array $A = N_{n_1} \times \cdots \times N_{n_d}$ *with bound* b $(b \in N)$ if $\max(\mathbf{r}(A)) \leqq b \cdot \prod n_i$.

[7] When $d = 2$, we denote the shape by $\mathbf{S} = \langle \mathbf{h}, \mathbf{w} \rangle$ for *height* and *width*.

(b) $\mathbf{r}$ is *b-linear* ($b \in N$) *on array shape* $\mathbf{S}$ if $\mathbf{r}$ stores every array of shape $\mathbf{S}$ with bound b.

(c) $\mathbf{r}$ is *compact on array shape* $\mathbf{S}$ *if* $\mathbf{r}$ *is b-linear on* $\mathbf{S}$ *for some* $b \in N$.
It is assumed, of course, that the array shapes of (*b*, *c*) are *d*-dimensional.

TABLE 1
Some array "shapes" and their formal analogues

"Shape"	Shape: Height	Width
square	$\lambda n[n]$	$\lambda n[n]$
even side square	$\lambda n[2n]$	$\lambda n[2n]$
$n \times (n + 1)$	$\lambda n[n]$	$\lambda n[n + 1]$
$n^2 \times n$	$\lambda n[n^2]$	$\lambda n[n]$

Our earlier remark that $\mathbf{r}_s$ stores square arrays "perfectly" can now be formalized: $\mathbf{r}_s$ is 1-linear on the shape $\langle \lambda n[n], \lambda n[n] \rangle$. The existence of analogues of $\mathbf{r}_s$ for arbitrary shapes can also be stated formally.

THEOREM 4.1. *Let* $\mathbf{S} = \langle \mathbf{h}_1, \cdots, \mathbf{h}_d \rangle$ *be any d-dimensional array shape. There is a realization* $\mathbf{r}$ *of* Ω_d *which is* 1*-linear on* $\mathbf{S}$.

Proof. We design $\mathbf{r}$ to linearize the following shell index $\mathbf{s}$ which exposes the structure of the shape $\mathbf{S}$. For $\langle n_1, \cdots, n_d \rangle \in N^d$,

$$\mathbf{s}(n_1, \cdots, n_d) = \max \{\text{least } k \text{ with } \mathbf{h}_1(k) \geqq n_1; \text{ least } k \text{ with } \mathbf{h}_2(k) \geqq n_2; \cdots; \text{ least } k \text{ with } \mathbf{h}_d(k) \geqq n_d\}.$$

For each $k \in N$, denote by A_k the array scheme $N_{\mathbf{h}_1(k)} \times \cdots \times N_{\mathbf{h}_d(k)}$. Informally, the first shell $\mathbf{s}^{-1}(1)$ is the set of positions of the array scheme A_1; for $k > 1$, the kth shell $\mathbf{s}^{-1}(k)$ comprises those positions of the array scheme A_k which are not positions of the array scheme A_{k-1}. (See Fig. 6.) Any onto realization $\mathbf{r}$ which linearizes $\mathbf{s}$ is easily seen to be 1-linear on $\mathbf{S}$. □

In order to illustrate that the realization constructed in the previous proof need not be computationally prohibitive, we remark that the following realization $\mathbf{r}$ of Ω_2 is 1-linear on the shape $\mathbf{S} = \langle \mathbf{h}, \mathbf{w} \rangle$: let $M = \mathbf{s}(i, j)$; then

$$\mathbf{r}(i, j) = \mathbf{h}(M) \cdot (j - 1) + \mathbf{h}(M - 1) \cdot (\mathbf{w}(M - 1) - \min \{j, \mathbf{w}(M - 1)\}) + i.$$

A derivation of $\mathbf{r}$ is given in the Appendix.

Remark. In [5] (the technical report underlying this paper), we used a more general notion of shape than here, by insisting only that *some* $\mathbf{h}_i$ be unbounded (so, for instance, all arrays of the form $N_1 \times N_n$ would be a shape). The price of this generalization is that the subsequent development gets very awkward; for instance, Theorem 4.1 must be weakened to assert that $\mathbf{r}$ stores all but finitely many arrays of shape $\mathbf{S}$ with bound 2. Since our main concern is with shapes as defined here, we have opted for cleanliness rather than generality. Should the reader be tempted to consider the more general notion of [5], we suggest that the constants used to measure storage bound and linearity be allowed to be rational rather than integral, for in the general case the constant 2 can usually be reduced to $1 + e$ for arbitrarily small $e > 0$.

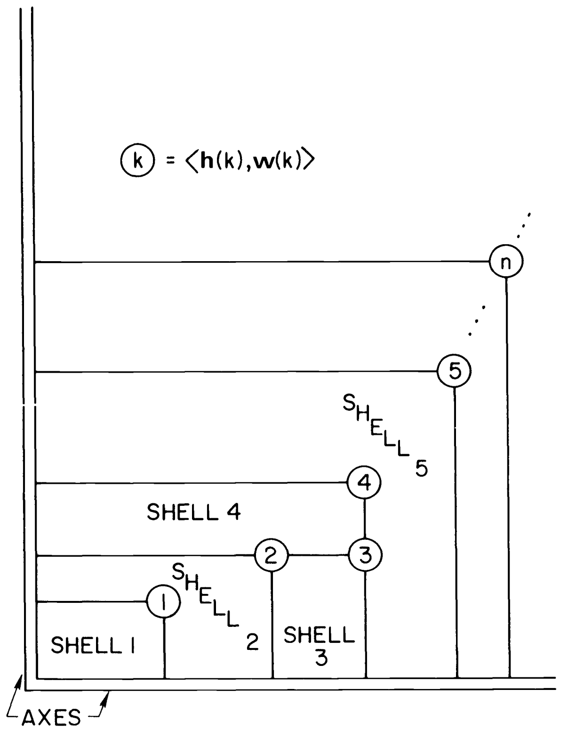


FIG. 6. *A possible layout of shells from the array shape* $\langle \mathbf{h}, \mathbf{w} \rangle$

4.2. Compact handling of sets of shapes. Theorem 4.1 can be combined with the following basic lemma to show that any finite set of shapes can be handled compactly by some realization.

THE DOVETAILING LEMMA. *Let* $\mathbf{r}_1, \mathbf{r}_2, \cdots, \mathbf{r}_k$ *be realizations of* Ω_d. *There exists a realization* $\mathbf{r}$ *of* Ω_d *such that, for all* $\pi \in N^d$,

$$\mathbf{r}(\pi) \leqq k \cdot \min \{\mathbf{r}_l(\pi) | l \in N_k\}.$$

Proof. Define $\mathbf{r}$ by

$$\mathbf{r}(\pi) = \min \{k \cdot (\mathbf{r}_l(\pi) - 1) + l | l \in N_k\}.$$

Now, $\mathbf{r}$ is total since the $\mathbf{r}_l$ are; $\mathbf{r}$ is one-to-one since each $\mathbf{r}_l$ is, and since $\mathbf{r}$ uses a distinct residue class modulo k to "select" from each of the $\mathbf{r}_l$; also, $\mathbf{r}(\varepsilon) = 1$. Intuitively, $\mathbf{r}$ is computed by giving each $\pi \in N^d$ the integers all of the $\mathbf{r}_l$'s would give it—adjusted so that each $\mathbf{r}_l$ uses only integers in the residue class l (mod k)—and then selecting the smallest integer as $\mathbf{r}(\pi)$. □

THEOREM 4.2. *Let* $\mathbf{S}_1, \cdots, \mathbf{S}_k$ *be array shapes. There is a realization* $\mathbf{r}$ *of* Ω_d *which is compact on all of the* $\mathbf{S}_i$. *Moreover,* $\mathbf{r}$ *can be chosen to spread any p-position array which is not of any of the shapes* $\mathbf{S}_i$ *over at most* $O(p \cdot (\log p)^{d-1})$ *locations.*

Proof and discussion. The Theorem can be proved using Theorem 4.1 and the Dovetailing Lemma. The most straightforward method of proof would take the k realizations to be combined and would combine them according to the prescription of the lemma's proof. (We assume that Theorem 4.1 has been used to translate

82	83	84	85	86	87	88	89	90	91
65	66	67	68	69	70	71	72	73	92
50	51	52	53	54	55	56	57	74	93
37	38	39	40	41	42	43	58	75	94
26	27	28	29	30	31	44	59	76	95
17	18	19	20	21	32	45	60	77	96
10	11	12	13	22	33	46	61	78	97
5	6	7	14	23	34	47	62	79	98
2	3	8	15	24	35	48	63	80	99
1	4	9	16	25	36	49	64	81	100

(a) $\mathbf{r}_s$—*square shell realization*

$>10^3$	$>10^3$	$>10^4$	$>10^4$	$>10^5$	$>10^5$	$>10^6$	$>10^6$	$>10^7$	$>10^7$
$>10^3$	$>10^3$	$>10^4$	$>10^4$	$>10^5$	$>10^5$	$>10^6$	$>10^6$	$>10^7$	$>10^7$
768	$>10^3$	$>10^3$	$>10^4$	$>10^4$	$>10^5$	$>10^5$	$>10^6$	$>10^6$	$>10^7$
384	$>10^3$	$>10^3$	$>10^4$	$>10^4$	$>10^4$	$>10^5$	$>10^5$	$>10^6$	$>10^6$
192	576	$>10^3$	$>10^3$	$>10^4$	$>10^4$	$>10^5$	$>10^5$	$>10^6$	$>10^6$
96	288	864	$>10^3$	$>10^3$	$>10^4$	$>10^4$	$>10^5$	$>10^5$	$>10^6$
48	144	432	$>10^3$	$>10^3$	$>10^4$	$>10^4$	$>10^5$	$>10^5$	$>10^5$
24	72	216	648	$>10^3$	$>10^3$	$>10^4$	$>10^4$	$>10^5$	$>10^5$
12	36	108	324	972	$>10^3$	$>10^3$	$>10^4$	$>10^4$	$>10^5$
1	3	5	7	9	11	13	15	17	19

(b) $\mathbf{r}_r$—*stores "row arrays" 2-linearly*

24	62	106	153	204	255	308	363	419	478
21	54	93	135	179	223	270	320	371	420
17	47	79	116	154	193	235	277	321	364
15	39	68	98	129	164	200	236	271	309
11	31	55	80	107	136	165	194	224	256
9	25	43	63	86	108	130	155	180	205
6	18	32	48	64	81	99	117	137	156
4	12	22	33	44	56	69	82	94	109
2	7	13	19	26	34	40	49	57	65
1	3	5	8	10	14	16	20	23	27

(c) $\mathbf{r}_h$—*hyperbolic shell realization*

FIG. 7. *Three realizations to be dovetailed*

shapes into realizations and that the hyperbolic shell realization has been added to the set to be combined, if desired.) If k realizations are so combined, then the number of locations over which a p-position array is spread by the resulting realization is k times the least number of positions it is spread over by any of the k input realizations. In particular, if an input realization stores a given shape c-linearly, then the resulting realization stores that shape ck-linearly. Often, one might wish to favor some k shapes but to *super-favor* certain of them; that is, one is unwilling to suffer the dilation factor of k for those certain shapes. One can easily use the lemma to attain such differential favoring as follows. Take the input realizations and represent them as the leaves of a rooted tree in such a way that (i) each nonleaf of the tree has at least two sons, and (ii) the less favored a realization is, the farther is its leaf from the root. Then, combine the realizations according to the lemma from the bottom up, according to the usual rules for evaluating such a tree. (A node "becomes" a realization, via the lemma, when all of its sons "are" realizations; the realization it "becomes" is the one obtained by combining its sons.) To illustrate these applications of the lemma, consider three realizations: $\mathbf{r}_s$ the square shell realization, $\mathbf{r}_r$ which stores row arrays 2-linearly, and $\mathbf{r}_h$ the hyperbolic shell realization which stores arrays in space roughly $p \cdot \log_2 p$ (See Fig. 7.) The straightforward combination $(\mathbf{r}_s, \mathbf{r}_r, \mathbf{r}_h)$ would store squares 3-linearly, row arrays 6-linearly, and all other arrays in space roughly $3p \cdot \log_2 p$. If one desires to super-favor $\mathbf{r}_s$, say, because most arrays of interest will be square, then one could combine the realizations according to the (encoded) tree $(\mathbf{r}_s, (\mathbf{r}_r, \mathbf{r}_h))$. The realization so obtained would store squares 2-linearly, row arrays 8-linearly, and all other arrays in space roughly $4p \cdot \log_2 p$. The layout of storage under these sample schemes is illustrated schematically in Fig. 8.

As to the constant of linearity of the resulting realization, we see now that we have some flexibility. However, the straightforward combination technique demonstrates that, when k shapes are combined, the resultant constant of linearity for each shape need never exceed k (or $k + 1$ if we wish to be gentle on unfavored arrays). This upper bound may not be optimal, but it cannot be more than a factor of 2 from optimal. To wit, say that one wishes to favor k two-dimensional shapes simultaneously. Assume that each shape contains a distinct 2^{k-1}-position array—there are precisely k of them. Then, since $U(2^{k-1}) > (k - 1) \cdot 2^{k-2}$, the constant of linearity of the resulting scheme must exceed $(k - 1)/2$ for some shape. We have thus proved the following more detailed version of Theorem 4.2.

THEOREM 4.3. (a) *Let* $\mathbf{S}_1, \cdots, \mathbf{S}_k$ *be d-dimensional array shapes. There is a realization* $\mathbf{r}$ *of* Ω_d *which is* $(k + u)$*-linear on shape* $\mathbf{S}_i$ $(i = 1, \cdots, k)$, *where* $u = 1$ *or* 0 *according as* $\mathbf{r}$, *respectively, does or does not achieve the* $O(p \cdot (\log p)^{d-1})$ *bound on unfavored arrays.*

(b) *For each k, there exist array shapes* $\mathbf{S}_1, \cdots, \mathbf{S}_k$ *such that any realization which is compact on all* $\mathbf{S}_i$ *must be c-linear on one of them for some* $c > (k - 1)/2$.

4.3. On favoring infinitely many shapes. The construction used in proving the dovetailing lemma can obviously not be used to combine infinitely many shapes. Barring degenerate cases, we know of no realization which stores infinitely many shapes compactly. We do not even know if such a realization exists.

Open problem. Is there an extendible array realization which is compact on

72	186	250	253	256	259	262	265	268	271
63	162	199	202	205	208	211	214	217	274
51	141	154	157	160	163	166	169	220	277
45	112	115	118	121	124	127	172	223	280
33	79	82	85	88	91	130	175	226	283
27	52	55	58	61	94	133	178	229	286
18	31	34	37	64	97	136	181	232	289
12	16	19	40	67	100	139	184	235	292
4	7	22	43	70	102	120	147	171	195
1	8	14	20	26	32	38	44	50	56

(a) $(\mathbf{r}_s, \mathbf{r}_r, \mathbf{r}_h)$

96	165	167	169	171	173	175	177	179	181
84	131	133	135	137	139	141	143	145	183
68	101	103	105	107	109	111	113	147	185
60	75	77	79	81	83	85	115	149	187
44	53	55	57	59	61	87	117	151	189
33	35	37	39	41	63	89	119	153	191
19	21	23	25	43	65	91	121	155	193
9	11	13	27	45	67	93	123	157	195
3	5	15	29	47	69	95	125	159	197
1	7	17	26	34	42	50	58	66	74

(b) $(\mathbf{r}_s, (\mathbf{r}_r, \mathbf{r}_h))$

FIG. 8. *The result of dovetailing* $\mathbf{r}_s$, $\mathbf{r}_r$, *and* $\mathbf{r}_h$: (a) *equal weighting*; (b) *super-favoring* $\mathbf{r}_s$

each of infinitely many array shapes? In order to bar trivial solutions, we require that each shape in the collection specify infinitely many arrays specified by no other shape. Perhaps this nontriviality condition can be weakened to the following: given any two shapes in the collection, each specifies infinitely many arrays not specified by the other. Any nontrivial solution to this problem would be interesting.

Appendix: A 1-linear storage scheme for the shape $\langle \mathbf{h}, \mathbf{w} \rangle$. We are presented with an array shape $\langle \mathbf{h}, \mathbf{w} \rangle$. From it we infer the shell index $\mathbf{s}$ presented in the proof of Theorem 4.1. We linearize the shells delineated by $\mathbf{s}$ by listing the positions in each shell, in turn, "by columns". Note that the realization we arrive at is computable—in the technical sense—whenever $\mathbf{h}$ and $\mathbf{w}$ are, and that it is not exceedingly more difficult to compute than the worse of those two functions.

The general paradigm for constructing shell realizations is to represent $\mathbf{r}$ as the sum of two functions λ and μ, where, for each position $\pi \in N^2$,

$$\lambda(\pi) = \text{the number of positions in shells lower than } \pi\text{'s};$$

$$\mu(\pi) = \pi\text{'s position in its shell}.$$

(cf. the specification of the hyperbolic shell realization.) We show how to compute $\mathbf{r}(\pi) = \lambda(\pi) + \mu(\pi)$ for one fixed arbitrary $\pi = \langle i, j \rangle$.

Conventions. To avoid repetitive use of long formulas, let

$$M = \mathbf{s}(\pi) \quad \text{and} \quad W = \mathbf{w}(M - 1),$$

so that π lies just outside the array $N_{\mathbf{h}(M-1)} \times N_W$.

Lower shells. For any shape S and any position π, $\lambda(\pi)$ will be the number of positions in the largest array not containing π. For our particular example,

$$\lambda(\pi) = \mathbf{h}(M - 1) \cdot \mathbf{w}(M - 1),$$

using the convention that $\mathbf{h}(0) = \mathbf{w}(0) = 0$.

Current shell. To store the current shell $\mathbf{s}^{-1}(M)$ by columns, we must take care of the portion of the current shell which lies above lower shells and the portion which extends all the way down to the axis. (Either of these portions can be empty, but both cannot; see Fig. 6.) To ease our way into $\mu(\pi)$, we separate these two portions. Note that the former portion is characterized by $j \leqq W$ and the latter by $j > W$.

$$\mu(\pi) = \begin{cases} i - \mathbf{h}(M-1) + (\mathbf{h}(M) - \mathbf{h}(M-1)) \cdot (j-1) & \text{if } j \leqq W, \\ i + W \cdot (\mathbf{h}(M) - \mathbf{h}(M-1)) + \mathbf{h}(M) \cdot (j - W - 1) & \text{if } j > W, \end{cases}$$

which simplifies to

$$\mu(\pi) = \begin{cases} i - \mathbf{h}(M) + (\mathbf{h}(M) - \mathbf{h}(M-1)) \cdot j & \text{if } j \leqq W, \\ i + W \cdot \mathbf{h}(M-1) + \mathbf{h}(M) \cdot (j-1) & \text{if } j > W, \end{cases}$$

which finally simplifies to

$$\mu(\pi) = i + \mathbf{h}(M) \cdot (j-1) - \mathbf{h}(M-1) \cdot \min\{j, W\}.$$

Alternatives to this μ function will readily occur to the reader. For instance, one could store shell by "bands" in analogy to the square-favoring realization (Fig. 3). In specific instances, a variety of strategies should be investigated, with an eye toward simplifying traversal of rows and columns.

Acknowledgments. I am grateful to Shmuel Winograd for many insightful suggestions and for asking whether "added information" might allow one to overcome the $p \cdot \log p$ bound of Theorem 3.1. Several discussions with Larry Stockmeyer concerning this work were invaluable, especially those leading to Theorem 3.3, to the current proof of the dovetailing lemma, and to the suggestion to present the utilization function $\mathscr{U}$ via the spread function $\mathscr{S}$. Conversations with several other colleagues, too numerous to cite individually, were helpful in organizing the current presentation of this work. Special thanks are due the referee for his many valuable suggestions and criticisms.

REFERENCES

[1] E.D.S. DE VILLIERS AND L. B. WILSON, *Hash coding methods for sparse matrices*, Tech. Rep. 45, Computing Lab., Univ. of Newcastle on Tyne, 1973. See also, *Hashing the subscripts of a sparse matrix*, BIT, 14 (1974), pp. 347–358.

[2] D. E. KNUTH, *The Art of Computer Programming I: Fundamental Algorithms*, Addison-Wesley, Reading, Mass., 1968.

[3] M. L. MINSKY, *Computation: Finite and Infinite Machines*, Prentice-Hall, Englewood Cliffs, N.J., 1967.

[4] A. L. ROSENBERG, *Allocating storage for extendible arrays*, J. Assoc. Comput. Mach., 21 (1974), pp. 652–670.

[5] ———, *On storing extendible arrays of specified shapes*, IBM Rep. RC-4450, Yorktown Heights, N.Y., 1973.

[6] W. SIERPINSKI, *Elementary Number Theory*, Panstwowe Wydawnictwo Nankowe, Warsaw, 1964.

[7] L. J. STOCKMEYER, *Extendible array realizations with additive traversal*, IBM Rep. RC-4578, Yorktown Heights, N.Y., 1973.

SIAM J. Comput.
Vol. 4, No. 3, September 1975

RECURSION STRUCTURE SIMPLIFICATION*

H. R. STRONG, JR.,† A. MAGGIOLO-SCHETTINI‡ AND B. K. ROSEN¶

Abstract. This paper discusses a family of algorithms for transforming a recursive program into an equivalent program with a simplified recursion structure. The simplification is performed by integrating copies of certain procedures into the bodies of other procedures. This procedure integration process is analogous to macroexpansion, involving procedure calls rather than macro-operators.

We measure the complexity of the recursion structure in terms of the calling graph of the program. This is a directed graph with nodes representing the procedures of the program and arcs representing the relation "calls". We say that the recursion structure is complex if there is a high degree of cross linkage resulting in a small number of large strongly connected components in the graph; and we say it is simple, if the strongly connected regions are small, preferably containing one node each.

One algorithm in our family is optimal in the sense that it finds a program with the same number of procedures and the maximum number of strongly connected regions obtainable via macroexpansion of procedure calls. We discuss suboptimal algorithms which require less running time, and we discuss open problems related to finding simplification algorithms optimal with respect to a wide range of transformations including the inverse of macroexpansion. In particular, we present a very simple open problem concerning the decidability of the word problem for certain semigroups.

Key words. recursion, copy rule, calling graph, strongly connected component, covering problem

1. Introduction.

We are concerned with the simplification of the recursion structure of programs as a first step toward the systematic removal of recursive procedure calls. There is now a somewhat extensive literature on the subject of recursion removal. Some entry points into this literature include [2], [4], [11], [17]. It has been remarked in several places in this literature that a natural first step in recursion removal is the decomposition of a recursive program into a system of mutual recursions, i.e., recursive subprograms consisting of procedures (or functions, etc.), each of which calls all the others directly or indirectly. Each of these mutual recursions is then to be converted to an equivalent nonrecursive subprogram, and all the subprograms are then recomposed to yield a nonrecursive program equivalent to the original. However, this natural decomposition of the program is not always best for further processing.

The complexity of recursion removal algorithms generally increases dramatically with the number of distinct procedures in the mutual recursion. Moreover, it is often possible to further decompose the program, reducing the size of mutually recursive components. One technique for further decomposition involves the substitution of a modified version of a procedure body for a call in some other procedure body in a way analogous to macroexpansion. In this paper, we present an algorithm for the optimal use of such substitutions.

The reader will need no familiarity with recursive programming and only a minimal familiarity with directed graphs to understand the algorithm. We do not

* Received by the editors December 14, 1973, and in revised form September 11, 1974.

† Computer Sciences Department, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.

‡ Laboratorio di Cibernetica, 80072 Arco Felice, and Istituto di Scienze dell'Informazione dell'Università, 84100 Salerno, Italy. The work of this author was done while visiting with the IBM Thomas J. Watson Research Center.

¶ Computer Sciences Department, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.

discuss application of the algorithm to any specific programming language, although some of our examples are presented in a pseudo-ALGOL.

In § 2 we present a definition of the problem. In § 3 we describe a family of algorithms aimed at its solution. Section 4 designates the optimal algorithm and provides a proof that it does solve the problem. Suboptimal but faster members of the family of algorithms are discussed in § 5. In § 6 we discuss a wider range of program transformations with respect to which our algorithm is strictly suboptimal. Here we present a number of open problems and areas for investigation. Finally, in § 7, we treat one of the most concise and algebraically oriented of these open problems in more detail.

The algorithm presented here is essentially that of [13]. A preliminary report of some of the work presented here appeared as [8].

2. Definition of the problem. No specific definition of the class of recursive programs will be used here; our results are applicable to many different formalizations of recursion, including branched recursion equations [14], recursion with embedded conditionals [9], program schemes with recursion [3] and high level programming languages such as ALGOL 60 and PL/I. Any notion of "recursive program" with some elementary general properties may be used.

We assume that each recursive program has associated with it a finite set $\{f, g, h, \cdots\}$ of "procedures" and a relation "calls" among the procedures. Thus some of the information expressed by writing a program can be displayed in a *calling graph* with a node for each procedure and an arc from f to g whenever f calls g. We will use $\bar{P}$ to represent the calling graph of a program P. We wish to transform a program in ways that preserve whatever the program computes but that simplify the calling graph by increasing the number of strongly connected components (while maintaining the same number of nodes). The resulting small collections of mutually recursive procedures may be more amenable to recursion removal techniques [15], [17] than the original components.

Only one kind of transformation is considered here: the ALGOL 60 copy rule [10] and its analogues in other formalisms. Wherever a procedure g is called from within the body of a procedure f, we may replace the call by an appropriately modified copy of the body of g. (The main modification is to replace formals by actuals. For call-by-value or call-by-reference it is also necessary to add evaluations of the actuals or their addresses.) We do not consider the syntactic details because all we need here is the following general property: for any program P and any procedures f, g and P, we can form a program Q equivalent to P that uses the same procedures and calling relation, except that f calls in Q all the procedures which g calls in P and f no longer calls g in Q unless g called itself in P. To get this, we have to apply the copy rule to all calls on g from within the body of f, even nested ones. If g does not call itself, then no calls on g will be left in the body of f.

Thus the copy rule, when applied to all calls on g in the body of f, maps P to a new program which we call $[f - g]P$, and $[f - g]\bar{P}$ the calling graph of $[f - g]P$ is formed from the calling graph $\bar{P}$ by this operation:

(i) Delete the arc (f, g).

(ii) Add a new arc (f, h) for each h such that the arc (g, h) is in $\bar{P}$ but (f, h)

is not in the graph resulting from (i).

If g calls itself in P, then (f, g) reappears in the calling graph $[f - g]\bar{P}$.

Our problem is to find a finite sequence $[f_n - g_n][f_{n-1} - g_{n-1}] \cdots [f_2 - g_2] \cdot [f_1 - g_1]P$ of these copy rule substitutions that maximizes the number of strongly connected components, at least compared to all other sequences of substitutions. Our solution is applicable to any definition of the substitution operation, so long as substitution preserves the function computed by a program and is reflected in the calling graph by the operation (i), (ii) above.

Consider, for example, the following pseudo-ALGOL program P:

```
procedure q;
begin
  procedure f(x, n);
  real x; integer n;
  begin
    if n = 0 then x := 0 else
      begin
        n := n - 1; x := x + 1;
        g(x, n);
        k(x, n);
        x := x - 1;
        h(x, n)
      end
  end of f;
  procedure g(y, m);
  real y; integer m;
  begin
    if m = 0 then y := 1 else
      begin
        m := m - 1;
        k(y, m);
        h(y, m);
        f(y, m)
      end
  end of g;
  procedure h(z, i);
  real z; integer i;
  begin
    k(z, i);
    i := i + 1;
    k(z, i)
  end of h;
  procedure k(w, j);
  real w; integer j;
  begin
    integer i;
    for i := 1 step 1 until j do g(w, i);
```

```
        j:= j − 1;
        f(w, j)
    end of k;
    f(w, w)
end of q.
```

For our purposes, the particular operations are irrelevant; we are concerned only with the calling graph $\bar{P}$, shown in Fig. 1.

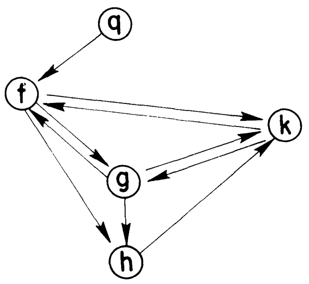


FIG. 1

The relevant part of $[f - h]P$ would be

```
        .
        .
        .
    procedure f(x, n);
        .
        .
        .
        x:= x − 1;
        begin
            k(x, n);
            n:= n + 1;
            k(x, n)
        end
        .
        .
        .
    end;
```

and $[f - h]\bar{P}$ would be as shown in Fig. 2,

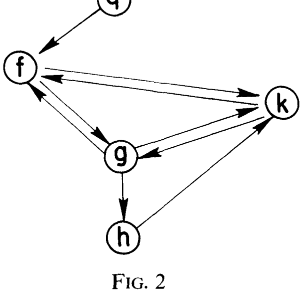


FIG. 2

while $[k - h][k - g][k - f]\bar{P}$ would be as shown in Fig. 3.

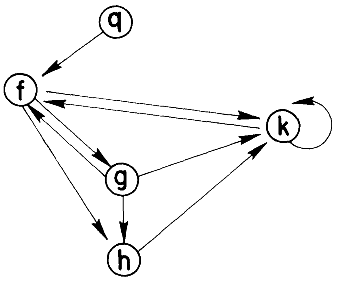


FIG. 3

Note that, in each of these example graphs, there are two components $\{q\}$ and $\{f, g, h, k\}$. Since we are interested in maximizing the number of components, the reader might find it helpful to try to find a sequence of substitutions that increases the number of components before studying the algorithms of the next section. The maximum attainable via these substitutions is four.

3. The algorithm. Given a recursive program P, the algorithm applies a variant of topsort [7] to its calling graph $\bar{P}$, simultaneously choosing a set * of nodes which is a covering of the graph. (By *path* we mean an alternating sequence of nodes and arcs, beginning and ending with nodes, such that each arc is directed from its predecessor to its successor node in the sequence. A *cycle* is a path that begins and ends with the same node. A node *cuts* a cycle if it appears in the sequence. A set of nodes *cuts* a cycle if one of its members does. A *covering* is a set of nodes that cuts all cycles.)

The result of the sorting is a linearization (LIST) of the partial order provided by the original graph with arcs into * nodes deleted. This linearization then determines the order of application of the substitutions. We present the algorithm below in flow chart form. The internal structures mentioned are lists for nodes (LIST and POOL), a queue for nodes (QUEUE), two cells for nodes (FOCUS, NEXT), a cell for arcs (PTR), and a structure for a set of arcs (GRAPH).

There are three phases in the algorithm. From a program we pass to a calling graph together with the number N of nodes in the graph and a list POOL of nodes. For the present, the only constraint on POOL is that it must be a covering.

The second phase manipulates the graph with no reference to the program text. It differs from topsort [7] in that an empty queue is no longer enough for termination. An empty queue together with a NO answer to "N NODES LISTED?" indicates that some cycles remain in the graph. (Only acyclic graphs are allowed in topsort.) To continue sorting we choose a node not already in LIST from POOL. We mark this node with * and delete its inarcs, so that any cycles involving this node will be cut. We queue this node and continue as in [7]. We only dip into POOL when necessary.

The third phase uses LIST and the * marks to direct a series of macro-expansions in the program. We have actually specified a *family* of related algorithms. The members of our family of algorithms can differ in the method of

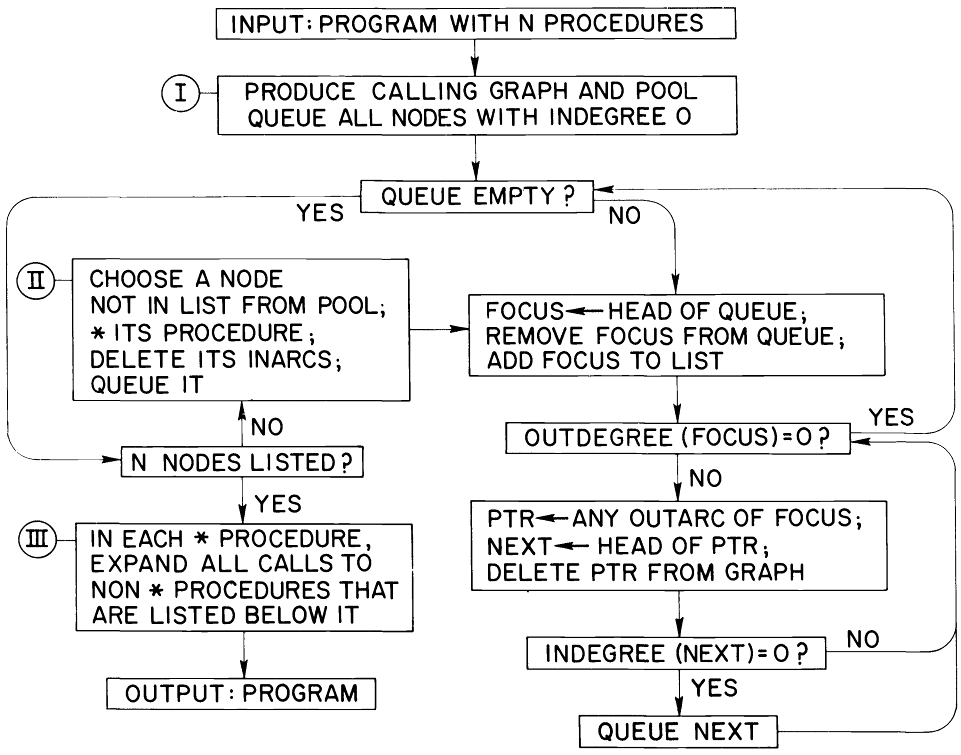


FIG. 4. *A family of algorithms. The list* POOL *of nodes formed in box* I *must include enough nodes to cut all cycles. The methods for forming* POOL *in box* I *and for choosing from* POOL *in box* II *are not specified*

constructing POOL initially and in the method of choosing which node in POOL to use in box II.

As an illustration we apply our algorithm to the example P of the last section. Alphabetical order will be used as the method of choice.

At point I, q is put on the queue and nodes f, g are put into POOL. The queue is not empty; so q is placed in FOCUS, the queue is emptied, and q is placed at the top of LIST. The outdegree of q is not zero; so (q, f) is placed in PTR, f is placed in NEXT and the arc (q, f) is deleted. We proceed to point II, at which f is chosen and its inarcs, (k, f) and (g, f), are deleted. After placing f on LIST and in FOCUS, we place (f, g) in PTR (using alphabetical order again to choose the outarcs), place g in NEXT, and delete (f, g). Since the indegree of g is still not zero, we place h in NEXT and delete (f, h). Similarly, we delete (f, k). Now, with outdegree $(f) = 0$, we return to point II and choose g, deleting (k, g). With g in FOCUS and h in NEXT, we delete (g, h) and queue h. After deleting (g, k), we place h in FOCUS, delete (h, k) and queue k.

At point III, LIST contains q, f^*, g^*, h, k. Thus the instructions at point III are to produce $[g - k][g - h][f - k][f - h]P$. The calling graph $[g - k][g - h]$ $\cdot [f - k][f - h]\bar{P}$ is shown in Fig. 5. The only changes to P are in the bodies of procedures f and g. But now the calling graph has four components $\{q\}$, $\{f, g\}$, $\{h\}$, and $\{k\}$. If the only entry to P is a call to q, then the structure can be simplified

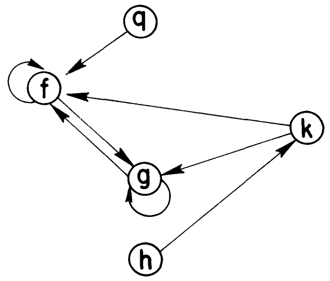


FIG. 5

further by deleting h and k. However, for the rest of this paper we will ignore this kind of simplification, assuming that any procedure of the program is a potential entry point.

4. The solution. The following results are concerned with the application of the algorithm to any program P. By "the algorithm" we really mean any algorithm in the family specified by Fig. 4.

LEMMA 4.1. *At any time when box* II *is entered during a computation,* GRAPH *contains a cycle.*

Proof. The assertion

(1) ALL NODES OF INDEGREE 0 ARE IN QUEUE OR IN LIST

is true on exit from the box I. The operations which falsify (1) are promptly followed by operations which make it true again, and (1) is true whenever box II is entered.

Let box II be entered at time t. By (1) and emptiness of QUEUE, there is a node f_0 which is not yet listed and which has positive indegree. Given f_i for any nonnegative integer i, if f_i has positive indegree we choose f_{i+1} such that (f_{i+1}, f_i) is in GRAPH at time t.

If the sequence $(f_0, f_1, f_2, \cdots)$ is infinite, then GRAPH has a cycle. Suppose instead that the sequence is finite, so some $i > 0$ has f_i of indegree 0. We will derive a contradiction.

By (1), f_i is in LIST at time t. Therefore f_i was added to LIST from FOCUS at a time prior to t. Between that time and t, the loop governed by "OUTDEGREE (FOCUS) = 0?" deleted (f_i, f_{i-1}) from GRAPH. But no arcs are added, so (f_i, f_{i-1}) is missing from GRAPH at time t. □

LEMMA 4.2. *The algorithm terminates normally.*

Proof. An abnormal termination is an attempt to do something impossible such as dividing by zero. The algorithm contains one box that might be impossible to execute at some point in a computation: box II presupposes that POOL has at least one more node not yet in LIST. By Lemma 4.1, GRAPH has cycles when box II is entered. Since POOL includes a covering C on exit from box I, there must be nodes in C with positive indegree when box II is entered. These nodes must be not in LIST and still in POOL, so execution of box II is indeed possible. Therefore, any terminating computation terminates normally.

In the flowchart presentation of the algorithm, the nodes corresponding to the tests "QUEUE EMPTY" and "OUTDEGREE(FOCUS) = 0" cut all cycles of the flowchart. On each return to "QUEUE EMPTY", the number of nodes not yet on LIST has been decreased by one. On each return to "OUTDEGREE(FOCUS) = 0" without passing through "QUEUE EMPTY", the number of arcs not yet deleted has been decreased by one. Thus there can be at most N returns to "QUEUE EMPTY" and at most $N + E$ returns to "OUTDEGREE(FOCUS) = 0", where there are N nodes and E edges in the original graph. Let c be the maximum length for cycle free paths in the flowchart. Then the algorithm terminates in no more than $2cN + cE + c$ steps □

For nodes or procedures (we use these terms interchangeably), we define an ordering: $g > f$ if g appears ahead of f on LIST. Thus, in our example, $q > f > g > h > k$. We also write $f \leqq g$ if f and g refer to the same node or $g > f$.

LEMMA 4.3. *If $f \leqq g$ and f calls g in P, then g is in* *.

Proof. The arc (f, g) must be deleted before g can be added to LIST. The only ways the arc (f, g) could be deleted during the operation of the algorithm are at point II or while f is in FOCUS. But since g appears earlier on LIST than f, all inarcs to g have been deleted before f is in FOCUS. □

COROLLARY 4.4. *The set * is a covering of $\bar{P}$.*

Proof. Because $>$ is transitive and no node h in a cycle can have $h > h$, at least one arc (f, g) in a cycle must have $f \leqq g$. Therefore g is in *. □

We write $\mathbf{A}P$ and $\mathbf{A}\bar{P}$ for the program and calling graph resulting from the application of the algorithm.

LEMMA 4.5. *Each non * node forms its own component in $\mathbf{A}\bar{P}$.*

Proof. By Lemma 4.3, each procedure in P calls only * procedures and procedures below it on LIST. At point III, the arcs corresponding to substitutions are those from * nodes to non * nodes in order of appearance on LIST. When $[f - g]$ is performed, the only arcs introduced are those to elements of * and nodes $< g$. Thus in $\mathbf{A}\bar{P}$ there are no arcs from * nodes to non * nodes. Consider a component of $\mathbf{A}\bar{P}$ with at least two nodes. It must contain nodes f and g and an arc (f, g) with $f \leqq g$. The algorithm does not alter arcs between non * nodes; so, if f and g were both non *, then f would call g in P, contradicting Lemma 4.3. Hence, at least one of the nodes must be in *. But, since there are no arcs from the node to non * nodes, the whole component must be in *. □

The following result is concerned with any sequence $\mathbf{B}$ of substitutions, not necessarily following the algorithm. Recall that a cycle is a sequence of nodes and arcs. A *prime cycle* is a cycle in which no proper contiguous subsequence is a cycle. A set of nodes cuts all cycles if and only if it cuts all prime cycles.

Consider the effect of the substitution $[f - g]$ on a prime cycle. For it to have any effect, the arc (f, g) must be in the cycle with $f \neq g$. Each occurrence of (f, g) in the cycle appears in a subsequence of the form f, (f, g), g, (g, h), h, with $h \neq g$. (If (f, g) is the last arc in a cycle from g to g, we take (g, h) to be the first arc in the cycle.) For each occurrence of (f, g), we replace this subsequence by the corresponding sequence f, (f, h), h to form the *descendant* prime cycle in $[f - g]\bar{P}$. (Note that if g begins the original cycle, then h will begin the new cycle.) The nodes of a descendant prime cycle are always a nonempty subset of the nodes of its ancestor.

LEMMA 4.6. *After any sequence* **B** *of substitutions, for each component* C *of* $\bar{P}$, *there is a component* D *of* $\mathbf{B}\bar{P}$ *which cuts all the cycles of* C *when viewed as a subset of the nodes of* $\bar{P}$.

Proof. Consider any two prime cycles α and β of C. Let $\mathbf{B}\alpha$ and $\mathbf{B}\beta$ be their descendant prime cycles in $\mathbf{B}\bar{P}$, so that $\mathbf{B}\alpha$ and $\mathbf{B}\beta$ lie in unique components $D\alpha$ and $D\beta$ of $\mathbf{B}\bar{P}$. It will suffice to show that $D\alpha = D\beta$, since then all descendants of prime cycles will lie in the same component D.

By symmetry, we can show that $D\alpha = D\beta$ by proving the existence of a path in $\mathbf{B}\bar{P}$ from a node of $\mathbf{B}\alpha$ to a node of $\mathbf{B}\beta$. To prove this by induction on the length of **B**, we need only consider the case $\mathbf{B} = [f - g]$.

Choose nodes a in $[f - g]\alpha$ and b in $[f - g]\beta$, with $b \neq g$ unless β is a self-loop $g, (g, g), g$. Let γ be a minimal length path from a to b in $\bar{P}$. If γ is also a path in $[f - g]\bar{P}$, there is nothing more to prove. Otherwise, γ must include an arc missing from $[f - g]\bar{P}$. Therefore, $[f - g]\bar{P}$ lacks (f, g) and $\bar{P}$ lacks (g, g). Therefore $b \neq g$ and γ has the form $\delta, (f, g), g, (g, h), \varepsilon$ for some path δ from a to f, node h, and path ε from h to b. Null paths are allowed, of course. By minimality of γ, the arc (f, g) does not appear in δ or ε. Therefore, $\delta, (f, h), \varepsilon$ is a path in $[f - g]\bar{P}$ from a to b. □

THEOREM 4.7. *Consider any algorithm* **A** *in the family such that the starred nodes form a covering of minimum cardinality. The algorithm* **A** *produces a program* $\mathbf{A}P$ *with a calling graph* $\mathbf{A}\bar{P}$ *which has the maximum number of components obtainable via substitutions.*

Proof. If there is no path from a to b in $\bar{P}$, then no sequence of substitutions can provide one. Thus the result of any sequence of substitutions is a refinement of the component structure: components are divided, but never coalesced.

Consider the effect of **A** on a single component C of $\bar{P}$. (Assume C contains at least one cycle.) If we ignore arcs into and out of C, the action of **A** on C is the same whether it is operating on all of $\bar{P}$ or on C only. The intersection of * with C is a minimal set of nodes which cuts all cycles of C. By Lemma 4.6, there is a component D of $\mathbf{A}C$ which cuts all the cycles of C.

Now there are two cases to consider. If D meets *, then D is included in * by Lemma 4.5, and so D is exactly the intersection of * with C by minimality. If D does not meet *, then $|D| = 1$ by Lemma 4.5, and so the intersection of * with C has just one member by minimality. In both cases, the intersection of * with C has $|D|$ members. Thus the number of components of $\mathbf{A}\bar{P}$ in C is one plus the number of non * nodes of C. By Lemma 4.6, this number is the maximum obtainable, since any component which cuts the cycles of C has at least as many nodes as the intersection of the * nodes with C. □

Note that Theorem 4.7 deals with any version of the algorithm that obtains a minimum cardinality covering, regardless of how this is done. One way to do this is to start with a listing of such a covering when forming POOL in box I as follows.

COROLLARY 4.8. *If* POOL *is formed by listing a covering of minimum cardinality, the algorithm stars exactly the nodes of this covering.*

Proof. This follows from Corollary 4.4 and the inclusion of * in POOL. □

5. Running time and suboptimality. In § 3 we specified a family of algorithms. Except perhaps for boxes I and II, the graph operations and tests in the flowchart can readily be programmed so as to require amounts of time essentially independent of the calling graph. Here we will also assume that the method of choice in box II is simply to take the next node in POOL not already in LIST. Thus we do not consider members of the family that reorder POOL dynamically.

Now we may assume that *all* boxes strictly between I and III in the flowchart have running times independent of the calling graph. From the proof of Lemma 4.2, it follows that this portion of the algorithm (the portion that does not involve the program text) has a running time linear in the size (N, E) of the calling graph.

The linearity just established would be of little consequence if the formation of POOL required more than linear time. Consider two extreme methods of forming POOL. The first is to list all the nodes in an arbitrary order. This is rapid but may lead to absurdly many * nodes. We cannot apply Theorem 4.7. The second method is to find a minimum cardinality covering and list it in some arbitrary order. By Corollary 4.8, the * nodes will be exactly this covering. Theorem 4.7 establishes optimality, but it is highly unlikely that POOL can be generated rapidly. The problem of finding a minimum cardinality covering is polynomial complete [6].

It will be more practical to form POOL by listing a small but perhaps nonminimal covering in linear time. The algorithm will then * only those members of the covering needed to cut cycles for topsort. Thus * will be a small covering obtained in linear time. Will the recursion structure obtained be close to optimal whenever the cardinality of * is close to minimal? Theorem 4.7 does not apply directly.

To analyze the situation, we consider a program P whose calling graph $\bar{P}$ has N nodes, K components, and M_0 as the minimum number of nodes in a covering.

LEMMA 5.1. *The maximum number of components obtainable via substitutions is* $K + N - M_0$.

Proof. Let algorithm $\mathbf{A}_{\text{opt}}$ in the family form POOL by finding and listing a minimum cardinality covering. The maximum number of components obtainable by substitutions does exist and is the number of components of $\mathbf{A}_{\text{opt}}\bar{P}$, by Theorem 4.7.

Let C be a component of $\bar{P}$ and let Q be the intersection of * with C, so that C contributes $1 + |C - Q| = 1 + |C| - |Q|$ components to $\mathbf{A}_{\text{opt}}\bar{P}$ in the proof of Theorem 4.7. Summing over all C leads to $K + N - M_0$. □

LEMMA 5.2. *Let M be the number of nodes starred when algorithm* $\mathbf{A}$ *is applied to P. The number of components of* $\mathbf{A}\bar{P}$ *is* $K + N - M$.

Proof. Let R be a program like P except that each * procedure in P calls itself as well as the procedures it calls in P. Thus $\bar{R}$ has enough self loops to make * be a minimum cardinality covering but is otherwise like $\bar{P}$. In particular, $K(\bar{R}) = K$, $N(\bar{R}) = N$, and $M_0(\bar{R}) = M$. By Lemma 5.1, $\mathbf{A}_{\text{opt}}\bar{R}$ has $K + N - M$ components. Choosing $\mathbf{A}_{\text{opt}}$ so as to treat R the way $\mathbf{A}$ treats P (except for carrying along the extra self loops), we find that $\mathbf{A}\bar{P}$ has $K + N - M$ components. □

THEOREM 5.3. *Let* MAX *be the maximum number of components obtainable via substitutions in P and let* ACT *be the actual number obtained when algorithm* $\mathbf{A}$ *is applied to P. Let M_0 be the minimum number of nodes required to cut all cycles in $\bar{P}$ and let M be the number of * nodes in* $\mathbf{A}\bar{P}$. *Then* $\text{MAX} - \text{ACT} = M - M_0$.

Proof. By Lemmas 5.1 and 5.2, MAX − ACT $= (K + N - M_0) - (K + N - M) = M - M_0$. □

These considerations come into sharper focus in the context where each program has but one entry point and is *not* guaranteed to call each procedure it declares. (This is indeed the application we are most concerned with.) Now the graph of interest is the subgraph formed by considering only those nodes in a larger graph which are accessible from a single node MAIN (representing entry to the program). In order to obtain this subgraph we would, of course, perform a depth-first search [16].

During a depth-first search, it is easy to list the nodes which have inarcs from themselves or from their descendants in the search [16]. These nodes form a covering which is reasonably small in many examples and which is obtained in linear time. Still within linear time, we can order POOL so that the nodes with inarcs from themselves precede all the other nodes, and the other nodes appear in the reverse of the order in which they were last visited during the depth-first search. This is called "rENDORDER" in [5].

Using the above method of forming POOL, the graph manipulations of the algorithm have been programmed in PL/I and subjected to some preliminary experiments. Postponing a detailed discussion until further experiments have been performed, we remark on two easily proved results suggested by the simple examples considered so far. For any positive integer X, we can construct a graph such that depth-first search uses X nodes to cut all cycles, but only one node is starred. We can also construct a graph such that X nodes are starred, but only one node is needed to cut all cycles.

6. A more general problem. The substitution of this paper has a definite one-way flavor. Once a particular substitution is made, the results cannot generally be undone by further substitution. For example, if we choose $[f - g]$ to operate on Fig. 6, we produce Fig. 7, which no further substitutions will change. However, by choosing $[g - f]$ instead, we obtain Fig. 8 with one more component.

While no substitution will reverse the action of $[f - g]$, the inverse transformation is applicable to programs. This information consists of replacing an appropriately modified copy of a procedure body by a call to that procedure. We cannot expect to be able to apply this transformation often, at least in a mechanical way. But it does correspond to a programming technique. We might hope to tell the programmer some of the effects in terms of recursion structure. Thus we would like to discover an algorithm for an optimal sequence of

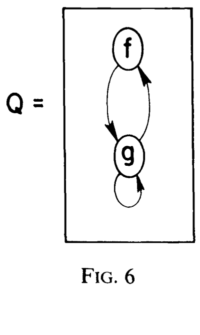


FIG. 6

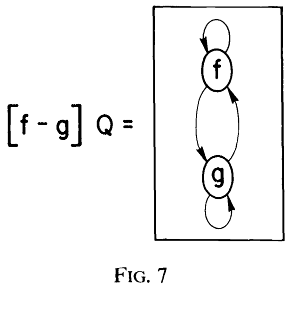


FIG. 7

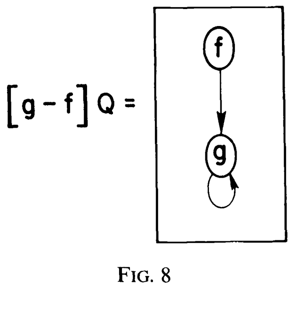


FIG. 8

transformations in the wider context including substitutions and their inverses, deciding when such an inverse is applicable and when it should be performed.

We know that our optimal algorithm is suboptimal in this wider context. It can do nothing for a program corresponding to $[f - g]Q$ in the previous example; but the sequence, $[f - g]^{-1}$ followed by $[g - f]$, will produce $[g - f]Q$. We have not been able to obtain an optimal algorithm. We have even encountered difficulty defining optimality in this context. For suppose that as a result of various transformations, two procedures become identical (except for names). We would certainly want to identify them and this would simplify recursion structure while possibly reducing the number of components. It might be that the sequence of transformations effecting this simplification would not otherwise have been optimal.

Temporarily assuming away the identification difficulty, one might suppose that we had only to perform all possible inverse transformations before applying our algorithm. However, it is possible that two inverse transformations be both applicable and incompatible, i.e., the application of one precluding that of the other. Moreover, it is possible for some inverse transformation to become (nontrivially) applicable only after certain substitutions have been performed. We have not found any way to bound the implied search tree.

These questions also force us to look more closely at the substitution transformation itself. In our algorithm, the only procedures modified are the * procedures, and the only procedures used to replace calls in the * procedures are non * procedures. But in general we might perform a sequence of the form $[f - g]$ (modifying f) followed by $[h - f]$ (modifying h with copies of f). The question then arises of whether to use the modified or original f in making the substitution. Either choice has its supporting arguments, especially in the context of inverse transformations where the modified f may be shorter than the original.

In this paper we leave this entire area open for further investigation.

7. A specific open problem. In this section we study a specific algebraically oriented problem related to the identification problem of the last section.

A standard result of recursive function theory tells us that we cannot expect to decide whether two procedures perform identically. Thus we will take the simplifying assumption that we would only wish to identify procedures that were syntactically identical (except for some trivially decidable name substitutions). Our problem is to decide whether two procedures can become identical via substitutions and their inverses. If the answer turned out affirmative, we could consider making all such identifications in advance and define an algorithm to be optimal if it maximized the number of components, as in the earlier sections of this paper. Otherwise, we would be stuck with a much more cumbersome definition and less hope of success.

We now greatly oversimplify our problem to obtain a semigroup word problem whose solution is likely to indicate that of the identification problem. The oversimplifications include treating each procedure as if it were simply a string of procedure calls. Substitution is then the replacement of one of these calls by its corresponding string; the inverse transformation is the contraction of some substring to an appropriate call. Thus, corresponding to a program, we have a semigroup for which the carrier alphabet is the set of procedures and which is presented by a set of equivalences of the form: one procedure is equivalent to a string of others. Our problem is then to decide whether two procedures (elements of the alphabet) are equivalent.

Consider, for example, the following semigroup on two generators a and b:

$$a \longleftrightarrow abaa, \qquad b \longleftrightarrow aaba.$$

We have the following expansions (substitutions) and contractions:

$$a \rightarrow abaa \rightarrow aaabaaa \leftarrow aaaa, \qquad b \rightarrow aaba \rightarrow aaaabaa \leftarrow aaaa.$$

Here the forward arrow indicates an expansion while the reverse arrow indicates a contraction. Thus a and b are equivalent in the semigroup.

If the semigroups corresponding to programs were arbitrary, then our word problem would be recursively unsolvable [12]. However, the restriction *that each generator appear as the left side of at most one equivalence and that there be no other relations on the semigroups* leaves the problem open.

In the example above, we could make all the expansions before all the contractions as follows:

$$a \rightarrow abaa \rightarrow a(aaba)aa \rightarrow aa(abaa)(abaa) \leftarrow aa(aaba)a \leftarrow aaba \leftarrow b.$$

This process generalizes to the following lemma.

LEMMA 7.1. *If there is a sequence of expansions and contractions connecting two words of one of these restricted semigroups, then there is a sequence connecting the words in which all expansions appear before all contractions.*

Proof. If an expansion immediately follows a contraction and does not reverse its effect, then it must expand some other element than the one contracted to. Thus the order of application can be reversed. □

COROLLARY 7.2. *Our word problem is equivalent to the intersection problem for a pair of context-free grammars which differ only in start symbol and are restricted to have at most one rule per nonterminal (except that each nonterminal has an additional rule of the form $A \rightarrow a$, where a is a terminal distinct from all the terminals corresponding to other nonterminals).*

8. Conclusion.

We have established a family of algorithms for simplifying recursion structure by means of "copy rule"-type expansions of procedure calls. We can apply the algorithms to other contexts. For example, within one procedure, we could expand gotos rather than calls in order to produce a program more suitable for storage in a paged memory.

The algorithms in the family vary according to the method for choosing a collection POOL of candidates for inclusion in a covering (set of nodes cutting all cycles) and the method for choosing nodes from POOL when they are needed. The collection POOL must at least contain a covering. The optimal algorithm in the family uses a minimum cardinality covering.

It appears likely that any method for finding a minimum cardinality covering of a directed graph must run in exponential time. Thus it is probably impractical to run our optimal algorithm on large programs. We have discussed one linear time heuristic method for the choice of POOL. Our family of algorithms has the advantage of operating with any such heuristic and producing a covering without listing all the prime cycles in advance.

Of the open areas presented in the later sections, we found the word problem of § 7 especially intriguing. We conjecture that the problem is solvable, but can offer as evidence only the failure of a few standard techniques for proving unsolvability.

Acknowledgments. The authors would like to acknowledge the assistance of Kaaren-Ann C. Fanti in the preparation of this manuscript and the helpful comments of R. E. Miller and the referees.

REFERENCES

[1] F. E. Allen and J. Cocke, *Graph-theoretic constructs for program control flow analysis*, RC 3923, IBM T. J. Watson Res. Center, Yorktown Heights, N.Y., 1973.

[2] A. K. Chandra, *Efficient compilation of linear recursive programs*, Proc. 14th Annual IEEE Symp. on Switching and Automata Theory, 1973, pp. 16–25.

[3] R. L. Constable and D. Gries, *On classes of program schemata*, this Journal, 1 (1972), pp. 66–118.

[4] J. Darlington and R. M. Burstall, *A system which automatically improves programs*, Proc. 3rd Internat. Joint Conf. on Artificial Intelligence, 1973, pp. 479–485.

[5] M. S. Hecht and J. D. Ullman, *Analysis of a simple algorithm for global data flow problems*, Proc. ACM Symp. on Principles of Programming Languages, 1973, pp. 207–217.

[6] R. M. Karp, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–103.

[7] D. E. Knuth, *The Art of Computer Programming. I: Fundamental Algorithms*, Addison-Wesley, Reading, Mass., 1968.

[8] A. Maggiolo-Schettini and H. R. Strong, *A graph-theoretic algorithm with application for transforming recursive programs*, AICA Convegno di Informatica Teorica, 1973, pp. 377–392.

[9] Z. Manna and A. Pneuli, *Formalization of properties of functional programs*, J. Assoc. Comput. Mach., 17 (1970), pp. 555–590.

[10] P. Naur, ed., *Revised report on the algorithmic language ALGOL 60*, Comm. ACM, 6 (1973), pp. 1–17.

[11] M. S. Paterson and C. E. Hewitt, *Comparative schematology*, ACM Conf. on Concurrent Systems and Parallel Computation, 1970, pp. 119–127.

[12] E. L. Post, *Recursive unsolvability of a problem of Thue*, J. Symbolic Logic, 12 (1947), pp. 1–11.

[13] B. K. Rosen and H. R. Strong, *Simplifying a recursion structure*, IBM Tech. Disclosure Bull., 16 (1973), pp. 858–859.

[14] H. R. Strong, *Translating recursion equations into flowcharts*, J. Comput. System Sci., 5 (1971), pp. 254–285.

[15] H. R. Strong and S. A. Walker, *Properties preserved under recursion removal*, Proc. ACM Conf. on Proving Assertions about Programs, 1972, pp. 97–103.

[16] R. Tarjan, *Testing flow graph reducibility*, Proc. 5th Annual ACM Symp. on Theory of Computing, 1973, pp. 96–107.

[17] S. A. Walker and H. R. Strong, *Characterizations of flowchartable recursions*, J. Comput. System Sci., 7 (1973), pp. 404–447.

SIAM J. Comput.
Vol. 4, No. 3, September 1975

INTERACTIVE COMPUTATION OF HOMOLOGY OF FINITE PARTIALLY ORDERED SETS*

R. BUMBY,† E. COOPER‡ AND D. LATCH‡

Abstract. We outline a method for practical use of an interactive system (APL) to compute the homology of finite partially ordered sets.

1. Prerequisites. All partially ordered sets (posets) are assumed finite.

Given a poset $\langle P, \leqq \rangle$, we say that *b covers a* if $b > a$ and $a \leqq c \leqq b$ implies $a = c$ or $b = c$. Since we deal with finite posets, the order relation can be obtained as the reflexive, transitive closure of the cover relation. Our programs allow the user to describe posets by the cover relation considered as a list of ordered pairs (more precisely, as an $N \times 2$ matrix). For its own convenience, the program only accepts cover relations which are subsets of the usual order on the natural numbers. There is no difficulty representing any poset in this fashion, e.g., the cover relation of a "labeled Hasse diagram" [1].

In order to calculate the homology of a poset, we define a functor, $C : \mathscr{P}_0 \to \mathscr{A}b^Z$, from the category of finite posets to the category of finite chain complexes of abelian groups. If P is a poset, then the group of n-chains, $C_n(P)$, is the free abelian group generated by symbols $a_0 < a_1 < \cdots < a_n$ in P. The *boundary operator* ∂ is defined on each generator $a_0 < a_1 < \cdots < a_n$ by the formula

$$\partial(a_0 < \cdots < a_n) = \sum_{0 \leqq i \leqq n} (-1)^i a_0 < \cdots < \hat{a}_i < \cdots < a_n,$$

where $a_0 < \cdots < \hat{a}_i < \cdots < a_n$ is the generator of $C_{n-1}(P)$ obtained from $a_0 < \cdots < a_i < \cdots < a_n$ by deleting the element a_i. The *n-th homology group* of P, $H_n(P)$, is defined to be the nth homology group of the complex $C(P)$. For the category of small categories, $\mathscr{C}$at, which includes $\mathscr{P}_0$, homology is usually defined as the homology of the simplicial set nerve of P, $N(P)$. It is well known [3], [6], that these homology theories are isomorphic.

2. Method. We begin by describing some of the functions in our APL-workspace:

PO: PO computes the graph of the $<$ relation in poset P and represents it as an $N \times N$ matrix called POMAT.

CHAIN: CHAIN computes the list of $K - 1$-chains in the poset P from the list of K-chains and POMAT.

BD: BD computes the matrix representing the boundary homomorphism. Input to this function consists of the list of K-chains and the list of $K + 1$-chains.

Following the sketch for computation of the homology of finite chain complexes found in Eilenberg and Steenrod [2, p. 138], we diagonalize the matrix giving the boundary map while saving the left transition matrix. For this diagonalization, we use a method of Nijenhuis [5] for determining the Smith canonical form of an integral matrix [4]. The functions actually used in the workspace are:

* Received by the editors June 27, 1974.

† Department of Mathematics, Rutgers University, New Brunswick, New Jersey 08903.

‡ Department of Mathematics, Lawrence University, Appleton, Wisconsin 54911.

NIJ1: NIJ1 reduces an integer matrix to a matrix whose nonzero entries are confined to main diagonal and an adjacent diagonal. Only these diagonals need to be stored for the remainder of the computation.

NIJ2: NIJ2 reduces output of NIJ1 to diagonal form.

Communication with user is accomplished via the function HOM. After loading the workspace, entering the single command HOM causes the response "ENTER POSET", followed by the request for input (□:). The user then enters the $N \times 2$ matrix of the cover relation, either directly or from a stored array. After computing the list of 1-chains, $H_0(P)$ is computed and displayed in the form

HO:RANK(#),

where (#) is the rank of $H_0(P)$. From here HOM enters a loop, which computes the chains of next highest length, computes the structure of the next homology group and displays it in a format similar to that used for $H_0(P)$ (see examples). If there are any elements of finite order in $H_k(P)$, the display includes the word "TORSION" followed by the orders of the factors in a direct sum decomposition.

3. Miscellaneous comments. An outline of an algorithm for performing this computation was developed by the second author. Actual programming was done by the first and third authors.

An early version of the workspace was produced fairly quickly, but proved too wasteful of space in the diagonalization routine. The appearance of Nijenhuis' abstract [5], while we were attempting to avoid WS-FULL errors, encouraged us to rewrite the workspace in the present form. In addition, this allowed a certain saving of time by not computing the Smith canonical form, but rather stopping as soon as the matrix was diagonalized. The workspace includes all functions necessary for the computation of the Smith canonical form of any given integer matrix.

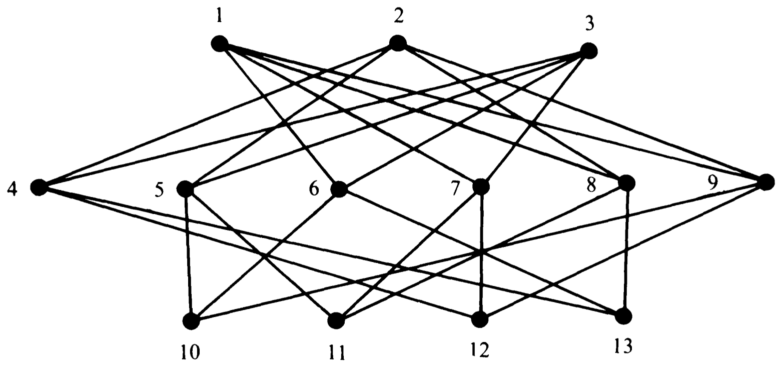
Computation of integral cohomology via the dual chain complex, $C^*(P)$, can be computed similarly, and is included in the workspace.

The workspace is currently being used on the Rutgers University System. A listing of the contents of the workspace will be furnished upon request from Professor R. Bumby.

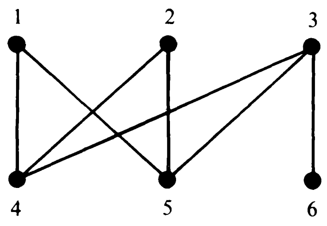
4. Examples. The cover relation A was derived from a triangulation of the projective plane. The arrays EL 291, etc., are cover relations of posets on at most 6 points named according to their occurrence in the list obtained by Ellis Cooper [1].

```
      HOM
ENTER POSET
□:
      A
H0:   RANK 1
H1:   RANK 0   TORSION 2
H2:   RANK 0
```

```
        A
 1   6
 1   7
 1   8
 1   9
 2   4
 2   5
 2   8
 2   9
 3   4
 3   5
 3   6
 3   7
 4  12
 4  13
 5  10
 5  11
 6  10
 6  13
 7  11
 7  12
 8  11
 8  13
 9  10
 9  12
```

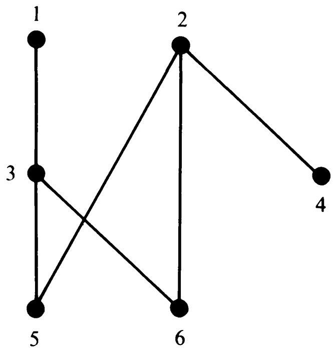


```
        HOM
ENTER POSET
□:
        EL291
H0:     RANK 1
H1:     RANK 2
          EL291
 1   4
 1   5
 2   4
 2   5
 3   4
 3   5
 3   6
```

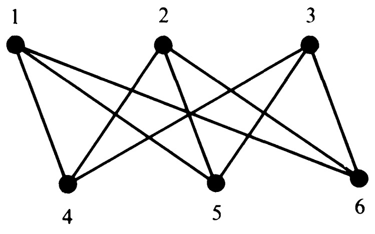


```
          EL137
 1   3
 2   4
 2   5
 2   6
 3   5
 3   6
```

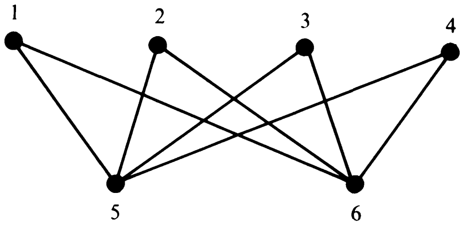
```
HOM
 ENTER POSET
 ⎕:
          EL137
 H0:   RANK 1
 H1:   RANK 1
 H2:   RANK 0
```



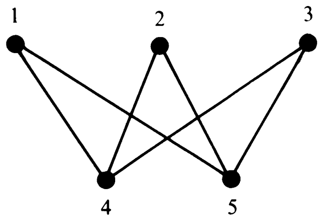
```
         EL316
  1   4
  1   5
  1   6
  2   4
  2   5
  2   6
  3   4
  3   5
  3   6
         HOM
ENTER POSET
⎕:
         EL316
H0:   RANK 1
H1:   RANK 4
```



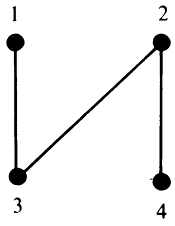
```
         HOM
ENTER POSET
⎕:
       ⎕←EL315
  1   6
  1   5
  2   5
  2   6
  3   5
  3   6
  4   5
  4   6
H0:   RANK 1
H1:   RANK 3
         HOM
ENTER POSET
⎕:
```



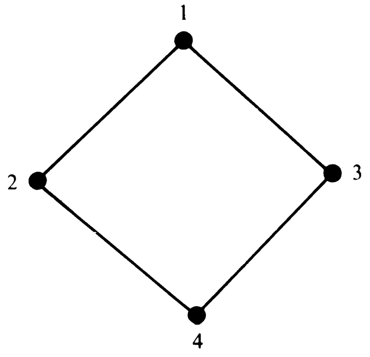
```
        □←EL 61
  1  4
  1  5
  2  4
  2  5
  3  4
  3  5
H0:   RANK 1
H1:   RANK 2
           HOM
ENTER POSET
□:
```



```
        □←EL 11
  1  3
  2  3
  2  4
H0:   RANK 1
H1:   RANK 0
           HOM
ENTER POSET
□:
```



```
        □←EL 13
  1  2
  1  3
  2  4
  3  4
H0:   RANK 1
H1:   RANK 0
H2:   RANK 0
```



REFERENCES

[1] E. Cooper, *Enumeration of finite partially ordered sets* (in manuscript).

[2] S. Eilenberg and N. Steenrod, *Foundations of Algebraic Topology*, Princeton University Press, Princeton, N.J., 1952.

[3] M. O. Laudal, *Sur les limites projectives et inductives*, Ann. Sci. École Norm. Sup., 82 (1965), pp. 241–296.

[4] M. Newman, *Integral Matrices*, Academic Press, New York, 1972.

[5] A. Nijenhuis, *Smith canonical forms of integer matrices*, Notices Amer. Math. Soc., 21 (1974), p. A-389.

[6] U. Oberst, *Homology of categories and exactness of direct limits*, Math. Z., 107 (1968), pp. 87–115.

SIAM J. COMPUT.
Vol. 4, No. 3, September 1975

THE POWER OF NEGATIVE THINKING IN MULTIPLYING BOOLEAN MATRICES*

VAUGHAN R. PRATT†

Abstract. We show that n^3 distinct *and*-gate inputs appear in any circuit constructed from *and*-gates and *or*-gates that computes the product of two $n \times n$ Boolean matrices. Using *not*-gates as well, it is possible to realize a circuit for this problem using only $O(n^{\log_2 7} \log^2 n)$ gates, whence we infer a much larger complexity gap between *and-or* and *and-or-not* circuits than was previously known.

Key words. Boolean, matrix, multiplication, monotone, lower bound, computational complexity, circuit, network

We are interested in combinational circuits synthesized from *and*-gates and *or*-gates. We first show that n^3 distinct *and*-gate inputs are needed to form the product of two $n \times n$ Boolean matrices, and hence $O(n^3)$ two-input *and*-gates are needed to compute the transitive closure of a Boolean matrix. While this result has the flavor of Kerr's (achievable) lower bound [3] of n^3 +-gates for computing the min/+ product of integer-valued matrices using only *min*-gates and +-gates, the problem turns out on closer inspection to be considerably more subtle, and in fact, using our methods, we have been able to come only to within a factor of two of the best known upper bound of n^3 *and*-gates. Paterson (private communication) has recently disposed of this factor.

Secondly, we use this result to study the effect on combinational circuits of not using *not*-gates (inverters). (With inverters and either *and* or *or*, it is clear that only a constant factor improvement can be had by increasing the variety of available types of gates.) A recent conjecture that this entailed a loss of at most a constant factor is defeated by the observations of Muller and Preparata [5], who point out that a circuit for sorting v 0- and 1-valued inputs can be built using $O(v)$ *and*-gates and inverters, and Lamagna and Savage [4] who show that at least $v \log v$ *and*-gates and *or*-gates are required if inverters are forbidden. This raises the question, what is the greatest loss of economy a circuit designer may incur in implementing monotonic functions using only *and*-gates and *or*-gates?

Since every monotone function of v variables may be implemented using $O(v2^v)$ such gates by constructing a circuit based on the disjunctive normal form of the function, the gap is at most a factor of 2^v. This is two exponentials larger than the above gap of a factor of $\log v$. We improve the gap by almost one exponential, to a factor of order $v^{(1/2)\log_2(8/7)}/\log^2 v$, using the result that Boolean matrix multiplication can be carried out on a Turing machine with the help of the Strassen [7]–Munro [6]–Fischer–Meyer [2] method in time $O(n^{\log_2 7} \log^2 n)$. This computation is performed obliviously—that is, the machine's head trajectories are a function solely of the length of the input. By a result of Fischer and Pippenger [1], the same computation may be carried out by a circuit using a number of *and*-gates and inverters proportional to the running time of the oblivious machine.

* Received by the editors May 30, 1974, and in revised form August 23, 1974. This research was supported by the National Science Foundation under Grant GJ-34671.

† Project MAC, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139.

THEOREM. *There are at least n^3 distinct* and-*gate inputs in a circuit for multiplying two $n \times n$ Boolean matrices A and B, when the only components permitted in the circuit are* and-*gates and* or-*gates.*

Proof. We shall take a syntactic, rather than semantic, approach to describing wires. With each wire we associate its expression, which is a partially simplified disjunctive normal form expression describing what function of the inputs this wire realizes. Formally, the *expression* of a wire is a list of terms. A *term* is a list of input-variables. A typical expression is $(a_{12}b_{32}, a_{34}, a_{24}a_{25}, a_{34})$, with corresponding function

$$(a_{12} \wedge b_{32}) \vee (a_{34}) \vee (a_{24} \wedge a_{25}) \vee (a_{34}).$$

We associate expressions with wires inductively, starting from the inputs. An input wire's expression is just one term consisting of the corresponding input variable. The expression of a wire attached to the output of an *or*-gate is the concatenation of the two lists of terms comprising the two respective expressions on the respective inputs of this *or*-gate. For example, if $(a_{11}, a_{11}b_{11})$ and (b_{12}, a_{12}) are the expressions of the two inputs, the output is $(a_{11}, a_{11}b_{11}, b_{12}, a_{12})$. For an *and*-gate, the Cartesian product for lists is taken instead of concatenation and for convenience each *term* is simplified by removing repeated variables. In the preceding example, replacing the *or*-gate with an *and*-gate, with the same input expressions, would change the output expression to

$$(a_{11}b_{12}, a_{11}a_{12}, a_{11}b_{11}b_{12}, a_{11}b_{11}a_{12}).$$

Note that we do not effect simplifications based on relations *between* terms. In this way, the expression carries along some information about the structure of the preceding circuitry, as well as about the corresponding function.

Let C be the product matrix AB. Then there are n^2 outputs, labeled c_{ij} for i, j in the range 1 to n. The next two lemmas supply a syntactic characterization of the function "Boolean matrix product" realized at these outputs.

LEMMA 1. *In the expression associated with the c_{ij} output, every term includes variables a_{ik} and b_{kj} for some k.*

Proof. Suppose some term did not have such variables. Now the pair of matrices A and B in which every entry named in this term is 1 and every other entry is 0 has the product $C = AB$ in which $c_{ij} = 0$, since there is no pair a_{ik} and b_{kj} simultaneously 1. But the c_{ij} output of the circuit is 1 because one term is 1 by construction, a contradiction. $\square$

LEMMA 2. *For each i, j and k in the range 1 to n, there is a term $a_{ik}b_{kj}$ in the c_{ij} output expression.*

Proof. Consider matrices A and B in which $a_{ik} = b_{kj} = 1$, and all other entries are 0. Then $c_{ij} = 1$ in the product. Hence some term in the c_{ij} output expression is 1, and therefore consists solely of a_{ik} and/or b_{kj}, all other variables being 0. By Lemma 1, both variables must be present. $\square$

These lemmas supply necessary conditions for an expression to realize matrix product. The reader may verify that they are also sufficient, a fact unused below.

We are now ready to prove the theorem. We shall give a method for finding n^3 *and*-gate inputs in a circuit Y for multiplying $n \times n$ Boolean matrices that

(i) has a minimal number of *and*-gate inputs and

(ii) with respect to the class of circuits satisfying (i), has a minimal number of gates.

(Condition (i) must take priority over condition (ii) because we are trying to bound from below the number of *and*-gate inputs. If a circuit with fewest *and*-gate inputs has n^3 of them, every circuit has n^3, but if it is a circuit with fewest gates, then the conclusion that it has n^3 *and*-gate inputs cannot immediately be applied to all circuits.)

For each of the n^3 triples i, j, k in the range 1 to n, we select an *and*-gate input whose expression has one term equal to b_{kj} and every term containing either b_{kj} or a_{il} for some l, and which is an input to an *and*-gate whose output expression includes the term $a_{ik}b_{kj}$.

We first verify the existence of such an input. Suppose to the contrary that every *and*-gate with $a_{ik}b_{kj}$ as a term in its output, and b_{kj} as a term in one of its inputs, also has a term in the same input that has no occurrence of b_{kj} or a_{il} for any l. We shall show that this would give rise to "noise" in the c_{ij} output.

LEMMA 3. *If every* and-*gate whose output expression includes the term $a_{ik}b_{kj}$, and which has the term b_{kj} in one of the input expressions, also has a term in the same input which has no occurrence of b_{kj} or of a_{il} for any l, then in the c_{ij} output there appears a "noise" term which does not simultaneously contain a_{im} and b_{mj} for any m.*

Proof. We prove by induction on the distance from the furthest circuit input that every expression containing the term $a_{ik}b_{kj}$ also contains a term with no occurrence of b_{kj} and no occurrence of a_{il} for $l \neq k$. This is vacuously true for expressions at the circuit inputs, and it follows by induction trivially for expressions at the output of *or*-gates. For *and*-gates, if $a_{ik}b_{kj}$ appears as a term in the output, then either a_{ik} or b_{kj} or $a_{ik}b_{kj}$ must occur as a term in each input. Accompanying b_{kj} is a term containing no occurrence of b_{kj} or of a_{il} for any l, and by induction, accompanying $a_{ik}b_{kj}$ is a term containing no occurrence of b_{kj} or a_{il} for $l \neq k$. Hence we know there is a term in the output formed by taking from each input either the term a_{ik}, or a term containing neither b_{kj} nor a_{il} for $l \neq k$, and concatenating them.

By Lemma 2, $a_{ik}b_{kj}$ appears in the output, whence a term accompanies it containing neither b_{kj} nor a_{il} for $l \neq k$, and hence does not simultaneously contain a_{im} and b_{mj} for any m. $\square$

This lemma together with the preceding hypothesis and Lemma 1 leads to a contradiction. Hence the desired *and*-gate input always exists.

We now claim that no gate input is selected twice. Suppose the contrary. Since b_{kj} is the only variable from the B matrix that can appear by itself as a term on this wire, only i can differ in the two selections; suppose the two selections are made for the triples i, j, k and m, j, k.

Every term on the selected wire contains either b_{kj} or both a_{il} and $a_{ml'}$ for some l, l'. We first show that the terms containing two such "a" inputs play no role.

LEMMA 4. *Given $i \neq m$ and a circuit Y that computes the Boolean matrix product, if the expression e of some gate's output consists of the term b_{kj} together with other terms each of which contains, for some l and l', the variables a_{il} and $a_{ml'}$, then an* *equivalent circuit Y' can be built from Y by omitting this gate and connecting the wire(s) formerly connected to its output directly to the input variable b_{kj}.*

Proof. Suppose Y' differs from Y for some input. Then by monotonicity, some output c_{pq} must be 1 in Y and 0 in Y', whence b_{kj} is 0 and one of the other terms in e is 1. Setting the ith row of A (that is, all elements a_{ir}, $1 \leqq r \leqq n$) to zero will then set c_{pq} to zero, again by monotonicity. Hence $p = i$. Similarly $p = m$, a contradiction. □

Hence if the expression at the selected input is other than b_{kj}, it follows that either condition (i) (number of *and*-gate inputs) has been violated, or if not, then both Y and Y' satisfy (i), whence Y violates (ii) (minimal number of gates within the class satisfying (i)). In either case, this contradicts the conditions on Y.

The *and*-gate whose input has been selected twice (an input which we now know has the expression b_{kj}) has on its output the terms $a_{ik}b_{kj}$ and $a_{mk}b_{kj}$. We show that b_{kj} alone will do the same job as this output.

LEMMA 5. *Given a circuit Y that computes the Boolean matrix product, if the expression of some* and-*gate's output includes terms $a_{ik}b_{kj}$ and $a_{mk}b_{kj}$ for $i \neq m$, and one input is just the expression b_{kj}, then an equivalent circuit Y' may be built from Y by omitting this* and-*gate and connecting the wire(s) formerly connected to its output directly to the input variable b_{kj}.*

Proof. Suppose Y' differs from Y for some input. Then, again by monotonicity, some output c_{pq} must by 0 in Y and 1 in Y', when b_{kj} is 1 and both a_{ik} and a_{mk} are 0. Hence changing either a_{ik} or a_{mk} to 1 will send c_{pq} to 1 in Y. This implies that $i = p = m$, a contradiction. □

Thus, if we had selected the same *and*-gate input twice, we could have found a circuit with fewer *and*-gate inputs, in violation of condition (i).

We have exhibited n^3 distinct *and*-gate inputs in a circuit with a minimal number of *and*-gate inputs. Hence every circuit has n^3 *and*-gate inputs. This completes the proof of the main theorem.

Discussion. As remarked earlier, the main theorem gives us a much larger lower bound for the complexity gap between *and/or* and *and/not* circuits than we have had previously. However, the improvement was by only one exponential, from a ratio of order $\log n$ to order $n^{(1/2)\log_2(8/7)}/\log^2 n$. The best upper bound to date for this ratio is 2^n.

In conclusion, we raise the question, is the complexity gap between *and-or* and *and-or-not* circuits for every function at most a polynomial in the number of inputs?

REFERENCES

[1] M. J. FISCHER AND N. PIPPENGER, private communication, 1973.

[2] M. J. FISCHER AND A. R. MEYER, *Boolean matrix multiplication and transitive closure*, IEEE Conf. Rec. 12th Ann. Symp. on Switching and Automata Theory, East Lansing, Mich., IEEE, New York, 1971, pp. 129–131.

[3] L. R. KERR, *The effect of algebraic structure on the computational complexity of matrix multiplication*, Ph. D. thesis, Cornell University, Ithaca, N.Y., 1970.

[4] E. A. LAMAGNA AND J. E. SAVAGE, *On the logical complexity of symmetric switching functions in monotone and complete bases*, Tech. Rep., Center for Computer and Information Sciences, Brown Univ., Providence, R.I., 1973.

[5] D. E. MULLER AND F. P. PREPARATA, *Minimum delay networks for sorting and for switching*, Proc. 6th Ann. Princeton Conf. on Information Sciences and Systems, Princeton Univ., Princeton, N.J., 1972, pp. 138–139.

[6] I. MUNRO, *Efficient determination of the transitive closure of a directed graph*, Information Processing Letters, 1 (1971), pp. 56–58.

[7] V. STRASSEN, *Gaussian elimination is not optimal*, Numer. Math., 13 (1969), pp. 354–356.

SIAM J. COMPUT.
Vol. 4, No. 3, September 1975

SPEED OF RECOGNITION OF CONTEXT-FREE LANGUAGES BY ARRAY AUTOMATA*

S. RAO KOSARAJU†

Abstract. The recognition speed of context-free languages (CFL's) using arrays of finite state machines is considered. It is shown that CFL's can be recognized by 2-dimensional arrays in linear time and by 1-dimensional arrays in time n^2.

Key words. context-free languages, array automata, recognition speed

1. Introduction. Recognition of context-free languages (CFL's) has attracted considerable attention [4], [10], [12]. Multitape Turing machines or list-like structures have been employed for such recognition. The known upper and lower bounds for the speed of CFL recognition are $n^{\log_2 7}$ and linear time, respectively. In this paper we show that the recognition speed of CFL's is tightly bounded by linear time on 2-dimensional array automata. We also present an n^2 algorithm for 1-dimensional array automata.

The array automaton model is introduced in § 2. A brief review of the CFL's, and the Younger's algorithm for CFL recognition are given in § 3. Sections 4 and 5 are devoted to recognition of CFL's by 2-dimensional and 1-dimensional array automata, respectively. Throughout we shall skip many trivial details, for clarity, and consequently some familiarity with the array automaton will be helpful.

2. Array automata. A *d-dimensional array automaton* (notationally d-AA) consists of a 1-way input tape together with a d-dimensional regular array of cells (d-dimensional integer space I^d), each of which contains a finite state machine (fsm). There is a single *input head* which scans a square of the input tape, and a single *array head* which scans a cell of the array. Any two cells which are separated by no more than a unit distance along every axis are *neighbors* of each other; i.e.,

$$\{\xi_1 + \delta_1, \xi_2 + \delta_2, \cdots, \xi_d + \delta_d| \ |\delta_i| \leqq 1 \text{ for } i = 1, 2, \cdots, d\}$$

is the set of neighbors of $(\xi_1, \xi_2, \cdots, \xi_d)$. Thus there are 3^d neighbors for any cell. When there is no ambiguity, the fsm in any cell is simply referred to by the cell itself.

A *step* of computation consists of an input head move and an array head move, together with a state transformation for each fsm in the array. The moves of the input and array heads depend on the input symbol scanned and the state of the cell scanned by the array head. The input head moves right only, and in one step it may move right by at most one square. In one step the array head may move to any one of the neighbors of the cell it is currently scanning. The next state of any cell depends upon the present state of each of its neighbors; the next state of the cell containing the array head also depends upon the input symbol being scanned.

The fsm's in all cells are identical and start in a designated starting state. There are two designated states: q_a (accept state) and q_r (reject state). When a cell

* Received by the editors July 27, 1973, and in revised form June 26, 1974.

† Electrical Engineering Department, Johns Hopkins University, Baltimore, Maryland 21218.

goes into either state, it cannot change into any other state. When the head scans a cell in either state, it cannot leave that cell, and we say that the automaton has *halted*.

For language recognition, to decide the membership of $a_1 a_2 \cdots a_n$, the string $a_1 a_2 \cdots a_n \#$ is placed on the input tape with the input head positioned on a_1; and each cell of the array is set to the starting state. $a_1 a_2 \cdots a_n$ is *accepted* (*rejected*) by the d-AA, M, if and only if the input head eventually scans $\#$ with q_a (q_r, respectively) as the state of the cell scanned. The *language accepted* by M, denoted by $L(M)$, is the set of strings it accepts.

Other variations of this model were considered in the literature [2], [3], [5], but our results are not sensitive to the particular model employed.

A d-AA, M, *recognizes* a language L *within time* $T(n)$ if and only if M halts on every input string of length n within $T(n)$ steps and $L(M) = L$. $T(n) = n$ is known as *real time*. $T(n) = cn$, where c is a constant, is known as *linear time*.

In the following, for ease of description, we might consider each cell of the array to consist of an (ordered) set of k registers, for some k. At any instant the state is given by the ordered k-tuple formed by the contents of the k registers. In addition, many trivial details are omitted.

3. Context-free languages. A *context-free grammar* (CFG) G is a 4-tuple: $G = (V_N, V_T, P, S)$, where V_N and V_T are finite nonterminal and terminal alphabets of G, respectively. S is the starting symbol, $S \in V_N$. P is a finite set of productions, each member of which has one of the forms (for simplicity, the null string Λ is excluded):

(a) $A \to a$, where $A \in V_N$ and $a \in V_T$,

(b) $A \to BC$, where $A, B, C \in V_N$.

If P contains the production $A \to \gamma$, then for any $\alpha, \beta \in (V_N \cup V_T)^*$, we write $\alpha A \beta \Rightarrow \alpha \gamma \beta$. Let $\overset{*}{\Rightarrow}$ be the reflexive and transitive closure of $\Rightarrow$. The language, $L(G)$, generated by G is given by

$$L(G) = \{x | x \in V_T^* \text{ and } S \overset{*}{\Rightarrow} x\}.$$

A language L is a *context-free language* (CFL), if and only if there exists a context-free grammar G such that $L(G) = L$.

In the following, we show that context-free languages can be recognized by 2-AA's within linear time and by 1-AA's within n^2 time. The algorithm we use is the classical algorithm of Younger [12]. For any string $a_1 a_2 \cdots a_n$, $n \geqq 1$ and each $a_i \in V_T$, let

$$\Delta_i^j = \{A | A \in V_N \text{ and } A \overset{*}{\Rightarrow} a_i \cdots a_j\}, \qquad 1 \leqq i \leqq j \leqq n.$$

ALGORITHM.

$$\Delta_i^i = \{A | A \in V_N \text{ and } A \to a_i \in P\}, \qquad 1 \leqq i \leqq n,$$

$$\Delta_i^j = \bigcup_{i \leqq k < j} \Delta_i^k * \Delta_{k+1}^j, \qquad 1 \leqq i < j \leqq n,$$

where $X * Y = \{A | (\exists B, C)(B \in X, C \in Y, \text{ and } A \to BC \in P)\}$.

$$a_1 a_2 \cdots a_n \in L(G) \iff S \in \Delta_1^n.$$

Thus the algorithm involves computing many "convolutions".

4. CFL's and 2-AA's.

THEOREM 1. *Any CFL can be recognized by some 2-AA within time $T(n) = (1 + \varepsilon)n$, for any given real number $\varepsilon > 0$.*

Proof. Let the array cell where the head starts be cell $(1, 1)$. If $a_1a_2 \cdots a_n$ is the input string, then at the end of the computation, Δ_i^j will be contained in cell (i, j) of the storage array, for $1 \leqq i \leqq j \leqq n$. From the input string and the grammar, $\Delta_1^1, \Delta_2^2, \cdots, \Delta_n^n$ can be easily computed and stored properly. For $1 \leqq i < j \leqq n$, cell (i, j) receives $\Delta_i^i, \Delta_i^{i+1}, \cdots, \Delta_i^{j-1}$ along the Y-axis (vertical) from its "down neighbor" $(i, j - 1)$. Simultaneously, cell (i, j) receives Δ_{i+1}^j, $\Delta_{i+2}^j, \cdots, \Delta_j^j$ along the $-X$-axis (horizontal) from its "right neighbor" $(i + 1, j)$. In a straightforward implementation, it is seen that the computation of Δ_1^n requires a time on the order of n^2.

In our method, each cell (i, j) receives the above convolution terms from its down and right neighbors in *folded form*, folded at the middle, with the middle terms coming first. When $j - i$ is odd, there is only one middle term. To maintain this folding, each cell needs to store only a finite amount of information.

Each cell contains 8 registers: V_1, V_2, V_3, H_1, H_2, H_3, B and A. The V_k and H_k registers are the vertical and horizontal propagation registers, respectively. These registers provide the necessary delay for maintaining the folded form. Register B of cell (i, j) is used for storing Δ_i^j. Every one of the registers V_1, V_2, V_3, H_1, H_2, H_3 and B can store any subset of the set of nonterminals of the grammar. Register A controls the various phases of the recognition process. For simplicity, we will not go into all the details of the operation of register A. Initially every one of these registers, except B, contains a special symbol $\neg$, different from the empty set $\varnothing$, and B contains $\varnothing$.

The productions of the grammar are built into the state transformations, as will be evident subsequently. The following recognition scheme gives the important details.

Phase 1. When the first symbol a_1 is received, Δ_1^1 is computed and stored in register B of cell $(1, 1)$. Register A of cell $(1, 1)$ is set to indicate that it is the starting cell. It also sends a control signal, which propagates at the rate of 1 cell/step along the Y-axis. The propagation of this signal suitably changes register A of each cell $(1, j)$, $j \geqq 1$, to facilitate the positioning of the array head on the cell $(1, n)$ in Phase 3. For ease of description, let us call this the α-signal. The head then moves to cell $(2, 2)$ in one step. For $i > 1$, when the ith symbol is received:

(a) if it is not #, then Δ_i^i is computed and stored in register B of the cell scanned by the head; i.e., cell (i, i). Then the head moves to cell $(i + 1, i + 1)$.

(b) if it is #, then the head moves from cell (i, i) to cell $(i - 1, i - 1)$ and initiates Phase 2.

Phase 1 takes $n + 1$ steps.

Phase 2. Cells (j, j), $j = 1, \cdots, n$, act as the classical *firing squad*, using register A of each cell. The firing squad operation simultaneously sets register A of each cell (j, j) into a designated "firing state". As soon as register A of a cell goes into the firing state, the content of its B register is copied into its V_3 and H_3 registers, and Phase 3 starts. Phase 2 takes $2n - 2$ steps.

Phase 3. This is the main computational phase. In this phase, each cell (i, j) computes Δ_i^j from the convolution terms received from cells $(i, j - 1)$ and $(i + 1, j)$.

At the beginning of this phase, V_1, V_2, V_3, H_1, H_2, H_3 and B of the cell (i, i), $1 \leqq i \leqq n$, contain $-$, $-$, Δ_i^i, $-$, $-$, Δ_i^i and Δ_i^i, respectively. In addition, for any other cell, each of V_1, V_2, V_3, H_1, H_2 and H_3 contains $-$; and B contains $\varnothing$.

At the end of this phase, in each cell (i, j), $1 \leqq i \leqq j \leqq n$, each of V_1, V_2, V_3, H_1, H_2 and H_3 will contain $-$, and B will contain Δ_i^j.

The registers of each cell undergo the following parallel transformations in *one step*. Notationally, the superscripts d and r stand for *down* and *right* neighbors, respectively; e.g., for any cell (i, j), V_k^d refers to V_k of cell $(i, j - 1)$, and H_k^r refers to H_k of cell $(i + 1, j)$. In any transformation, the right-hand side gives the contents before the step, and the left-hand side after the step. The transformation $(\alpha_1, \alpha_2, \cdots, \alpha_m) \leftarrow (\beta_1, \beta_2, \cdots, \beta_m)$ stands for $\alpha_1 \leftarrow \beta_1, \alpha_2 \leftarrow \beta_2, \cdots, \alpha_m \leftarrow \beta_m$.

$(V_2, H_2) \leftarrow (V_1, H_1)$,

(V_1, V_3, H_1, H_3, B)

$$\leftarrow \begin{cases} (V_2^d, V_3^d, H_2^r, H_3^r, B \cup V_2^d * H_3^r \cup V_3^d * H_2^r) \quad \text{if } V_2^d \neq -, & \text{(I)} \\ (V_3^d, -, H_3^r, -, B \cup V_3^d * H_3^r) \quad \text{if } V_2^d = -, \text{and } V_3^d \neq -, & \text{(II)} \\ (-, B, -, B, B) \quad \text{if } V_2^d, V_3^d = -, \text{ and } V_1 \neq -, & \text{(III)} \\ (-, -, -, -, B) \quad \text{otherwise.} & \text{(IV)} \end{cases}$$

In parallel with these transformations, the head moves left (along $-X$-axis) at the rate of 1 cell/2 steps and reaches cell $(1, n)$ in exactly $2(n - 1)$ steps. Cell $(1, n)$ is located by the α-signal sent in Phase 1. In the next step, cell $(1, n)$ goes into state q_a or q_r, depending upon whether or not S is in the set stored in B of cell $(1, n)$.

This phase takes $2n - 1$ steps. Now we prove that when the head scans cell $(1, n)$, its B register contains Δ_1^n, which establishes the validity of the recognition process.

Let us count steps from the instant when Phase 3 starts. Notationally, let the contents of registers $X_1, X_2, \cdots, X_m$ of cell (i, j) at instant t be represented by $(X_1, X_2, \cdots, X_m)_t^{i,j}$.

LEMMA 1.1. *For any* $1 \leqq i \leqq i + \delta \leqq n$ *and* $t \geqq 0$,

$$(V_2, H_2)_t^{i,i+\delta} = (V_1, H_1)_{t-1}^{i,i+\delta} \; (\textit{assuming } (V_1, H_1)_{-1}^{i,i+\delta} = (-, -)).$$

Proof. Registers V_2 and H_2 of any cell always undergo the transformation $(V_2, H_2) \leftarrow (V_1, H_1)$. Note that in addition for any cell $(i, i + \delta)$, $(V_2, H_2)_0^{i,i+\delta} = (-, -)$. Q.E.D.

LEMMA 1.2. *For any* $1 \leqq i \leqq i + \delta \leqq n$ *and* $t \geqq 0$, $(V_1, V_3, H_1, H_3, B)_t^{i,i+\delta}$ *is given by the following two cases*:

Case 1. $\delta = 2j$ *(δ even).*

(1.1) $\quad (-, -, -, -, \varnothing)$ *if* $t < 3j$,

(1.2) $\quad (\Delta_i^{i+j-k}, \Delta_i^{i+j+k-1}, \Delta_{i+j+k}^{i+2j}, \Delta_{i+j-k+1}^{i+2j}, \bigcup_{-(k-1) \leqq m \leqq k} (\Delta_i^{i+j-m} * \Delta_{i+j-m+1}^{i+2j}))$

if $t = 3j + k - 1, 1 \leqq k \leqq j$, (1.3) $(-, \Delta_i^{i+2j}, -, \Delta_i^{i+2j}, \Delta_i^{i+2j})$ *if* $t = 4j$,

(1.4) $(-, -, -, -, \Delta_i^{i+2j})$ *if* $t > 4j$.

Case 2. $\delta = 2j + 1$ (δ *odd*).

(2.1) $(-, -, -, -, \varnothing)$ *if* $t < 3j + 1$,

(2.2) $(\Delta_i^{i+j}, -, \Delta_{i+j+1}^{i+2j+1}, -, \Delta_i^{i+j} * \Delta_{i+j+1}^{i+2j+1})$ *if* $t = 3j + 1$,

(2.3) $(\Delta_i^{i+j-k}, \Delta_i^{i+j+k}, \Delta_{i+j+k+1}^{i+2j+1}, \Delta_{i+j-k+1}^{i+2j+1}, \bigcup_{-k \leqq m \leqq k} (\Delta_i^{i+j-m} * \Delta_{i+j-m+1}^{i+2j+1}))$

if $t = 3j + k + 1, 1 \leqq k \leqq j$,

(2.4) $(-, \Delta_i^{i+2j+1}, -, \Delta_i^{i+2j+1}, \Delta_i^{i+2j+1})$ *if* $t = 4j + 2$,

(2.5) $(-, -, -, -, \Delta_i^{i+2j+1})$ *if* $t > 4j + 2$.

Proof. We prove this by induction on δ. If $\delta = 0$, then $j = 0$, and $1 \leqq i \leqq n$. When Phase 3 starts, $(V_1, V_3, H_1, H_3, B)_0^{i,i} = (-, \Delta_i^i, -, \Delta_i^i, \Delta_i^i)$, satisfying (1.3). For any $t \geqq 1$, transformation IV applies, and hence $(V_1, V_3, H_1, H_3, B)_t^{i,i} = (-, -, -, -, \Delta_i^i)$, satisfying (1.4).

As inductive hypothesis we assume that the lemma holds for any $\delta \leqq n - 2$, and then show that the lemma must hold for $\delta + 1$. We treat the case of δ even (i.e., $\delta = 2j$) and leave the other case as an exercise. The validity for $\delta + 1$ is established by induction on the instant t. When Phase 3 starts, $(V_1, V_3, H_1, H_3, B)_0^{i,i+2j+1} = (-, -, -, -, \varnothing)$, satisfying (2.1). Let Case 2 hold for cell $(i, i + 2j + 1)$ at any instant t.

If $0 \leqq t \leqq 3j - 1$ or $t \geqq 4j + 2$, then $(V_2)_t^{i,i+2j} = (V_1)_{t-1}^{i,i+2j} = -$, $(V_3)_t^{i,i+2j} = -$, and $(V_1)_t^{i,i+2j+1} = -$. Hence by transformation (IV) above,

$$(V_1, V_3, H_1, H_3, B)_{t+1}^{i,i+2j+1} = (-, -, -, -, (B)_t^{i,i+2j+1}),$$

satisfying (2.1) and (2.5).

If $t = 3j$, then $(V_2)_t^{i,i+2j} = (V_1)_{t-1}^{i,i+2j} = -$, and $(V_3)_t^{i,i+2j} = \Delta_i^{i+j} \neq -$. Also $(H_3)_t^{i+1,i+2j+1} = \Delta_{i+j+1}^{i+2j+1}$. Hence by transformation (II) above,

$$(V_1, V_3, H_1, H_3, B)_{3j+1}^{i,i+2j+1} = (\Delta_i^{i+j}, -, \Delta_{i+j+1}^{i+2j+1}, -, \varnothing \cup \Delta_i^{i+j} * \Delta_{i+j+1}^{i+2j+1}),$$

satisfying (2.2).

If $t = 3j + k$, for $1 \leqq k \leqq j$, then $(V_2)_t^{i,i+2j} = (V_1)_{t-1}^{i,i+2j} \neq -$. Hence by transformation (I) above and Lemma 1.1,

$$(V_1, V_3, H_1, H_3)_{3j+k+1}^{i,i+2j+1} = (\Delta_i^{i+j-k}, \Delta_i^{i+j+k}, \Delta_{i+j+k+1}^{i+2j+1}, \Delta_{i+j-k+1}^{i+2j+1}),$$

and

$$(B)_{3j+k+1}^{i,i+2j+1} = (B)_{3j+k}^{i,i+2j+1} \cup \Delta_i^{i+j-k} * \Delta_{i+j-k+1}^{i+2j+1} \cup \Delta_i^{i+j+k} * \Delta_{i+j+k+1}^{i+2j+1},$$

satisfying (2.3).

If $t = 4j + 1$, then $(B)_t^{i,i+2j+1} = \Delta_i^{i+2j+1}$, $(V_2, V_3)_t^{i,i+2j} = (-, -)$, and $(V_1)_t^{i,i+2j+1} \neq -$. Hence by transformation (III) above,

$$(V_1, V_3, H_1, H_3, B)_{4j+2}^{i,i+2j+1} = (-, \Delta_i^{i+2j+1}, -, \Delta_i^{i+2j+1}, \Delta_i^{i+2j+1}),$$

satisfying (2.4). Hence the induction on t is complete. Q.E.D.

As a consequence, for any cell $(i, i+2j)$, $1 \leqq i \leqq i+2j \leqq n$, B contains Δ_i^{i+2j} at any instant $t \geqq 4j$; and for any cell $(i, i+2j+1)$, $1 \leqq i < i+2j+1 \leqq n$, B contains Δ_i^{i+2j+1} at any instant $t \geqq 4j+2$. Hence for cell $(1, n)$, B contains Δ_1^n at any instant $t \geqq 2(n-1)$. Thus when the head first scans cell $(1, n)$, the B register contains Δ_1^n.

The complete recognition process takes $n+1+2n-2+2n-1 = 5n-2$ steps. Now the theorem follows from linear speedup. (The classical linear speedup for Turing machines by tape compression can be trivially adapted for d-AA's.)

Q.E.D.

This is the best we can attain using d-AA's, since there is a CFL which cannot be recognized in real time by any d-AA [2], [3].

5. CFL's and 1-AA's.

THEOREM 2. *Any CFL can be recognized by some* 1-*AA within time* $T(n) = n^2$.

Proof. Let the input be $a_1 a_2 \cdots a_n\#$, where $n \geqq 2$. The computations are performed in n^2 contiguous cells, divided into n blocks numbered $0, 1, 2, \cdots, n-1$, left to right. Block b, $0 \leqq b \leqq n-2$, is of length $n+1$, and block $n-1$ is of length 1. Let the cells in the bth block, $b \geqq 0$, be addressed $(b, 0), (b, 1), \cdots$, from left to right. Thus block b, $0 \leqq b \leqq n-2$, consists of cells $(b, 0), (b, 1), \cdots, (b, n)$.

Block b, $1 \leqq b \leqq n-1$, computes $\Delta_1^{b+1}, \Delta_2^{b+2}, \cdots, \Delta_{n-b}^n$. To compute these, cell $(b, 0)$ receives the sequence of terms

$$(\Delta_1^b, \Delta_{b+1}^{b+1}), (\Delta_1^{b-1}, \Delta_b^{b+1}), \cdots, (\Delta_1^1, \Delta_2^{b+1}), (-, -)^{n-b-1}, (\Delta_2^{b+1}, \Delta_{b+2}^{b+2}), \cdots,$$
$$(\Delta_2^2, \Delta_3^{b+2}), (-, -)^{n-b-1}, \cdots, (\Delta_{n-b}^{n-1}, \Delta_n^n), \cdots, (\Delta_{n-b}^{n-b}, \Delta_{n-b+1}^n),$$

in $(n-b-1)(n-1)+b$ consecutive steps from the rightmost cell of block $b-1$, i.e., cell $(b-1, n)$. From these convolution terms, cell $(b, 0)$ computes Δ_1^{b+1}, $\Delta_2^{b+2}, \cdots, \Delta_{n-b}^n$, and the other cells of block b provide proper delays so that correct input sequence is presented to block $b+1$. The details are given below.

Each cell consists of 8 registers: D_0, D_1, V_0, V_1, H_0, H_1, B and A. Registers D_0, D_1, V_0, V_1, H_0 and H_1 provide the necessary delays, and initially each of them contains $-$, different from $\varnothing$. Register A provides proper control sequencing. For cell $(b, 0)$, $1 \leqq b \leqq n-1$, register B stores partial results of computation of convolutions, and initially it contains $\varnothing$. The computations are performed in 3 phases as given below.

Phase 1. Let the cell where the head starts be cell $(0, 0)$. When the symbol a_i is received, if $a_i \neq \#$, then Δ_i^i is computed and stored in register B of the cell scanned (i.e., cell $(0, i-1)$), and the head moves right by one cell. If the symbol received is $\#$, then the head moves left by one cell, and starts Phase 2. This phase takes $n+1$ steps.

Phase 2. By proper control signals, a 0 is stored in register A of cell (b, j), for $1 \leqq b \leqq n-2$ and $1 \leqq j \leqq n$; a 1 is stored in register A of cell $(b, 0)$, for $1 \leqq b \leqq n-1$; and cell $(n-1, 0)$ is marked distinctly, to facilitate the positioning of the head over cell $(n-1, 0)$ in Phase 3. Then Phase 3 is initiated; and in parallel, cell $(0, n)$ successively places

$$(\Delta_1^1, \Delta_2^2), (-, -)^{n-2}, (\Delta_2^2, \Delta_3^3), (-, -)^{n-2}, \cdots, (\Delta_{n-1}^{n-1}, \Delta_n^n)$$

in registers (V_0, H_0) of cell $(1, 0)$ in $(n-2)(n-1)+1$ consecutive steps ($((-,-)^{n-2}$ stands for $(-,-),(-,-),\cdots,(-,-)$ ($n-2$ times)). We skip the details, which are rather involved but trivial in nature. The part of Phase 2, done before Phase 3 starts, can be performed in cn^2 steps, for some constant c; and the rest of Phase 2 requires $(n-2)(n-1)+1$ steps.

Phase 3. This is the main computational phase; let the instant when this phase starts be instant 0. From the description of Phase 2, $(V_0, H_0)_t^{1,0}$ is $(\Delta_{i+1}^{i+1}, \Delta_{i+2}^{i+2})$ if $t = (n-1)i + 1$, for $0 \leqq i \leqq n-2$ and $(-,-)$ otherwise. Also, $(D_0, D_1, V_0, V_1, H_0, H_1, B)_0^{b,j} = (-,-,-,-,-,-,\varnothing)$, for $1 \leqq b \leqq n-2$ and $0 \leqq j \leqq n$, or $(b, j) = (n-1, 0)$. The cells undergo the following parallel transformations (notationally, superscript l stands for the left neighbor):

$A = 1$ *cells.*

$$\text{(I)}\qquad D_0 \leftarrow \begin{cases} D_0 & \text{if } D_1^l = -, \\ D_1^l & \text{otherwise;} \end{cases}$$

$$\text{(II)}\qquad (V_0, H_0) \leftarrow \begin{cases} (V_1^l, H_1^l) & \text{if } V_1^l, H_1^l \neq -, \\ (D_0, H_1^l) & \text{if } V_1^l = - \text{ and } H_1^l, D_0 \neq -, \\ (-,-) & \text{otherwise;} \end{cases}$$

$$\text{(III)}\qquad B \leftarrow \begin{cases} B \cup V_0 * H_0 & \text{if } V_0 \neq -, \\ \varnothing & \text{otherwise;} \end{cases}$$

$$\text{(IV)}\qquad V_1 \leftarrow V_0;$$

$$\text{(V)}\qquad (H_1, D_1) \leftarrow \begin{cases} (B, B) & \text{if } V_0 = - \text{ and } V_1 \neq -, \\ (H_0, -) & \text{otherwise.} \end{cases}$$

$A = 0$ *cells.*

$$\text{(VI)}\qquad (D_1, V_0, V_1, H_1) \leftarrow (D_1^l, V_1^l, V_0, H_1^l);$$

$$\text{(VII)}\qquad (D_0, H_0, B) \leftarrow (D_0, H_0, B).$$

In parallel with these transformations, the head (which is on cell $(0, n)$) moves right 1 cell/2 steps until it scans cell $(n-1, 0)$ (marked in Phase 2); and in the next step, cell $(n-1, 0)$ goes into state q_a or q_r, depending upon whether or not its register B contains S. The head scans cell $(n-1, 0)$ at the $(2(n-2)(n+1)+2)$nd instant. In the following, we show that register B of cell $(n-1, 0)$ contains Δ_1^n at instant $2(n-2)(n+1)+2$, verifying the correctness of the recognition process.

Notation. For $1 \leqq b \leqq n-1$, let $\tau_b = (b-1)(2n+1)$ and $\gamma_b = n - b - 1$.

LEMMA 2.1. *For* $1 \leqq b \leqq n-1$ *and* $t \geqq 0$, $(V_0, H_0)_t^{b,0}$ *is given by*

$$\begin{cases} (\Delta_{i+1}^{b+i-j+1}, \Delta_{b+i-j+2}^{b+i+1}) & \textit{if } t = \tau_b + (n-1)i + j, \quad \textit{where } 0 \leqq i \leqq \gamma_b, \quad 1 \leqq j \leqq b, \\ (-,-) & \textit{otherwise.} \end{cases}$$

Proof. Lemma 2.1 holds for $b = 1$ (from Phase 2). Assume that the lemma holds for any $b \leqq n-2$. From transformation (III) above, it can be easily proved that

$$(B)_t^{b,0} = \begin{cases} \bigcup_{1 \leqq m \leqq j} (\Delta_{i+1}^{b+i-m+1} * \Delta_{b+i-m+2}^{b+i+1}) & \text{if } t = \tau_b + (n-1)i + j + 1, \\ & 0 \leqq i \leqq \gamma_b, \quad 1 \leqq j \leqq b, \\ \varnothing & \text{otherwise.} \end{cases}$$

Thus

$$(1) \qquad (B)_{\tau_b+(n-1)i+b+1}^{b,0} = \Delta_{i+1}^{b+i+1} \quad \text{for } 0 \leqq i \leqq \gamma_b.$$

By transformation (IV) above, $(V_1)_t^{b,0} = (V_0)_{t-1}^{b,0}$. Hence $(V_0)_t^{b,0} = -$ and $(V_1)_t^{b,0} \neq -$ if and only if $t = \tau_b + (n-1)i + b + 1, 0 \leqq i \leqq \gamma_b$. Hence from transformation (V) and (1),

$$(H_1)_t^{b,0} = \begin{cases} \Delta_{b+i-j+2}^{b+i+1} & \text{if } t = \tau_b + (n-1)i + j + 1, \quad 0 \leqq i \leqq \gamma_b, \\ & 1 \leqq j \leqq b+1, \\ - & \text{otherwise;} \end{cases}$$

$$(D_1)_t^{b,0} = \begin{cases} \Delta_{i+1}^{b+i+1} & \text{if } t = \tau_b + (n-1)i + b + 2, \quad 0 \leqq i \leqq \gamma_b, \\ - & \text{otherwise.} \end{cases}$$

From transformations (IV) and (VI), $(V_1)_t^{b,n} = (V_1)_{t-2n}^{b,0} = (V_0)_{t-2n-1}^{b,0}$, and $(H_1, D_1)_t^{b,n} = (H_1, D_1)_{t-n}^{b,0}$. Hence

$$(V_1)_t^{b,n} = \begin{cases} \Delta_{i+1}^{b+i-j+1} & \text{if } t = \tau_b + (n-1)i + j + 2n + 1 \\ & \quad = \tau_{b+1} + (n-1)i + j, \quad 0 \leqq i \leqq \gamma_b, \\ & 1 \leqq j \leqq b, \\ - & \text{otherwise;} \end{cases}$$

$$(H_1)_t^{b,n} = \begin{cases} \Delta_{b+i-j+2}^{b+i+1} & \text{if } t = \tau_b + (n-1)i + j + 1 + n \\ & \quad = \tau_{b+1} + (n-1)i + j - n, \quad 0 \leqq i \leqq \gamma_b, \\ & 1 \leqq j \leqq b+1, \\ - & \text{otherwise;} \end{cases}$$

$$(D_1)_t^{b,n} = \begin{cases} \Delta_{i+1}^{b+i+1} & \text{if } t = \tau_b + (n-1)i + b + 2 + n \\ & \quad = \tau_{b+1} + (n-1)i + b + 1 - n, \quad 0 \leqq i \leqq \gamma_b, \\ - & \text{otherwise.} \end{cases}$$

Now from transformation (I),

$$(D_0)_t^{b+1,0} = - \quad \text{if and only if} \quad t \leqq \tau_{b+1} + b - 1 - n;$$

and

$$(D_0)_{\tau_{b+1}+(n-1)i}^{b+1,0} = \Delta_{i+1}^{b+i+1} \quad \text{for } 0 \leqq i \leqq \gamma_b.$$

Thus $(V_1)_t^{b,n}$, $(H_1)_t^{b,n} \neq -$ if and only if $t = \tau_{b+1} + (n-1)i + j$, $0 \leqq i \leqq \gamma_b - 1$, $1 \leqq j \leqq b$. $(V_1)_t^{b,n} = -$ and $(H_1)_t^{b,n}, (D_0)_t^{b,n} \neq -$ if and only if $t = \tau_{b+1} + (n-1)i$,

$0 \leqq i \leqq \gamma_b - 1$. Hence by transformation (II),

$$(V_0, H_0)_t^{b+1,0} = \begin{cases} (\Delta_{i+1}^{b+i+1}, \Delta_{b+i+2}^{b+i+2}) & \text{if } t = \tau_{b+1} + (n-1)i + 1, \\ & \quad 0 \leqq i \leqq \gamma_b - 1, \\ (\Delta_{i+1}^{b+i-j+2}, \Delta_{b+i-j+3}^{b+i+2}) & \text{if } t = \tau_{b+1} + (n-1)i + j, \\ & \quad 0 \leqq i \leqq \gamma_b - 1, \quad 2 \leqq j \leqq b+1, \\ (-,-) & \text{otherwise.} \end{cases}$$

This can be rewritten in the form

$$(V_0, H_0)_t^{b+1,0} = \begin{cases} (\Delta_{i+1}^{b+1+i-j+1}, \Delta_{b+1+i-j+2}^{b+1+i+1}) & \text{if } t = \tau_{b+1} + (n-1)i + j, \\ & \quad 0 \leqq i \leqq \gamma_{b+1}, \quad 1 \leqq j \leqq b+1, \\ (-,-) & \text{otherwise.} \end{cases}$$

This proves the lemma. Q.E.D.

Thus for $b = n - 1$,

$$(V_0, H_0)_t^{n-1,0} = \begin{cases} (\Delta_1^{n-j}, \Delta_{n-j+1}^{n}) & \text{if } t = (n-2)(2n+1) + j, \quad 1 \leqq j \leqq n-1, \\ (-,-) & \text{otherwise.} \end{cases}$$

Thus by transformation (III), $(B)_{(n-2)(2n+1)+n}^{n-1,0} = \Delta_1^n$, which verifies the recognition process.

The whole process takes $n + 1 + cn^2 + 2(n-2)(n+1) + 3$ steps $= c'n^2$, for some constant c'. Now the theorem follows from linear speedup. When $n = 1$, cell $(0, 0)$ handles input $a_1 \#$ as a special case. Q.E.D.

6. Conclusions. We studied the recognition of CFL's, using Younger's algorithm, by arrays of finite state machines. It would be interesting to investigate whether Earley's [4] algorithm could give equally efficient realizations on array machines.

Acknowledgments. This is a part of the author's thesis [8] done under the stimulating guidance of Professors Aravind Joshi and Hisao Yamada. Thanks are due to one of the referees for his many significant suggestions.

REFERENCES

[1] W. T. Beyer, *Recognition of topological invariants by iterative arrays*, Project MAC, Tech. Rep. TR-66, Mass. Inst. of Tech., Cambridge, Mass., 1969.

[2] S. N. Cole, *Real time computation by n-dimensional iterative arrays of finite-state machines*, IEEE Trans. Computers, 18 (1969), pp. 349–365.

[3] ———, *Deterministic pushdown store machines and real time computation*, J. Assoc. Comput. Mach., 18 (1971), pp. 306–328.

[4] J. Earley, *An efficient context-free parsing algorithm*, Comm. ACM, 13 (1970), pp. 94–102.

[5] V. C. Hamacher, *A class of parallel processing automata*, Ph.D. thesis (Tech. Rep. TR-68-6), Syracuse Univ., 1968.

[6] J. E. Hopcroft and J. D. Ullman, *Formal Languages and their Relation to Automata*, Addison-Wesley, Reading, Mass., 1969.

[7] S. R. Kosaraju, *Recognition of context-free and stack languages*, IEEE Switching and Automata Theory Conf., 1969, pp. 129–132.

[8] ———, *Computations on iterative automata*, Ph.D. thesis, Univ. of Pennsylvania, Philadelphia, 1969.

[9] A. R. SMITH III, *Two-dimensional formal languages and pattern recognition by cellular automata*, IEEE Switching and Automata Theory Conf., 1971, pp. 144–152.

[10] L. G. VALIANT, *General context-free recognition in less than cubic time*, Tech. Rep., Carnegie–Mellon Univ., Pittsburgh, January 1974.

[11] A. WAKSMAN, *An optimum solution to the firing squad synchronization problem*, Information and Control, 9 (1966), pp. 66–78.

[12] D. H. YOUNGER, *Recognition and parsing of context-free languages*, Ibid., 10 (1967), pp. 189–208.

SIAM J. COMPUT.
Vol. 4, No. 3, September 1975

OPTIMUM PARTITIONS OF TREE ADDRESSING STRUCTURES*

W. H. HOSKEN†

Abstract. We consider the problem of finding the best partition of a binary tree addressing structure where the maximum block size is given and a one-block buffer is available. An algorithm is presented for finding an optimum partition. The algorithm operates in time proportional to $N \cdot n^2$, where N is the number of nodes in the tree and n is the block size.

Key words. data base, index, tree, partition, access

Introduction. In large data base systems the directories or indexes involved in the access method are often large files in their own right. In this case, the access information must be partitioned into blocks and, assuming random access to blocks, the dominant cost of a search becomes the number of block accesses in a search. We consider here the case of a tree structured access method. Searches are assumed to begin at the root of the tree and proceed to the descendants. Weights (e.g., the frequency that a node is the terminal point of a search) are assumed for each node.

We present an algorithm to find a partition of a weighted binary tree so that the "weighted path length" is minimal where "path length" to a node v is the number of times blocks are entered in a path from the root to v. The algorithm operates in time proportional to $N \cdot n^2$, where N is the number of nodes in the tree and n is the maximum number of nodes in a block.

The algorithm may be extended in a straight-forward way to more general trees. However, the operating time increases drastically.

This problem suggested itself in reading Knuth [1], and variants of it are treated in Muntz and Uzgalis [2] and Kernighan [3], [4].

1. Preliminary definitions and notation. A weighted binary tree is a structure $\tau = \langle V, I, r, L, R, W \rangle$ where

V is the finite set of nodes,

$I \subseteq V$ is the set of internal nodes,

$V - I$ is the set of leaves,

$r \in V$ is the root,

$L : I \to V$ determines the left son,

$R : I \to V$ determines the right son,

$W : V \to \mathbf{N} \cup \{0\}$ assigns nonnegative integer weights to nodes.

The following conditions hold:

1. $L(v) = L(v')$ implies $v = v'$, $R(v) = R(v')$ implies $v = v'$, $L(v) \neq R(v)$ for any v.

2. The root r is the only node not in the range of L or R. For nonroot nodes v, $F(v)$ (the father of v) is the node such that $R(F(v)) = v$ or $L(F(v)) = v$.

A path from v_1 to v_m in τ is a sequence of nodes $v_1, v_2, \cdots, v_m$ where $v_{i-1} = F(v_i)$ for $i = 2, \cdots, m$. Of course, if a path exists between nodes, it is unique.

* Received by the editors May 28, 1974, and in final revised form September 3, 1974.

† Whitmore Laboratory, Computer Science Department, Pennsylvania State University, University Park, Pennsylvania 16802.

The height, $ht(v)$, of a node v is the length of the longest path from v to any leaf, i.e.,

$$\begin{aligned} ht(v) &= 1 \quad \text{for leaves } v, \\ &= \max\,[ht(L(v)), ht(R(v))] + 1 \quad \text{for nonleaves}. \end{aligned}$$

The subtree of τ at v is the binary tree with v as root and including all of the descendants (in τ) of v. A part of a binary tree is a set B of nodes of a subtree τ' of τ which includes the root of τ' and for any $v \in B$ not the root of τ', includes the father of v. The function $s(v)$ is the number of nodes in the subtree at v.

An n-partition P of τ is a partition of V with at most n nodes in any block. To simplify a later statement it will be useful to define an n-j-partition as an n-partition with at most j nodes in the block with the root.

If P is a partition of a weighted binary tree τ, the cost of P is defined as

$$\text{Cost}\,(P) = \sum_{v \in V} M(v)W(v),$$

where $M(v)$ is the P-path length defined as follows: Let $r = v_1, v_2, \cdots, v_m = v$ be a path in τ from root r to node v. Then

$$M(v_1) = 1,$$

$$M(v_{i+1}) = \begin{cases} M(v_i) & \text{if } v_i \text{ and } v_{i+1} \text{ are in the same block}, \\ M(v_i) + 1 & \text{otherwise}. \end{cases}$$

2. Preliminary remarks. Before presenting the algorithm it will be useful to show that least cost n-partitions of a simple form can be found.

LEMMA 1. *Let τ be a weighted binary tree. There is a least cost n-partition P of τ with blocks that are parts of τ.*

Proof. First observe that a block B containing the root will not be a part of τ if and only if there is another node v in B such that $F(v)$ is not in B. For other blocks B, B will not be a part of τ if and only if there are distinct nodes u and v in B such that $F(u) \notin B$ and $F(v) \notin B$.

Suppose a block B contains distinct nodes $v_1, v_2, \cdots, v_m$ with $m > 1$ and $F(v_i) \notin B$ (or $v_i = r$) for $i = 1, 2, \cdots, m$. A new partition P' can be defined, splitting B into m blocks $B_1, B_2, \cdots, B_m$ as follows: Include in B_i the node v_i and all nodes v in B for which the entire path from v_i to v lies in B. Note that each block B_i will be a part of τ. The cost of P' is the same as the cost of P since the P-path length M is the same as the P'-path length M' for each node $v \in V$.

This process may be repeated for all blocks that are not parts of τ. Q.E.D.

In the proof of Lemma 1 it appears that requiring blocks to be parts of τ results in many partially full blocks. Lemma 2 shows that least cost partitions can be found where the only partially full blocks have no descendant blocks.

LEMMA 2. *Let τ be a weighted binary tree. There is a least cost n-partition P of τ into blocks that are parts of τ. Furthermore, for all blocks B with a node v such that $F(v) \in B$ but $v \notin B$, there are n nodes in B.*

Proof. Let P be a least cost n-partition of τ with blocks that are parts of τ. Let B be a block of P with fewer than n nodes. Let v be a node such that $F(v) \in B$ but $v \notin B$. Let v be in block C. A new n-partition P' can be formed changing B to

$B' = B \cup \{v\}$ and C to $C' = C - \{v\}$. Note that Cost $(P') \leqq$ Cost (P). If C' is not a part of τ, a new n-partition P'' can be formed using the block splitting process described in the proof of Lemma 1, so that

$$\text{Cost}\,(P'') = \text{Cost}\,(P') \leqq \text{Cost}\,(P).$$

This process can be repeated so long as blocks violating the condition of the lemma exist. Since P is itself a tree structure (with blocks taken as nodes, etc.) and the process of obtaining P' from P increases the size of parent blocks, eventually the only partially full blocks will have no descendant blocks. Q.E.D.

From these lemmas it can be seen that the problem of finding a least cost n-partition of a weighted binary tree τ can be reduced to the problem of finding the best part B of τ including the root and then repeating this on the subtrees with roots that are sons of members of B but not in B. A nondeterministic process forming the block with the root may be described as follows:

(i) Include the root in the block.

(ii) If there are no nodes remaining, stop.

(iii) If there are n nodes in the block, stop. Otherwise, include some son of some member of the block.

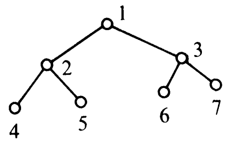
(iv) repeat step (ii).

Example 1 shows that the heuristic process of including in step (iii) the son which is the root of the highest weight subtree does not always yield a least cost n-partition.

Example 1.

$V = \{1, 2, 3, 4, 5, 6, 7\}$,
$r = 1$,
$W(1) = W(2) = W(3) = 0$,
$W(4) = W(5) = W(6) = W(7) = 1$,
$N = 7$,
$n = 3$,

τ:



(a) $P_1 = \{B_1, B_2, B_3, B_4, B_5\}$,
$B_1 = \{1, 2, 3\}$,
$B_2 = \{4\}$, $B_3 = \{5\}$, $B_4 = \{6\}$, $B_5 = \{7\}$,
$$\begin{aligned}\text{Cost} &= 1 \cdot W(1) + 1 \cdot W(2) + 1 \cdot W(3) + 2 \cdot W(4) + 2 \cdot W(5) \\ &\quad + 2 \cdot W(6) + 2 \cdot W(7) \\ &= 8.\end{aligned}$$

(b) $P_2 = \{B_1, B_2, B_3\}$,
$B_1 = \{1, 2, 4\}$,
$B_2 = \{5\}$,
$B_3 = \{3, 6, 7\}$,

$$\begin{aligned}\text{Cost} &= 1 \cdot W(1) + 1 \cdot W(2) + 2 \cdot W(3) + 1 \cdot W(4) + 2 \cdot W(5) \\ &\quad + 2 \cdot W(6) + 2 \cdot W(7) \\ &= 7.\end{aligned}$$

3. The algorithm. The algorithm may now be stated. A two pass process is envisioned. During pass one each internal node v is labeled (bottom up) with numbers $\operatorname{tag}(v,j)$, where $0 \leqq \operatorname{tag}(v,j) \leqq j - 1$ and $j = 1, 2, \cdots, n$. Each $\operatorname{tag}(v,j)$ is the number of nodes from the subtree at $L(v)$ to be included with v in attaining a least cost n-j-partition of the subtree at v.

In order to calculate $\operatorname{tag}(v, j)$ at any node, auxiliary information must be used.

Let $W'(v) = W(v)$ for v a leaf,

$= W'(L(v)) + W'(R(v)) + W(v)$ for v a nonleaf.

For $j = 1, 2, \cdots, n$:

If $s(v) \leqq j$, then $C(v, j) = W'(v)$, and if v is a leaf, $\operatorname{tag}(v, j) = 0$, otherwise $\operatorname{tag}(v, j) = s(L(v))$.

If $s(v) > j$, then

$$C(v, j) = \min_{i=0}^{j-1} [C(L(v), i) + C(R(v), j - i - 1)] + W(v)$$

and $\operatorname{tag}(v, j)$ is any i which achieves this $C(v, j)$.

For $j = 0$, $C(v, 0) = C(v, n) + W'(v)$.

During pass two the tags are used to determine the blocks. To determine the block B containing the root, proceed as follows. If $s(r) \leqq n$, then B consists of all the nodes. Otherwise, B is defined according to the following process.

The root r is included in B and $b(r) = n$.

If v is a node and $F(v)$ has been included in B, then v will be included in B and have value $b(v)$ as follows:

Case 1. If v is a left son of $F(v)$ and $\operatorname{tag}(F(v), b(F(v))) \neq 0$, then v is included in B and $b(v) = \operatorname{tag}(F(v), b(F(v)))$.

Case 2. If v is a right son of $F(v)$ and $\operatorname{tag}(F(v), b(F(v))) \neq b(F(v)) - 1$, then v is included in B and $b(v) = b(F(v)) - \operatorname{tag}(F(v), b(F(v))) - 1$.

Once the block B is defined, this process is repeated recursively on each of the subtrees with roots $v \notin B$ where $F(v) \in B$. In this way, an n-partition is defined.

To see that the n-partition so defined is a least cost n-partition, consider the definition of the tags. The value $b(v)$ at node v indicates that a part of size $b(v)$ is to be used at v. Now $\operatorname{tag}(v, b(v))$ is the number of nodes from the subtree at $L(v)$ to be used in the least cost n-partition of the subtree at v with the added condition that there are only $b(v)$ nodes in the block containing v. Thus $L(v)$ should be included in the same block with v if $\operatorname{tag}(v, b(v)) \neq 0$. Similarly for $R(v)$, but to make up a part of size $b(v)$, $b(v) - \operatorname{tag}(v, b(v)) - 1$ nodes come from the subtree at $R(v)$.

Example 2. It will be instructive to work out an example. Let τ be as in Example 1 and $n = 3$.

First, since 4 is a leaf,

$$C(4, 1) = C(4, 2) = C(4, 3) = 1, \qquad C(4, 0) = 2.$$

Similarly,

$$C(5, j) = C(6, j) = C(7, j) = C(4, j), \qquad j = 0, \cdots, 3.$$

Therefore,

$$\text{tag}(4, 1) = \text{tag}(4, 2) = \text{tag}(4, 3) = 0,$$

and

$$\text{tag}(5, j) = \text{tag}(6, j) = \text{tag}(7, j) = \text{tag}(4, j), \qquad j = 1, 2, 3.$$

Next,

$$C(2, 1) = C(4, 0) + C(5, 0) = 4 \quad \text{and } \text{tag}(2, 1) = 0,$$

$$C(2, 2) = \min\begin{Bmatrix} C(4, 0) + C(5, 1) = 3 \\ C(4, 1) + C(5, 0) = 3 \end{Bmatrix} = 3 \quad \text{so tag}(2, 2) = 0,$$

(Whenever two values of tag (v, j) are possible, the smaller has been chosen.)

$$C(2, 3) = W'(2) = 2 \quad \text{so tag}(2, 3) = 1,$$

$$C(2, 0) = C(2, 3) + W'(2) = 4.$$

Similarly,

$$C(3, j) = C(2, j), \qquad j = 0, \cdots, 3.$$

$$\text{tag}(3, j) = \text{tag}(2, j), \qquad j = 1, 2, 3.$$

For the root,

$$C(1, 1) = C(2, 0) + C(3, 0) = 8 \quad \text{so tag}(1, 1) = 0,$$

$$C(1, 2) = \min\begin{Bmatrix} C(2, 0) + C(3, 1) = 8 \\ C(2, 1) + C(3, 0) = 8 \end{Bmatrix} = 8 \quad \text{so tag}(1, 2) = 0,$$

$$C(1, 3) = \min\begin{Bmatrix} C(2, 0) + C(3, 2) = 7 \\ C(2, 1) + C(3, 1) = 8 \\ C(2, 2) + C(3, 0) = 7 \end{Bmatrix} = 7 \quad \text{so tag}(1, 3) = 0,$$

$$C(1, 0) = C(1, 3) + W'(1) = 11.$$

In pass two:

$b(1) = 3$ and tag $(1, 3) = 0$ so 2 is not included.
$3 - \text{tag}(1, 3) - 1 = 2 \neq 0$ so 3 is included.
$b(3) = 2$ and tag $(3, 2) = 0$ so 6 is not included.
$2 - \text{tag}(3, 2) - 1 = 1 \neq 0$ so 7 is included.
$b(7) = 1$.

The block with 1 is $\{1, 3, 7\}$ and the process is repeated on the trees at 2 and 6. Since $b(2) = 3$ and $s(2) = 3$, the block with 2 is $\{2, 4, 5\}$. Finally, $b(6) = 3$ and $s(6) = 1$, so the block with 6 is $\{6\}$.

4. Concluding remarks. The operating time for the algorithm is proportional to $N \cdot n^2$. The operation $\min_{i=1}^{j-1} (C(L(v), i) + C(R(v), j - i - 1))$, $j = 1, \cdots, n$, is done at each node and contributes the factor of n^2.

No examples have been found where the heuristic process (including with the father, the son with the highest weight subtree) gives a partition of more than 30%

of the least cost. Example 3 shows that some partitions with full internal blocks are quite poor with respect to optimum.

Example 3.

Let $v = \{1, 2, \cdots, 4 \cdot m, 4 \cdot m + 1\}$,

$r = 1$,

$I = \left\{ \begin{matrix} 1, 2, \cdots, m \\ m + 2, m + 3, \cdots, 2 \cdot m + 1 \end{matrix} \right\}$,

$L(i) = i + 1, i = 1, \cdots, m$,

$L(i) = i + m, i = m + 2, \cdots, 2 \cdot m + 1$,

$R(i) = i + m + 1, i = 1, \cdots, m$,

$R(i) = i + 2 \cdot m, i = m + 2, \cdots, 2 \cdot m + 1$,

$W(v) = 1$ for leaves,

$= 0$, otherwise.

$n = 4$.

(a) Let $P = \{B_1, B_2, \cdots, B_m, B_{m+1}\}$ where

$B_i = \{i, R(i), L(R(i)), R(R(i))\}, i = 1, \cdots, m$,

$B_{m+1} = \{m + 1\}$,

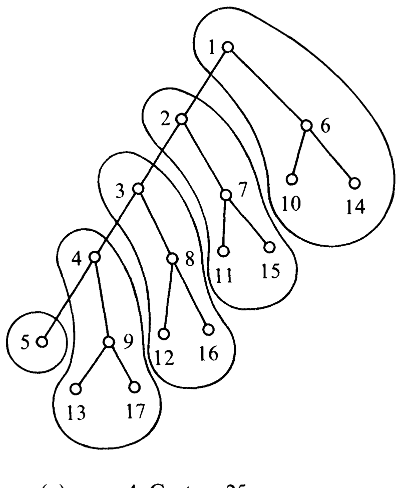
$$\begin{aligned} \text{Cost}(P) &= 2 \cdot 1 + \cdots + 2 \cdot i + \cdots + 2 \cdot m + (m + 1) \\ &= (m + 1)^2 \\ &\approx m^2 \quad \text{for large } m. \end{aligned}$$

(b) Let $P' = \{B'_1, B'_2, \cdots, B'_{m/4}, C'_1, \cdots, C'_m, C'_{m+1}\}$, where

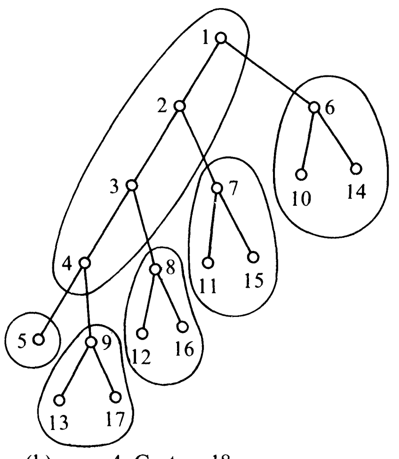
$B'_i = \{4 \cdot (i - 1) + 1, 4 \cdot (i - 1) + 2, 4 \cdot (i - 1) + 3, 4 \cdot (i - 1) + 4\}$,
$i = 1, \cdots, m/4$,

$C'_i = \{R(i), L(R(i)), R(R(i))\}, i = 1, \cdots, m$,

$C'_{m+1} = \{m + 1\}$,



(a) $m = 4$, Cost $= 25$



(b) $m = 4$, Cost $= 18$

$$\begin{aligned}\text{Cost}\,(P') &= 8\cdot 2 + 8\cdot 3 + \cdots + 8\cdot(m/4 + 1) + (m/4 + 1)\\ &= 4\cdot[(m/4 + 1)\cdot(m/4 + 2) - 2] + m/4 + 1\\ &\approx m^2/4 \quad \text{for large } m.\end{aligned}$$

Acknowledgments. The author wishes to thank E. G. Coffman, Jr., and J. Bruno for helpful discussions.

REFERENCES

[1] D. E. KNUTH, *The Art of Computer Programming*, vol. 3, Addison-Wesley, Reading, Mass., 1973, pp. 472–473.

[2] R. MUNTZ AND R. UZGALIS, *Dynamic storage allocation for binary trees in a two level memory*, Proc. Princeton Conf. on Inf. Sci. and Systems, 4 (1970), pp. 345–349.

[3] B. W. KERNIGHAN, *Optimal partitions for a class of subroutine graphs*, Ibid., 4 (1970), pp. 350–354.

[4] ———, *Optimal sequential partitions of graphs*, J. Assoc. Comput. Mach., 18 (1971), pp. 34–40.

SIAM J. Comput.
Vol. 4, No. 3, September 1975

PARALLELISM IN COMPARISON PROBLEMS*

LESLIE G. VALIANT†

Abstract. The worst-case time complexity of algorithms for multiprocessor computers with binary comparisons as the basic operations is investigated. It is shown that for the problems of finding the maximum, sorting, and merging a pair of sorted lists, if n, the size of the input set, is not less than k, the number of processors, speedups of at least $O(k/\log\log k)$ can be achieved with respect to comparison operations. The algorithm for finding the maximum is shown to be optimal for all values of k and n.

Key words. parallel algorithms, comparison problems, sorting merging, tournaments, complexity

Introduction. We investigate the worst-case time complexity of parallel binary-comparison algorithms for the classical problems of merging, sorting, and finding the maximum. We do this for a model that in several senses can be regarded as embodying the intrinsic difficulty of solving these problems on a multiprocessor computer. Any lower bound on the time complexity of a task for this model will necessarily also be a bound for any other model of parallelism that has binary comparisons as the basic operations. Furthermore the best constructive upper bounds will correspond to the fastest algorithms for independent processor machines whenever the time taken to perform a comparison dominates all the overheads.

For each problem the input consists of a set of elements on which there is a linear ordering. The ordering relationship between any pair of elements can be discovered by performing a comparison operation on them. In our model there are k processors available, and therefore k comparisons can be performed simultaneously. The processors are synchronized so that within each time interval each of them completes a comparison. At the end of the interval the algorithm decides, by inspecting the ordering relationships that have already been established, which k (not necessarily disjoint) pairs of elements are to be compared during the next interval, and assigns processors to them. The computation terminates when the relationships that have been discovered are sufficient to specify the solution to the given problem.

The time complexity of each problem will be expressed as a function of the number of processors, and of the size of the input set. The function will give the number of time intervals taken for a worst-case input by the comparison algorithm that requires the least time in the worst case. Thus we define $\max_k(n)$ to be this measure of complexity for the problem of finding the maximum of n elements on a k processor machine. $\mathrm{Sort}_k(n)$ is defined analogously for putting n elements in order, and $\mathrm{Merge}_k(m, n)$ for merging two sorted lists of length m, n respectively.

The phenomena we exhibit for the three problems share certain qualitative features. For a given size of input set, the more processors we have available, the shorter the computation time. However, the price paid for increased speed is increased total number comparisons. Intuitively, we can say that the larger k is, the larger the number of comparisons that at each step we have to choose on the

* Received by the editors February 5, 1974.

† Centre for Computer Studies, University of Leeds, Leeds LS2 9JT, England. This research was carried out at Carnegie-Mellon University, Pittsburgh, Pennsylvania.

basis of fixed previous information, and consequently the lower the "average quality" of the choices made. For any given task P, we can conveniently measure this phenomenon by the "speedup factor" P_1/P_k, where P_i is the worst-case time complexity on P on i processors. The success of parallelization can then be judged by observing how close this speedup factor is to k.

That there are mathematically degenerate extreme cases has been observed before. All the problems can be solved in unit time if there are enough processors for every element to be compared with every other simultaneously. The speedup then, however, is rather small ($\sim\sqrt{k}$). At the other extreme, as the input set becomes very large in relation to k, then, as observed by Borodin and Munro [5], optimal speedups can be approached. Furthermore, such speedups can be attained by algorithms that use the processors largely independently (as in Corollaries 3, 6, 7, 9 below) and that are therefore efficient even on machines for which inter-processor communication is relatively expensive.

Here, however, we shall focus especially on the intermediate cases. As the fastest parallel algorithms previously studied for the case $k = n = m$ are those that can be realized on sorting networks (Batcher [2], Knuth [6]), it will be of interest to compare the results for these with our analysis. Thus, to find the maximum of k elements on k processors can be done, and requires $\lceil \log_2 k \rceil$ steps on a network. It is natural to ask whether better utilization of the available processors can be made if the network restriction is removed. For merging two lists of k elements on k processors again $O(\log_2 k)$ time is necessary and can be achieved. In this case, it has, furthermore, been proved (by R. W. Floyd [6]) that $O(k \log_2 k)$ comparisons are necessary, and hence that, under the network constraint, near optimal use of the k processors is being made. The question is whether the $\log_2 k$ bound represents the intrinsic complexity of the merging problem or is a consequence only of the extra constraints.

Even if the network restriction is relaxed to allow arbitrary disjoint comparisons, it is easy to see that the $\log_2 k$ lower bound remains for both problems. What our results show is that for the more general model, this barrier no longer exists. We note, however, that the overheads implied by our algorithms may grow as $\log k$, i.e., faster than the bounds we shall derive. Thus although we may validly ignore overheads for any fixed value of k, it will not be meaningful to do so asymptotically.

1. The maximum.

We give a worst-case analysis of the problem of finding the maximum of n elements using k processors. We consider the case of $k = n$ first, and then show how solutions to all the others can be derived. The theorems are stated in the form of asymptotic inequalities. However, it will be apparent that the analysis itself is complete in the sense that given any k and n, a provably optimal algorithm can be developed using the observations made in the proofs. Although, for simplicity, we shall not explicitly consider the possibility of two elements being equal, our arguments apply to that case as well, as long as just one of the maximal elements is being sought.

THEOREM 1. *For $k = n > 1$,*

$$\max_k(n) \geqq \log\log n - \text{const}.$$

Proof. Consider the execution of an arbitrary comparison algorithm for finding the maximum of n elements. Let C_i be the set of all elements that up to time i have not been shown to be smaller than any other. Call these the *candidates* at time i, and denote their number by c_i.

To prove the theorem, we show that given $k = n$ and c_i, the value of c_{i+1} can be bounded from below. The result is then deduced by induction on i.

Suppose that in the next time interval in every comparison between a noncandidate and a candidate the candidate turns out to be the larger. Then the results of these comparisons will clearly not contribute to any reduction in the candidate population at all. Clearly, this will also be true for any comparisons that involve only noncandidates. Therefore, in this worst case the only comparisons that do contribute to reducing the number of candidates are those among the candidates themselves.

To obtain the bound, we show that if n comparisons are made on c_i elements, then there must be a sufficiently large subset of these elements in which no pair has been compared directly. In the worst case, it is possible that the elements in this subset happen each to be larger than each of the elements outside this subset. In that case, they will clearly all still be candidates at time $i + 1$.

The inductive step can be reduced to a graph theoretic formulation if we identify elements with nodes and comparisons with arcs in the obvious way. We call a subset of the nodes of a graph *stable* if no pair from it is connected by an arc. We can then express the relationship we require as follows:

$$c_{i+1} \geqq \min \{\max \{h | G \text{ contains a stable set of size } h\} | \; G \text{ is a graph with } c_i \text{ nodes and } k \text{ arcs}\}.$$

As a corollary to Turan's theorem, it can be shown [3], that

$$c_{i+1} \geqq \frac{c_i^2}{2k + c_i}.$$

By solving this inequality, we get that, if $c_0 = n = k$, then for some constant, c_i will exceed unity as long as

$$i < \log \log n - \text{const}.$$

The result follows. □

COROLLARY 1. *If* $4 \leqq 2n \leqq k \leqq n(n-1)/2$, *then*

$$\max_k (n) \geqq \log \log n - \log \log (k/n) - \text{const}.$$

Proof. Solving the same inequality as above, i.e.,

$$c_{i+1} \geqq \frac{c_i^2}{2k + c_i}$$

with $c_0 = n$ gives the claimed solution. □

As we shall now indicate, not only are the known bounds on stability achievable, but the extremal graphs are such that comparison algorithms based on them do reduce the candidate population at an optimal rate.

THEOREM 2. *For $k = n > 1$,*

$$\max_k(n) \leqq \log\log n + \text{const}.$$

Proof. It is known [3], [7] that any graph with p nodes that has no stable set larger than x has at least as many arcs as the graph $G_{p,x}$. $G_{p,x}$ is defined to be the graph with p nodes that consists of x disjoint cliques of which $p - x(q - 1)$ have q nodes and the remaining $xq - p$ have $q - 1$ nodes, where $q = \lceil p/x \rceil$. It is easily shown that such a graph has $(q - 1)(2p - xq)/2$ nodes altogether.

In our parallel algorithm we shall at time i perform comparisons as dictated by some such graph with $p = c_i$. Clearly, c_{i+1} will equal x, since to each clique there will correspond exactly one candidate at time $i + 1$. To minimize c_{i+1} we shall have to use from among the graphs

$$\{G_{c_i,x} | x = 1, 2, \cdots ; G_{c_i,x} \text{ has fewer than } k \text{ arcs}\}$$

the one with the smallest index x. We therefore have that

$$c_{i+1} = \min\{x | (\lceil c_i/x \rceil - 1) \cdot (2c_i - x \cdot \lceil c_i/x \rceil)/2 \leqq k\}.$$

This relation gives the inequality

$$c_{i+1} \leqq \frac{c_i^2}{k \cdot \text{const.}}.$$

Solving for $c_0 = n = k$ gives that for some constant, $c_i = 1$ for some $i \leqq \log\log n + \text{const}$. The result follows.

From the considerations in the proof of Theorem 1, it is immediate that if $G_{c_i,x}$ is chosen at each step so as to minimize x, the implied algorithm is indeed optimal. □

COROLLARY 2. *For $4 \leqq 2n \leqq k \leqq n(n - 1)/2$,*

$$\max_k(n) \leqq \log\log n - \log\log(k/n) + \text{const}. \qquad \square$$

The remaining case, that of $k < n$, can be dealt with by the following observations. Clearly with just k comparisons we can reduce c_i by at most k at each step. However, as long as $c_i \geqq 2k$, we can achieve this optimal reduction by having k disjoint pairs from c_i compared. Furthermore, once c_i is less than $2k$, the algorithm of the previous theorem can take over. We therefore conclude the following.

COROLLARY 3. *For $1 < k < n$,*

$$n/k + \log\log k - \text{const.} < \max_k(n) < n/k + \log\log k + \text{const}. \qquad \square$$

For each case we have arrived at upper and lower bounds that differ only by additive constants. Furthermore, the method of deriving a provably optimal algorithm for any given values of k and n is implicit in our analysis. We conclude by mentioning that for the special case of $k = n$, we can state the exact result explicitly as follows.

COROLLARY 4. *The sequence $s_i, s_2 \cdots$ with the property that $s_i = \max\{y | \max_y(y) = i\}$ is defined by*

$$s_i = 3 \quad \textit{and } s_{i+1} = (2s_i + 1)s_i.$$

For some real number K, $s_i = \lfloor K^{2^i}/2 \rfloor$.

Proof. By induction on i. The given solution of the recurrence follows from the analysis of [1]. $\square$

2. Merging. We now give an algorithm for merging that is considerably faster than any corresponding algorithm previously known.

THEOREM 3. *For* $k = \lfloor\sqrt{mn}\rfloor$ *and* $1 < n \leqq m$,

$$\mathrm{Merge}_k(n, m) \leqq 2 \log\log n + \text{const}.$$

Proof. We proceed inductively, by showing how, given $\lfloor\sqrt{mn}\rfloor$ processors, we can, in two time intervals, reduce the problem of merging two lists of length n, m, respectively, to one of merging a number of pairs of lists, the shorter of each of which has length less than $\sqrt{n}$. The pairs of lists are so created that we can distribute the $\lfloor\sqrt{mn}\rfloor$ processors amongst them at the next stage in such a way as to ensure that for each pair there will be enough processors allocated to satisfy the inductive assumption.

Consider the following algorithm for the sorted lists $X = (x_1, x_2, \cdots, x_n)$, $Y = (y_1, y_2, \cdots, y_m)$.

(a) Mark the elements of X that are subscripted by $i \cdot \lceil\sqrt{n}\rceil$ and those of Y subscripted by $i \cdot \lceil\sqrt{m}\rceil$ for $i = 1, 2, \cdots$. There are at most $\lfloor\sqrt{n}\rfloor$ and $\lfloor\sqrt{m}\rfloor$ of these, respectively. The sublists between successive marked elements and after the last marked element in each list we call *segments.*

(b) Compare each marked element of X with each marked element of Y. This requires no more than $\lfloor\sqrt{nm}\rfloor$ comparisons and can be done in unit time.

(c) The comparisons of (b) will decide for each marked element the segment of the other list into which it needs to be inserted. Now compare each marked element of X with every element of the segment of Y that has thus been found for it. This requires at most

$$\lfloor\sqrt{n}\rfloor \cdot (\lceil\sqrt{m}\rceil - 1) < \lfloor\sqrt{nm}\rfloor$$

comparisons altogether and can also be done in unit time.

On the completion of (a), (b) and (c) we have identified where each of the marked elements of X belongs in Y. Thus there remain to be merged the disjoint pairs of sublists $(X_1, Y_1), (X_2, Y_2), \cdots$ where each X_i is a segment of X and, therefore, of length $|X_i| \leqq \lfloor\sqrt{n}\rfloor$. Furthermore, $\sum |X_i| < n$ and $\sum |Y_i| < m$ since the sublists are disjoint. But by Cauchy's inequality,

$$\sum \sqrt{(|X_i| \cdot |Y_i|)} \leqq \sqrt{(\sum |X_i| \cdot \sum |Y_i|)}.$$

It follows that

$$\sum \lfloor\sqrt{(|X_i| \cdot |Y_i|)}\rfloor \leqq \sum \sqrt{(|X_i| \cdot |Y_i|)} \leqq \lfloor\sqrt{mn}\rfloor.$$

There are therefore enough processors altogether that we can assign $\lfloor\sqrt{(|X_i| \cdot |Y_i|)}\rfloor$ to merge (X_i, Y_i) for each i simultaneously.

We have therefore established that the inductive process of successively splitting a pair of lists into a set of pairs of sublists can continue with the given number of processors. Furthermore, the length of the shorter component of each sublist pair is inductively bounded by the square root of the shorter component of the list pair. Thus at time $2i$, each pair of lists produced has a component of length no more than λ_i where

$$\lambda_i = \lfloor\sqrt{\lambda_{i-1}}\rfloor,$$

and $\lambda_0 = n$. Solving $\lambda_i \leqq \sqrt{\lambda_{i-1}}$ gives $\lambda_i \leqq n^{1/2^i}$. The merging process clearly terminates locally whenever a pair of sublists with a null component is produced. Thus merging must be complete before $\lambda_i = 0$. This gives that

$$\mathrm{Merge}_k(n, m) \leqq 2\lceil\log\log n + \mathrm{const.}\rceil$$

The additive constant can be shown to be less than unity if logarithms to the base 2 are taken. □

COROLLARY 5. *For* $k = \lfloor r\sqrt{nm}\rfloor$ *where* $1 < n \leqq m$ *and* $r \geqq 2$,

$$\mathrm{Merge}_k(n, m) \leqq 2(\log\log n - \log\log r) + \mathrm{const}.$$

Proof. We use the same algorithm as above, except that at step (a) the objects marked are those subscripted by $i \cdot \lceil\sqrt{(n/r)}\rceil$ in X and by $i \cdot \lceil\sqrt{(m/r)}\rceil$ in Y for $i = 1, 2, \cdots$. It is easily verified that steps (b) and (c) then each require no more than k comparisons, and can thus be done in unit time. Now $\lambda_i < \sqrt{(\lambda_{i-1}/r)}$, from which the result follows. □

COROLLARY 6. *For* $1 < k \leqq n \leqq m$,

$$\mathrm{Merge}_k(n, m) \leqq (n + m)/k + \log(mn \log k/k) + \mathrm{const}.$$

Proof. Mark $k - 1$ elements in each list so as to induce k segments of about uniform size (i.e., n/k and m/k) in each one. Merge the two lists of marked elements as in the above theorem. Insert each of the $2(k - 1)$ marked elements into the segment to which it belongs in the other list. This can be done in time $\log(mn/k)$. This leaves $2k$ pairs of disjoint sublists to be merged, in which no pair contains more than $(n + m)/k$ elements. It only remains to schedule how this merging is to be done on the k processors in time $(n + m)/k$ (as opposed to time $2(n + m)/k$).

The first observation is that the problem of merging a given pair of lists by the standard sequential algorithm (Knuth [6, p. 160]) can be split arbitrarily into two independent subproblems with no loss of efficiency. If the two lists have x elements altogether, then for any y we can divide the task into processes that take y and $x - y - 1$ steps, respectively. The two processes simply execute the first y and $x - y - 1$ steps, respectively, of the standard merging algorithm, but start from different ends of the lists.

With this freedom to break up the merging of a pair arbitrarily, we can schedule the whole task optimally as follows. We symbolically assign the ith processor jointly to the ith *segments* of the two lists. These segments have $(m + n)/k$ elements between them. To any sublist pair which has say z elements in common with this pair of segments, we assign z steps of the ith processor. Then clearly, we are assigning no more than $(m + n)/k$ steps altogether to each processor. Furthermore, since, by construction, each sublist is totally contained in some segment, each sublist pair will be assigned to at most two processors. With this scheduling, we can therefore carry out the remainder of the computation optimally. □

This last corollary is an improvement on one described in [5] (and attributed to Kirkpatrick) for the case $k \ll n = m$. Asymptotically, a speedup of k is clearly achieved, since it is known [6] that the merging of two lists of length n requires $2n - 1$ comparisons in the worst case.

The method suggested in [5] is essentially that described in the first paragraph of the proof above, with the suggestion that the number of elements initially marked in each list be not $k - 1$ but some function of n, such as $\log n$. Even with naive scheduling (i.e., whenever a processor becomes free supply it with an unmerged sublist pair) a speedup of k can be achieved asymptotically in this way. Although this is less efficient than our algorithm, the idea can be used to show that even in the general case of $n \leqq m$, optimal speedup is achievable in various asymptotic senses, such as the following.

COROLLARY 7. *If* $m = \alpha n$ *where* $\alpha \geqq 1$, *then*

$$\mathrm{Merge}_1(n, m)/\mathrm{Merge}_k(n, m) \to k \quad \text{as } n \to \infty.$$

Proof. Execute the first paragraph of the proof of Corollary 6 but with $\log n$ elements marked in each list. This requires $o(n)$ comparisons and time. Clearly the total number of comparisons required to merge the sublist pairs produced is no more than $\mathrm{Merge}_1(n, m)$. Even with the naive scheduling indicated above, if optimal sequential merging algorithms are used for the sublist pairs, the total time taken is no more than $\mathrm{Merge}_1(n, m)/k + o(n)$. Since $\mathrm{Merge}_1(n, m) > n$ the result follows. □

Note that the asymptotic behavior of $\mathrm{Merge}_1(n, m)$ itself is at present unknown [6].

3. Sorting. The well-known information theoretic argument gives that the sorting of n elements requires, in the worst case, $n \log n - O(n)$ comparisons. This immediately gives the following lower bound for sorting on n processors:

$$\mathrm{Sort}_n(n) \geqq \log n - \text{const}.$$

We now derive a corresponding upper bound.

THEOREM 4.

$$\mathrm{Sort}_{n/2}(n) \leqq 2 \log n \log \log n + O(\log n).$$

Proof. We show that the binary-merge sorting algorithm requires only this time if merging is done fast, as in Theorem 3.

We first consider the case $n = 2^j$ for some j. We assume inductively that after the ith stage, we have 2^{j-1} disjoint sorted lists each of length 2^i. By assigning 2^i processors to each such pair and using the fast merging algorithm, we clearly arrive at the inductive assumption of the following stage after time $2 \log i + \text{const}$. But sorting of the whole list will be complete when $i = j$. The total time needed is therefore no more than

$$\sum_{i=1}^{\log n} (2 \log i + \text{const.}) \leqq 2 \log n \log \log n + O(\log n).$$

In the general case, when n is not a power of two, there may be a fragmentary sorted list left over at each stage. However, the above argument applies in that case as well. □

COROLLARY 8. *For* $4 \leqq 2n < k \leqq n(n-1)/2$,

$$\mathrm{Sort}_k(n) \leqq 2(\log n - \log(k/n))(\log\log n - \log\log(k/n) + \text{const.}).$$

Proof. With k processors we can split the input into sets of size $\lceil k/n \rceil$ and sort each such set completely in one step. We then need $\log n - \log(k/n)$ stages of merging in the manner of Corollary 5. □

COROLLARY 9. *For* $1 < k < n$,

$$\mathrm{Sort}_k(n) \leqq (n \log n)/k + 2 \log k \cdot \log(n \log k/k).$$

Proof. As in [5], we split the input into k equal sets and sort each of these sequentially in time $(n/k)\log(n/k)$. We then successively merge pairs of these, in $\log k$ stages, using the algorithm of Corollary 6. At each stage, there will clearly be twice as many processors available per merge as at the previous one, and if we always use these, then the time taken for each stage will be about n/k. □

4. Conclusion.

We have shown that for the most basic model of parallelism for comparison problems, algorithms for merging, sorting and finding the maximum exist that are much more efficient than any previously known. We suggest our model and analysis as part of the theoretical background against which parallelism for these problems can be studied and in appropriate instances exploited. In practice, to derive good algorithms suitable for a specific multiprocessor machine, additional considerations have also to be taken into account. In particular, the tradeoffs between optimizing the sequencing of the comparisons (which is what our analysis attempts), and minimizing the overheads (e.g., inter-processor communication), have to be weighed.

Of the many further questions implied, theoretically the most tantalizing is perhaps that of parallelism in the problem of finding the median. Since this can be done in linear time sequentially [4], but cannot be solved in less than time $\sim \log\log n$ on n processors (by implication, Theorem 1), it follows that for the case $k = n$, $O(k/\log\log k)$ is an upper bound on the attainable speedup. Since we have shown that for merging, sorting, and finding the maximum, a speedup of that order is attainable, any substantial lowering of this upper bound for the median, which we conjecture to be possible, would put this problem in a class of its own. It would confirm that near optimal sequential algorithms for the median problem need to be "more carefully sequenced" than those for any of the others, and would go some way to explaining why they have proved more difficult to find. By examining parallelism, we may in this way gain deeper insights into specific computational problems than is offered by sequential analyses alone.

REFERENCES

[1] A. V. AHO AND N. J. A. SLOANE, *Some doubly exponential sequences*, Fibonacci Quart., 2 (1973), no. 4, pp. 429–437.

[2] K. E. BATCHER, *Sorting networks and their applications*, Proc. AFIPS Spring Joint Computer Conf., 32 (1968), pp. 307–314.

[3] C. BERGE, *Graphs and Hypergraphs*, North-Holland, London, 1973.

[4] M. BLUM, R. W. FLOYD, V. PRATT, R. L. RIVEST, AND R. E. TARJAN, *Time bounds for selection*, J. Comput. System Sci., 7 (1973), pp. 448–461.

[5] A. B. BORODIN AND I. MUNRO, Notes on "Efficient and Optimal Algorithms", 1972.

[6] D. E. KNUTH, *The Art of Computer Programming*, vol. 3, Addison-Wesley, Reading, Mass., 1973.

[7] P. TURAN, *On the theory of graphs*, Colloq. Math., 3 (1954), pp. 19–34.

SIAM J. COMPUT.
Vol. 4, No. 3, September 1975

THE RENEWAL MODEL FOR PROGRAM BEHAVIOR*

H. OPDERBECK AND W. W. CHU†

Abstract. A model for program behavior, the renewal model, is introduced; its properties are discussed, and its ability to model the behavior of real programs is investigated. Using this renewal model, several theorems are derived which describe the performance of the working set replacement algorithm.[1] Then the renewal model is used to evaluate the performance of a replacement algorithm for two-level directly addressable memory hierarchies.

Key words. models of program behavior, renewal model, working set algorithm, two-level directly addressable memory hierarchies

1. Introduction. The development of virtual memory systems is one of the most important advances in computer architecture over the last decade. Virtual memory systems have successfully shifted the allocation of main memory from the programmer to the system. While the programmer is relieved from this burden, the system now has to decide what parts of a program must be present in main memory during a given interval of execution. These decisions must be made with respect to the behavior of the executing programs.

Virtual memory is usually divided into blocks of contiguous locations to allow an efficient mapping of the logical addresses into the physical address space. If these blocks are of equal size, the system is called a *paging system* and the blocks are called *pages*. Main memory is equipartitioned into page frames of the same size, and any page can be put into any available page frame. The occurrence of a reference to a page that is currently not in main memory is called a *page fault*. A page fault results in the interruption of the program and the transfer of the referenced page from secondary storage.

Since the main memory has only a limited capacity, pages already in main memory must continually be removed to make room for pages entering from secondary memory. The decisions as to when and what pages are to be removed from main memory are critical for the efficient operation of the system. The replacement algorithm is that part of the system which makes these decisions. The objective of a replacement algorithm is twofold. Firstly, it is to keep those pages in main memory that are currently being used. This is necessary to keep the page fault frequency as low as possible. Secondly, the replacement algorithm is to free page frames as soon as there is a low probability that they will be referenced in the near future. This is a requirement for the efficient utilization of main memory by all processes. If the second level memory is directly addressable by the CPU, the objective of the replacement algorithm is to guarantee an efficient use of the high-speed (first level) memory.

If a program's page references were randomly distributed over all pages according to a uniform distribution, it would not matter what page is chosen

* Received by the editors February 20, 1974, and in revised form July 24, 1974.

† Computer Science Department, University of California, Los Angeles, Los Angeles, California 90024. This research was supported by the U.S. Office of Naval Research, Mathematical and Information Sciences Division, under Contract N00014-69-A-0200-4027, NR 048-129.

[1] The results of theorems 1 and 2 and the intervening lemmas are roughly equivalent to results obtained earlier in [4], [6], [8].

for replacement. The page fault frequency would only depend on the number of allocated page frames. However, programs tend to reference pages unequally; they tend to cluster references to certain pages in short time intervals. This property is exhibited to varying degrees by many practical programs and commonly known as locality. The properties of locality are [10]:

1. A program distributes its references nonuniformly over its pages, some pages being favored over others.
2. The density of references to a given page tends to change slowly in time.
3. Two reference string segments are highly correlated when the interval between them is small, and tend to become uncorrelated as the interval between them becomes large.

The cause for the occurrence of locality in practical programs can be found in the behavior and style of programmers. Programmers tend to organize their code into modules; they frequently use loops in their control structures. It is also common practice to group data into content-related blocks. The modularity of code and data layout is the primary reason for the clustering of references in space and time.

For the purpose of studying memory management, we will use the following abstraction of the notion of a process. A *process* is a sequence of references (either fetches or stores) to a set of information, called a *program*. When talking about processes in execution, we must distinguish between real time and virtual time. *Virtual time* is the time seen by an active process, as if there were no page wait interruptions. By definition, a process generates one information reference per unit virtual time. *Real time* is a succession of virtual time intervals and page wait intervals. A virtual time unit is the time between two successive page references in a process, and usually the memory cycle time of the computer in which the process operates. As a matter of convenience, we let the time for 1,000 references to the first level memory be equal to 1 msec.

Let $N = \{1, 2, \cdots, n\}$ be the set of pages in the logical address space of an n-page program. The dynamic behavior of the program for given input data can be modeled in machine-independent terms by its reference string ω, which is a sequence

$$\omega = r_1 r_2 \cdots r_t,$$

each r_t being in N. $r_t = i$ implies that page i was referenced at the tth reference; thus t measures the virtual processing time, which is discrete.

The performance of a replacement algorithm depends largely on the behavior of the running programs which, for our purposes, will be described by their reference strings. These reference strings can be obtained in two ways. Firstly, a program is interpretively executed and its reference string is recorded. Secondly, the reference string is generated by a model of program behavior. In the first case, simulation techniques are usually used to evaluate the performance of various replacement algorithms. This method has been used successfully in the past [1], [2], [3], [12]. However, only short reference strings (compared with the length of real programs) are usually processed, since these simulations are rather expensive. This is one of the reasons why the modeling of program behavior has recently found increasing interest [9], [10], [16]. In this case, the reference string is only described in terms of its statistical properties. These properties are then used to evaluate the performance of replacement algorithms, thereby considerably decreasing the overall effort in terms of cost and time. The analytical study is, of course, only as good as the underlying model. Therefore, it is necessary to develop models of program behavior which allow us to model the behavior of real programs with reasonable accuracy (depending on the kind of application).

In this paper, we shall first review several available program models. Then we will describe the renewal model for program behavior, derive its properties, and investigate its ability to model the behavior of real programs. Using this renewal model for program behavior, several theorems are derived which describe the performance of the working set replacement algorithm [6]. The renewal model is also used to evaluate the performance of a replacement algorithm for two-level directly addressable memory hierarchies.

2. Models for program behavior. The random reference model (RRM) [7] is a program behavior model which assumes that each page is equally likely to be referenced at any time. In this case, the time between successive references to the same page, called the *interreference interval*, is geometrically distributed. The probability that any given page is referenced is $1/n$, where $n = |N|$ is the number of pages which comprise program and data. The independent reference model (IRM) is a generalization of the RRM. In this model, the page references $r_1, r_2, \cdots, r_t, \cdots$ are assumed to be independent trials under some fixed probability distribution. In other words, the probability of referencing page i at the kth reference is given by $\Pr[r_k = i] = \beta_i$. Note that consecutive page references are taken according to these probabilities without regard to the previous references. The interreference intervals are again geometrically distributed. The average interreference interval for the ith page is $m_i = 1/\beta_i$. The RRM is the special case of an IRM where $\beta_i = 1/n$ for all i.

The page fault frequency of the LRU and the working set algorithm was derived for the IRM [4], [8]. Also, page reference strings generated by the IRM were compared with reference strings of real programs [17]. The results show that the IRM is a poor approximation. The observed page fault frequency of the working set algorithm differed from the theoretical value for the IRM in many cases by more than one order of magnitude.

The locality model of program behavior [9] was defined as $(L_1, t_1), (L_2, t_2), \cdots, (L_i, t_i), \cdots$, where L_i is the ith locality and t_i the holding time in L_i. The L_i are members of a specified set of localities associated with the program. A specialization of this model is the so-called very simple locality model (VSLM). This model assumes a fixed size locality, i.e., the localities L_i are all of the same size x, where $1 \leqq x < n$. At any given time t, the probability of referencing a page in the locality L_i is $1 - \lambda$, and the probability of referencing a page not in L_i and therefore making a transition to another locality is λ.

Experimental results [17] show that the VSLM more closely approximates the behavior of programs than does the IRM. However, the VSLM does not do as well as the simple LRU stack model which will be discussed next.

The simple LRU stack model (SLRUM) is based on the memory contention stack generated by the LRU algorithm [14]. To create the SLRUM, we assign to each position of the stack a fixed probability. We will denote these probabilities $q_1, q_2, \cdots, q_k$ where k is the largest integer such that $q_k > 0$ and $q_j = 0$ for all $j > k$. All the q_j's are independent of each other. The q_j are termed stack distance probabilities, with j being the distance from the top of the stack. At any time, stack position j will be chosen with probability q_j; if it is chosen, the page in that position becomes the current reference and is brought to the top of the stack. The pages at stack positions 1 through $j - 1$ are pushed down one position. In general, there is a nonzero probability q_∞ which denotes the probability of referencing a page that has not been accessed before; if such a page is chosen, it is put on top of the stack, and all previously referenced pages are pushed down one position. Since q_∞ is generally not equal to zero, there is a finite probability of referencing a page which is not in the LRU stack even after a very long execution time. Because of this behavior, the stack is steadily growing in size. However, all pages with a stack distance larger than k will never be accessed again, because $q_j = 0$ for $j > k$. Whenever $q_\infty > 0$, there are some pages which are "passing through the set of all those pages to which references may be directed, i.e., the top k pages of the LRU stack. The rate at which these pages enter the stack at the top is equal to the rate at which they drop below stack distance k, thereby becoming inaccessible. This kind of behavior is frequently exhibited by pages used in input/output operations. In this case, the probability of referencing a new page which contains input data is larger than zero even after a long virtual execution time. But once all the data items in the input page have been read, this page will never be accessed again. A similar observation can be made for pages used in output operations.

3. Renewal model. As mentioned before, the interreference intervals for the IRM are geometrically distributed. This corresponds to an exponential distribution on a continuous time scale. Using a continuous time scale, we can describe the referencing of pages as a Poisson arrival process. The arrival of a customer corresponds to the referencing of a page, interarrival times correspond to interreference intervals. The Poisson arrival process consists of the superposition of n independent Poisson processes, one for each page. The arrival rate for page i corresponds to the reference probability β_i of the IRM. We therefore have again

$$\sum_{i=1}^{n} \beta_i = 1.$$

This represents a complete alternative description of the IRM on a continuous time scale. It is the special case of a renewal model where the interreference intervals are exponentially distributed. If we generalize this latter assumption and replace the exponential distribution by a general cumulative distribution $F(t)$, we obtain the renewal model for program behavior. In this model, we assume that the page interreference intervals are statistically independent. Note that under this general assumption, only the referencing of individual pages constitutes a renewal process. The process, which is formed by superposing (or pooling) the n individual processes, is generally not a renewal process. The n individual processes are assumed to be independent.

Figure 1 shows an example of a renewal process for a program which consists of five pages. The horizontal axis represents virtual processing time. The page reference string is formed by projecting the page references of each page on a common time axis.

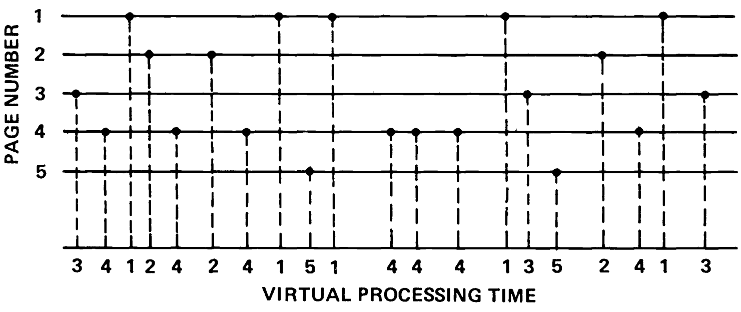


FIG. 1. *Example of a reference string generated by the renewal model*

DEFINITION OF THE RENEWAL MODEL. $R = (n, V)$ is a renewal model for program behavior iff

1. n is a positive integer;
2. V is a list of n cumulative distribution functions $F_i(x)$, $i = 1, 2, \cdots, n$, where $m_i = \int_0^\infty [1 - F_i(x)]\, dx$ exists and $\sum_{i=1}^n 1/m_i = 1$.

Here m_i is the average interreference interval of page i. $\sum_{i=1}^n 1/m_i = 1$ is a normalizing condition. If this condition holds, we have, on the average, one page reference per unit time.

Let us now consider what type of cumulative distribution functions $F_i(t)$ may be used for the representation of interreference intervals. For the continuous time IRM, the probability of referencing some page i during the small time interval Δt is constant and equal to $\beta_i \cdot \Delta t$. Let us call β_i the immediate reference density (ird). For exponentially distributed interreference intervals, the ird β_i is independent of the current backward distance of page i, that is, the time interval between the last reference to page i and the current time. Intuitively, this appears to be rather a questionable assumption since it contradicts the principle of locality. The principle of locality implies that the larger the current backward distance of page i, the smaller its reference probability. Thus the ird β_i should not be a constant but a decreasing function of the backward distance. This explains why the IRM is not a good model of program behavior.

In renewal theory, the ird is known as the *age-specific failure rate*. Renewal processes for which $\beta_i(t)$ is a decreasing function of the backward distance t are called processes with *negative aging* [5]. The ird $\beta_i(t)$ uniquely determines the probability density function (pdf) $f_i(t)$ and the cumulative distribution $F_i(t)$:

$$f_i(t) = \beta_i(t) \cdot \exp\left\{ -\int_0^t \beta_i(x)\, dx \right\},$$

$$F_i(t) = 1 - \exp\left\{ -\int_0^t \beta_i(x)\, dx \right\}.$$

4. Application of the renewal model to the working set algorithm. For renewal models, the page interreference intervals are assumed to be statistically independent. Further, the reference probability of each page is independent of the reference probability of any other page. The sequence of references for each page is modeled separately. Therefore renewal models can be applied most successfully to replacement algorithms for which the replacement decision depends solely on the behavior of individual pages. The working set replacement algorithm is an example of such a replacement algorithm. In this case, pages are always replaced when they have not been referenced during the last τ msec. This decision is clearly independent of the reference distribution of all the other pages.

We are now going to derive the average page fault frequency and the average working set size for the working set algorithm. The average page fault frequency can be calculated very easily because it is directly related to the distribution function $F_i(x)$ of each page. The following three lemmas are used for the calculation of the average working set size. These lemmas will also be used later for the evaluation of two-level directly addressable memories.

THEOREM 1. *The average page fault frequency, $Q(\tau)$, of the working set algorithm with window size τ for the renewal model is*

$$Q(\tau) = \sum_{i=1}^{n} \frac{1 - F_i(\tau)}{m_i}.$$

Proof. Since page i does not belong to the current working set if its interreference interval is larger than τ, $1 - F_i(\tau)$ is the probability that the next reference results in a page fault caused by page i, given that page i is referenced next. Recall that m_i denotes the average interreference interval of page i. Therefore, $(1 - F_i(\tau))/m_i$ is the average page fault frequency due to page i. By summing up these average page fault frequencies for all the distinct pages in the program, we obtain Theorem 1.

LEMMA 1.

$$\int_0^\tau x \cdot f(x)\,dx + \int_\tau^\infty [1 - F(x)]\,dx = m - \tau[1 - F(\tau)].$$

Proof.

$$\begin{aligned}\int_0^\tau x \cdot f(x)\,dx &+ \int_\tau^\infty [1 - F(x)]\,dx \\ &= \tau \cdot F(\tau) - \int_0^\tau F(x) \cdot dx + \int_\tau^\infty [1 - F(x)]\,dx \\ &= \tau \cdot F(\tau) - \tau + \int_0^\infty [1 - F(x)]\,dx = m - \tau[1 - F(\tau)].\end{aligned}$$

LEMMA 2. *The expected time interval a page resides in main memory is*

$$\tau + \int_0^\tau x \cdot f(x)\,dx/(1 - F(\tau)).$$

Proof. The page is replaced at the first instance of time when an interreference interval larger than τ occurs. The probability of this event is $1 - F(\tau)$. Thus the probability that the kth interreference interval is the first one that is larger than τ is $[F(\tau)]^{k-1} \cdot (1 - F(\tau))$.

Let $E\{x|\tau\}$ be the expected length of the interreference interval, given that it is shorter than τ. The expected time interval a page resides in main memory is

$$(1) \qquad [1 - F(\tau)] \cdot \sum_{k=1}^{\infty} (k - 1) \cdot E\{x|\tau\} \cdot [F(\tau)]^{k-1} + \tau = \frac{E\{x|\tau\} \cdot F(\tau)}{1 - F(\tau)} + \tau.$$

The probability $\Pr\{y \leqq x|\tau\}$ that the interreference interval y is less than or equal to x, given that it is less than or equal to τ, is

$$\Pr\{y \leqq x|\tau\} = \begin{cases} \dfrac{F(x)}{F(\tau)} & \text{for } x \leqq \tau, \\ 0 & \text{for } x > \tau. \end{cases}$$

Thus

$$(2) \qquad E\{x|\tau\} = \frac{1}{F(\tau)} \cdot \int_0^{\tau} x \cdot f(x)\, dx,$$

where $f(x) = dF(x)/dx$. Substituting (2) into (1) yields Lemma 2.

LEMMA 3. *The expected number of references to a page between page faults (excluding the reference which caused the page transfer) for the working set algorithm is* $F(\tau)/(1 - F(\tau))$.

Proof. The lemma follows directly from the proof of Lemma 2.

Since $[1 - F_i(\tau)]/m_i$ is the average page fault frequency due to page i, $m_i/[1 - F_i(\tau)]$ is the average inter-page-fault interval for page faults caused by page i. At the beginning of such an inter-page-fault interval, page i resides in main memory. Lemma 2 gives the expected length of this residency in main memory. The expected fraction of time page i resides in main memory is the ratio of the expected time page i resides in main memory to the average inter-page-fault interval of page i and is equal to

$$(3) \qquad \frac{1}{m_i}\left\{\int_0^{\tau} x \cdot f_i(x)\, dx + \tau[1 - F_i(\tau)]\right\}.$$

Using Lemma 1, (3) reduces to

$$1 - \frac{1}{m_i}\int_{\tau}^{\infty} [1 - F_i(x)]\, dx = \frac{1}{m_i}\int_0^{\tau} [1 - F_i(x)]\, dx.$$

By summing up these time fractions, we obtain the following theorem.

THEOREM 2. *The average working set size,* $s(\tau)$, *of the working set algorithm with window size* τ *for the renewal model is*

$$s(\tau) = \sum_{i=1}^{n} \frac{1}{m_i}\int_0^{\tau} [1 - F_i(x)]\, dx.$$

Note that $\lim_{\tau \to 0} s(\tau) = 0$ and $\lim_{\tau \to \infty} s(\tau) = n$.

COROLLARY 1.

$$\frac{d}{d\tau}s(\tau) = Q(\tau).$$

Proof. Differentiating $s(\tau)$ of Theorem 2 yields directly $Q(\tau)$ of Theorem 1. Corollary 1 is analogous to Denning and Schwartz' result [8] for the case of a discrete time scale.

As mentioned before, we should use a renewal process with negative aging for the modeling of program behavior if we want to derive a better program model than the IRM. This implies that the immediate reference density $\beta_i(t)$ must be a decreasing function of the backward distance t. If we choose

$$\beta_i(t) = \alpha_i \rho_i (\rho_i \cdot t)^{\alpha_i - 1},$$

then $\beta_i(t)$ is a decreasing function of t for $\alpha < 1$, $t > 0$. Since the ird $\beta_i(t)$ uniquely determines the pdf $f_i(t)$ and the cumulative distribution $F_i(t)$, we obtain

$$f_i(t) = \alpha_i \rho_i (\rho_i \cdot t)^{\alpha_i - 1} \cdot \exp\{-(\rho_i t)^{\alpha_i}\}$$

and

$$F_i(t) = 1 - \exp\{-(\rho_i t)^{\alpha_i}\}.$$

$F_i(t)$ is the so-called Weibull distribution [18]. It is easy to verify that the mean of the Weibull distribution is

$$m_i = \frac{\Gamma(1 + 1/\alpha_i)}{\rho_i}, \tag{4}$$

and the coefficient of variation (the ratio of standard deviation to the mean) is

$$c_i = \left[\frac{\Gamma(1 + 2/\alpha_i)}{\{\Gamma(1 + 1/\alpha_i)\}^2} - 1\right]^{1/2}. \tag{5}$$

In order to obtain estimates for the parameters α_i and ρ_i, we measured the mean and the coefficient of variation of the interreference interval distribution for two sample programs (FORTRAN and FORTCOMP). An estimate for α_i was then derived from the coefficient of variation (equation (5)). This estimate of α_i and the measured mean is then used to determine ρ_i (equation (4)). Tables 1 and 2 give the mean m_i, the coefficient of variation c_i, and the estimates for α_i and ρ_i for FORTRAN and FORTCOMP, respectively. The results in these tables reveal that different pages have great differences in the average interreference interval m_i. For the FORTCOMP program, for example, the smallest interreference interval is 3.6379 references and the largest interreference interval is 166,630 references. A similar statement is true for its coefficient of variation. This shows that different pages may exhibit a completely different behavior. The intuitive explanation for this observation is, among other things, that pages are used for the storing and fetching of instructions as well as data.

The page fault frequency for the case in which the interreference interval is distributed according to a Weibull distribution can be derived from Theorem 1 and is equal to

$$\sum_{i=1}^{n} \frac{\exp\{-(\rho_i \tau)^{\alpha_i}\}}{m_i}.$$

Likewise, the average working set size for the Weibull distribution can easily be derived from Theorem 2 and is equal to

$$\sum_{i=1}^{n} \frac{1}{m_i} \int_0^{\tau} \exp\{-(\rho_i t)^{\alpha_i}\}\, dt.$$

This integral can be evaluated by numerical integration.

Figures 2 and 3 show the average page fault frequency for the FORTRAN and the FORTCOMP program, respectively. The dashed curves represent the measurement results. The two solid curves represent the average page fault frequency for the renewal model with the Weibull distribution and IRM, respectively. The exponential distribution can be viewed as a special case of the Weibull distribution for which $\alpha_i = 1$ and $\rho_i = 1/m_i$.

Figures 4 and 5 show the results for the average working set size for the same two sample programs. As can be seen from these figures, the renewal model represents a better approximation of program behavior than the IRM. The results for the IRM are similar to those obtained by Spirn and Denning [17].

In the renewal model, we have assumed that page interreference intervals are statistically independent. We know, however, that in real programs the interreference intervals are somewhat dependent. Comparing the program behavior derived from the renewal model with the program behavior derived from measurements, the results show that the assumption of independent interreference intervals in the renewal model gives a fairly good approximation.

TABLE 1
Input parameters for the renewal model (FORTRAN program)

Page Number	m_i	c_i	α_i	ρ_i
1	7.2601	87.598	0.2077	12.085
2	11.601	60.762	0.2177	5.2498
3	68.537	43.704	0.2279	0.63832
4	158.78	31.026	0.2398	0.19592
5	46.714	20.631	0.2565	0.44153
6	148.94	28.481	0.2431	0.19146
7	322.69	21.901	0.2538	0.06799
8	1336.5	11.539	0.2861	0.0086463
9	1052.3	20.082	0.2576	0.019123
11	3.3813	239.34	0.1854	70.438
12	19.189	99.796	0.2045	5.1890
13	146.11	61.684	0.2173	0.42261
14	42.199	315.33	0.1802	7.4553
15	45.403	303.92	0.1809	6.6665
18	60.942	250.63	0.1844	4.1168
19	20.669	45.218	0.2268	2.1895
20	100.49	21.873	0.2540	0.21731
21	4.5785	45.907	0.2263	10.039
22	35.642	24.942	0.2483	0.70178
23	4544.0	28.834	0.2426	0.006358
25	26308.0	12.371	0.2822	0.0004698
26	41654.0	10.086	0.2943	0.0002426
32	2202.0	47.107	0.2254	0.02147

TABLE 2
Input parameters for the renewal model (*FORTCOMP program*)

Page Number	m_i	c_i	α_i	ρ_i
2	1741.8	12.878	0.2137	0.04025
3	166630.0	1.4573	0.7020	7.576E-6
4	31243.0	2.0555	0.5320	5.745E-5
5	14.113	124.34	0.1228	3887.45
6	74.6	53.605	0.1457	51.366
8	8.1907	12.625	0.2151	8.1422
9	8.8770	10.449	0.2296	4.6809
10	3.6379	41.857	0.1541	503.77
11	5.6674	34.610	0.1612	186.56
12	52.571	36.731	0.1590	23.694
13	90.315	21.083	0.1838	2.8669
14	298.27	15.949	0.1996	0.40926
15	54.591	36.331	0.1593	22.306
16	321.27	26.127	0.1734	1.4548
17	9.7798	43.766	0.1526	212.21
18	27.492	40.187	0.1555	59.515
20	278.49	16.401	0.1979	0.47184
21	31.394	6.9421	0.2687	0.50709

5. Application to a two-level directly addressable memory hierarchy. In the case where only the first level of the memory hierarchy is directly addressable by the CPU, an entire page must be transferred whenever an information item is referenced which is not in the first level. Because of the locality of page references this is in many cases an efficient policy. However, as Tables 1 and 2 show, there are usually some pages which are referenced rather infrequently. For these infrequently used pages, it would be more efficient to transfer the referenced information item directly to the CPU, leaving the corresponding page in the second level memory. Memory hierarchies that use this strategy are called *two-level directly addressable paged memories.* The IBM 360/67 installation at Carnegie-Mellon University is an example of a computer system with this type of memory hierarchy [11], [13], [20]. As the costs of high-speed large memories such as bulk core storage and semiconductor memory decrease, systems with two-level directly addressable memories become increasingly attractive. Experimental performance [19] and some theoretical properties [15] of these memory systems have been reported recently.

To demonstrate the utility of the renewal model, we are now going to apply it to the evaluation of a two-level directly addressable memory hierarchy. In the case where only the first level of the memory hierarchy is directly addressable, a demand paging algorithm has to decide when to remove a page from the first level memory and what page or pages are to be removed. The same decisions must also be made in a two-level directly addressable memory hierarchy. In this later case,

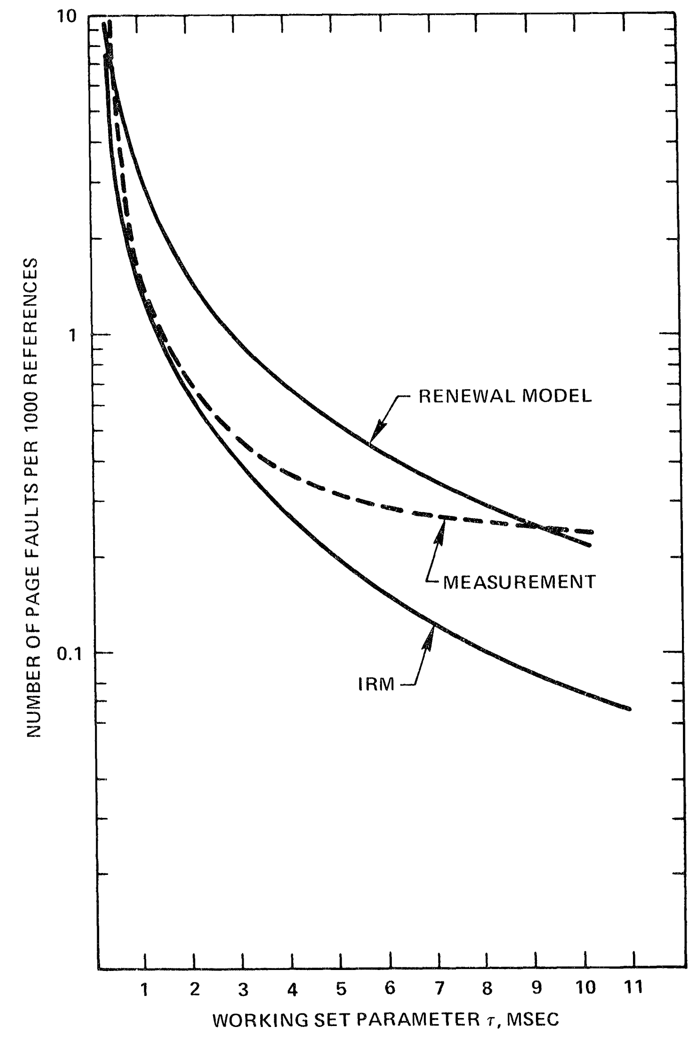


FIG. 2. *Average page fault frequency* (*FORTCOMP program*)

however, we now have a choice as to whether or not to promote a page to the first level. We therefore must establish a second decision rule which tells when to promote a page.

There is a variety of promotion and replacement decision rules. These decision rules may or may not depend on the reference string. Any combination of a promotion rule with a replacement rule represents a memory management algorithm for a two-level directly addressable memory hierarchy. If the promotion and replacement rules are not applied at the same time, the amount of allocated first level memory varies in time, and we have a dynamic storage partitioning scheme. In what follows, we will restrict ourselves to a particular combination of a promotion and replacement rule which uses the dynamic storage partitioning scheme.

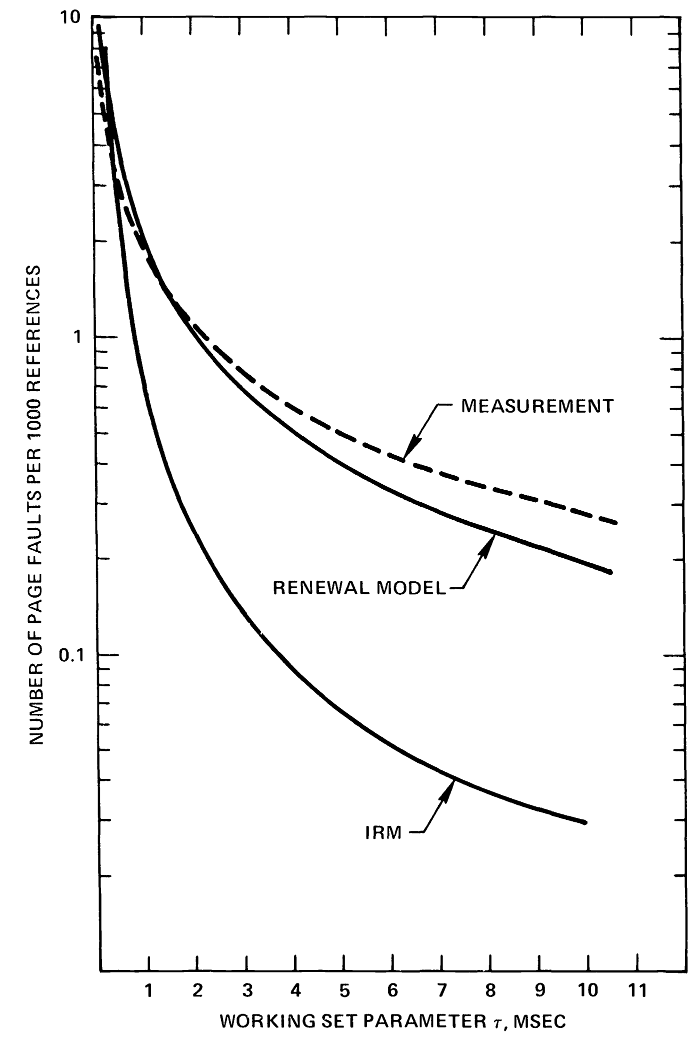


FIG. 3. *Average page fault frequency* (*FORTCOMP program*)

If only the first level memory is directly addressable, the choice of a specific replacement rule with its parameter determines uniquely the average page fault frequency and the average number of allocated first level memory page frames. In the case of two-level directly addressable memories, there is not only one combination of parameters for the promotion and replacement rule, but there is, in general, a large number of such combinations which all achieve the same average page fault frequency. We are therefore interested in an answer to the following two questions:

1. Given that the average number of allocated first level memory page frames should not be greater than *n*, how should we select parameters for the promotion and replacement rule that minimize the total processing time?

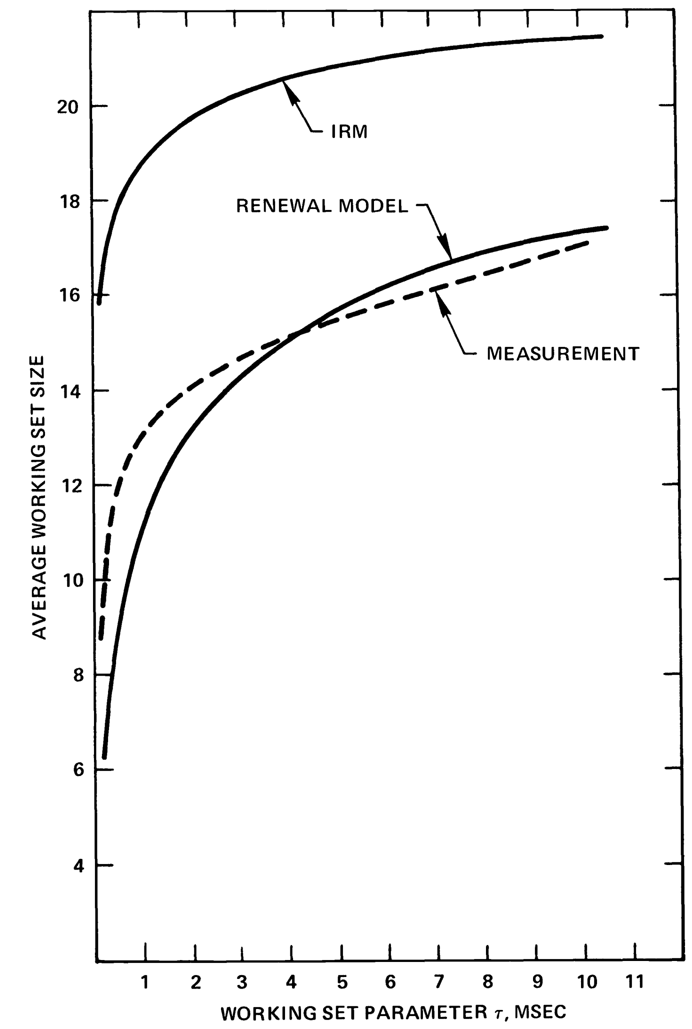


FIG. 4. *Average working set size* (*FORTRAN program*)

2. Given that the total processing time should not be greater than t, how should we select parameters for the promotion and replacement rule that minimize the average number of allocated first level memory page frames?

We are going to use the "working set rule" as a replacement rule; i.e., a page is removed from the first level memory whenever it has not been referenced during the last τ msec. As a promotion rule, we will use a rule that was first suggested by Williams [19]: whenever a page that resides in the second level memory is referenced, it will be promoted according to some fixed probability p. Although this promotion rule appears to be very simple, it can be quite effective since it causes those pages that are referenced more frequently to be promoted to the

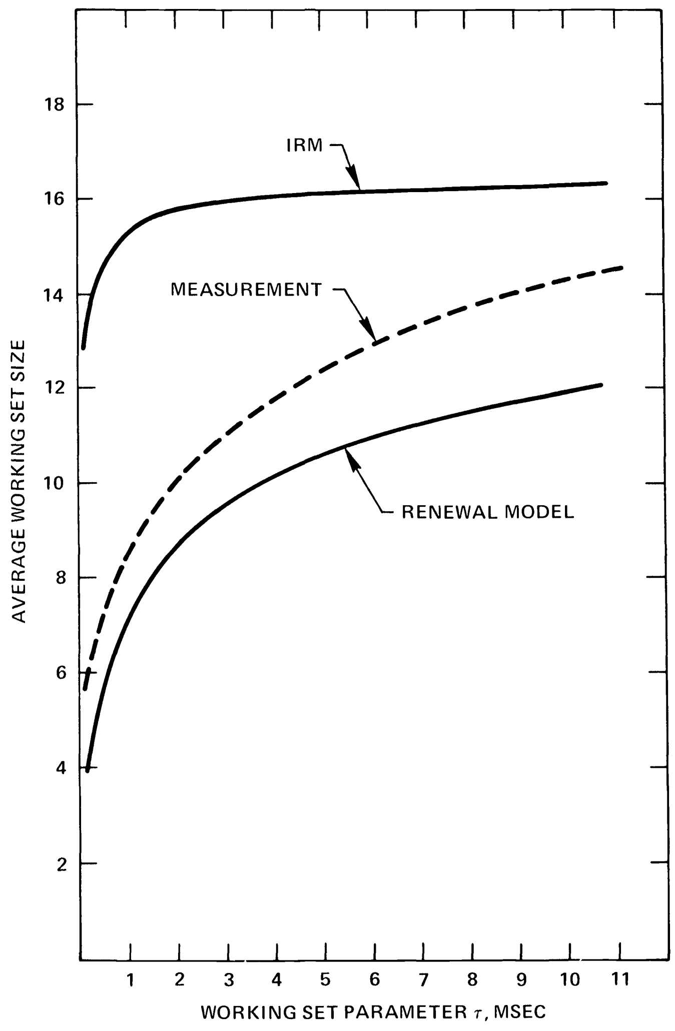


FIG. 5. *Average working set size* (*FORTCOMP program*)

first level memory. With this particular combination of promotion and replacement rule, the parameters to be optimized are the working set parameter τ and the promotion probability p. Let us call this particular combination of promotion and replacement rule a (p, τ)-*algorithm.*

We define v as the number of references to a page while it resides in the second level memory; i.e., the number of references between removal and promotion to the first level memory. v is a random variable which is geometrically distributed. The mean value of v is

$$E\{v\} = p \cdot \sum_{v=1}^{\infty} v \cdot (1 - p)^{v-1} = \frac{1}{p}.$$

Note that we have the same value $E\{v\}$ for each page because the promotion probability p is independent of the number of the referenced page. However, the expected period of time a page stays in the second level memory is different for different pages. This time interval can be computed as a sum which consists of two parts. The first part is the expected period of time between the removal from the first level memory and the first reference to the same page in the second level memory. The second part is equal to $(1/p - 1)\cdot m_i$, that is, the expected number of interreference intervals times the average length of an interreference interval for page i.

Let s denote the expected time interval between the removal from the first level memory and the first reference to the removed page in the second level memory. We know that a page which is transferred to the second level memory has a backward distance equal to τ. The random variable s has therefore the following cumulative distribution:

$$F_i(s|\tau) = \frac{F_i(s+\tau) - F_i(\tau)}{1 - F_i(\tau)}.$$

The mean of the random variable s, $E_i\{s|\tau\}$, can be derived as follows:

$$\begin{aligned} E_i\{s|\tau\} &= \frac{1}{1 - F_i(\tau)} \int_0^\infty s \cdot f_i(s+\tau)\, ds \\ &= \frac{1}{1 - F_i(\tau)} \int_\tau^\infty (s-\tau) f_i(s)\, ds \\ &= \frac{1}{1 - F_i(\tau)}(-1) \int_\tau^\infty (s-\tau)\frac{d[1 - F_i(s)]}{ds}\cdot ds \\ &= \frac{1}{1 - F_i(\tau)}(-1)\cdot\left\{ [(s-\tau)[1 - F_i(s)]]_{s=\tau}^{\infty} - \int_\tau^\infty [1 - F_i(s)]\, ds \right\}. \end{aligned}$$

From Lemma 1, we know that $\lim_{s\to\infty} s\cdot(1 - F_i(s)) = 0$ for $m_i < \infty$. Therefore

$$E_i(s|\tau) = \left(\int_\tau^\infty [1 - F_i(s)]\, ds\right) \bigg/ (1 - F_i(\tau)).$$

Thus the expected period of time a page stays in the second level memory is

$$\text{(6)} \qquad \left(\int_\tau^\infty [1 - F_i(s)]\, ds\right) \bigg/ (1 - F_i(\tau)) + \left(\frac{1}{p} - 1\right)\cdot m_i.$$

We now make use of Lemma 2 to calculate the expected fraction of time a page stays in the first level memory. Summing these fractions over all pages, we obtain the following theorem.

THEOREM 3. *The average number of allocated first level memory page frames for the* (p, τ)*-algorithm is*

$$\sum_{i=1}^{n} \left\{ \left(\frac{p}{m_i}\int_0^\tau [1 - F_i(s)]\, ds\right) \bigg/ (p + [1 - F_i(\tau)][1 - p]) \right\}.$$

For $p = 1$, we obtain the same result which was previously derived for the working set algorithm (Theorem 2).

THEOREM 4. *The expected fraction σ of references to the first level memory for the (p, τ)-algorithm is*

$$\sigma = \sum_{i=1}^{n} \frac{\sigma_i}{m_i}, \quad \textit{where } \sigma_i = \frac{p \cdot F_i(\tau)}{F_i(\tau)(p-1)+1}.$$

Proof. As mentioned above, the expected number of references to a page in the second level memory is $1/p$. Lemma 3 gives the same information for the first level memory. From this we derive the expected fraction of references to each page. The sum of these fractions weighted by the average reference frequency $1/m_i$ gives Theorem 4.

Let T_1 denote the real processing time for a program operated under a (p, τ)-algorithm (the processing time including the access times to single information items in the first level memory and the second level memory and the page transfer times for transferring from the second level to the first level memory). Further, let T_0 be the fastest execution time which can be achieved if the whole program is loaded into the first level memory. T_0 is equal to the total number of page references. The ratio $\delta = T_1/T_0$ is called the *expansion factor*. The expansion factor δ is a measure for the delay which is introduced by the use of a slower secondary memory. We will use δ as a performance measure for the evaluation of the (p, τ)-algorithm.

The expansion factor δ depends on the following variables: (i) the promotion probability p, (ii) the working set parameter τ, (iii) the speed ratio R of the memory hierarchy and (iv) the page transfer time T_s. T_s includes the processing time which is spent to transfer the modified pages back from main memory to secondary storage. For convenience we shall use the access time of the first level memory as a time unit for T_s. In order to compute δ, we need to know the expected number of page transfers for page i, which is given by $p(1 - \sigma_i)T_0/m_i$. Recall that σ_i is the fraction of references to the first level memory for page i. Let us assume that a page which is promoted to the first level will be referenced by the CPU only after it has been promoted. Then the real processing time due to references to the first level memory is

$$\sum_{i=1}^{n} \frac{T_0}{m_i}[\sigma_i + p(1-\sigma_i)]. \tag{7}$$

The real processing time due to references to the second level memory is

$$\sum_{i=1}^{n} \frac{R \cdot T_0}{m_i}(1-\sigma_i)(1-p), \tag{8}$$

and the real processing time due to page transfers is

$$\sum_{i=1}^{n} \frac{T_s \cdot T_0}{m_i} p(1-\sigma_i). \tag{9}$$

Adding (7), (8) and (9) and dividing by T_0, we obtain the following theorem.

THEOREM 5. *The expansion factor δ for a two-level directly addressable memory hierarchy with speed ratio R and page transfer time T_s and operated under a (p, τ)-algorithm is*

$$\delta = \sum_{i=1}^{n} \frac{1}{m_i}[\sigma_i + (1 - \sigma_i)(pT_s + R(1 - p) + p)].$$

To demonstrate the use of the renewal model for the evaluation of two-level directly addressable memory hierarchies, the average number of first level memory pages (Fig. 6) and the expansion factor (Fig. 7) were evaluated for different values of the working set parameter τ and the promotion probability p. In these figures, the FORTCOMP program was used as a sample program (see Table 2). Further, we chose $R = 10$ and $T_s = 5{,}120$ units.

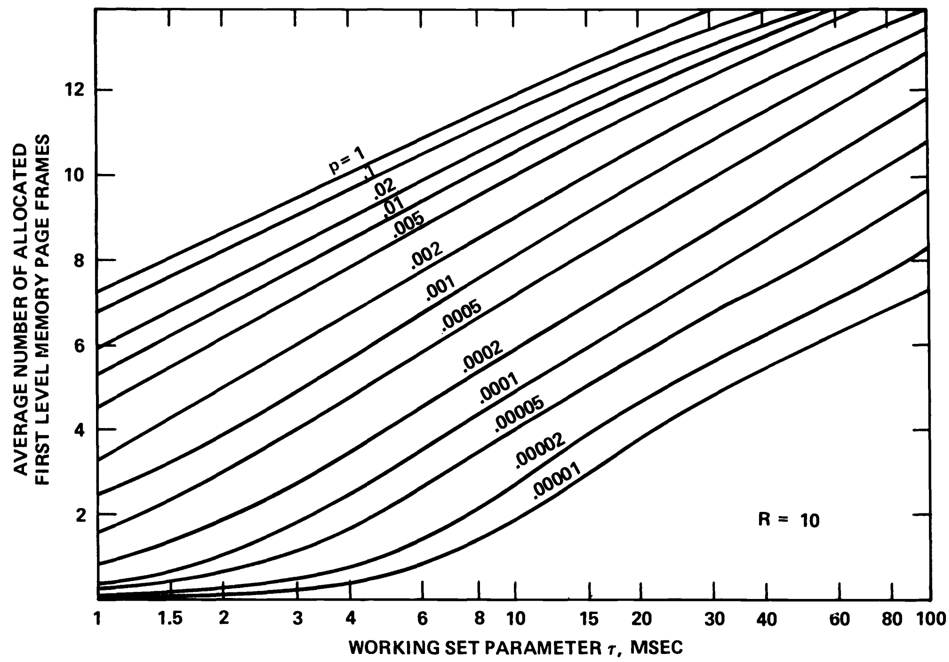


FIG. 6. *Average number of first level memory page frames for the (p, τ)-algorithm*

The average number of allocated first level memory page frames is an increasing function of both the working set parameter τ and the promotion probability p. No such general statement can be made with respect to the expansion factor δ. This is because a greater promotion probability increases the number of references to the first level memory and may therefore speed up the computation. However, it also increases the number of page promotions, thereby slowing down the computation. This explains why two of the curves of Fig. 7 cross each other.

Figures 6 and 7 taken together allow us now to answer the two questions which had been posed in the beginning of this section. For instance, the dashed line of Fig. 7 connects all those points for which the average size of the allocated first level memory space is equal to five page frames. The minimum of this curve represents the optimal combination of parameters for the promotion and replacement rule that minimizes the total processing time, given that the average number of allocated first level memory page frames is not greater than 5. In this case, the optimal combination is approximately $\tau = 10$ msec, and $p = 10^{-4}$.

These figures represent only an example for the application of the renewal model and highlight the technique that can be used for the optimization of two-level directly addressable memories. The optimization of a two-level directly addressable memory with such parameters as promotion and replacement rule, page size, speed ratio R, etc., is beyond the scope of this paper.

Much more theoretical and experimental work has to be done to evaluate the performance of these promotion and replacement rules. In particular, the following questions should be investigated: how can the promotion probability be implemented (possibly by promoting a page after every pth page reference to the second level memory)? What other promotion and replacement rules are available? What is the effect of different speed ratios R? It appears that the renewal model is a valuable tool for the investigation of these problems. Also, it is quite possible that the use of a different distribution function for the page interreference intervals may produce closer approximations of the behavior of real programs.

6. Conclusion. One of the difficulties of modeling resource allocation in modern computer systems has been the proper representation of memory allocation. The primary reason for this difficulty can be found in the varying memory requirements of the running programs. Many efforts have been made to

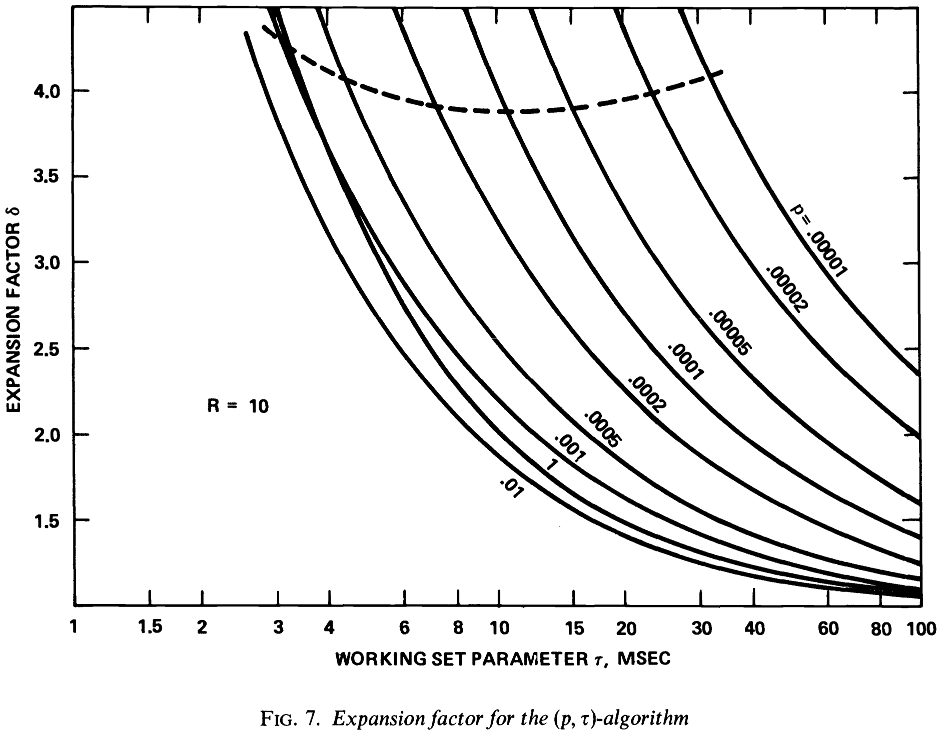


FIG. 7. *Expansion factor for the* (p, τ)*-algorithm*

describe this complex process in terms of a model of program behavior. However, no universal model that can describe all kinds of program behavior satisfactorily has yet been developed, and it is not clear whether this can be done. Like all the other models of program behavior that have been used in the past, the renewal model is limited in its applicability. However, in certain cases it can be a very useful tool to study the performance of memory allocation algorithms. This was shown for the working set algorithm and the (p, τ)-algorithm. Such studies allow us to gain insight into the performance characteristics of these algorithms. This, in turn, may significantly contribute to the improvement of overall system performance.

Acknowledgment. The authors wish to thank the referees for their constructive comments.

REFERENCES

[1] L. A. Belady, *A study of replacement algorithms for virtual-storage computers*, IBM Systems J., 5 (1966), pp. 78–101.

[2] W. W. Chu and H. Opderbeck, *The page fault frequency replacement algorithm*, Proc. AFIPS 1972 Fall Joint Computer Conf., 41, no. 1, pp. 597–609.

[3] E. G. Coffman and L. C. Varian, *Further experimental data on the behavior of programs in a paging environment*, Comm. ACM, 11 (1968), pp. 471–474.

[4] E. G. Coffman and P. J. Denning, *Operating System Theory*, Prentice-Hall, Englewood Cliffs, N. J., 1973.

[5] D. R. Cox, *Renewal Theory*, Methuen, London, 1967.

[6] P. J. Denning, *The working set model for program behavior*, Comm. ACM, 11 (1968), pp. 323–333.

[7] ———, *Thrashing: Its causes and prevention*, Proc. AFIPS 1968 Fall Joint Computer Conf., 33, pp. 915–922.

[8] P. J. Denning and S. C. Schwartz, *Properties of the working set model*, Comm. ACM, 15 (1972), pp. 191–198.

[9] P. J. Denning, J. E. Savage and J. R. Spirn, *Some thoughts about locality in program behavior*, Proc. Symp. on Computer Communications Networks and Teletrafic, Polytechnic Inst. of Brooklyn, 1972, pp. 101–112.

[10] P. J. Denning, *On modeling program behavior*, Proc. AFIPS 1972 Spring Joint Computer Conf., 40, pp. 937–944.

[11] R. E. Fikes, H. C. Lauer and A. L. Vareha, *Steps towards a general purpose time-sharing system using large-capacity core storage and RSS/*360, Proc. 23rd National ACM Conf., 1968, pp. 7–18.

[12] M. Joseph, *An analysis of paging and program behavior*, Comput. J., 13 (1970), pp. 38–54.

[13] H. C. Lauer, *Bulk-core in a* 360/67 *time sharing system*, Proc. AFIPS 1967 Fall Joint Computer Conf., pp. 601–609.

[14] R. L. Mattson, J. Gecsei, D. R. Slutz and I. L. Traiger, *Evaluation techniques for storage hierarchies*, IBM Systems J., 9 (1970), pp. 78–117.

[15] R. R. Muntz and H. Opderbeck, *Stack replacement algorithms for two-level directly addressable paged memories*, this Journal, 3 (1974), pp. 11–22.

[16] G. S. Shedler and C. Tung, *Locality in page reference strings*, this Journal, 1 (1972), pp. 218–241.

[17] J. R. Spirn and P. J. Denning, *Experiments with program locality*, Proc. AFIPS 1972 Fall Joint Computer Conf., 41, no. 1, pp. 611–621.

[18] W. Weibull, *A statistical distribution function of wide applicability*, J. Appl. Mech., 18 (1951), pp. 293–297.

[19] J. G. Williams, *Experiments in page activity determination*, Proc. AFIPS 1972 Spring Joint Computer Conf., 40, pp. 739–747.

[20] A. L. Vareha, R. M. Rutledge and M. M. Gold, *Strategies for two-level memories in a paging environment*, Proc. 2nd ACM Symp. on Operating Systems Principles, 1969, pp. 54–59.

SIAM J. COMPUT.
Vol. 4, No. 3, September 1975

ON FINDING AND UPDATING SPANNING TREES AND SHORTEST PATHS*

P. M. SPIRA† AND A. PAN‡

Abstract. We consider one origin shortest path and minimum spanning tree computations in weighted graphs. We give a lower bound on the number of analytic functions of the input computed by a tree program which solves either of these problems equal to half the number of worst-case comparisons which well-known algorithms attain. We consider the work necessary to update spanning tree and shortest path solutions when the graph is altered after the computation has terminated. Optimal or near-optimal algorithms are attained for the cases considered. The most notable result is that a spanning tree solution can be updated in $O(n)$ when a new node is added to an n-node graph whose minimum spanning tree is known.

Key words. spanning trees, shortest paths, lower bounds on computation, graph computations

1. Synopsis of results. Dijkstra [2] has given an algorithm to find all shortest paths from a single origin in a directed graph with positive arc weights and Prim [1] has given an algorithm to find a minimal spanning tree in an undirected graph. We discuss the optimality of these algorithms in the sequel and show that no program whose unit operation is the evaluation and testing for positivity of an analytic function of the weights can better these algorithms by more than a factor of two. We then consider the problem of updating previous shortest path and minimum spanning tree solutions when parameters of the graph are changed. We consider what must be done when nodes are added or deleted and when weights on arcs are increased or decreased. We obtain lower bounds and optimal or near optimal algorithms for these problems in terms of how many analytic functions of the weights must be considered.

2. Definitions and preliminaries. Let G be an n-node with d_{ij} the distance from node i to node j so that G is undirected if $d_{ij} = d_{ji}$ for all i and j.

DEFINITION 2.1. An *analytic tree program* T is one defined by a rooted tree. Each internal node and the root are labeled by analytic functions, and each leaf is labeled by an answer—the output of the program. Computation begins at the root. At each node the analytic function is evaluated and the next node visited is the left or right successor of the present node. Computation terminates when a terminal node is reached. The *depth of* T, $d(T)$, is the length of the longest branch.

DEFINITION 2.2. Let $l_1, \cdots, l_m$ be linear maps from R^d to R, where R is the real numbers, and let $C \subseteq R^d$ be a convex set. Let $L^+ = \{x \in R^d : l_i(x) \geqq 0, 1 \leqq m\}$. A *complete analytic proof of* L^+ *on* C is a matrix

$$\mathscr{P} = \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1k} \\ \vdots & \vdots & & \vdots \\ p_{r1} & p_{r2} & \cdots & p_{rk} \end{bmatrix},$$

* Received by the editors January 1, 1974, and in revised form July 26, 1974. This research was supported in part by the U.S. Army Research Office, Durham, under Contract DA-ARO-D-31-124-71-6174 and in part by the National Science Foundation under Grant GJ-356 04.

† Department of Electrical Engineering and Computer Sciences and the Electronics Research Laboratory, University of California at Berkeley, Berkeley, California 94720.

‡ Department of Electrical Engineering and Computer Sciences and the Electronics Research Laboratory, University of California at Berkeley, Berkeley, California. Now at Bell Laboratories, Naperville, Illinois 60540.

where each $p_{ij}: R^d \to R$ is analytic and such that $x \in L^+ \Leftrightarrow \exists i, 1 \leqq i \leqq k$, with $p_{ij}(x) \geqq 0, 1 \leqq j \leqq k$. We call k the *width of* $\mathscr{P}$.

The reason for defining a complete proof is that any lower bound on the width of a complete proof for L^+ is a lower bound on the depth of a tree program. In fact, we have the following lemma.

LEMMA 2.3. *Let $l_1, \cdots, l_m$ be linear maps from R to R. Let $C \subseteq R^d$. Let k be the minimum width of any analytic proof of L^+ on C. Let T be a program which, given any point $x \in C$, determines whether or not $x \in L^+$. Then $d(T) \geqq k$.*

Proof. The proof is direct from the definition of complete analytic proof. Q.E.D.

We shall use in all our lower bound proofs the following theorem.

THEOREM 2.4 (Rabin). *Let $l_1, \cdots, l_m$ be linear forms from R^d to R, with $m \leqq d$. Let $C \subseteq R^d$ contain a point for any given of the 3^m possible $+, 0, -$ sign conditions of the l_i. Then any complete analytic proof of L^+ on C has width at least m.*

This theorem says that under the given hypotheses, the easiest thing to do to verify that a point $x \in C$ is in L^+ is to compute $l_1(x), \cdots, l_m(x)$ and see if they are all nonnegative.

3. Spanning trees.

Prim's [1] well-known procedure finds the minimum spanning tree in an undirected graph. There are two types of comparisons employed. The first type finds the closest unconnected node to the set of nodes already connected. This closest node becomes a connected node. The second type compares for each unconnected node the distance to it via the last connected node and the distance to it which was minimal before the last node was connected. If the algorithm is properly programmed by introducing a tree of depth $\lceil \log_2 (n - k) \rceil$ for the arcs from the kth node brought in, then it will take between $\frac{1}{2}(n - 1) \cdot (n - 2)$ and $(n - 1) \cdot (n - 2)$ comparisons, depending upon the number of new arcs brought into consideration in the second type of comparison. We show that any analytic tree program will have depth at least $\frac{1}{2}(n - 1) \cdot (n - 2)$ for this problem. In fact, more strongly, we have the next theorem.

THEOREM 3.1. *Let T be an analytic tree program which, given a complete undirected weighted graph and $n - 1$ arcs, determines whether or not these arcs form a minimum spanning tree. Then $d(T) \geqq \frac{1}{2}(n - 1) \cdot (n - 2)$.*

Proof. Let D be the set of $n - 1$ arcs. Let $d = \max \{d_{ij}$: the arc from node i to node j is in $D\}$. Then the arcs of D form a minimum spanning tree if and only if they form a tree and $d_{ij} \geqq d$ for each i and j such that the arc from i to j is not in D. But this is a set of $\frac{1}{2}(n - 1) \cdot (n - 2)$ inequalities which satisfy Rabin's hypothesis. Q.E.D.

We now discuss updating minimum spanning tree solutions when graph parameters are changed. First we consider adding a new node to the graph.

THEOREM 3.2. *Let an n-node weighted undirected graph G be given, together with $n - 1$ arcs known to be a minimum spanning tree. Let an $(n + 1)$-st node be added to G, together with at least two arcs connecting it to the original n nodes. Then any analytic tree program to compute the minimum spanning tree of the new graph has depth at least n.*

Proof. Consider the case in which there are two arcs from the new node which, together with the given minimum spanning tree, form a cycle of length $n + 1$.

Then the new minimum spanning tree will contain each of these arcs except that arc with the maximum weight. Q.E.D.

The reader can easily construct an $O(n \log n)$ algorithm to update the minimum spanning tree if he or she notes the fact that the only eligible arcs are the $n - 1$ arcs now in the tree and the at most n new arcs connected to the new node. In fact, there is an $O(n)$ algorithm which we now present. Also, the algorithm uses storage proportional to n.

THEOREM 3.3. *There is an algorithm to update the minimum spanning tree of an n-node graph to which a new node has been added which uses $O(n)$ comparisons and $O(n)$ storage.*

Proof. We give the algorithm. The input to the algorithm is the set of arcs in the old tree and the set of arcs to the new node. All arcs appear with their weights.

ALGORITHM.

1. Find minimum weight arc incident upon each node.
2. Find the connected components of the set of arcs found in step 1.
3. Find the minimum arc between each pair of trees found in step 2 such that there is at least one such arc.
4. Collapse each tree found in step 2 to a new node, and go to step 1 if there is more than one such node.

Step 1 requires at most $4n$ comparisons. Step 2 is linear in n if we use Tarjan's [4] connected components algorithm. Step 3 can be done by processing each edge not found in step 1 once and uses linear storage. To see this, note that there can be no more than one arc between any two trees unless one of them contains the newly added node, or there would have been a cycle in the original spanning tree. So we only need to process arcs that go to the component containing the new node and hence use linear storage. In the process we will throw out all nonminimal connecting arcs, so that step 4 is trivial. When we return to step 1, we have the original problem on at most half as many nodes. Hence for a constant c, we have a recursion for the work, $F(n)$, given by

$$F(n) \leqq F(n/2) + cn,$$

so that $F(n) \leqq 2cn$. Q.E.D.

We note that Johnson and Simon [5] have independently discovered an entirely different $O(n)$ algorithm for this problem.

The rest of the results on updating spanning trees are now stated as Theorem 3.4.

THEOREM 3.4. *Let G be an n-node undirected weighted graph whose minimum spanning tree is specified. Then*:

(i) If the value of a tree arc is increased any analytic tree program to update the minimum spanning tree has depth at least $n/4$ for n even and $(n^2 - 4)/4$ for n odd. Furthermore there is an algorithm using this many comparisons in the worst case.

(ii) If the value of a nontree arc is decreased in weight, then an algorithm using $n - 1$ comparisons in the worst case will yield the new minimum spanning tree and no analytic tree program with depth less than $n - 1$ can solve this problem.

(iii) If a node is deleted from the graph together with all of its arcs, then an analytic tree program to update the solution has depth at least $\frac{1}{2}(n - 2) \cdot (n - 3)$ (although it will usually be easier than this).

Proof. (i) Consider all arcs running between the two subtrees formed by deleting the arc whose weight has increased. Then the new tree will be the union of the subtrees and the connecting arc of minimum weight. If the subtrees have i and $n - i$ nodes, there are $n(n - i)$ such arcs. Hence the result follows.

(ii) The arc of decreased weight is in the new tree if and only if it is no longer the maximum weight arc in the cycle it forms when added to the old minimum spanning tree.

(iii) The worst case occurs when the deleted node was a root of degree $n - 1$ of the old tree. Then no old information is useful. Q.E.D.

4. Shortest paths. In this section we discuss finding and updating shortest paths from a single origin in positively weighted directed graphs (digraphs). Dijkstra's [2] procedure for finding a shortest path from a root to every other node in an n-node graph requires between $\frac{1}{2}(n - 1) \cdot (n - 2)$ and $(n - 1) \cdot (n - 2)$ comparisons. Similar considerations apply as in the spanning tree problem. Also, similarly to Theorem 3.1, we have Theorem 4.1.

THEOREM 4.1. *Let T be an analytic tree program which verifies that a tree rooted at node 1 specifies a shortest path from node 1 to each other node in a positively weighted digraph. Then $d(T) \geqq \frac{1}{2}(n - 1) \cdot (n - 2)$.*

Proof. Let D_{ij} be the shortest distance from node 1 to node j in the given tree for each $1 < j \leqq n$. Assume with no loss of generality that $D_{12} \leqq D_{13} \leqq \cdots \leqq D_{in}$. Then for each $1 \leqq i \leqq j \leqq n$ such that d_{ij} is not in the proposed shortest path tree, we must verify that $d_{ij} \geqq D_{ij} - D_{ij}$ and this set of $\frac{1}{2}(n - 1) \cdot (n - 2)$ inequalities cannot be proven by an analytic proof of width less than $\frac{1}{2}(n - 1) \cdot (n - 2)$. Q.E.D.

In contrast to the case of spanning trees, when a new node is added, it requires an $O(n^2)$ algorithm to update the solution. In fact, the updating problems we considered for shortest paths all require $O(n^2)$ steps.

THEOREM 4.2. *Let G be an n-node positively weighted digraph for which a shortest path tree from node 1 to each other node is specified. Then:*

(i) *If a new node is added, any analytic tree program to update the solution will have depth at least $\frac{1}{2}(n - 1) \cdot (n - 2)$.*

(ii) *If a node is deleted, any analytic tree program for updating the set of paths will have depth at least $\frac{1}{2}(n - 2) \cdot (n - 3)$.*

(iii) *If the weight of some arc in a path is increased, any updating program will have depth at least $\frac{1}{2}(n - 2) \cdot (n - 3)$.*

(iv) *If the weight of some arc in a path is decreased, the minimum depth of an updating program is at least $\frac{1}{2}(n - 2) \cdot (n - 3)$.*

(v) *If the weight of an arc not in the shortest path tree is decreased, then any analytic tree program to update the solution has depth at least $\frac{1}{2}(n - 2) \cdot (n - 3)$.*

Proof. (i) Consider the case in which

$$d_{ij} = 1, \qquad 1 \leqq j \leqq n,$$

$$d_{ij} < \frac{1}{n}, \quad \text{all other } i \text{ and } j \text{ with } 1 \leqq i \neq j \leqq n + 1,$$

$$d_{i,n+1} = \min \{d_{ij} : 1 \leqq i \neq j \leqq n + 1\}.$$

Then the old tree had a direct arc from node 1 to each other node, but the new tree will not use any of these arcs. The new solution will have a direct path only from node 1 to node $n + 1$, and an entirely new solution for the rest of G which will entail finding a shortest path from node $n + 1$ to each other node.

(ii) Consider the case

$$\begin{aligned} &d_{12} = 1, \\ &d_{2j} = 1, \qquad 3 \leqq j \leqq n, \\ &d_{ij} > 2, \quad \text{all other } i \text{ and } j. \end{aligned}$$

Then if node 2 is deleted, an entirely new problem must be solved on nodes $1, 3, \cdots, n$.

(iii) Take

$$\begin{aligned} &d_{12} = 1, \\ &d_{2j} = 1, \qquad 3 \leqq j \leqq n, \\ &d_{ij} > 2, \qquad i \neq 2, \quad j \neq 2, \\ &d_{i2} > \sum_{j \neq 2} d_{ij}. \end{aligned}$$

Then the original solution is to go from node 1 to node 2 and thence directly to each other node. Now let d_{12} increase to be the maximum of all weights, and we must solve a new problem from node 1 to nodes 3 through n.

(iv) Let

$$\begin{aligned} &d_{ij} = 1, \qquad 1 < j \leqq n, \\ &d_{ij} < \frac{1}{n}, \quad \text{all other } i \text{ and } j, \end{aligned}$$

and now let d_{12} decrease to be the minimum weight arc. So we must solve a shortest path problem from node 2 to each other node.

(v) Let

$$\begin{aligned} &d_{12} = 1, \\ &d_{2j} = 1, \qquad 2 < j, \\ &d_{ij} > 2, \qquad j \neq 2, \\ &d_{ij} < \frac{1}{n}, \quad \text{all other } i \text{ and } j. \end{aligned}$$

Now let d_{13} decrease to $1/n$. Then we must solve a new problem from node 3 to nodes $2, 4, \cdots, n$. Q.E.D.

5. Further considerations. In this concluding section we make several further remarks about shortest paths and spanning trees. Firstly, there is an algorithm for shortest paths or for the spanning tree problem which uses an average of $\frac{1}{2}n^2 + O(n \log^2 n)$ comparisons. To see this, let G be a graph in which the weights are chosen independently from any probability distribution which has zero probability of yielding negative values. Then Spira's [6] algorithm for the all shortest path problem can be adapted to either of the above problems to yield an algorithm which uses $\frac{1}{2}n^2 + O(n \log^2 n)$ comparisons on the average. Secondly, we have discussed updating where only the answer to the problem considered is retained. It seems likely that if intermediate information in obtaining the original solution is kept, improvements will be possible. We have not investigated this. Thirdly, we have not considered sparse graphs. A major open problem is whether there are $O(E)$ algorithms for these computations in the case where E, the number of edges actually present, is small.

Acknowledgment. We acknowledge a helpful discussion with Professor Shimon Even concerning Theorem 3.3.

REFERENCES

[1] R. C. Prim, *Shortest interconnection network and some generalizations*, Bell System Tech. J., 36 (1967), pp. 1389–1401.

[2] E. W. Dijkstra, *A note on two problems in connection with graphs*, Numer. Math., 1 (1959), pp. 269–271.

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

[4] R. Tarjan, *Depth-first and linear graph algorithms*, this Journal, 1 (1972), pp. 146–160.

[5] R. Johnson and Janos Simon, Private communication.

[6] P. M. Spira, *A new algorithm for finding all shortest paths in a graph of positive arcs in average time $O(n^2 \log^2 n)$*, this Journal, 2 (1973), pp. 28–32.

SIAM J. COMPUT.
Vol. 4, No. 3, September 1975

ON THE NUMBER OF MULTIPLICATIONS/DIVISIONS EVALUATING A POLYNOMIAL WITH AUXILIARY FUNCTIONS*

L. REVAH†

Abstract. The number of multiplications/divisions (we use the notation m/d) necessary to evaluate an nth degree polynomial with auxiliary functions are studied. Motzkin proved, that in general, $\lceil (n + 1)/2 \rceil$ m/d are required when operations on coefficients of the polynomial are not counted and presented an algorithm which requires $\lceil (n + 2)/2 \rceil$ m/d.

The main purpose of this work is to treat the question, "Is the minimum number of m/d $\lceil (n + 1)/2 \rceil$ or $\lceil (n + 2)/2 \rceil$?" We obtain the following results:

(a) for $n < 9$, the minimum number is $\lceil (n + 2)/2 \rceil$,

(b) for $n \geqq 9$, we prove by exhibiting an algorithm that almost all polynomials of degree n can be evaluated with $\lceil (n + 1)/2 \rceil$ multiplications over a complex field.

Key words. algebraically closed field, algorithm, auxiliary functions, lower bounds, polynomials, preconditioning

1. Introduction.

We are interested in establishing the number of arithmetical operations required to evaluate the nth-degree polynomial $P_n(x) = \sum_{i=0}^{n} u_i x^i$. One can evaluate $x^2, x^3, \cdots, x^n$, then multiply a_i by x_i ($1 \leqq i \leqq n$), and finally add these products and a_0. This method requires $2n - 1$ m/d and n additions/subtractions (we use the notation a/s).

When a polynomial of degree n is computed at one point by Horner's method based on the following identity:

$$P_n(x) = \sum_{i=0}^{n} u_i x^i = (\cdots (u_n x + u_{n-1})x + \cdots)x + u_0$$

or

$$P_n(x) = P_{n-1}(x)x + u_0,$$

then n m/d and n a/s are involved. Horner's method turns out to be optimal from the point of view of operation count for computing a polynomial at one point (Pan [2]).

When the same polynomial is to be evaluated at several points, we can manipulate in some way the coefficients of the polynomial once and for all and then use these "adapted"- (we shall say preconditioned) coefficients in all subsequent evaluations.

A scheme with preconditioning is a sequence of steps:

(1) $$p_i = R_i' \circ R_i'', \qquad i = 1, 2, \cdots, r,$$

in which:

(a) $\circ$ is one of these operations: addition, subtraction, multiplication, division;

(b) R_i', R_i'' is either x, p_j ($j < i$) or a function $\alpha_j = f_j(u_0, u_1, \cdots, u_n)$;

(c) $p_r(x; \alpha_1, \alpha_2, \cdots, \alpha_k) = P_n(x) = \sum_{i=0}^{n} u_i x^i$.

The number of m/d in the scheme is the number of times $\circ$ is either multiplication or division. α_j is called a *parameter* or an *auxiliary function.*

* Received by the editors February 12, 1974.

† Department of Computer Science, University of Toronto, Toronto, Ontario, Canada M5S 1AL. This research was carried out at the Department of Computer Science, Technion-Israel Institute of Technology, Haifa, Israel.

P_j is called a *chain* step if either none of the R'_j, R''_j is a parameter or if $p_j = \alpha : R''_j$ and R''_j is not a parameter. Otherwise p_j is said to be a *parametric* step.

Motzkin proved that every scheme with preconditioning which computes a general nth-degree polynomial at a general point requires at least $\lceil (n+1)/2 \rceil$ m/d and at least n a/s, if operations on coefficients are not counted.

Motzkin [1] indicated a scheme for evaluating an nth-degree polynomial with $\lceil (n+2)/2 \rceil$ multiplications and $n+1$ a/s (for simplicity, we do not follow the indexing of (1), but instead, indicate the degree of a polynomial obtained at each multiplication step):

$$
\begin{aligned}
&p_2 = x(x+\alpha_1),\\
&p_4 = (p_2 + x + \alpha_2)(p_2 + \alpha_3) + \alpha_4,\\
&p_{2s+2} = p_{2s}(p_2 + \alpha_{2s+1}) + \alpha_{2s+2}, \qquad s = 2, 3, \cdots, k-1,\\
&p_n(x) = \begin{cases} u_n p_{2k} & \text{for even } n = 2k, \\ u_n p_{2k} x + u_0 & \text{for odd } n = 2k+1. \end{cases}
\end{aligned} \tag{2}
$$

(See also [5, pp. 7–10].) Other known methods do not achieve the lower bound of $\lceil (n+1)/2 \rceil$ for the number of m/d, which fact gave rise to the question, "Is the minimum number $\lceil (n+1)/2 \rceil$ or $\lceil (n+2)/2 \rceil$?" Since $\lceil (n+2)/2 \rceil$ turns out to be $\lceil (n+1)/2 \rceil$ when n is even, we deal only with polynomials of odd degree. (This question was raised in Knuth [3, § 4.6.4, Prob. 40]. Rabin and Winograd [4] gave a scheme for the case $n = 13$ using 7 multiplications.)

First we shall establish the necessary conditions for schemes evaluating all nth-degree polynomials with $\lceil (n+1)/2 \rceil$ m/d. Then we shall prove that for $1 < n < 9$, the lower bound on the number of m/d is $\lceil (n+2)/2 \rceil$.

In the last section we shall prove that for $n \geqq 9$ the lower bound is $\lceil (n+1)/2 \rceil$ by constructing four schemes over a complex field corresponding to four cases of odd n:

1. $n = 4k - 1, k \geqq 3$;
2. $n = 4k - 3, k > 3$ odd;
3. $n = 4k - 3, k > 2$ even;
4. $n = 9$.

2. Necessary conditions for an algorithm to compute any nth-degree polynomial with $\lceil (n+1)/2 \rceil$ m/d. We cite the following result.

LEMMA 1 (Pan [2, p. 113], Belaga [5, p. 11]). *The number k of parameters involved in a scheme* (1) *computing an n-th-degree polynomial satisfies the inequality $k \geqq n + 1$.*

With no loss of generality, we can assume that the scheme (1) with t m/d is of the form: (we write $\divideontimes$ for m/d)

$$
\begin{aligned}
&q_i = T_i^{(1)} \divideontimes T_i^{(2)}, \qquad i = 1, \cdots, t,\\
&q_{t+1} \equiv P_n(x) = q_t \pm T_t^{(3)},
\end{aligned} \tag{3}
$$

where $T_j^{(i)}$ $(1 \leqq j \leqq t,\ 1 \leqq i \leqq 3)$ is some sum of q_s $(s < j)$, x and parameters α. Then it is easy to prove the following lemma.

LEMMA 2 (Knuth [3, § 4.6.4, Prob. 30, p. 443]). *For any scheme of the form* (1) *with t_1 chain and t_2 parametric m/d, it is possible to construct a scheme of the form*

(3) *which evaluates the same polynomial as the preceding one with at most* $2t_1 + t_2$ *parameters.*

From these two lemmas we deduce the following corollary.

COROLLARY 1. *If a scheme* (3) *evaluates any n-th-degree polynomial with* $\lceil (n+1)/2 \rceil$ *m/d when n is odd,* $n \geqq 3$, *then all m/d must be* chain *operations.*

Proof. Indeed, it is possible to construct a scheme (3) with $t_1 + t_2 = (n+1)/2$ m/d and at most $2t_1 + t_2$ parameters (Lemma 2). But $2t_1 + t_2 \geqq n + 1$ (Lemma 1). Therefore $t_2 = 0$. □

DEFINITION 1. We call the operation $p_i = R_i' \circ R_i''$ a *nonreducing operation* if either one of the following conditions holds:

(i) if $\circ$ is $\div$, $R_j' = P$ and $R_j'' = Q$, then

$$\text{degree}(P) > \text{degree}(Q);$$

(ii) if $\circ$ is $\pm$, $R_j' = P/Q$ and $R_j'' = P'/Q'$, then

$$\text{degree}(PQ' \pm P'Q) = \max(\text{degree}(PQ'), \text{degree}(P'Q)),$$

where P, P', Q and Q' are some polynomials in x.

A scheme is called a *nonreducing scheme* if it contains only nonreducing operations. Otherwise the scheme is said to be *reducing.*

THEOREM 1. *If a nonreducing scheme does not contain any parametric m/d, then the leading coefficients of polynomials in numerator and denominator obtained at each step* p_j *are rational numbers. Therefore every nonreducing scheme without parametric m/d evaluates only polynomials with rational leading coefficient.*

Proof. We prove this theorem by use of induction on the number of m/d in the scheme (3). □

This leads to the following theorem.

THEOREM 2. *The necessary conditions for a scheme to evaluate any n-th-degree polynomial when n is odd,* $n > 1$, *with* $\lceil (n+1)/2 \rceil$ *m/d are:*

(i) *the scheme must not contain parametric m/d;*

(ii) *the scheme has to be a reducing one.*

The point is that in other methods, the monic polynomial p_n of degree n was obtained first, and then, in order to compute the general polynomial $P_n(x) = \sum_{i=1}^{n} u_i x^i$, the monic polynomial $p_n(x)$ had to be multiplied by a parameter u_n. We avoid this loss of multiplication.

In fact, we will try to obtain two monic polynomials of the same degree using chain multiplications only and then subtract these two polynomials obtaining a polynomial with a general leading coefficient:

$$R_j' = x^s + a_{s-1}x^{s-1} + \cdots + a_0,$$
$$R_j'' = x^s + b_{s-1}x^{s-1} + \cdots + b_0,$$

and if $p_j = R_j' - R_j''$, then

$$p_j = (a_{s-1} - b_{s-1})x^{s-1} + \cdots + (a_0 - b_0).$$

Theorem 2 and Lemma 1 lead us to the following result concerning the number of a/s in a scheme evaluating any nth-degree polynomial with $\lceil (n+1)/2 \rceil$ m/d.

COROLLARY 2. *For odd n, the lower bounds of* $\lceil (n+1)/2 \rceil$ *for m/d and n for a/s cannot both be achieved. A scheme evaluating any n-th-degree polynomial with* $\lceil (n+1)/2 \rceil$ *m/d contains at least* $n+1$ *a/s.*

Proof. There are at least $n+1$ parameters in a scheme evaluating an nth-degree polynomial (Lemma 1). Parametric *m/d* are not allowed (Theorem 2). Therefore there are at least $n+1$ parametric a/s defining the $n+1$ parameters. □

When we try to apply the above-stated idea for a polynomial of odd degree n, $1 < n < 9$, we cannot achieve the lower bound of $\lceil (n+1)/2 \rceil$ for the number of m/d.

3. Lower bound for polynomials of degrees $n = 3, 5, 7$. The necessary conditions of the preceding section do not imply the achievement of the lower bound for $n = 3, 5, 7$.

LEMMA 3. *If* $p_m = p_k - p'_k$ *is the first reducing operation in the scheme not containing any other multiplication, except the first one, by a first-degree polynomial (i.e., there is no multiplication of the form* $p_j = (\sum_{i<j} u_i p_i + vx + \alpha)(wx + \beta)$, *where* u_i, v *and* w *are integers), then the leading coefficient of* x^{t-1} *in* p_m *is an integer (possibly 0), t being the degree of* p_k *(and* p'_k*).*

Proof. Only chain m/d may be used in the scheme (3). Let us prove the lemma in the case in which only multiplications are used. Without loss of generality, we may assume that the first chain multiplication is of the form $p_2 = x(nx + \alpha)$, where n is an integer.

It follows from the hypothesis of the lemma that any polynomial obtained before the reducing operation p_m is of even degree with an integer leading coefficient. Moreover, any rth-degree polynomial obtained in the scheme before the operation p_m has its two highest terms of the form

$$nsx^r + \left(s \cdot \frac{r}{2} \cdot \alpha + u\right)x^{r-1},$$

where u and s are integers. (This can be proved easily by use of induction on r.)

Therefore, in the reducing operation p_m, p_k and p'_k are of the following form:

$$p_k = ns_1x^t + \left(\frac{s_1t}{2}\alpha + u_1\right)x^{t-1} + \cdots,$$

$$p'_k = ns_2x^t + \left(\frac{s_2t}{2}\alpha + u_2\right)x^{t-1} + \cdots,$$

where u_1, u_2, s_1 and s_2 are integers and $s_1 = s_2$. Thus $p_m = (u_2 - u_1)x^{t-1} + \cdots$, and its leading coefficient is equal to $u_2 - u_1$, which is an integer number, as required. □

With this lemma, we can prove, by use of combinatorial analysis, the impossibility of achieving the bound $\lceil (n+1)/2 \rceil$ for $n = 3, 5, 7$.

THEOREM 3. *For odd n,* $1 < n < 9$, *there is no method evaluating all n-th-degree polynomials with less than* $\lceil (n+2)/2 \rceil$ *m/d.*

Proof. In order to illustrate our method of proof, we shall consider a scheme (3) without divisions. (When divisions are allowed, the proof is similar but a little cumbersome.) We consider a four-level tree (see Fig. 1), each level corresponding to chain multiplication. (Remember that parametric multiplications are not allowed.) A node p_i represents a polynomial of degree at most i whose leading coefficient is an integer number, and p_j^* is a polynomial of degree at most j with a general leading coefficient.

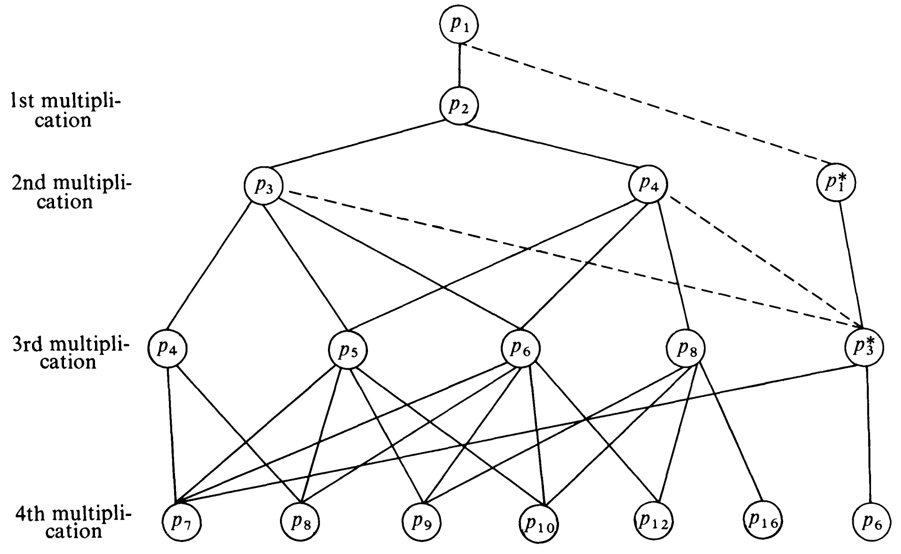


FIG. 1

The root of the tree is a polynomial of first degree with integer leading coefficient obtained without multiplication.

The dotted line joining the node p_i to the node p_j or p_j^* indicates one multiplication and one reducing subtraction. We can multiply only those polynomials appearing in the path from the root to the node p_i or p_i^*.

The first level corresponds to the first chain multiplication, which gives a polynomial of degree 2 with integer leading coefficient.

The second level corresponds to two multiplications, and its nodes represent integer leading coefficient polynomials p_3, p_4. p_1^* appears also in the second level; it results from subtraction of two different second-degree polynomials with equal leading coefficients obtained by 2 different multiplications.

In the fourth level, we include only polynomials of degree seven or higher, since we are interested in obtaining the general seventh-degree polynomial with 4 multiplications. Therefore there appear the nodes p_7, p_8, p_9, p_{10}, p_{12}, p_{16} representing the corresponding polynomials evaluated by use of 4 multiplications. (Other polynomials are of lower degrees and are of no interest in obtaining the bound.)

For example, monic seventh-degree polynomials can be calculated with 4 multiplications in one of the following ways:

$$p_2, p_3, p_4; \qquad p_2, p_3, p_5; \qquad p_2, p_4, p_5; \qquad p_2, p_4, p_6.$$

From the above considerations and from Lemma 3, we can conclude that there is no way to obtain p_3^* in the second level, p_5^* in the third level and p_7^* in the fourth level. □

4. Algorithms for computing polynomial of degree n with $\lceil (n+1)/2 \rceil$ multiplications. These algorithms will be reducing. We can point out that for all algorithms we will start with one even-degree monic polynomial, p_{2t}, and two odd-degree monic polynomials P_{2t_1+1} and P_{2t_2+1}, which we obtain by known methods, and then we construct the required algorithm.

THEOREM 4. *Let $x^2 + \lambda x$ be given. Then every monic polynomial p_{2t+1} of odd degree $2t + 1$ can be evaluated with t multiplications (and $2t + 1$ a/s).*

Proof. We use induction on t. The proof is obvious for $t = 1$. (Since for any λ a monic polynomial $x^3 + \sum_{i=0}^{2} a_i x^i$ can be written as $(x^2 + \lambda x + \alpha_1)(x + \alpha_0) + \gamma_1$, where $\alpha_0 = a_2 - \lambda$, $\alpha_1 = a_1 - \alpha_0\lambda$ and $\gamma_1 = a_0 - \alpha_1\alpha_2$, then only one multiplication and 3 a/s are required.) Assume the theorem to be true for every $1 < l < t$.

For a given λ, any monic polynomial P_{2t+1} can be written as

$$P_{2t+1} = \sum_{i=0}^{t} a_{2i}\left(x + \frac{\lambda}{2}\right)^{2i} + \left(x + \frac{\lambda}{2}\right)\sum_{i=0}^{t} a_{2i+1}\left(x + \frac{\lambda}{2}\right)^{2i}$$

for a suitable choice of a_i's. So, for every α_t,

$$P_{2t+1} = \left(\left(x + \frac{\lambda}{2}\right)^2 - \alpha_t\right)P_{2t-1} + \beta_t\left(x + \frac{\lambda}{2}\right) + \gamma_t,$$

where

$$\beta_t = \sum_{i=0}^{t} a_{2i+1}\alpha_t^{2i} \quad\text{and}\quad \gamma_t = \sum_{i=0}^{t} a_{2i}\alpha_t^{2i}.$$

Now choose α_t, a root of $\sum_{i=0}^{t} a_{2i+1} z^{2i}$; then

$$P_{2t+1} = P_{2t-1}\left(x^2 + \lambda x + \frac{\lambda^2}{4} - \alpha_t\right) + \gamma_t,$$

and P_{2t+1} can be obtained from P_{2t-1} given $x^2 + \lambda x$ by one more multiplication and 2 more additions involving α_t' and γ_t (where $\alpha_t' = \lambda^2/4 - \alpha_t$). By the induction hypothesis on P_{2t-1}, the theorem follows. □

THEOREM 5. *There exists a scheme evaluating s monic polynomials P_{2t_i+1} of odd degree $2t_i + 1$ $(i = 1, 2, \cdots, s)$ and a monic polynomial P_{2t} of even degree $2t$ using $t + \sum_{i=1}^{s} t_i$ multiplications (and $2t + 1 + 2\sum_{i=1}^{s} t_i + s$ a/s).*

Proof. Let

$$P_{2t} = x^{2t} + u_{2t-1}x^{2t-1} + \cdots + u_0,$$
$$P_{2t_i+1} = x^{2t_i+1} + v_{2t_i}^{(i)}x^{2t_i} + \cdots + v_0^{(i)}, \qquad i = 1, 2, \cdots, s.$$

First we compute P_{2t} with Motzkin's method:

$$\begin{aligned} &p_2 = x(x+\lambda),\\ &p_4 = (p_2 + x + \alpha_2)(p_2 + \alpha_3) + \alpha_4,\\ &p_{2s+2} = p_{2s}(p_2 + \alpha_{2s+1}) + \alpha_{2s+2}, \qquad s = 2, \cdots, t-1,\\ &P_{2t}(x) \equiv p_{2t}. \end{aligned} \tag{4}$$

The first multiplication in this algorithm is $p_2 = x(x + \lambda)$ with $\lambda = (u_{2t-1} - 1)/t$. The number of multiplications used is t (the number of a/s is $2t + 1$). Now the quadratic polynomial $x^2 + \lambda x$ is given. Therefore each of the polynomial P_{2t_i+1} can be evaluated using t_i multiplications and $2t_i + 1$ a/s by the scheme of Theorem 4. □

Let $Q_{2m+1}(x) = x^{2m+1} + \sum_{i=0}^{2m} q_i x^i$, and let $\hat{C}_r(x) = \sum_{i=0}^{r} \hat{c}_i x^i$.

LEMMA 4. *For any monic Q_{2m+1}, any $\hat{C}_r$ ($r \leqq \lceil m/2 \rceil - 1$) and any number α, there exists a monic polynomial $P_m(x) = x^m + \sum_{i=0}^{m-1} a_i x^i$ such that degree $(Q_{2m+1} - P_m(P_m + \hat{C}_r)(x + \alpha)) \leqq m$. Furthermore, a_{m-j} ($1 \leqq j \leqq m - r - 1$) is a polynomial in α and q_{2m-i} ($0 \leqq i \leqq j - 1$), $2a_{r-j} + \hat{c}_{r-j}$ ($0 \leqq j \leqq r$) is a polynomial in α, q_i ($m + r - j + 1 \leqq i \leqq 2m$), and $2a_{r-s} + \hat{c}_{r-s}$ ($s < j$).*

Proof. Let us consider the polynomial

$$S = P_m(P_m + \hat{C})(x + \alpha) = P_m^2 x + P_m^2 \alpha + P_m \hat{C}_r x + P_m \hat{C}_r \alpha. \tag{5}$$

The coefficient s_j of x^j in S, $j = m + t$, $r + 1 < t < m$, is

$$s_j = 2a_{t-1} + \sum_{i=1}^{m-t} a_{m-i} a_{t+i-1} + \alpha \sum_{i=0}^{m-t} a_m \quad a_{t+i}. \tag{6}$$

For $j = m + t$, $0 < t \leqq r + 1$,

$$\begin{aligned} s_j = 2a_{t-1} + \hat{c}_{t-1} &+ \sum_{i=1}^{r-t+1} a_{m-i}(2a_{t+i-1} + \hat{c}_{t+i-1}) + \alpha \sum_{i=0}^{r-t} a_{m-i}(2a_{t+i} + \hat{c}_{t+i})\\ &+ \sum_{i=r-t+2}^{m-r-1} a_{m-i} a_{t+i-1} + \alpha \sum_{i=r-t+1}^{m-r-1} a_{m-i} a_{t+i}. \end{aligned} \tag{7}$$

So we can choose a_k recursively in the following manner: for $r + 1 < t \leqq m$,

$$a_{t-1} = \frac{1}{2}\left[q_{m+t} - \sum_{i+1}^{m-t} a_{m-i} a_{t+i-1} - \alpha \sum_{i=0}^{m-t} a_{m-i} a_{t+i} \right]. \tag{8}$$

For $0 < t \leqq r + 1$, we choose a_{t-1} such that

$$\begin{aligned} 2a_{t-1} + \hat{c}_{t-1} = q_{m+t} &- \sum_{i=1}^{r-t+1} a_{m-i}(2a_{t+i-1} + \hat{c}_{t+i-1}) - \sum_{i=r-t+2}^{m-r-1} a_{m-i} a_{t+i-1}\\ &- \alpha \sum_{i=0}^{2-t} a_{m-i}(2a_{t+i} + \hat{c}_{t+i}) - \alpha \sum_{i=r-t-1}^{m-r-1} a_{m-i} a_{t+i}. \end{aligned} \tag{9}$$

Therefore $P_m(x) = \sum_{i=0}^{m} a_i x^i$. □

LEMMA 5. *Let S be as in Lemma 4. Then*

(i) *for $2r + 1 \leqq j \leqq m$, $s_s = f_j(\alpha, 2a_0 + \hat{c}_0, \cdots, 2a_r + \hat{c}_r, \hat{c}_r, a_r, a_{r+1}, \cdots, a_m)$;*

(ii) *for* $j = r + t,\ 0 < t \leqq r + 1,\ s_j = \hat{c}_r a_{t-1} + f_j\ (\alpha, 2a_0 + \hat{c}_0, 2a_r + \hat{c}_r, a_t, \cdots, a_m, \hat{c}_t, \cdots, \hat{c}_r)$.

Proof. (i)

$$\begin{aligned} s_j = {} & \sum_{i=0}^{r-1} a_{j-i-1}(2a_i + \hat{c}_i) + a_{j-r-1}\hat{c}_r + \sum_{i=0}^{j-r-1} a_{j-i-1}a_i \\ & + \alpha \sum_{i=0}^{r} a_{j-i}(2a_i + \hat{c}_i) + \alpha \sum_{i=r+1}^{j-r-1} a_{j-i}a_i. \end{aligned} \tag{10}$$

(ii) For $t < r + 1$,

$$\begin{aligned} s_j \neq {} & \hat{c}_r a_{t-1} + \sum_{i=0}^{t-1} a_{j-i-1}(2a_i + \hat{c}_i) + \alpha \sum_{i=0}^{t-1} a_{j-i}(2a_i + \hat{c}_i) \\ & + \sum_{i=t}^{r-1} a_{j-i-1}(a_i + \hat{c}_i) + \alpha \sum_{i=t}^{r} a_{j-i}(a_i + \hat{c}_i). \end{aligned} \tag{11}$$

For $t = r + 1$,

$$s_{2r+1} = \hat{c}_r a_r + \sum_{i=0}^{r-1} a_{2r-i}(2a_i + \hat{c}_i) + \alpha \sum_{i=0}^{r} a_{2r-i+1}(2a_i + \hat{c}_i) + a_r^2. \quad \square \tag{12}$$

4.1. Algorithm for $n = 4k - 1, k \geqq 3$. Let $u(x) = \sum_{i=0}^{n} u_i x^i$ be given. Let

$$P_{2k-1}(x) = x^{2k-1} + \sum_{i=0}^{2k-2} a_i x^i, \qquad C_{k-2}(x) = x^{k-2} + \sum_{i=0}^{k-3} c_i x^i,$$

$$D_{k-1}(x) = x^{k-1} + \sum_{i=0}^{k-2} d_i x^i.$$

THEOREM 6. *For almost every polynomial $u(x)$ of degree $n = 4k - 1$, $k \geqq 3$, there exists three monic polynomials P_{2k-1}, D_{k-1}, C_{k-2} and numbers α, α^*, β and γ such that*

$$\begin{aligned} u(x) = {} & [P_{2k-1}(x + \alpha) - (P_{2k-1} + C_{k-2})(x + \alpha^*) + \beta] \\ & \cdot [P_{2k-1}(x + \alpha) + \gamma] + D_{k-1}. \end{aligned} \tag{13}$$

Proof. We have to find $4k$ parameters:

$$a_{2k-2}, \cdots, a_0; \quad c_{k-3}, \cdots, c_0; \quad d_{k-2}, \cdots, d_0; \quad \alpha, \alpha^*, \gamma, \beta. \tag{14}$$

Clearly, $\alpha - \alpha^* = u_n$. If $Q(x) = x^{4k-1} + \sum_{i=0}^{4k-2} q_i x^i = u(x)/(\alpha - \alpha^*)$, then

$$\begin{aligned} Q(x) = {} & P_{2k-1}\left[P_{2k-1} - C_{k-2}\left(\frac{x + \alpha^*}{\alpha - \alpha^*}\right) + \frac{\beta}{\alpha - \alpha^*}\right](x + \alpha) \\ & + \gamma\left[P_{2k-1} - C_{k-2}\left(\frac{x + \alpha^*}{\alpha - \alpha^*}\right) + \frac{\beta}{\alpha - \alpha^*}\right] + \frac{D_{k-1}}{\alpha - \alpha^*}. \end{aligned} \tag{15}$$

Using Lemmas 4 and 5 with

$$r = k - 1, \qquad m = 2k - 1,$$

$$\hat{C}_r = -C_{k-2}\left(\frac{x + \alpha^*}{\alpha - \alpha^*}\right) + \frac{\beta}{\alpha - \alpha^*} \quad \text{and} \quad \alpha^* = \alpha - u_n,$$

it is possible to obtain a_i, $i = 2k-2, \cdots, k$, and $2a_i + \hat{c}_i$, $i = k-1, \cdots, 0$, as some functions of α and of coefficients of Q such that degree $(Q - S) = 2k - 1$, where $S = P_{2k-1}(P_{2k-1} + \hat{C}_{k-1})(x + \alpha)$ (Lemma 4).

Continuing to compare coefficients in both sides of (15) and using Lemma 5, we obtain formulas for computing γ in terms of α and $q_{2k-1}, \cdots, q_{4k-3}, q_{4k-2}$, for a_{k-j-1}, $j = 1, 2, \cdots, k-1$, in terms of α and $q_{4k-2}, \cdots, q_{2k-j-1}$, and therefore for $\hat{c}_{k-j-1}$ $(\hat{c}_{k-1} = -1/u_n)$.

Comparing coefficients of x^{k-1} in (15), we obtain some polynomial equation in α.

Indeed, from (15), we obtain $q_{k-1} = s_{k-1} + \gamma(a_{k-1} + \hat{c}_{k-1}) + 1/u_n$, or

$$\begin{aligned}\sum_{i=0}^{k-2} (a_{k-i-2} + \alpha a_{k-i-1})(a_i + \hat{c}_1) + \alpha a_0(\alpha_{k-1} + \hat{c}_{k-1}) \\ + \gamma(a_{k-1} + \hat{c}_{k-1}) + 1/u_n - q_{k-1} = 0.\end{aligned} \tag{16}$$

The left side of (16) is a polynomial in α; we denote it $P(\alpha)$. In order to prove the solvability of $P(\alpha) = 0$, we have to show that $P(\alpha)$ is not identically a constant (for each set of $u_n, q_{4k-1}, \cdots, q_0$). (We work in the field of complex numbers, which is algebraically closed.)

Let us write $P(\alpha)$ as a sum of two polynomials: the first is a polynomial in α and q_i, $i \neq k$, $k - 1 \leqq i \leqq 4k - 1$, and does not depend on q_k; the second is a polynomial in α and depends on q_k. With Lemma 5, we conclude that q_k does not appear in a_i, $i > 0$; q_k enters only in the computation of a_0.

Now we write

$$\begin{aligned}P(\alpha) = \sum_{i=1}^{k-3} (a_{k-i-2} + \alpha a_{k-i-1})(a_i + \hat{c}_i) \\ + a_0[2a_{k-2} + \hat{c}_{k-2} + (2a_{k-1} + \hat{c}_{k-1})] + \hat{c}_0(a_{k-2} + \alpha a_{k-1}) \\ + \alpha a_1(a_{k-2} + \hat{c}_{k-2}) + \gamma(a_{k-1} + \hat{c}_{k-1}) + \frac{1}{u_n} - q_{k-1}.\end{aligned}$$

We obtain $P(\alpha) = R(\alpha) - q_k u_n(\hat{c}_{k-2} - \alpha/u_n)$, where $R(\alpha)$ does not depend on q_k. Denoting $\phi(\alpha) = \hat{c}_{k-2} - \alpha/u_n$ and using the equation (10) for the coefficient of x^{2j-2}, $j = 1, 2, \cdots, k-1$, and the equation (11) for $t = k-1$, and using (12) with $t = k$, we obtain $\phi(\alpha) = g(\alpha) - \alpha/(4u_n) - q_{4k-2}/2$, and $1/u_n$ does not appear in $g(\alpha)$. The coefficient of α in $\phi(\alpha)$ is a polynomial in $1/u_n$. Since the coefficient of $1/u_n$ is $-\frac{1}{4}$, therefore $\phi(\alpha)$ cannot be a constant for every set of $u_n, q_{4k-2}, \cdots, q_0$ (it contains at least a linear member as a polynomial in α). Therefore $P(\alpha) = R(\alpha) - q_k u_n \phi(\alpha)$ cannot be a constant identically. So $P(\alpha) = 0$ almost always has some solution.

Once α is fixed, we compute $a_{2k-1}, \cdots, a_0, c_{k-3}, \cdots, c_0, \beta, \gamma$ and α^*.

Finally, equating the coefficients of x^{k-j}, $j = 2, 3, \cdots, k$, on both sides of (15), we obtain d_{k-j}. $\square$

LEMMA 6. *There exists a scheme computing P_{2k-1}, D_{k-1} and C_{k-2} with $2k-3$ multiplications (and $4k-3$ a/s).*

Proof. We have two monic polynomials of odd degree, P_{2k-1} and either D_{k-1} or C_{k-2}, and one monic even-degree polynomial, either C_{k-2} or D_{k-1}. So we can use Theorem 5. □

THEOREM 7. *There exists a scheme computing almost every polynomial of degree* $n = 4k - 1$, $k \geqq 4$, *with* $\lceil (n+1)/2 \rceil$ *multiplications* (*and* $n + 5$ *a/s*).

It is clear how we compute $u(x)$. First we evaluate all the parameters (14) such that (13) holds. Then we use Lemma 6 to evaluate P_{2k-1}, D_{k-1} and C_{k-2} with $2k - 3$ multiplications. We need 3 additional multiplications in order to obtain $u(x)$ by (13).

Example. Evaluation of a polynomial $u(x) = \sum_{i=0}^{11} u_i x^i$ of degree 11 will proceed as follows:

1. *Preconditioning.*
   (i) compute parameters $a_4, a_3, a_2, a_1, a_0, \alpha, \alpha^*, \beta, c_0, \gamma, d_1$ and d_0 such that the equality
   $$u(x) = [P_5(x+\alpha) - (P_5 + x + c_0)(x + \alpha^*) + \beta][P_5(x+\alpha) + \gamma] + x^2 + d_1 x + d_0$$
   holds, where $P_5(x) = x^5 + \sum_{i=0}^{4} a_i x^i$.
   (ii) compute parameters $\alpha_2, \alpha_1, \beta_2, \beta_1, \beta_0$ (as rational functions of $a_4, a_3, a_2, a_1, a_0, d_1$) satisfying the equality:
   $$P_5(x) = [(x^2 + d_1 x + \alpha_1)(x + \beta_0) + \beta_1](x^2 + d_1 x + \alpha_2) + \beta_2.$$
2. Evaluate $x^2 + d_1 x = x(x + d_1)$ using one multiplication (and one addition).
3. Compute $P_5(x)$ using the equality in (ii) with 2 additional multiplications (and 5 a/s).
4. Evaluate $u(x)$ by the equality in (i) using 3 multiplications (and 9 a/s).

The total number of multiplications involved is 6 (and for a/s is 15).

4.2. Algorithm for $n = 4k - 3$, when $4 \leqq k$ is even. As for the preceding scheme, the present one will be reducing. We use one monic even-degree polynomial,
$$P_{2k-2} = x^{2k-2} + \sum_{i=0}^{2k-3} a_i x^i,$$
and two odd-degree monic polynomials,
$$C_{k-3} = x^{k-3} + \sum_{i=0}^{k-4} c_i x^i \quad \text{and} \quad D_{k-1} = x^{k-1} + \sum_{i=0}^{k-2} d_i x^i.$$

THEOREM 8. *For almost every polynomial* $u(x) = \sum_{i=0}^{n} u_i x^i$, $n = 4k - 3$, $k \geqq 5$, *k even, there exist three monic polynomials* $P_{2k-2}, C_{k-3}, D_{k-1}$ *and numbers* α, α^*, β *and* γ, *such that*

(17)
$$u(x) = [P_{2k-2}(x+\alpha) - (P_{2k-2} + C_{k-3})(x+\alpha^*) + \beta][P_{2k-2}(x+\alpha) + \gamma] + D_{k-1}.$$

Proof. The method of proof is similar to that of Theorem 6.

From Theorem 5, the proof of the following lemma is immediate.

LEMMA 7. *There exists a scheme computing P_{2k-2}, D_{k-1} and C_{k-3} with $2k-4$ multiplications (and $4k-5$ a/s).*

THEOREM 9. *There exists a scheme computing almost every polynomial of degree $n = 4k-3$, when $k \leqq 4$ is even, with $\lceil (n+1)/2 \rceil$ multiplications (and $n+5$ a/s).*

4.3. Algorithm for $n = 4k - 3$, $k > 3$ odd.

THEOREM 10. *For almost every polynomial $u(x) = \sum_{i=0}^{n} u_i x^i$, $n = 4k-3$, $k > 3$ odd, there exists three monic polynomials*

$$P_{2k-2} = x^{2k-2} + \sum_{i=0}^{2k-3} a_i x^i, \qquad C_{k-2} = x^{k-2} + \sum_{i=0}^{k-3} c_i x^i,$$

$$D_{k-4} = x^{k-4} + \sum_{i=0}^{k-5} d_i x^i,$$

and numbers α, α^*, β, γ, δ and b such that

$$u(x) = [P_{2k-2}(x+\alpha) - (P_{2k-2}+\beta)(x+\alpha^*) + C_{k-2}][P_{2k-2}(x+\alpha)+\gamma]$$
$$+ (C_{k-2} + D_{k-4})(x^2 + ax + b) + C_{k-2} + \delta,$$

where $a = (a_{2k-3} - 1)/(k-1)$.

LEMMA 8. *There exists a scheme computing $P_{2k-2}, x^2 + ax$, D_{k-4} and C_{k-2} with $2k-5$ multiplications (and $4k-7$ a/s).*

Proof. As in Theorem 5, we first compute P_{2k-2} by Motzkin's method (4). The first multiplication in this method is $x^2 + ax$, where $a = (2_{2k-3} - 1)/(k-1)$. Then we use the scheme of Theorem 4 to compute odd-degree monic polynomials D_{k-4} and C_{k-2} with first multiplication $x^2 + ax$. □

THEOREM 11. *There exists a scheme computing almost every polynomial of degree $n = 4k-3$, when $k > 3$ is odd, with $\lceil (n+1)/2 \rceil$ multiplications (and $n+7$ a/s).*

4.4. Algorithm for $n = 9$.

THEOREM 12. *For almost every polynomial $u(x) = \sum_{i=0}^{9} u_i x^i$, there exist a monic polynomial P_4 and numbers α, α^*, β, γ, δ and ε such that*

$$u(x) = [P_4(x)(x+\alpha) - (P_4(x)+\beta)(x+\alpha^*) + \gamma][P_4(x)(x+\alpha)+\delta] + \varepsilon.$$

THEOREM 13. *There exists a scheme computing almost every polynomial of degree 9 with $\lceil (n+1)/2 \rceil = 5$ multiplications (and $n+3$ a/s).*

Proof. Compute P_4 by the two first multiplications of (4). This requires 2 multiplications of 5 a/s. Then compute $u(x)$ by the identity of the Theorem 12.

Remarks. We have proved the existence of a scheme "for almost every" polynomial $u(x)$. "Almost every" was due to the fact that we succeeded in proving the solvability of the polynomial equation $P(\alpha) = 0$ for almost every polynomial $u(x)$. However, we feel that the following conjecture is true.

Conjecture. Theorems 7, 9, 11 and 13 are true for all nth-degree polynomials, when n is odd. Therefore, the corresponding algorithms can evaluate every nth-degree general polynomial.

We have seen in Theorems 6, 8 and 10 that in order to obtain the required parameters, we have to solve $P(\alpha) = 0$. If there exists some real solution α, then the parameters are real for real coefficients u_i. However, the algorithm of Theorem 5 may involve complex parameters for real coefficients of P, C and D. Therefore, the question, "Can the lower bound of $\lceil (n + 1)/2 \rceil$ for the number of m/d be achieved over the field of real numbers?" remains open.

Acknowledgment. I would like to acknowledge the valuable guidance of Professor S. Winograd of Technion-Israel Institute of Technology, my research advisor, throughout the course of this master's thesis research.

REFERENCES

[1] T. S. Motzkin, *Evaluation of polynomials and evaluation of rational functions*, Bull. Amer. Math. Soc., 61 (1955), p. 163.
[2] V. Ya. Pan, *Methods of computing values of polynomials*, Russian Math. Surveys, 21 (1966), pp. 106–136.
[3] D. E. Knuth, *The Art of Computer Programming*, vol. 2, Addison-Wesley, Reading, Mass., pp. 422–444.
[4] M. O. Rabin and S. Winograd, *Fast evaluation of polynomials by rational preparation*, RC 4802, IBM T. J. Watson Research Center, Yorktown Heights, N.Y., 1971.
[5] E. G. Belaga, *Evaluation of polynomials of one variable with preliminary processing of the coefficients*, Problemy Kibernet., 5 (1961), pp. 7–15.

SIAM J. COMPUT.
Vol. 4, No. 3, September 1975

AN ALGORITHM FOR DETERMINING WHETHER THE CONNECTIVITY OF A GRAPH IS AT LEAST k*

SHIMON EVEN†

Abstract. The algorithm presented in this paper is for testing whether the connectivity of a large graph of n vertices is at least k. First the case of undirected graphs is discussed, and then it is shown that a variation of this algorithm works for directed graphs. The number of steps the algorithm requires, in case $k < \sqrt{n}$, is bounded by $O(kn^3)$.

Key words. algorithm, connectivity, graph

1. Introduction. Let G be a finite undirected graph with n vertices and e edges. We assume that G has no self-loops and no parallel edges. A set of vertices, S, is called a *separating set* if there exists two vertices $a, b \notin S$ such that all paths between a and b pass through at least one vertex of S. The *connectivity*, c, of G is defined in the following way:

(i) if G is completely connected,[1] then $c = n - 1$,
(ii) if G is not completely connected, then c is the least number of vertices in a separating set.

Menger's theorem [1] states that if the connectivity of G is c, then for every two vertices a and b there exist c vertex-disjoint paths connecting a and b;[2] and conversely, if for every two vertices a and b there exist c vertex-disjoint connecting paths, then the connectivity of G is at least c. Dantzig and Fulkerson [2] introduced the relation between connectivity and network flow. Thus the Ford and Fulkerson [3] algorithm can be used to determine the connectivity of a graph. In fact, the max-flow min-cut theorem (and algorithm) immediately translates to the following: the maximum number of vertex-disjoint paths connecting vertices a and b is equal to the minimum cardinality separating set between a and b, in case there is no edge between a and b; otherwise the number of paths is one more than the minimum cardinality of a set separating a from b after the edge between them has been deleted.

Thus one can find the connectivity of a graph in the following way: for each pair of vertices, find the maximum number of vertex-disjoint paths. The minimum value over all pairs is the connectivity.

For each pair of vertices, we construct a flow network whose number of vertices is $2n$ and the number of edges is $2e + n$. The capacities are all one.[3] Each labeling and augmenting path realization costs $O(e)$ steps, and it corresponds to one path between the two vertices. Since the connectivity can be as high as $n - 1$, the whole procedure for finding the maximum number of vertex-disjoint paths connecting this pair of vertices is at most of cost $O(ne)$, or $O(n^3)$ if $O(e) = O(n^2)$.

* Received by the editors September 12, 1973, and in revised form August 7, 1974.

† Department of Computer Science, Technion-Israel Institute of Technology, Haifa, Israel. On leave of absence from Weizmann Institute of Science, Rehovot, Israel.

[1] Each pair of vertices is connected by an edge. In this case, G has no separating sets.

[2] Clearly, the vertices a and b are shared by all c paths, but no other vertex, and therefore no edge, is shared.

[3] For more details, see, for example, [4, p. 226]. There, some of the capacities are infinite and some are unit. Changing them all to one unit does not change anything.

Repeating this for all pairs will cost, then, at most $O(n^5)$. Recently, Even and Tarjan [5] have reduced these to $O(n^{2.5})$ and $O(n^{4.5})$, respectively.

Assume now that we are not interested in the connectivity itself, but rather would like to check whether the connectivity is at least k, where k is much smaller than n. It is natural to investigate the question of whether we can find an algorithm which requires less than $O(n^{4.5})$ steps.

Kleitman [6] has shown a method which takes at most $O(k^2n^3)$ steps. In § 2 I shall present a method, an improvement of Kleitman's technique, which takes at most $O(kn^3 + k^3n^2)$ steps. Directed graphs are discussed in § 3.

It is proper to comment here that the case of $k = 1$ is trivially solvable in $O(e)$ steps. The case $k = 2$ was solved in $O(e)$ steps by Hopcroft and Tarjan [7], who proceeded to solve the case $k = 3$ in $O(e)$ steps, too [8]. Their methods are different from the ones described above. They use the powerful technique of depth-first search (which was already known in the 19th century as a maze threading technique. See, for example, Lucas' [9] report of Trémaux's work). I do not believe that their methods will extend for higher k's.

2. The algorithm for undirected graphs. Let G be an undirected graph with n vertices and e edges. Let $L = \{v_1, v_2, \cdots, v_l\}$ be a set of vertices of G and u be a vertex of G not in L. Let k be a positive integer such that $k \leqq l$.

Let us add to G a new vertex a and connect it by an edge to each of the vertices in L. The new graph, $\tilde{G}$, will be called the *augmented graph.*

LEMMA 1. *If in G each vertex v_i $(1 \leqq i \leqq l)$ can be connected to u via k vertex-disjoint paths, then in $\tilde{G}$ there are k vertex-disjoint paths between a and u.*

Proof. Assume not. Then there is a separating set S, $|S| < k$, such that all paths from u to a pass through at least one vertex of S. Consider the set of vertices, U, such that, as for u, all the paths from them to a pass through at least one vertex of S. None of the vertices in L can be in U, since each vertex of L is connected by an edge to a. Thus there exists a vertex v in L which is not in U and not in S. Every path from u to v must pass through at least one vertex of S. Thus there cannot be k vertex-disjoint paths between u and v, a contradiction. Q.E.D.

Assume the set of G's vertices is $\{1, 2, \cdots, n\}$. Let j be the least vertex such that for some $i < j$ there are no k vertex-disjoint paths connecting i and j in G.

LEMMA 2. *Let j be as defined above and $\tilde{G}$ be the augmented graph where $L = \{1, 2, \cdots, j - 1\}$. There are no k vertex-disjoint paths connecting a and j in $\tilde{G}$.*

Proof. Consider a minimum separating set S, such that all paths between i and j pass through at least one vertex of S. It follows that $|S| < k$. Let U be the set of all vertices such that all paths from them to i must pass through at least one vertex of S. Clearly, $j \in U$. If a vertex $j' < j$ is in U, then there are no k vertex-disjoint paths from j' to i, and jth choice was erroneous. Thus j is the least vertex in U, or $L \cap U = \emptyset$. Namely, all paths from j to vertices in L must pass through, or end in, a vertex in S. It follows that in $\tilde{G}$ there are no k vertex-disjoint paths between a and j. Q.E.D.

We are now ready for the algorithm for determining whether the connectivity of G is at least k.

ALGORITHM 1.

1. For every i and j such that $1 \leqq i < j \leqq k$, check whether there are k

vertex-disjoint paths between them. If for some i and j the test fails, then G's connectivity is less than k.

2. For every j, $k + 1 \leqq j \leqq n$, form[4] $\tilde{G}$ and check whether there are k vertex-disjoint paths between a and j. If for some j the test fails, then G's connectivity is less than k.
3. The connectivity of G is at least k.

The proof of validity of this algorithm is as follows: if G's connectivity is at least k, then by Lemma 1, step 2 will detect no failure, and the algorithm will halt with the correct answer. If G's connectivity is less than k, then by Lemma 2, failure will occur, and again the algorithm will halt with the correct answer.

Step 1 of the algorithm requires at most $O(k^3 \cdot e)$ elementary steps, and step 2 requires at most $O(k \cdot n \cdot e)$. Thus the whole algorithm requires at most $O(k^3 \cdot e + k \cdot n \cdot e)$ steps. Since $O(e) \leqq n^2$, the number of steps is bounded by $O(k^3 \cdot n^2 + k \cdot n^3)$. For $k < \sqrt{n}$, $O(k \cdot n^3)$ is an upper bound.

The algorithm is an improvement of Kleitman's algorithm [6], which may require $O(k^2 \cdot n^3)$ steps. The saving is achieved by the addition of the vertex a and checking its connectivity to every vertex j (in $\tilde{G}$) instead of repeating this test for k different vertices.

3. The algorithm for directed graphs.

Let G be a directed graph whose set of vertices is $\{1, 2, \cdots, n\}$ and with e edges. An *(i, j)-separating set*, S, is a set of vertices such that every directed path from i to j passes through at least one vertex in S. The *connectivity* of G is defined as follows:

(i) if the graph is completely connected (namely, $e = n(n - 1)$), then $c = n - 1$,
(ii) if the graph is not completely connected, then c is the least cardinality of a separating set.

Menger's theorem holds in this case, too, and the network flow technique applies. The straightforward technique of checking if there are k vertex-disjoint directed paths between every ordered pair of vertices takes at most $O(kn^4)$ steps.

Let $\vec{G}$ be an augmented graph constructed for j as follows: add a new vertex a to the graph and connect a by an edge to each of the vertices in $L = \{1, 2, \cdots, j - 1\}$. Similarly, $\overleftarrow{G}$ is constructed by adding a new vertex a and edges from each of the vertices in L to a. Assume now that $j > k$.

LEMMA 3. *If in G each $i \in L$ can be connected to j via k vertex-disjoint directed paths, then in $\vec{G}$ there are k vertex-disjoint directed paths from a to j.*

The proof of this Lemma and the following one is analogous to that of Lemma 1.

LEMMA 4. *If, in G, j can be connected to each $i \in L$ via k vertex-disjoint directed paths, then in $\overleftarrow{G}$, there are k vertex-disjoint directed paths from j to a.*

Let j be the least vertex such that for some $i < j$ either there are no k vertex-disjoint directed paths from i to j or there are no k vertex-disjoint directed paths from j to i.

LEMMA 5. *Assume j is as defined and that there are no k vertex-disjoint directed paths from i to j (from j to i). There are no k vertex-disjoint paths from a to j in $\vec{G}$ (from j to a in $\vec{G}$).*

The proof is analogous to that of Lemma 2.

[4] Clearly, $L = \{1, 2, \cdots, j - 1\}$.

ALGORITHM 2.

1. For every i and j such that $1 \leqq i < j \leqq k$, check whether there are k vertex-disjoint directed paths from i to j and also if there are k such paths from j to i. If one of these tests fails, then G's connectivity is less than k.
2. For every j, $k + 1 \leqq j \leqq n$, form $\vec{G}$ and check whether there are k vertex-disjoint directed paths from a to j; also form $\overleftarrow{G}$ and check whether there are k such paths from j to a. If for some j one of these tests fails, then G's connectivity is less than k.
3. The connectivity of G is at least k.

The proof of validity is similar to that of Algorithm 1. Again, step 1 takes at most $O(k^3n^2)$ and step 2, $O(kn^3)$. If $k < \sqrt{n}$, then the whole algorithm takes at most $O(kn^3)$ steps.

Acknowledgment. This work was done while the author was with the Department of Computer Science, Cornell University, during the summer of 1973. It is a revision of Report TR-73-184 of the Department of Computer Science, Cornell University, Ithaca, New York, 1973.

REFERENCES

[1] K. MENGER, *Zur allgemeinen Kurventheorie*, Fund. Math., 10 (1927), pp. 96–115.

[2] G. B. DANTZIG AND D. R. FULKERSON, *On the max-flow min-cut theorem of networks*, Linear Inequalities and Related Systems, Annals of Math. Study, no. 38, Princeton University Press, Princeton, N.J., 1956, pp. 215–221.

[3] L. R. FORD AND D. R. FULKERSON, *Flows in Networks*, Princeton University Press, Princeton, N.J., 1962.

[4] S. EVEN, *Algorithmic Combinatorics*, Macmillan, New York, 1973.

[5] S. EVEN AND R. E. TARJAN, *Network flow and testing graph connectivity*, this Journal, to appear.

[6] D. J. KLEITMAN, *Methods for investigating connectivity of large graphs*, IEEE Trans. Circuit Theory, CT-16 (1969), pp. 232–233.

[7] J. HOPCROFT AND R. TARJAN, *Algorithm* 447; *Efficient algorithms for graph manipulation*, Comm. ACM, 16 (1973), pp. 372–378.

[8] ———, *Dividing a graph into triconnected components*, this Journal, 2 (1973), pp. 135–158.

[9] E. LUCAS, *Récreations Mathématiques*, Paris, 1882.

SIAM J. COMPUT.
Vol. 4, No. 4, December 1975

COMPLEXITY RESULTS FOR MULTIPROCESSOR SCHEDULING UNDER RESOURCE CONSTRAINTS*

M. R. GAREY AND D. S. JOHNSON†

Abstract. We examine the computational complexity of scheduling problems associated with a certain abstract model of a multiprocessing system. The essential elements of the model are a finite number of identical processors, a finite set of tasks to be executed, a partial order constraining the sequence in which tasks may be executed, a finite set of limited resources, and, for each task, the time required for its execution and the amount of each resource which it requires. We focus on the complexity of algorithms for determining a schedule which satisfies the partial order and resource usage constraints and which completes all required processing before a given fixed deadline. For certain special cases, it is possible to give such a scheduling algorithm which runs in low order polynomial time. However, the main results of this paper imply that almost all cases of this scheduling problem, even with only one resource, are NP-complete and hence are as difficult as the notorious traveling salesman problem.

Key words. complexity of algorithms, NP-complete problems, scheduling theory

1. Introduction. In recent years, there has been considerable interest in scheduling problems associated with a certain abstract model of a multiprocessing system (see [6] for a survey). Much effort has been directed toward obtaining efficient algorithms for scheduling a given set of tasks to achieve the least possible finishing time. In a number of important special cases [7], [4], [9], [1], such efficient optimization algorithms have been obtained. In contrast, Ullman [10] has shown that certain other cases belong to the class of NP-complete problems, which is tantamount to proving that they are computationally intractable. In this paper, we consider the augmented multiprocessing model of [5], which allows for the possibility that certain tasks may require the use of various limited resources during their execution, and examine how such constraints affect the computational complexity of the associated scheduling problems.

We first describe the augmented multiprocessing model. Its three main components are *processors*, *resources* and *tasks*. The processors are all identical, each able to execute at most one task at a time, and there are at most a finite number of them. The set $\mathscr{R} = \{R_1, R_2, \cdots, R_r\}$ of resources is also finite and, for each resource R_j, there is a bound B_j which gives the total amount of that resource available at any given time. The tasks form a third finite set $\mathscr{T} = \{T_1, T_2, \cdots, T_m\}$, the elements of which are to be executed by the processors subject to a number of constraints.

First, associated with each task T_i is a positive task time τ_i which is the time required for execution of T_i. If a processor begins executing T_i at time t, it will complete the execution of T_i at time $t + \tau_i$, and until then can execute no other tasks.

Second, $\mathscr{T}$ is partially ordered by a binary relation $\prec$ which must be respected in the execution of $\mathscr{T}$ as follows: if $T_i \prec T_j$, then the execution of T_i must be completed before the execution of T_j can begin. It is frequently convenient to describe the partial order by a directed graph G with node set $\mathscr{T}$ and a directed edge $\langle T_i, T_j \rangle$ if and only if $T_i \prec T_j$.

* Received by the editors July 30, 1974.

† Bell Laboratories, Murray Hill, New Jersey 07974.

Finally, for each resource R_j and task T_i, there is a nonnegative resource requirement $R_j(T_i) \leqq B_j$ which is the amount of resource R_j required by T_i at all times during its execution. The execution of tasks is constrained by the requirement that no subset of tasks can be executed simultaneously if the sum of their requirements for any resource R_j exceeds B_j, the total amount of that resource available.

For the purposes of this paper, we shall assume that all τ_i, B_j and $R_j(T_i)$ are integers. This entails no loss of generality for the type of results we present, since it is equivalent to permitting arbitrary rational values.

A particular *input* for the general scheduling problem consists of the following information: a number n of processors; a set $\mathcal{R} = \{R_1, R_2, \cdots, R_r\}$ of resources, together with a bound B_j for each R_j; a set $\mathcal{T} = \{T_1, T_2, \cdots, T_m\}$ of tasks, together with a partial order $\prec$ on $\mathcal{T}$ and, for each T_i, a corresponding task time τ_i and resource requirements $R_1(T_i)$ through $R_r(T_i)$; a positive integer *deadline* D.

A function $f:\mathcal{T} \to \{0, 1, 2, \cdots, D - 1\}$ is a *valid schedule* for such an input if it obeys the following four conditions:

(i) for each $T_i \in \mathcal{T}$, $f(T_i) + \tau_i \leqq D$;

(ii) for each T_i, $T_j \in \mathcal{T}$, if $T_i \prec T_j$, then $f(T_i) + \tau_i \leqq f(T_j)$;

(iii) for each integer t, $0 \leqq t < D$, the set

$$E_f(t) = \{T_i \in \mathcal{T} : f(T_i) \leqq t < f(T_i) + \tau_i\}$$

satisfies $|E_f(t)| \leqq n$, and

(iv) for each integer t, $0 \leqq t < D$, and each j, $1 \leqq j \leqq r$,

$$\sum_{T_i \in E_f(t)} R_j(T_i) \leqq B_j.$$

The set $E_f(t)$ defined in (iii) is called the set of tasks being executed at time t under schedule f.

Some additional terminology will be useful to describe certain special properties which may be satisfied by inputs. The partial order $\prec$ is said to be a *forest* if whenever $T_i \prec T_k$ and $T_j \prec T_k$, we have either $T_i \prec T_j$ or $T_j \prec T_i$. An input will be said to have the *saturated processor* property if $\sum_{i=1}^m \tau_i = n \cdot D$. Any valid schedule f for an input with this property must have $|E_f(t)| = n$ for each integer t, $0 \leqq t < D$, i.e., no processor can be idle until the deadline D is reached. An input has a *saturated resource* R_j if $\sum_{i=1}^m \tau_i R_j(T_i) = D \cdot B_j$. Any valid schedule f for an input with resource R_j saturated must satisfy $\sum_{T_i \in E_f(t)} R_j(T_i) = B_j$ for each integer t, $0 \leqq t < D$, i.e., the full amount of resource R_j must be in continued use until the deadline D is reached. We shall be dealing mainly with inputs for which all $\tau_i = 1$, in which case the saturated processor property can be written $|\mathcal{T}| = n \cdot D$, and resource R_j will be saturated if $\sum_{T \in \mathcal{T}} R_j(T) = D \cdot B_j$.

In this paper we shall be concerned with algorithms which, given any input I, are capable of answering the question, "Does there exist a valid schedule for I?". Notice that we phrase the question in such a way that the answer is always "yes" or "no". Such phrasing is convenient to use in proving computational complexity results and will not seriously limit the applicability of the results. The related problems of actually generating a valid schedule for a given input or (treating the deadline D as a variable) generating a valid schedule which achieves the minimum possible deadline D have computational complexity within a polynomial of that for the simple existence problem.

In particular, we shall be concerned with algorithms which apply only to certain restricted classes of possible inputs. It will be convenient to denote such a *subproblem* by $\mathrm{MS}[\varphi_1, \varphi_2, \cdots, \varphi_k]$, which represents the subproblem having input domain restricted to only those inputs satisfying all of the constraints φ_1 through φ_k. For instance, $\mathrm{MS}[n \leqq 2$, each $\tau_i = 1]$ has input domain consisting only of those inputs with at most two processors and all task times equal to one. Occasionally we may include a vacuous restriction for emphasis. The subproblem $\mathrm{MS}[n \leqq 2$, each $\tau_i = 1, r$ arbitrary$]$ is identical with the above but may be specified this way when compared to $\mathrm{MS}[n \leqq 2$, each $\tau_i = 1, r = k]$ for some fixed k.

For a variety of subproblems, we will either give an algorithm for the subproblem which operates in time bounded by a polynomial in the length of the input or prove that the subproblem belongs to the class of "NP-complete" problems. There is a significant difference between these two possibilities, since, as a result of work by Cook [2] and Karp [8], it is widely believed that both possibilities cannot hold for the same problem. This belief is motivated by the knowledge that either all or none of the NP-complete problems can be solved with polynomial-time algorithms, along with the fact that this class contains a number of extensively studied problems, such as set covering and the "traveling salesman" problem, for which no such algorithm has ever been found. The reader who is unfamiliar with the terminology related to NP-complete problems, or who would like his belief in their intractability reinforced, is referred to [2] and [8] for a complete discussion. (Note that the term "NP-complete" is synonymous with "polynomial complete".) For the purposes of this paper, we shall use the following informal definitions.

A *problem* $P = (\mathscr{D}, \pi)$ consists of an *input domain* $\mathscr{D}$ and a *property* π. An algorithm for $P = (\mathscr{D}, \pi)$ is an algorithm which for any input $I \in \mathscr{D}$ determines whether or not property π holds for I (briefly, whether or not $\pi(I)$ holds). Observe that we have described the general scheduling problem and its subproblems in this format, all having the same property π = "there exists a valid schedule for input I".

For any two problems $P = (\mathscr{D}, \pi)$ and $P' = (\mathscr{D}', \pi')$, we say that P is *polynomially reducible* to P' if there exists a total function $g:\mathscr{D} \rightarrow \mathscr{D}'$ such that (i) g can be performed in polynomial time by a deterministic Turing machine, and (ii) for any $I \in \mathscr{D}$, $\pi(I)$ holds if and only if $\pi'(g(I))$ holds. If P is polynomially reducible to P', we write $P \propto P'$.

The class of problems NP consists of all problems which can be solved with a polynomial-time-bounded *non*deterministic Turing machine. A problem P is said to be NP-*complete* if both $P \in \mathrm{NP}$ and, for every $P' \in \mathrm{NP}$, $P' \propto P$. An immediate consequence of the transitivity of the reducibility relation $\propto$ is that, in order to prove $P \in \mathrm{NP}$ is NP-complete, one need only exhibit a known NP-complete problem P' such that $P' \propto P$. This is the method that will be used in our proofs, where we leave to the reader the trivial observation that $P \in \mathrm{NP}$ and the straightforward verification that the mapping g can be executed in polynomial time.

We now briefly outline the contents of the paper. In § 2 we examine subproblems with $n = 2$ processors and show that MS[$n = 2$, $r = 1$, $\prec$ a forest, each $\tau_i = 1$] is NP-complete, while a polynomial time algorithm can be given for MS[$n = 2$, r arbitrary, $\prec$ empty, each $\tau_i = 1$]. In § 3, we examine subproblems with $n \geqq 3$ and derive, by a series of reductions, the NP-completeness of MS[$n = 3$, $r = 1$, $\prec$ empty, each $\tau_i = 1$]. In the final section, we discuss a number of different senses in which these results are "best possible" and state some additional results and open problems.

2. Two-processor subproblems. In this section we discuss special cases of the general scheduling problem for which the number of processors is restricted to $n = 2$. Quite a bit is known about such subproblems when the set of resources is required to be empty. For example, the following two subproblems are NP-complete [10]:

$$\text{MS}[n = 2, r = 0, \prec \text{ empty}, \tau_i \text{ arbitrary integers}];$$

$$\text{MS}[n = 2, r = 0, \prec \text{ arbitrary, each } \tau_i \in \{1, 2\}].$$

On the other hand, [4], [9] and [1] give polynomial time algorithms for MS[$n = 2$, $r = 0$, $\prec$ arbitrary, each $\tau_i = 1$]. We shall focus on subproblems for which $r > 0$.

We first observe that a polynomial-time scheduling algorithm can be given for the case MS[$n = 2$, r arbitrary, $\prec$ empty, each $\tau_i = 1$]. Given any input for this subproblem, one can construct an m-node graph G, having each node labeled by a distinct task, with an edge joining T_i to T_j if and only if

$$R_k(T_i) + R_k(T_j) \leqq B_k$$

for all k, $1 \leqq k \leqq r$. Thus tasks T_i and T_j can be executed at the same time if and only if there is an edge joining the corresponding nodes. One then applies the maximal matching algorithm of Edmonds [3] to obtain a maximal cardinality set E of edges from G such that no two edges share a common endpoint. It is not difficult to see that a valid schedule exists for this input if and only if $D \geqq m - |E|$. Since we can construct G in time proportional to $r \cdot m^2 \cdot \log(\max_j B_j)$ and since Edmonds' matching algorithm requires only time polynomial in the number m of graph nodes, the entire algorithm just described runs in polynomial time.

We now show, however, that if we widen the class of inputs slightly by permitting the partial order $\prec$ to be nonempty but no more complex than a forest, then the resulting subproblem is NP-complete, even if restricted to $r = 1$. This problem will be shown to be NP-complete by reducing the following NP-complete problem to it.

NODE COVER [8].

Input: Graph $G = (N, A)$, positive integer k.

Property: There exists a node cover of size k for G, i.e., a subset S of the set N of nodes, with $|S| = k$, such that S contains at least one endpoint from every edge in A.

THEOREM 2.1. NODE COVER $\propto$ MS[$n = 2$, $r = 1$, $\prec$ a forest, each $\tau_i = 1$].

Proof. Given a graph $G = (N, A)$ and a positive integer k, we construct a corresponding scheduling input I as follows: Let $N = \{N_1, N_2, \cdots, N_p\}$ and

$A = \{A_1, A_2, \cdots, A_q\}$, where $p = |N|$ and $q = |A|$. The input I is specified by

$$n = 2;$$

$$\mathscr{T} = \{T_i : 1 \leqq i \leqq p + 2q\} \cup \{V_i : 1 \leqq i \leqq p\} \cup \{E_i, \bar{E}_i : 1 \leqq i \leqq q\};$$

all task times equal 1; partial order is defined by

$$T_i \prec T_{i+1}, \qquad 1 \leqq i < p + 2q;$$

$$V_i \prec E_l \quad \text{whenever } A_l = \{N_i, N_j\} \in A \text{ and } j > i;$$

$$V_i \prec \bar{E}_l \quad \text{whenever } A_l = \{N_i, N_j\} \in A \text{ and } j < i;$$

single resource R_1 with bound $B_1 = 2q$;

$$R_1(T_i) = q, \qquad 1 \leqq i \leqq k \quad \text{or} \quad q + k < i \leqq p + q;$$

$$R_1(T_{k+i}) = R_1(T_{p+q+i}) = q + i, \qquad 1 \leqq i \leqq q;$$

$$R_1(V_i) = q, \qquad 1 \leqq i \leqq p;$$

$$R_1(E_i) = R_1(\bar{E}_i) = q - i, \qquad 1 \leqq i \leqq q;$$

deadline $D = p + 2q$.

It is easy to verify that input I is in the required domain. Furthermore, observe that it satisfies the saturated processor property and that R_1 is a saturated resource.

Intuitively, the V_i tasks represent the nodes of G, and the E_l and $\bar{E}_l$ tasks represent the arcs, with the pair E_l and $\bar{E}_l$ both representing arc A_l, one for each endpoint. The T_i tasks form a chain designed to be a backbone for the entire schedule, since in order to achieve the deadline these must be executed in sequence, with each task T_i executed starting at time $i - 1$. Thus during each time unit there will be a T_i task and one other task. Because of the saturated resource property, the resource requirement of each task T_i puts strict limits on the possibilities for the concurrently executed task. Thus a pattern is imposed on any valid schedule. The first k time units must all be filled with node tasks, and the next q time units must be filled with arc tasks, one from each pair $\{E_l, \bar{E}_l\}$. Because of the precedence constraints, neither E_l nor $\bar{E}_l$ can occur in this section unless a node task representing one of the endpoints of arc A_l is present in the first section. Thus we can have a valid schedule if and only if the k node tasks in the first section form the desired node cover.

With this intuition, we now prove that G has a node cover S with $|S| = k$ if and only if there exists a valid schedule for I.

First, suppose $S = \{s_1, s_2, \cdots, s_k\}$ is a node cover for G, and let $U = N - S = \{u_1, u_2, \cdots, u_{p-k}\}$. Then consider the function f, defined below:

$$f(T_i) = i - 1, \qquad 1 \leqq i \leqq p + 2q;$$

$$f(V_j) = i - 1 \quad \text{if } N_j = s_i \in S;$$

$$f(V_j) = q + k + i - 1 \quad \text{if } N_j = u_i \in U;$$

$$f(E_l) = k + l - 1 \quad \text{and} \quad f(\bar{E}_l) = p + q + l - 1$$

$$\text{if } A_l = \{N_i, N_j\}, i < j, \text{ and } N_i = s_h \in S;$$

$$f(E_l) = p + q + l - 1 \quad \text{and} \quad f(\bar{E}_l) = k + l - 1$$
$$\text{if } A_l = \{N_i, N_j\},\ i < j, \text{ and } N_i = u_h \in U.$$

One can easily check that f is a valid schedule for the input I.

Conversely, suppose we are given a valid schedule f for I. Observe first that, due to the partial order constraints and deadline $D = p + 2q$, we must have $f(T_i) = i - 1$ for $1 \leqq i \leqq p + 2q$. Thus the tasks of this type form a backbone for the schedule, and, due to the saturated resource R_1, each of the remaining tasks can be executed only at certain times. To be specific, we must have

$$\text{(2.1a)} \qquad \{f(V_j): 1 \leqq j \leqq p\} = \{t: 0 \leqq t \leqq k\} \cup \{t: q + k \leqq t < q + p\}$$

and, for $1 \leqq l \leqq q$,

$$\text{(2.1b)} \qquad \{f(E_l), f(\bar{E}_l)\} = \{k + l - 1, p + q + l - 1\}.$$

We claim that $S = \{N_i: f(V_i) < k\}$ forms a node cover of size k for G. By (2.1a) and the saturated processor property, we immediately have $|S| = k$. Consider any edge $A_l = \{N_i, N_j\} \in A$, where $i < j$. By (2.1b), either $f(E_l) = k + l - 1$ or $f(\bar{E}_l) = k + l - 1$. In the first case, since $V_i \prec E_l$ and $k + l - 1 < q + k$, we must have $f(V_i) < k$ and hence $N_i \in S$. In the second case, since $V_j \prec \bar{E}_l$, we similarly obtain $N_j \in S$. Thus in either case, we find that at least one endpoint of A_l belongs to S. Since A_l was an arbitrary edge from A, S must be a node cover.

Hence G has a node cover of size k if and only if the associated scheduling input I has a valid schedule, and the theorem follows. $\Box$

3. Three-processor subproblems. In this section we turn to special cases of the general scheduling problem for which the number n of processors is permitted to be larger than 2. Again, subproblems for which the set of resources is required to be empty have been studied previously. In [7] a polynomial-time algorithm is given for MS[n arbitrary, $r = 0$, $\prec$ a forest, each $\tau_i = 1$]. In contrast, Ullman [10] has shown that MS[n arbitrary, $r = 0$, $\prec$ arbitrary, each $\tau_i = 1$] is NP-complete. However, it is not known whether there is any fixed k for which MS[$n \leqq k$, $r = 0$, $\prec$ arbitrary, each $\tau_i = 1$] is NP-complete.

The principal result of this section is that if we substitute a single resource for the arbitrary partial order in the last problem above, requiring $\prec$ to be empty, it then is NP-complete for k as small as 3. Specifically, we show that MS[$n = 3$, $r = 1$, $\prec$ empty, each $\tau_i = 1$] is NP-complete.

We arrive at this result via a series of lemmas about scheduling problems obeying the further restriction that all inputs have the saturated processor property and all their resources saturated. For convenience, we use the abbreviation P[k, j] for MS[$n = k$, $r = j$, $\prec$ empty, each $\tau_i = 1$, saturated processors, all resources saturated]. Since the domain for P[3, 1] is a subset of the domain for MS[$n = 3$, $r = 1$, $\prec$ empty, each $\tau_i = 1$], it will suffice to show that P[3, 1] is NP-complete.

Our first lemma is as follows.

LEMMA 3.1. *For all integers* $k \geqq 1$, $j \geqq 2$, P[k, j] $\propto$ P[k, 1].

Proof. We actually show P[k, j] $\propto$ P[$k, j - 1$]. The result then follows by induction from the transitivity of $\propto$. Let I be an input for P[k, j], with tasks T_i,

$1 \leqq i \leqq m$, resources R_l, $1 \leqq l \leqq j$, bounds B_l, $1 \leqq l \leqq j$, and deadline D. (We omit reference to those items in the input which must be the same for all inputs to the problem under consideration, such as $\prec$ empty, each $\tau_i = 1$, and $n = k$.) The corresponding input I' for P$[k, j-1]$ will have

$$\mathscr{T}' = \{T'_i : 1 \leqq i \leqq m\};$$
$$\mathscr{R}' = \{R'_l : 1 \leqq l \leqq j-1\};$$
$$B'_l = B_l, \qquad 2 \leqq l \leqq j-1;$$
$$B'_1 = B_1 + B_j(kB_1 + 1);$$
$$R'_l(T'_i) = R_l(T_i), \qquad 2 \leqq l \leqq j-1, \quad 1 \leqq i \leqq m;$$
$$R'_1(T'_i) = R_1(T_i) + (k \cdot B_1 + 1) \cdot R_j(T_i), \qquad 1 \leqq i \leqq m;$$

and deadline $D' = D$.

Intuitively, this merely encodes resources R_1 and R_j into the single resource R'_1. The multiplier $(kB_1 + 1)$ on R_j used in this encoding is chosen to be large enough so that, for any k or fewer tasks which might be executed simultaneously, their total usage of resource R'_1 uniquely determines the corresponding usage of both R_1 and R_j. This causes any valid schedule for I' to be, in effect, a simulation of a valid schedule for I.

Observe first that since I has the saturated processor property, $|\mathscr{T}'| = |\mathscr{T}|$ and $D' = D$, it follows that I' also has the saturated processor property. Similarly, we can see that R'_2 through R'_{j-1} are saturated in I' because the corresponding resources are saturated in I. Finally observe that for R'_1

$$\sum_{i=1}^{m} R'_1(T'_i) = \sum_{i=1}^{m} (R_1(T_i) + (k \cdot B_1 + 1) \cdot R_j(T_i))$$
$$= D \cdot B_1 + (k \cdot B_1 + 1) \cdot D \cdot B_j = D' \cdot B'_1$$

since R_1 and R_j are saturated in I. Thus R'_1 is saturated in I' and I' is in the input domain for P$[k, j-1]$.

To complete the proof, we must show that I' has a valid schedule if and only if I does. Suppose f is such a schedule for I. Define f' by $f'(T'_i) = f(T_i)$. Clearly f' has the proper domain and range and obeys all requirements of the definition of valid schedule, except possibly condition (iv) for R'_1, since f is a valid schedule for I. Since R_1 and R_j are saturated for I, we have for all t, $0 \leqq t \leqq D - 1$,

$$\sum_{T'_i \in E_{f'}(t)} R'_1(T'_i) = \sum_{T_i \in E_f(t)} (R_1(T_i) + (k \cdot B_1 + 1) \cdot R_j(T_i))$$
$$= B_1 + (k \cdot B_1 + 1) \cdot B_j = B'_1,$$

where we recall that $E_f(t) = \{T \in \mathscr{T} : f(T) = t\}$. Thus (iv) is satisfied for R'_1 also and f' is a valid schedule for I'.

Conversely, suppose f' is a valid schedule for I' and define f by $f(T_i) = f'(T'_i)$. The only way f could fail to be a valid schedule for I would be for resource constraint (iv) to be violated for R_1 or R_j. We show this to be impossible.

Since R'_1 is saturated for I' and f' is a valid schedule, we know that for all t, $0 \leqq t \leqq D - 1$,

$$\sum_{T_i \in E_f(t)} (R_1(T_i) + (k \cdot B_1 + 1) \cdot R_j(T_i)) = B_1 + (k \cdot B_1 + 1) \cdot B_j.$$

If for any t we had $\sum_{T_i \in E_f(t)} R_j(T_i) > B_j$, then since B_j and all the $R_j(T_i)$ are integers, we would have

$$\sum_{T_i \in E_f(t)} R_1(T_i) \leqq B_1 + (k \cdot B_1 + 1) \cdot B_j - (k \cdot B_1 + 1) \cdot (B_j + 1)$$
$$\leqq B_1 - k \cdot B_1 - 1 < 0,$$

which is impossible since all resource requirements are nonnegative. Similarly, if $\sum_{T_i \in E_f(t)} R_j(T_i) < B_j$, we would have

$$\sum_{T_i \in E_f(t)} R_1(T_i) \geqq B_1 + (k \cdot B_1 + 1) \cdot B_j - (k \cdot B_1 + 1) \cdot (B_j - 1)$$
$$\geqq B_1 + k \cdot B_1 + 1 > k \cdot B_1,$$

which is impossible since each $R_1(T_i) \leqq B_1$ and $|E_f(t)| = k$. Thus we must have $\sum_{T_i \in E_f(t)} R_j(T_i) = B_j$ for all t, $0 \leqq t \leqq D - 1$, which in turn implies that $\sum_{T_i \in E_f(t)} R_1(T_i) = B_1$ for all required t. That is, neither resource constraint can be violated, and f is a valid schedule.

Thus I' has a valid schedule if and only if I does, and the desired reduction has been demonstrated. □

LEMMA 3.2. *For any integer* $k \geqq 1$, P$[k, 1] \propto$ P$[k + 1, 1]$.

Proof. Suppose I is an input for P$[k, 1]$ with tasks T_i, $1 \leqq i \leqq m$, resource R_1 with bound B_1, and deadline D. The corresponding input I' for P$[k + 1, 1]$ will have

$$\mathscr{T}' = \{T'_i : 1 \leqq i \leqq m\} \cup \{S_i : 1 \leqq i \leqq D\};$$
$$\mathscr{R}' = \{R'_1\}; \qquad B'_1 = 3 \cdot B_1;$$
$$R'_1(T'_i) = R_1(T_i), \qquad 1 \leqq i \leqq m;$$
$$R'_1(S_i) = 2 \cdot B_1, \qquad 1 \leqq i \leqq D;$$

and deadline $D' = D$.

The basic idea here is that the resource requirements of the S_i-tasks insure that no two of them can be executed simultaneously. Since there are D such tasks, any valid schedule must execute exactly one of them during each time unit, which has the effect of using up the extra processor.

The reader may verify easily that I' has the saturated processor property and saturated resource R'_1 and hence is in the input domain for P$[k + 1, 1]$. It is also easy to see that if f is a valid schedule for I, then f' defined by

$$f'(T'_i) = f(T_i), \qquad 1 \leqq i \leqq m;$$
$$f'(S_i) = i - 1, \qquad 1 \leqq i \leqq D,$$

is a valid schedule for I'.

Conversely, suppose that f' is any valid schedule for I' and define $f:\mathcal{T} \rightarrow \{0, 1, \cdots, D-1\}$ by $f(T_i) = f'(T'_i)$. We shall show that f is a valid schedule for I.

Clearly f has the proper domain and range and obeys properties (i) and (ii) of the definition of valid schedule. For (iii) and (iv), we first observe that each $E_{f'}(t)$, $0 \leqq t \leqq D-1$, contains exactly one S_i task since $R'_1(S_i) > \frac{1}{2}B'_1$ and there are D such tasks. Thus

$$|E_f(t)| = |E_{f'}(t) \cap \{T'_i : 1 \leqq i \leqq m\}| = k + 1 - 1 = k,$$

and property (iii) is satisfied. Furthermore, since R'_1 is saturated in I', we have for each t, $0 \leqq t \leqq D-1$,

$$\sum_{T_i \in E_f(t)} R_1(T_i) = B'_1 - 2B_1 = B_1,$$

so f also satisfies (iv) and hence is a valid schedule for I.

Thus I' has a valid schedule if and only if I does and the desired reduction has been demonstrated. □

LEMMA 3.3. *For each integer* $k \geqq 2$, $\mathrm{P}[2k, 1] \propto \mathrm{P}[k+1, 1]$.

Proof. We actually show that $\mathrm{P}[2k, 1] \propto \mathrm{P}[k+1, 2]$, and the desired result follows from that by an application of Lemma 3.1. Let I be an input for $\mathrm{P}[2k, 1]$ with tasks T_i, $1 \leqq i \leqq m$, resource R_1 with bound B_1, and deadline D. Our construction of the corresponding input I' for $\mathrm{P}[k+1, 2]$ is somewhat more complicated than the preceding constructions. Define

$$\mathscr{A} = \left\{ A \subseteq \{1, 2, \cdots, m\} : |A| = k \text{ and } \sum_{j \in A} R_1(T_j) \leqq B_1 \right\},$$

with $M = |\mathscr{A}|$. (Observe that $M \leqq m^k$.) In addition, let

$$T = \{T'_i : 1 \leqq i \leqq m\};$$
$$F = \{F_{i,j} : 1 \leqq i \leqq M - D, 1 \leqq j \leqq k-1\};$$
$$U = \{U_A : A \in \mathscr{A}\}; \qquad V = \{V_A : A \in \mathscr{A}\}.$$

The input I' for $\mathrm{P}[k+1, 2]$ corresponding to I has

$$\mathscr{T}' = T \cup F \cup U \cup V;$$
$$\mathscr{R}' = \{R'_1, R'_2\};$$
$$B'_1 = B_1; \qquad B'_2 = 2k;$$
$$R'_1(T'_i) = R_1(T_i) \quad \text{and} \quad R'_2(T'_i) = 1 \qquad \text{for } T'_i \in T;$$
$$R'_1(F_{i,j}) = R'_2(F_{i,j}) = 0 \qquad \text{for } F_{i,j} \in F;$$
$$R'_1(U_A) = \sum_{j \in A} R_1(T_j) \quad \text{and} \quad R'_1(V_A) = B_1 - R'_1(U_A) \qquad \text{for } A \in \mathscr{A};$$
$$R'_2(U_A) = R'_2(V_A) = k \qquad \text{for } A \in \mathscr{A};$$

and deadline $D' = M + D$.

Before embarking on the formal proof, a few intuitive comments concerning the correspondence between valid schedules for I and I' may be helpful. We want single time units in the $2k$ processor case to correspond to *pairs* of time units in the $k + 1$ processor case, with each set of $2k$ concurrently executed tasks in the one case partitioned into two sets of k concurrently executed tasks in the other. We need the $(k + 1)$st processor and the U_A and V_A tasks to insure that this happens in the right way. The resource requirements for U_A and V_A are designed so that essentially only one of two things can happen: either U_A and V_A are executed together, with "filler" tasks from F occupying the remaining $k - 1$ processors, or else U_A and V_A are executed at different times, each together with a set of k T_i' tasks, such that the two corresponding sets of T_i tasks could all be executed concurrently in the $2k$ processor case.

We first observe that $m = 2kD$ since I has the saturated processor property, which implies

$$|\mathscr{T}'| = 2kD + (M - D)(k - 1) + 2M = (M + D)(k + 1).$$

Thus I' has the saturated processor property. (The tasks in F are essentially "filler" to make this the case.) The reader may verify similarly that resources R_1' and R_2' are saturated, showing that I' indeed belongs to the input domain for P$[k + 1, 2]$.

Suppose now that f is a valid schedule for I. Since I has the saturated processor property, we have $|E_f(t)| = 2k$ for each t, $0 \leqq t \leqq D - 1$. Define

$$X(t) = \{i \in E_f(t) : |\{1, 2, \cdots, i\} \cap E_f(t)| \leqq k\}$$

and

$$Y(t) = E_f(t) - X(t)$$

for $0 \leqq t \leqq D - 1$. Then $|X(t)| = |Y(t)| = k$, $X(t) \cup Y(t) = E_f(t)$, and both $X(t)$ and $Y(t)$ belong to $\mathscr{A}$. Let $S = \mathscr{A} - \{X(t) : 0 \leqq t \leqq D - 1\}$ and label the elements of S as $A_1, A_2, \cdots, A_{M-D}$. We now are prepared to define the schedule $f' : \mathscr{T} \to \{0, 1, 2, \cdots, M + D - 1\}$. For each t, $0 \leqq t \leqq D - 1$, set

$$f'(V_{X(t)}) = f'(T_i') = 2t \quad \text{for } i \in X(t);$$

$$f'(U_{X(t)}) = f'(T_i') = 2t + 1 \quad \text{for } i \in Y(t).$$

To complete the definition of f', set

$$f'(U_{A_i}) = f'(V_{A_i}) = f'(F_{i,j}) = 2D - 1 + i \quad \text{for } 1 \leqq j \leqq k - 1,$$

for each $A_i \in S$, $1 \leqq i \leqq M - D$. It is not difficult to check that f' satisfies all the properties required for a valid schedule.

Now suppose f' is a valid schedule for I'. Define $W = U \cup V$, and for each integer $i \geqq 0$ define

$$H_i(f') = \{t : 0 \leqq t \leqq M + D - 1 \text{ and } |E_{f'}(t) \cap W| = i\}.$$

Thus $H_i(f')$ is the set of integer times at which exactly i tasks from W are being executed under f'. Since $B_2' = 2k$ and each $w \in W$ has $R_2'(w) = k$, we know that $H_i(f')$ must be empty for $i > 2$. Furthermore, since each task not in W requires at most 1 unit of R_2' and R_2' is saturated in I', we must also have $H_0(f') = \varphi$.

Thus $|H_1(f')| + |H_2(f')| = M + D$, and, since $|W| = 2M$, it follows that $|H_1(f')| = 2D$ and $|H_2(f')| = M - D$.

Also, for any $t \in H_2(f')$, we know that $E_{f'}(t) \cap T = \varphi$ because the two tasks from W already use up the full amount of R'_2. Thus each T'_i satisfies $f'(T'_i) \in H_1(f')$ and, by the saturated processor property, each $F_{i,j}$ must satisfy $f'(F_{i,j}) \in H_2(f')$.

Define

$$\text{MATCH}(f') = \{t \in H_2(f') : \exists\, A \in \mathscr{A} \text{ such that } \{U_A, V_A\} \subseteq E_{f'}(t)\}.$$

Without loss of generality, we may assume that f' satisfies

$$|\text{MATCH}(f')| \geqq |\text{MATCH}(f'')|$$

for every valid schedule f'' for I'. We claim that this implies that $\text{MATCH}(f') = H_2(f')$. For, suppose there were a $t_0 \in H_2(f')$ such that $t_0 \notin \text{MATCH}(f')$. Let W_1 and W_2 be members of W belonging to $E_{f'}(t_0)$. Let W_3 be such that $\{W_2, W_3\} = \{U_A, V_A\}$ for some $A \in \mathscr{A}$, and let $t' = f'(W_3)$. Notice that this and the saturated resource property imply that

$$R'_1(W_3) = B_1 - R'_1(W_2) = R'_1(W_1).$$

Then the function f'', which is identical to f' except that $f''(W_3) = t_0$ and $f''(W_1) = t'$, is a valid schedule for I' with

$$|\text{MATCH}(f'')| > |\text{MATCH}(f')|,$$

contradicting the choice of f'. Therefore we may assume $\text{MATCH}(f') = H_2(f')$.

Since $\text{MATCH}(f') = H_2(f')$, it immediately follows that for every $A \in \mathscr{A}$, $f'(U_A) \in H_1(f')$ if and only if $f'(V_A) \in H_1(f')$. Accordingly, let

$$S = \{A \in \mathscr{A} : f'(U_A) \in H_1(f')\},$$

and sequence the members of S as $S_1, S_2, \cdots, S_D$. We now can define a valid schedule f for I by

$$E_f(t-1) = \{T_i \in \mathscr{T} : f'(T'_i) \in \{f'(U_{S_t}), f'(V_{S_t})\}\}$$

for each t, $1 \leqq t \leqq D$.

The function f is a total function from $\mathscr{T}$ to $\{0, 1, 2, \cdots, D - 1\}$ since $f'(T'_i) \in H_1(f')$ for all $T'_i \in T$. Thus f trivially satisfies properties (i) and (ii) required of a valid schedule. Moreover, $|E_f(t)| = 2k$ for $0 \leqq t \leqq D - 1$ since, for each $t \in H_1(f')$, $|E_{f'}(t) \cap T| = k$. Thus f also obeys property (iii). Finally, observe that for each t, $0 \leqq t \leqq D - 1$,

$$\begin{aligned}\sum_{T_i \in E_f(t)} R_1(T_i) &= (B'_1 - R'_1(U_{S_{t+1}})) + (B'_1 - R'_1(V_{S_{t+1}})) \\ &= 2B'_1 - B'_1 = B'_1 = B_1,\end{aligned}$$

so property (iv) also is satisfied, and f is indeed a valid schedule for I.

Thus I' has a valid schedule if and only if I has a valid schedule, and the required reduction has been demonstrated. □

The final lemma of this section will show that P[5, 8] is NP-complete, enabling us to apply the preceding lemmas to obtain our main result. First, let us define the NP-complete three-dimensional matching problem.

THREE-DIMENSIONAL MATCHING (3DM) [8].
Input: Finite sets T and $S \subseteq T \times T \times T$.
Property: S contains a complete matching, i.e., a subset $S' \subseteq S$ with $|S'| = |T|$ such that for any two members $\langle x_1, y_1, z_1\rangle$ and $\langle x_2, y_2, z_2\rangle$ of S', $x_1 \neq x_2$, $y_1 \neq y_2$, and $z_1 \neq z_2$.

LEMMA 3.4. 3DM $\propto$ P[5, 8].

Proof. Let T and $S \subseteq T \times T \times T$ be given. We may assume without loss of generality that $T = \{1, 2, \cdots, N\}$ where $N = |T|$. For integers i, k, $1 \leqq i \leqq N$, $1 \leqq k \leqq 3$, let $m_k(i)$ be the number of ordered triplets in S having their kth component equal to i. We may assume that each $m_k(i) \geqq 1$, since if any $m_k(i) = 0$, we would know immediately that S contains no complete matching. The corresponding input I to P[5, 8] is the following:

$$n = 5;$$

$$\begin{aligned}\mathscr{T} = {} & \{S[i,j,k] : \langle i,j,k\rangle \in S\} \cup \{X_0[i], Y_0[i], Z_0(i); 1 \leqq i \leqq N\} \\ & \cup \{X[i;l] : 1 \leqq i \leqq N, 1 \leqq l < m_1(i)\} \cup \{Y[i;l] : 1 \leqq i \leqq N, 1 \leqq l < m_2(i)\} \\ & \cup \{Z[i;l] : 1 \leqq i \leqq N, 1 \leqq l < m_3(i)\} \cup \{F_i : 1 \leqq i \leqq N\} \\ & \cup \{G_i : 1 \leqq i \leqq |S| - N\};\end{aligned}$$

$$\mathscr{R} = \{R_j : 1 \leqq j \leqq 8\},$$

with resource bounds and task resource requirements as given in Table 1, and with deadline $D = |S|$.

Though the input may appear rather complicated, the basic idea behind it is quite simple. The resource requirements and bounds insure that during each time unit of any valid schedule, the five tasks being executed will consist of one S-type task, one X_0- or X-type task, one Y_0- or Y-type task, one Z_0- or Z-type task, and one F- or G-type task. The S-type tasks each represent an ordered triple from S. The N F-type tasks select the triples which form the matching, i.e., those $\langle i, j, k\rangle$ for which $S[i, j, k]$ is executed concurrently with some F-type task. Finally, the X_0-type tasks, Y_0-type tasks and Z_0-type tasks are used to insure that the N selected S-type tasks actually represent a matching. This is done by specifying the resource requirements so that, since all resources are saturated, the only tasks which can be executed simultaneously with F_l and $S[i, j, k]$ are $X_0[i]$, $Y_0[j]$, and $Z_0[k]$. Since there is exactly one $X_0[i]$, $Y_0[i]$, and $Z_0[i]$ for each i, $1 \leqq i \leqq N$, the saturated processor property will guarantee that in any valid schedule the selected $S[i, j, k]$ tasks represent disjoint triples and thus give the required matching. We now give the formal proof.

Observe first that

$$\begin{aligned}|\mathscr{T}| &= |S| + \sum_{k=1}^{3} \sum_{i=1}^{N} m_k(i) + N + (|S| - N) \\ &= |S| + 3 \cdot |S| + |S| = 5 \cdot |S| = n \cdot D,\end{aligned}$$

so that I has the saturated processor property. The reader may verify similarly that all eight resources are saturated, so I belongs to the input domain for P[5, 8].

TABLE 1
Resource requirements and bounds for Lemma 3.4

Task W	$R_1(W)$	$R_2(W)$	$R_3(W)$	$R_4(W)$	$R_5(W)$	$R_6(W)$	$R_7(W)$	$R_8(W)$
$S[i, j, k]$	$N - i$	$N - j$	$N - k$	1	0	0	0	0
$X_0[i]$	i	0	0	0	1	0	0	0
$X[i; l]$	i	0	0	0	1	0	0	1
$Y_0[j]$	0	j	0	0	0	1	0	0
$Y[j; l]$	0	j	0	0	0	1	0	1
$Z_0[k]$	0	0	k	0	0	0	1	0
$Z[k; l]$	0	0	k	0	0	0	1	1
F_i	0	0	0	0	0	0	0	3
G_i	0	0	0	0	0	0	0	0
Bound	N	N	N	1	1	1	1	3

Suppose S contains a complete matching S'. We must construct a valid schedule f for I. Order the elements of S as $S_1, S_2, \cdots, S_{|S|}$ so that $S' = \{S_l: 1 \leqq l \leqq N\}$. For each $S_l \in S$, let the components of S_l be denoted by $\langle i(l), j(l), k(l)\rangle$. We specify f by giving the sets $E_f(t)$, $0 \leqq t < D = |S|$. For $1 \leqq t \leqq N$, let

$$E_f(t - 1) = \{S[i(t), j(t), k(t)], X_0[i(t)], Y_0[j(t)], Z_0[k(t)], F_t\}.$$

For each t, $N + 1 \leqq t \leqq |S|$, define

$$L_1(t) = |\{t' : N + 1 \leqq t' \leqq t \text{ such that } i(t') = i(t)\}|,$$

$$L_2(t) = |\{t' : N + 1 \leqq t' \leqq t \text{ such that } j(t') = j(t)\}|,$$

$$L_3(t) = |\{t' : N + 1 \leqq t' \leqq t \text{ such that } k(t') = k(t)\}|.$$

Then for each t, $N + 1 \leqq t \leqq |S|$, let

$$E_f(t - 1) = \{S[i(t), j(t), k(t)], X[i(t); L_1(t)], Y[j(t); L_2(t)], Z[k(t); L_3(t)], G_{t-N}\}.$$

We leave to the reader the straightforward verification that f is a valid schedule for I.

Conversely, suppose we have a valid schedule f for I. Since there are $|S|$ tasks of the form $S[i, j, k]$, the constraints on resource R_4 and the fact that $D = |S|$ imply that there is exactly one $S[i, j, k]$ in each $E_f(t)$, $0 \leqq t \leqq D - 1$. Similarly, if we let

$$X = \{X_0[i], X[i; l] : 1 \leqq i \leqq N, 1 \leqq l < m_1(i)\},$$

$$Y = \{Y_0[i], Y[i; l] : 1 \leqq i \leqq N, 1 \leqq l < m_2(i)\},$$

$$Z = \{Z_0[i], Z[i, l] : 1 \leqq i \leqq N, 1 \leqq l < m_3(i)\},$$

then, due to resources R_5, R_6, and R_7, each $E_f(t)$ contains exactly one element of X, one element of Y, and one element of Z. Since I has the saturated processor property, we can conclude that each $E_f(t)$ also must contain exactly one element of

$$\{F_i: 1 \leqq i \leqq N\} \cup \{G_i: 1 \leqq i \leqq |S| - N\}.$$

Let us define

$$H = \{t : 0 \leqq t < D\} \text{ such that } E_f(t) \cap \{F_i : 1 \leqq i \leqq N\} \neq \varphi\}.$$

Our previous arguments imply that $|H| = N$. Because of resource R_8, we know that for each $t \in H$, $E_f(t)$ contains no tasks of the form $X[i; l]$, $Y[i; l]$ or $Z[i; l]$, and hence $E_f(t)$ has the form

$$\{S[i, j, k], X_0[i'], Y_0[j'], Z_0[k'], F_l\}.$$

Furthermore, by saturated resources R_1, R_2 and R_3, it must be the case that $i = i'$, $j = j'$ and $k = k'$. It follows that

$$S' = \{\langle i, j, k\rangle \in S : f(S[i, j, k]) \in H\}$$

is a complete matching for S.

Thus I has a valid schedule if and only if S has a complete matching, and the desired reduction has been demonstrated. □

We now have our main result.

THEOREM 3.5. *MS*$[n = 3, r = 1, \prec$ *empty, each* $\tau_i = 1]$ *is* NP-*complete.*

Proof. By starting with Lemma 3.4 and proceeding via an application of Lemma 3.1, an application of Lemma 3.2, and two applications of Lemma 3.3, we obtain

$$\text{3DM} \propto \text{P}[5, 8] \propto \text{P}[5, 1] \propto \text{P}[6, 1] \propto \text{P}[4, 1] \propto \text{P}[3, 1].$$

Since 3DM is NP-complete, we conclude that P[3, 1] is NP-complete. The theorem follows since the input domain for P[3, 1] is included in the input domain for MS$[n = 3, r = 1, \prec$ empty, each $\tau_i = 1]$. □

4. Concluding remarks. Though the two main scheduling problems which we have proved to be NP-complete may seem rather specialized, the results immediately imply that many other scheduling problems are NP-complete. In particular, for arbitrary integers $k \geqq 2$ and $j \geqq 1$, each of the following problems, and any problem whose input domain contains the input domain for such a problem, is NP-complete:

1. MS$[n = k, r = j, \prec$ a forest, each $\tau_i = 1]$;
2. MS$[n = k + 1, r = j, \prec$ empty, each $\tau_i = 1]$.

Furthermore, our results are best possible in the sense that further natural restrictions on the input domains lead to problems which can be solved with polynomial time algorithms. That is, if the input domains for problems 1 and 2 above are restricted further by choosing $k < 2$, $j < 1$, or (in problem 1) requiring $\prec$ empty, then the resulting problems can be solved in polynomial time by algorithms which have been mentioned previously.

Our results have thus determined the "boundary" between NP-completeness and polynomial time solvability with respect to the number of processors, number of resources, and type of partial order. Implicit in our proofs is another restriction on input domains for which we can determine a fairly narrow "frontier".

The NP-completeness of *some* problems hinges quite strongly on the fact that one can use the expressive power of binary notation, that is, one can write an integer of magnitude n using only $\log_2 n$ symbols. For instance, MS$[n = k, r = 0,$

$\prec$ empty, τ_i arbitrary] is NP-complete for any $k \geqq 2$; but for every polynomial p and integer $k \geqq 1$, MS[$n = k$, $r = 0$, $\prec$ empty, each $\tau_i \leqq p(|\mathscr{T}|)$] can be solved in polynomial time (although the degree of the polynomial depends on k and p, and can be quite large).

In contrast, our proofs in each case can be used to construct a polynomial p such that the problem remains NP-complete *even if* we include the restriction that each $R_j(T_i) \leqq p(|\mathscr{T}|)$. The other side of the frontier is provided by the fact that for any *finite* set S, and integers j, $k \geqq 0$, MS[$n = j$, $r = k$, $\prec$ empty, each $\tau_i = 1$, each $R_j(T_i) \in S$] can be solved in polynomial time (although the degree of the polynomial again can be quite large, depending on S, j, and k). Further information is provided by the fact that MS[$n = 3$, r arbitrary, $\prec$ empty, each $\tau_i = 1$, each $R_j(T_i) \in \{0, 1\}$] *is* NP-complete. It is not yet known whether there exists an integer $k \geqq 0$ and finite set S such that MS[n arbitrary, $r = k$, $\prec$ empty, each $\tau_i = 1$, each $R_j(T_i) \in S$] is NP-complete.

REFERENCES

[1] E. G. Coffman and R. L. Graham, *Optimal scheduling for two-processor systems*, Acta Informat., 1 (1972), pp. 200–213.

[2] S. A. Cook, *The complexity of theorem proving procedures*, 3rd Ann. ACM Symp. on Theory of Computing, 1971, pp. 151–158.

[3] J. Edmonds, *Paths, trees, and flowers*, Canad. J. Math., 17 (1965), pp. 449–467.

[4] M. Fujii, T. Kasami and K. Ninomiya, *Optimal sequencing of two equivalent processors*, SIAM J. Appl. Math., 17 (1969), pp. 784–789.

[5] M. R. Garey and R. L. Graham, *Bounds on scheduling with limited resources*, 4th Symp. on Operating System Principles, 1973, pp. 104–111.

[6] R. L. Graham, *Bounds on multiprocessing anomalies and related packing problems*, AFIPS Conf. Proc., 40 (1972), pp. 205–217.

[7] T. C. Hu, *Parallel scheduling and assembly line problems*, Operations Res., 9 (1961), pp. 841–848.

[8] R. M. Karp, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.

[9] Y. Muraoka, *Parallelism exposure and exploitation in programs*, Ph.D. thesis, Univ. of Illinois, Urbana, 1971.

[10] J. D. Ullman, *Polynomial complete scheduling problems*, 4th Symp. on Operating System Principles, 1973, pp. 96–101.

SIAM J. COMPUT.
Vol. 4, No. 4, December 1975

PROVING THEOREMS WITH THE MODIFICATION METHOD*

D. BRAND†

Abstract. A method for proving theorems in first order predicate calculus theories with equality is described and proven complete. Completeness of this "Modification Method" implies completeness of Paramodulation without the functionally reflexive axioms, thus proving a conjecture of Wos and Robinson (1969). Moreover, completeness holds with some other restrictions, such as limiting paramodulation into variables. Experimental results using the Modification Method are included.

Key words. theorem proving, equality, predicate calculus, paramodulation, model, congruence, transitivity, symmetry

1. Introduction and notation.

Note. This paper is a shortened version of the technical report Brand (1973). The reader is referred there for omitted parts. Therefore we retain numbering (of definitions, etc.) used in that technical report.

Notation. The reader is assumed to have some familiarity with predicate calculus (see Enderton (1972)) and resolution (see Robinson (1965a) or Chang and Lee (1973)). We will stress only points needed in this treatment.

We assume a countable language containing function symbols, predicate symbols and at least one constant symbol. It has the predicate = (equality), and we write equations and inequalities in infix form ($a=b$, $a\neq b$). [We use "$\equiv$" for equality in the metalanguage ($a\equiv b$, $a\not\equiv b$).] Logical symbols are $\vee$, &, $\supset$ (implication), $\leftrightarrow$ (equivalence), $\forall$, $\exists$, $\neg$. Usually we write, say, $\bar{P}x$ instead of $\neg Px$, and for a literal L (positive or negative atomic formula), $\bar{L}$ is the negation of L. Parentheses are often omitted to improve readability.

Whenever we write a clause containing variables, they are meant to be universally quantified.

We say that a term s appears on argument level if it is an argument to a predicate $P(\dots, s, \dots)$ (including the case of equality, e.g., $s=t$).

We work within first order predicate calculus without equality; i.e., equality is a predicate like any other. In particular, a formula is consistent if it has a model that may interpret the equality symbol as any binary relation. To express that a formula is consistent in predicate calculus with equality, we will say explicitly that it has a normal model. The three phrases: a formula is consistent, satisfiable and has a model are used interchangeably to denote the same thing.

Quite often we will use a phrase such as "s is on the left of $=$". This refers to a literal of the form $s=t$. For example, in $f(a)=b$, $f(a)$ is on the left of $=$, but a is not.

Summary of notation.

$=$	formal equality,
$\equiv$	equality in metalanguage,
u, v, w, x, y, z	variables,
p, q, r, s, t	terms,

* Received by the editors June 28, 1974, and in revised form January 13, 1975.

† Department of Computer Science, University of Toronto, Toronto, Canada M5S 1A7. This research was supported by the National Research Council of Canada.

f, g	function symbols,
P, Q	predicate symbols,
L, K	literals,
a, b, c, d	constant symbols,
C, D, E	clauses,
S, R	set of clauses,
$L[s], t[s]$	exhibits an occurrence of s in L, and in t,
θ, ϕ, σ	substitutions,
i, k, l, m, n, N	natural numbers,
B	a formula,
M	an interpretation, model,
m.g.u.	most general unifier,
I.H.	induction hypothesis,
A	the axiom $A \equiv (x \neq y \vee x \neq z \vee y = z)$,
$\square$	the empty clause,
w.r.t.	with respect to.

Motivation. Consider the set $S = \{Pa, \bar{P}b, a = b\}$, which is contradictory in predicate calculus with equality. If we modify S into $\tilde{S} = \{(a \neq w \vee Pw), \bar{P}b, a = b\}$, then $\tilde{S}$ is contradictory in predicate calculus without equality. This paper develops that observation.

In § 2, we show that all the equality axioms expressing the congruence requirement are not necessary provided the given set of clauses is in a so-called "E-form". This can be achieved by pulling nonvariable terms from argument lists as illustrated in the previous paragraph. It should be remarked that we have a choice of modifying $(Pa \vee Qa)$ either into $(a \neq v \vee Pv \vee a \neq w \vee Qw)$ or into $(a \neq v \vee Pv \vee Qv)$. Both modifications are sound and complete, and practical considerations in choosing between them will be mentioned in § 4. The effect of resolving $a \neq v$ with $a = b$ is to substitute b for a into all positions where v appears.

Section 3 shows that the equality axioms expressing symmetry and transitivity are not necessary if the given set of clauses is in "ST-form". This leaves us only with the reflexivity axiom $x = x$.

Section 4 gives an effective way of modifying a set of clauses S into $\tilde{S}$ which is in "E-form" as well as "ST-form". Thus proving the inconsistency of S together with all the equality axioms can be replaced by proving the inconsistency of $\tilde{S}$ together with the reflexivity axiom only.

Section 5 proves completeness of paramodulation without the functionally reflexive axioms.

Section 6 quotes some experimental results using the modification method.

2. Equality as a congruence relation. Here we prove the main result—namely, how to ensure substitutivity properties of equality (without most of the equality axioms). We will need to ensure that equality is an equivalence relation; therefore, in this section we include the axioms $x = x$ and $A \equiv (x \neq y \vee x \neq z \vee y = z)$. The axiom A will be removed in § 3.

We are going to define the notion of an S-interpretation (S for symmetry). In general, by an interpretation we mean a truth assignment to all ground literals, and we represent it by a set of those literals that are assigned true. Since we will always interpret equality symmetrically (the axioms $A \equiv (x \neq y \vee x \neq z \vee y = z)$ and $x = x$ imply symmetry), we will include only one direction of every equation and inequality. We choose the one with left-hand side more complex than the right side. (This is defined precisely in Definition 2.1.) For example, an S-interpretation will contain either $f(a) = a$ or $f(a) \neq a$ but neither $a = f(a)$ nor $a \neq f(a)$. It is implied that the truth value of $a = f(a)$ is the same as that of $f(a) = a$. The literals of an S-interpretation will be ordered according to increasing complexity; for example, $P(f(a))$ will be above $P(a)$.

First we order all ground literals using Gödel numbers. Assign a unique odd natural number to every constant, function and predicate symbol of the (assumed countable) language. This gives the basis for the following inductive definition of $G(t)$ for a term t. Suppose $t \equiv f(t_1, \cdots, t_n)$, where $G(f)$, $G(t_i)$ have been defined. Define

$$G(t) \equiv 2^{G(f)} \cdot \prod_{i=1}^{n} \mathrm{Pr}_{i+1}^{G(t_i)},$$

where Pr_i is the ith prime.

The only important properties of this definition are the following: Suppose $G(s) > G(t)$ and s is a subterm of r (including the case $s = r$), and let r' be obtained from r by replacing an occurrence of s by t. Then $G(r') > G(r)$. (Proof by induction on the structure of r.) Also, if t is a subterm of s, then $G(s) \geqq G(t)$, with equality holding iff $s \equiv t$.

For a ground atomic formula L define

$$G(L) \equiv \begin{cases} 2^{G(P)} \cdot \prod_{i=1}^{n} \mathrm{Pr}_{i+1}^{G(t_i)} & \text{if } L \equiv P(t_1, \cdots, t_n),\ P \text{ not equality,} \\ G(s) & \text{if } L \equiv s = t. \end{cases}$$

For a negation, $G(\bar{L}) \equiv G(L)$. The important property of this definition is that for every literal L containing s, $G(s = t) < G(L[s])$, except possibly for the case $L \equiv s = r$ or $L \equiv s \neq r$. (It is also not true for $L \equiv p = q$ or $L \equiv p \neq q$ with $G(p) < G(q)$; but we will not allow this in our definition of an S-interpretation.)

The Gödel number $G(C)$ of a ground clause C is the sum of the Gödel numbers of all its terms appearing on argument level (see notation). The important property of this definition is that, if $G(s) > G(t)$ and C' is obtained from C by replacing one or more occurrences of s by t, then $G(C) > G(C')$.

Definition 2.1. An S-*interpretation* is an assignment of true or false to every ground literal (L is assigned true iff $\bar{L}$ is assigned false), which interprets equality symmetrically, i.e., $s = t$ and $t = s$ have the same truth value.

With every S-interpretation, we associate a consistent set of ground literals in a natural way (we call this set also an S-interpretation, for no confusion can arise). A literal L is in an S-interpretation M iff it is true under M and it is not of the form $t = s$, $t \neq s$ where $G(t) < G(s)$.

A *partial* S-*interpretation* is an assignment of true or false to some ground literals where equality is interpreted symmetrically (in particular $s = t$ is defined iff $t=s$ is). In the set notation, a partial S-interpretation is a subset of an S-interpretation. An S-interpretation is a special case of a partial S-interpretation.

The literals of a partial S-interpretation are ordered into a sequence according to their increasing Gödel numbers G; call this order "$<$", "below" or "above". Literals with the same Gödel number are equations and inequalities with identical left sides (there is only a finite number of such for each left side), and they are ordered according to the increasing Gödel numbers of their right sides.

Note that if $s=t$ is in an S-interpretation, then s is not a proper subterm of t.

DEFINITION 2.2. A ground clause is *true* under an S-interpretation if at least one of its literals is true; it is false otherwise.

DEFINITION 2.3. An S-interpretation M is an *S-model* for a set of clauses S iff every ground instance of every clause from S is true under M.

Remark. S has an S-model iff S, together with the symmetry axiom $x \neq y \vee y=x$, has any model.

DEFINITION 2.4. An E*-inconsistency w.r.t. an equation* $s=t$ in a partial S-interpretation M is a pair of literals $L[s], \bar{L}[t]$ which are both true under M. ($L[t]$ is obtained by replacing by t the exhibited occurrence of s in $L[s]$.)

An E*-inconsistency* in M is a triple $\{s=t, L[s], \bar{L}[t]\}$ such that $s=t$ is true under M, and $\{L[s], \bar{L}[t]\}$ is an E-inconsistency w.r.t. $s=t$.

A partial S-interpretation is E*-consistent w.r.t.* $s=t$ iff it contains no E-inconsistency w.r.t. $s=t$. It is E*-consistent* iff it contains no E-inconsistency.

Remark. Literals of an S-interpretation M are ordered in such a way that if M is a model for the axioms $A \equiv (x \neq y \vee x \neq z \vee y=z)$ and $x=x$, and $\{s=t, L[s], \bar{L}[t]\}$ is an E-inconsistency in M, then $L[s]$ is above both $s=t$ and $\bar{L}[t]$.

DEFINITION 2.5. *An S-interpretation M* is an E*-model* for a set of clauses S, iff M is an S-model for $S \cup \{x=x\}$, and M is E-consistent.

Remark. M is an E-model for S iff M is a model for S together with the equality axioms.

Proof. ($\Rightarrow$). Suppose that M does not satisfy an axiom $x \neq y \vee P(x_1, \cdots, x, \cdots, x_n) \vee \bar{P}(x_1, \cdots, y, \cdots, x_n)$. Then there are three literals $s=t$, $\bar{P}(r_1, \cdots, s, \cdots, r_n)$, $P(r_1, \cdots, t, \cdots, r_n)$ true under M, which contradicts E-consistency of M w.r.t. $s=t$. Suppose that M does not satisfy an axiom $x \neq y \vee f(x_1, \cdots, x, \cdots, x_n) = f(x_1, \cdots, y, \cdots, x_n)$. Then there are two literals $s=t$, $f(r_1, \cdots, s, \cdots, r_n) \neq f(r_1, \cdots, t, \cdots, r_n)$ true under M; since $f(r_1, \cdots, s, \cdots, r_n) = f(r_1, \cdots, s, \cdots, r_n)$ is true, M is not E-consistent w.r.t. $s=t$.

($\Leftarrow$). If M satisfies all the equality axioms, then it must interpret equality as a congruence relation; therefore, M is E-consistent. Q.E.D.

The following definition of $M[s/t]$ gives the basic notion used in the proof of Theorem 2.1. Given an S-interpretation M and a pair of terms s, t (s is not a subterm of t), we construct an S-interpretation $M[s/t]$ that will be E-consistent w.r.t. $s=t$. It is done by redefining the truth values of literals in M containing s, according to the literals obtained by replacing all occurrences of s by t. For example, $P(s)$ will have the same truth value as $P(t)$ and $f(s)=s$ will have the same truth value as $f(t)=t$. It is done by going from the bottom to the top of the interpretation, possibly changing signs of the literals.

DEFINITION 2.6. Let M be an S-interpretation and s, t terms such that

$G(s) > G(t)$. We define inductively a partial S-interpretation M_L for each literal $L \in M$ with the properties:

(i) for each $K \in M$, $K \leqq L$, exactly one of K, $\bar{K}$ is in M_L, and no other literals are in M_L;

(ii) M_L is E-consistent w.r.t. $s = t$;

(iii) if $K \leqq L$, then $M_K \subseteq M_L$ (i.e., M_L extends M_K).

Suppose that we have defined M_K for every $K \in M$, $K < L$. Let $M'_L \equiv \bigcup_{K<L} M_K$. Note that $M'_L \equiv M_{L_0}$, where L_0 is the predecessor of L, if L has one; $M'_L \equiv \phi$ otherwise.

Case 1. L does not contain s. Then define $M_L \equiv M'_l \cup \{L\}$. Conditions (i), (iii) are clearly satisfied. Condition (ii) is satisfied because an E-inconsistency w.r.t. $s = t$ not involving L cannot be in M_L by I.H., and any E-inconsistency w.r.t. $s = t$ involving L would imply that L is of the form $L[t]$, and $\bar{L}[s]$ is true under M_L, which is not possible because $L[t] < \bar{L}[s]$.

Case 2. L does contain s. Let L' be obtained from L by replacing any occurrence of s by t. (Then $L' < L$.) Let

$$M_L \equiv \begin{cases} M'_L \cup \{L\} & \text{if } L' \text{ is true under } M'_L, \\ M'_L \cup \{\bar{L}\} & \text{otherwise.} \end{cases}$$

Note that this is a proper definition, i.e., independent of the choice of the occurrence of s in obtaining L'. If a different choice should give a different result, it would mean that L is of the form $L[s, s]$ and $L[s, t]$ has the opposite truth value of $L[t, s]$. But then one of them would give an E-inconsistency w.r.t.s. $s = t$ with $L[t, t]$ in M'_L (contradicting I.H.).

Conditions (i) and (iii) are clearly satisfied; (ii) is satisfied because we gave L just that truth value so as not to create any E-inconsistency w.r.t. $s = t$. Now we define $M[s/t] \equiv \bigcup_{L \in M} M_L$. It follows that $M[s/t]$ is an S-interpretation that agrees with M on literals not containing s, and $M[s/t]$ is E-consistent w.r.t. $s = t$. (But $s = t \in M[s/t]$ only if $t = t \in M$.)

DEFINITION 2.7. A clause is in E-*form* iff it has nonvariable terms only as arguments of $=$, $\neq$.

Example.

$(P(x, y) \vee fx = y)$ is in E-form.

$(P(x, y) \vee w = a \vee fx \neq w)$ is in E-form.

$(fa \neq ga)$ is not in E-form.

$(fw \neq gw \vee a \neq w)$ is in E-form.

LEMMA 2.1: *Let M be an* S-*model for a set S of clauses in* E-*form, containing the axioms A and* $x = x$. *Suppose that* $s = t \in M$, $G(s) > G(t)$. *Then* $M[s/t]$ *is an* S-*model for S.*

Proof. Suppose the contrary, and let C be a ground instance of a clause C_0, so that C has the minimal Gödel number among all such clauses false under $M[s/t]$.

Case 1. C does not contain the term s. Then the truth value of C under $M[s/t]$ is the same as under M, i.e., true.

Case 2. There is an occurrence of s in C in a position other than an argument of equality. Let ϕ be the substitution such that $C \equiv C_0\phi$. Since C_0 has nonvariable terms only as arguments of equality, ϕ must assign a term $r[s]$ to a variable x of C_0.

Let ϕ' be obtained from ϕ by changing the assignment to x to be $r[t]$. Then $G(C_0\phi') < G(C)$, hence $C_0\phi'$ is true under $M[s/t]$. But the truth value of all literals of C is the same as those of $C_0\phi'$ under $M[s/t]$. Therefore C must be true under $M[s/t]$.

Case 3. All occurrences of s in C are as arguments of equality. But truth values of the literals of the form $s = r$, $r = s$, $s \neq r$, $r \neq s$ are the same as in M. The reason is, that in order for, say, $s = r$ to become false in $M[s/t]$ from true in M, we would have to have in M $s = r$, $t \neq r$; but that would contradict the axioms A, $x = x$ (in view of $s = t \in M$). Q.E.D.

THEOREM 2.1. *Let S be a satisfiable set of clauses in* E*-form, containing the axioms $A = (x \neq y \vee x \neq z \vee y = z)$ and $x = x$. Then S has an* E*-model.*

Proof. Every model of S is symmetrical because of A and $x = x$. If S did not have an E-model, then there would be a finite subset of the Herbrand universe in which S together with the equality axioms would be unsatisfiable. Therefore it is enough to show that we can construct S-models of S with an arbitrarily large initial segment that is E-consistent.

The idea is to apply Lemma 2.1 successively. The ordering of literals will ensure that by fixing a literal, we do not spoil anything below.

Notation. M_n is the initial segment of an S-interpretation M, consisting of the first n literals ($n \geqq 0$).

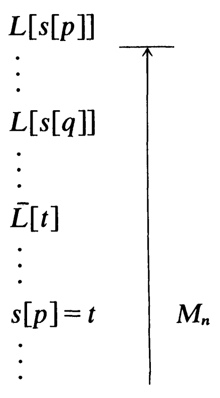
Suppose that M is an S-model for S, and M_n is the largest initial segment that is E-consistent. We construct an S-model M^*, where M^*_{n+1} is E-consistent.

We have $s = t$, $L[s]$, $\bar{L}[t]$ in M, where $L[s]$ is the (n+1)st literal ($L[s]$ cannot be $s = r$ or $s \neq r$ because of the axioms A and $x = x$).

Let $M^* \equiv M[s/t]$. Because of E-consistency of M_n, we have $M^*_n \equiv M_n$. Thus to show that M^*_{n+1} is E-consistent, we only need to verify that there is no E-inconsistency involving the ($n+1$)st literal of M^* (which is $\bar{L}[s]$) with some literals below.

Suppose, to the contrary, there is an E-inconsistency w.r.t. $p = q$ ($G(p) > G(q)$). This means that $L[s]$ also contains p. Referring to s or p below, we will mean their occurrences in L, such that replacing s by t, we get $L[t]$, which is false in M_n, and replacing p by q, we obtain $L[q]$ which is true in M_n.

Case 1. p is a subterm of $s \equiv s[p]$. Since $s[p] = t \in M_n$, then $s[q] = t \in M_n$ (or $t = s[q] \in M_n$; without loss of generality suppose the former). The situation in M is



$$\begin{array}{l|}
s[q]=t \\
\vdots \\
p=q \\
\vdots
\end{array}$$

Then $s[q]=t$, $\bar{L}[t]$, $L[s[q]]$ would give us an E-inconsistency in M_n.

Remark. The literals in M_n may be ordered differently from the order suggested by the picture, but the order inside M_n is irrelevant.

Case 2. s is a subterm of $p \equiv p[s]$. Since $p[s]=q \in M_n$, then $p[t]=q \in M_n$ (or $q=p[t] \in M_n$; again we assume the former). The situation in M is

$$\begin{array}{l|l}
L[p[s]] & \\
\vdots & \\
\bar{L}[p[t]] & \\
\vdots & \\
L[q] & \\
\vdots & \\
p[s]=q & M_n \\
\vdots & \\
p[t]=q & \\
\vdots & \\
s=t & \\
\vdots &
\end{array}$$

We have a contradiction with E-consistency of M_n.

Case 3. s and p are not subterms of each other, so the $(n+1)$st literal is of the form $L[s,p]$. Then M_n must contain $\bar{L}[t,p]$, and also $L[s,q]$. But this is a contradiction with E-consistency of M_n w.r.t. $s=t$ or $p=q$—depending on the truth value of $L[t,q]$. Q.E.D.

3. Equality as an equivalence relation. Here we show how to avoid including the axiom A. A set of clauses S will be modified into $\tilde{S}$ so that $\tilde{S} \cup \{x=x\}$ is consistent iff $S \cup \{x=x, A\}$ is consistent.

The idea is to replace every equation $s=t$ in S by $t \neq w \lor s=w$, where w is a new variable. This will take care of transitivity. Symmetry will be taken care of by considering both sides of every equation (i.e., $s=t$ and $t=s$). So $s=t$ will be modified into $t \neq w \lor s=w$ and $s \neq w \lor t=w$.

DEFINITION 3.1. A clause C is in T-*form* (T for transitivity) iff each equation of C is in the form $s = w$, and C contains a literal $t \neq w$, where w has only those two occurrences in C. (But w may, of course, appear in other clauses.)

DEFINITION 3.2. A set of clauses S is in ST-*form* (S for symmetry) iff each clause of S is in T-form, and whenever a clause $(t \neq w \vee s = w \vee C')$ is in S, then there is also a clause $(s \neq w \vee t = w \vee C')$ in S.

THEOREM 3.1. *Let $\tilde{S}$ be a set of clauses in* ST-*form, and suppose that $\tilde{S} \cup \{x = x\}$ is consistent. Then $\tilde{S} \cup \{x = x, A\}$ is consistent.*

Proof. See Brand (1973).

4. The modification method. Theorems 2.1, 3.1 give us a way of proving theorems with equality. Given a set of clauses S, containing equality, we "pull out" nonvariable terms and make them companions. Then we transform the result into ST-form and add $x = x$ to obtain the modification $\tilde{S}$. Then to find if S has an E-model, we can apply ordinary resolution to find if $\tilde{S}$ has any model. Since each step of the modification process corresponds to a resolution with an equality axiom, the whole process can be thought of as applying the equality axioms a fixed number of times, and then throwing them away.

First we define the notion of companions. The motivation is the following: We "pull out" a term $f(a)$ from inside a literal $Pf(a)$, and make it a companion. The result will be $f(a) \neq w \vee P(w)$, where w is a new variable, and $f(a) \neq w$ is said to be a *companion* of $P(w)$. The level of $P(w)$ will be 0 (level of ordinary literals). The level of $f(a) \neq w$ will be 1 (its position in the original $P(fa)$). Then we pull out the a, getting $a \neq u \vee f(u) \neq w \vee Pw$. Again we call $a \neq u$ a companion of $f(u) \neq w$ (which is a companion of Pw), and it has level increased by 1; i.e., the level of $a \neq u$ is 2.

The reason for introducing the levels is to ensure that we do not get circularity, e.g., that a literal is not a companion of itself. This will be used only for the definition of positive resolution, which is in turn used only in § 5 (paramodulation); i.e., the modification method itself does not need it.

DEFINITION 4.1. A *normalized clause* C is a forest of literals, such that every literal L that is not a root is of the form $s \neq w$, and the variable w has exactly one other occurrence in C—inside the direct predecessor K of L. We call L a companion of K. If K is of the form $t = w$ (i.e., w is on the right of equality), then we call L a T-companion of K, $\{L, K\}$ is a T-companion pair. Roots are assigned level 0, companions of a literal K are assigned level one greater than the level of K. (Roots of C can be arbitrary literals).

A *clause* C is an instance $C_0\phi$ of a normalized clause C_0. Companions and levels are preserved—i.e., $L\phi$ from C is a companion (T-companion) of $K\phi$ iff L is a companion (T-companion) of K in C_0. (A normalized clause is a special case of a clause).

Remark. Usually we exhibit literals in the form $s \neq r \vee L[r]$. This is to imply that $s \neq r$ is a companion (or T-companion) of $L[r]$.

Examples.

$$(a \neq b \vee P(b)), \qquad (f(a, b) \neq w \vee g(w) \neq c \vee P(x, c)),$$
$$(a \neq b \vee c = b \vee a \neq b \vee f(b) = c).$$

Remark. Looking at a clause as a tree does not change the intended meaning—it is still a disjunction of its literals. The introduction of levels is just a syntactical tool.

Note that any set of literals can be organized into a clause; all literals are at level 0. This would be the normal arrangement; modification then introduces companions.

Note that if a normalized clause C contains a companion $s \neq w$ of a literal K which is also a companion, then K cannot be of the form $t \neq w$; K must be of the form $t[w] \neq v$; otherwise w would be forced to have at least three occurrences in C.

DEFINITION 4.2. We define the E-*modification* $\tilde{C}$ of a clause C inductively. If all nonvariable terms of C appear as arguments of equality, let $\tilde{C} \equiv C$. Otherwise let C' be obtained from C as follows: Pick a nonvariable term s appearing on an argument level of a predicate P (other than $=$); e.g., $P(\cdots, s, \cdots)$, or appearing as an argument of a function, which is an argument to equality; e.g., $f(\cdots, s. \cdots) \neq v$. Then rewrite the literal $L[s]$ into $L[w]$, where w is a new variable appearing nowhere else, and give $L[w]$ a new companion $s \neq w$. In our examples $P(\cdots, s, \cdots)$ is replaced by $s \neq w \vee P(\cdots, w, \cdots)$. $f(\cdots, s, \cdots) \neq v$ is replaced by $s \neq w \vee f(\cdots, w, \cdots) \neq v$. The rest of the clause is unchanged. The E-modification of C (unique up to renaming of variables) is then the E-modification of C'.

Example. E-modification of $(Pf(a, gx) \vee f(x, y) = g(a))$ is

$$(a \neq u \vee gx \neq v \vee f(u, v) \neq w \vee Pw \vee a \neq z \vee f(x, y) = gz).$$

DEFINITION 4.3. Let C be a clause without any T-companions. The T-*modification* $\tilde{C}$ of C is obtained by replacing each equation $s = t$ in C by the T-companion pair $t \neq w \vee s = w$, where w is a new variable referred to as a T-new variable. Note that T-companions have level 1.

DEFINITION 4.4. The ST-modification $\tilde{S}$ of a set S of clauses with all literals on level 0 is obtained as follows. Let S' be the smallest set of clauses containing S such that whenever $(s = t \vee C) \in S'$, then $(t = s \vee C) \in S'$. All literals in S' have level 0. Then $\tilde{S}$ is the set of T-modifications of clauses from S'.

Remark. Intuitively, S' is obtained from S by flipping equations in all clauses of S in all possible ways. Thus a clause from S with n equations produces 2^n clauses in S'.

Example. $S \equiv \{(x = y \vee Px, y), (f(x) = x \vee g(x) = x), a \neq b\}$; then the ST-modification $\tilde{S}$ has the 7 clauses:

$y \neq w \vee x = w \vee Px, y,$
$x \neq w \vee y = w \vee Px, y,$
$x \neq u \vee f(x) = u \vee x \neq w \vee g(x) = w,$
$x \neq u \vee f(x) = u \vee g(x) \neq w \vee x = w,$
$f(x) \neq u \vee x = u \vee x \neq w \vee g(x) = w,$
$f(x) \neq u \vee x = u \vee g(x) \neq w \vee x = w,$
$a \neq b.$

DEFINITION 4.5. Let S be a set of clauses and S' the set of E-modifications of clauses of S. *Then the* STE-*modification* of S is the ST-modification of S'.

Remark. The STE-modification of S is both in ST-form and E-form, and it consists of normalized clauses. Further, every step in producing an STE-modification can be done by resolution with an equality axiom (or with a clause derivable from the axioms). Or equivalently, the modifications can be done by resolving with the second order equality axiom $x \neq y \vee \bar{P}x \vee Py$ (see Darlington (1968) or Jensen and Pietrzykowski (1972)). For example, $f(a)=b$ would be modified into $a \neq y \vee f(y)=b$ using $P \leftarrow \lambda x \cdot f(x)=b$.

Note that if all companions of an E-modification or of a T-modification are resolved with $x=x$, then we obtain the original clause.

THEOREM 4.1. *A set of clauses S has an* E*-model iff its* STE*-modification $\tilde{S}$ together with $x=x$ has any model.*

Proof. ($\Rightarrow$). If S has an E-model, then any set obtained from S by resolving with the equality axioms also has an E-model; in particular, $\tilde{S}$ does.

($\Leftarrow$). If $\tilde{S} \cup \{x=x\}$ has a model, then by Theorem 3.1 $\tilde{S} \cup \{x=x, A\}$ has a model; so by Theorem 2.1 $\tilde{S} \cup \{x=x, A\}$ has an E-model M. Every set of clauses obtained from $\tilde{S}$ by resolving with $x=x$ has the same model M, in particular S does. Q.E.D.

Practical considerations. In practice, we do not need to "pull out" a term s which is not unifiable with any s' appearing in $s'=t$, because then the companion $s \neq w$ could be resolved only with $x=x$, which puts s right back.

The definition of E-form does not use the notion of companions. Therefore in modifying a formula we do not need to worry about properties required for companions, as long as the modification is sound. In particular, we can have one companion to more than one literal. For example $(Pa \vee Qa)$ can be modified into $(a \neq w \vee Pw \vee Qw)$. (We do not allow this in the definition of E-modification for simplicity. Resolving with $a \neq w$ would correspond to two applications of the paramodulation rule.) In practice, this would be advantageous only if our intuition told us that $a \neq w$ will be resolved only with $x=x$. Otherwise a more awkward (thus harder to find) proof may be required, if the natural proof would substitute different terms for a in Pa and Qa.

In producing ST-modification, we are allowed to declare any literal a T-companion, provided it satisfies Definition 3.1. For example, the T-modification of $C \equiv (a \neq x \vee b=x)$ can be C itself, rather than $(a \neq x \vee x \neq w \vee b=w)$. Thus $a=b$ and $a \neq x \vee b=x$ may be modified into equivalent ST-forms.

5. Paramodulation.

The purpose of this section is to prove completeness of paramodulation without the functionally reflexive axioms. We will use some special inference rules which can be simulated by paramodulation.

As usual, we assume consistent renaming of variables, so that two clauses operated on by Rule VII (resolution) or by Rule VIII (paramodulation) do not share any variables.

Rule VI (positive factoring). From the clause $C_0 \equiv (L_1 \vee L_2 \vee C_0')$ which contains no companions, where L_1 and L_2 have an m.g.u. θ, infer $C \equiv (L_1\theta \vee C_0'\theta)$. All literals of C have level 0.

Rule VII (positive resolution). From a clause $E \equiv (L_1 \vee E')$ without companions, and a clause $D \equiv (\bar{L}_2 \vee D')$ where $\bar{L}_2$ has no companions, and

θ is an m.g.u. of L_1 and L_2, infer $C \equiv (E'\theta \vee D'\theta)$. Literals of C have the same level as their ancestors in E' and D'. If $\bar{L}_2$ was a companion of a literal $K \in D'$, then $K\theta$ has one less companion in C; otherwise there is no change in companions.

Remark. We do not require any of the literals figuring in Rules VI, VII to be positive. We only place restrictions on the presence of companions (which are inherently negative). Since these restrictions are very much like those of P1-resolution (Robinson (1965b)), we call the rules positive. In fact the completeness of these rules will be proven using hyper-resolution.

Example (companions as suggested). $E \equiv (a = b)$, $D \equiv (a \neq u \vee f(u) \neq v \vee Pv)$ resolve into $C \equiv (f(b) \neq v \vee Pv)$. In C, $f(b) \neq v$ does not have a companion any more.

LEMMA 5.1. *Suppose S is an inconsistent set of clauses. Then there exists a refutation of S by Rules* VI, VII.

Proof. There is a hyper-resolution proof of S (Robinson (1965b)).

Resolving a hyper-clash is done:

(i) factor the positive electrons,
(ii) factor the nucleus D,
(iii) resolve electrons against all the negative literals of the nucleus.

Step (i) is positive factoring, for electrons (being positive) contain no companions. We do not perform step (ii), because D may contain companions. Thus we enter step (iii) with some duplicate literals in D.

Step (iii) will be performed resolution by resolution. The next resolution will always be an electron against a literal with maximal level in the resolvent obtained to that point. This ensures that the resolution is positive. We may have to repeat the resolution against duplicate literals. For example, if hyper-resolution has the factored nucleus $(\bar{L} \vee C')$, we may be dealing with $(\bar{L} \vee \bar{L} \vee C')$. Thus we resolve twice with $(E' \vee L)$ into $(E' \vee E' \vee C')$.

We add another step that will delete the duplicate remainders E' of the electrons from the final resolvent. This factoring is positive because the resolvent is positive.

LEMMA 5.2. *Let S be a set of normalized clauses and C be a clause derived from S by Rules* VI *and* VII. *Then C is a normalized clause.*

Proof. It is enough to show that the clause produced by either of the rules is a normalized clause whenever the input clauses to that rule are normalized clauses.

(i) This is clear for Rule VI (positive factoring) because the clause produced has no companions.

(ii) For Rule VII we need to verify the condition of Definition 4.1 stating that every companion L of the resolvent is of the form $s \neq w$, and w has exactly one other occurrence—inside the predecessor of L. Every companion in the resolvent results from applying the m.g.u. θ to a companion from one of the parent normalized clauses. Each such original companion is of the form $s \neq w$ by assumption. In order that the m.g.u. θ assigns anything to w, or it assigns w to another variable, w would have to appear in one of the literals resolved upon. But this cannot happen because Rule VII resolves upon literals without companions. Q.E.D.

For our purposes we define paramodulation as Rule VIII.

Rule VIII. From clauses $D \equiv (L[s'] \vee D')$ and $E \equiv (s = t \vee E')$, where θ is an m.g.u. of s and s', infer $C \equiv (L\theta[t\theta] \vee D'\theta \vee D'\theta \vee E'\theta)$.

For a set of clauses S, we show how to simulate a refutation of the STE-modification $\tilde{S}$, which uses positive factoring and resolution (Rules VI, VII), by a refutation of S using Rules VI, VII, VIII. Note that Rules VI, VII constitute unrestricted resolution for a set of clauses without companions.

This then gives completeness of paramodulation without the functionally reflexive axioms, and with some other restrictions following from the STE-form; for example, the only type of variables we need to paramodulate into are on the left of $=$.

THEOREM 5.1. *Let S be a set of clauses, which has no* E-*model. Then there exists a refutation of $S \cup \{x = x\}$ by Rules* VI, VII, VIII.

Proof. Let $\tilde{S}$ be an E-modification of S. By Theorem 2.1 $\tilde{S} \cup \{A, x = x\}$ is inconsistent, and by Lemma 5.1 there is a refutation of $\tilde{S} \cup \{A, x = x\}$ using Rules VI, VII. Using a hyper-resolution argument, we can replace the axiom A by Rule IV.

Rule IV. From $E_1 \equiv (E_1' \vee s = t)$, $E_2 \equiv (E_2' \vee s' = r)$, infer $C \equiv (E_1'\theta \vee E_2'\theta \vee t\theta = r\theta)$, *where* θ is an m.g.u. of s and s', and neither E_1 nor E_2 contain any companions. The result C will contain no companions.

(Note that Rule IV is just a special case of paramodulation.)

So, using Lemma 5.2 together with the fact that Rule IV produces a normalized clause, there is a refutation $\tilde{\alpha}$ of $\tilde{S} \cup \{x = x\}$ by Rules IV, VI and VII, and $\tilde{\alpha}$ consists of normalized clauses.

We will define inductively a 1–1 correspondence ψ that assigns to every clause $\tilde{C}$ of $\tilde{\alpha}$ a clause C of a paramodulation proof α of S, being defined. All literals in the proof α will have level 0.

For a normalized clause $\tilde{C}$, we define inductively a map ψ that assigns an expression to every literal $\tilde{L}$ of $\tilde{C}$. The expression is obtained by resolving all the literals in the subtree of $\tilde{L}$ with $x = x$ (which has the effect of substituting the s from a companion $s \neq w$ for the variable w).

We define ψ as follows: Let $\tilde{L}_1, \tilde{L}_2, \cdots, \tilde{L}_n$ ($n \geqq 0$) be all the companions of $\tilde{L}$. Then by Lemma 5.2 each $\tilde{L}_i$ is of the form $s_i \neq w_i$, and $\tilde{L}$ is the form $\tilde{L}[w_1, \cdots, w_n]$. Let $L \equiv \tilde{L}[\psi(L_1), \cdots, \psi(L_n)]$. If $\tilde{L}$ has level 0, then let $\psi(\tilde{L}) \equiv L$; otherwise L is of the form $s \neq w$, let $\psi(\tilde{L}) \equiv s$. (This is a definition by induction on the tree structure. If $\tilde{L}$ is a leaf, i.e., $n = 0$, then $\tilde{L} \equiv L$.)

Note that ψ has been defined so that if $\tilde{L}$ is a companion of $\tilde{K}$, then $\psi(\tilde{L})$ is a subterm of $\psi(\tilde{K})$.

For a normalized clause $\tilde{C}$, we define $\psi(\tilde{C})$ as the clause $\psi(\tilde{C}) \equiv \{\psi(\tilde{L}) | \tilde{L} \in \tilde{C}$, $\tilde{L}$ has level 0$\}$ (all literals of $\psi(\tilde{C})$ have level 0). Note that the E-modification and T-modification $\tilde{C}$ of a clause C have been defined so that $\psi(\tilde{C}) \equiv C$ (Definitions 4.2, 4.3.).

The paramodulation proof α will consist of the clauses $C \equiv \psi(\tilde{C})$, $\tilde{C} \in \tilde{\alpha}$. Before defining α, we give an example. Corresponding clauses are written on the same line. The thing to note is that resolution against companions corresponds to paramodulation.

	$\tilde{\alpha}$	α	
1.	$a \neq v \vee fv \neq u \vee a \neq w \vee Pu, w$	Pfa, a	
2.	$b \neq v \vee fv \neq u \vee b \neq w \vee \bar{P}u, w$	$\bar{P}fb, b$	original
3.	$a = b$	$a = b$	$S \cup \{x = x\}$
4.	$x = x$	$x = x$	
5.	$fb \neq u \vee b \neq w \vee \bar{P}u, w$	$\bar{P}fb, b$	from 2., 4.
6.	$b \neq w \vee \bar{P}fb, w$	$\bar{P}fb, b$	from 5., 4.
7.	$\bar{P}fb, b$	$\bar{P}fb, b$	from 6., 4.
8.	$fb \neq u \vee a \neq w \vee Pu, w$	Pfb, a	from 1., 3.
9.	$a \neq w \vee Pfb, w$	Pfb, a	from 8., 4.
10.	Pfb, b	Pfb, b	from 9., 3.
11.	$\square$	$\square$	from 7., 10.

The basis for α is constituted by the original clauses of S. These clauses correspond to those of $\tilde{S}$ in the way established by the process of E-modification.

Suppose that we have defined an initial segment of α. The next step of α is defined according to the next step of $\tilde{\alpha}$.

Case 1 (positive factoring of $\tilde{C}_0$ into $\tilde{C}$). Since $\tilde{C}_0$ contains no companions, we have $\tilde{C}_0 \equiv \psi(\tilde{C}_0) \equiv C_0$. So we perform the same factoring in C_0, producing $C \equiv \tilde{C}$.

Case 2 (resolution with companions corresponds to paramodulation). A companion $s' \neq w$ of $\tilde{L}[w]$ in $\tilde{D} \equiv (s' \neq w \vee \tilde{L}[w] \vee \tilde{D}')$ is resolved against $s = t$ in $\tilde{E} \equiv (s = t \vee \tilde{E}')$. This involves applying the m.g.u. θ of s' and s, and the substitution $w \leftarrow t\theta$. Thus the resolvent is $\tilde{C} \equiv (L\theta[t\theta] \vee E'\theta \vee \tilde{D}'\theta)$.

Since $\tilde{E}$ contains no companions, we have $E \equiv \tilde{E}$. And since $s' \neq w$ has no companions, $\psi(s' \neq w) \equiv s'$, which appears inside $\psi(\tilde{L})$ in D. So after applying the m.g.u. of s' and s, we can paramodulate from $(s = t \vee E')$ into D with the result C. The correspondence ψ between literals of $\tilde{C}$ and C carries over from $\tilde{D}$ and $\tilde{E}$. Note that if $E \equiv (x = x)$, then $C \equiv D$.

Case 3 (resolution upon literals with level 0). Neither of the literals resolved upon can have any companions, so their counterparts in α are identical to them; therefore, we can make the same resolution in α. The correspondence ψ carries over from the parent clauses.

Case 4 (Rule IV produces $\tilde{C}$ from $\tilde{E}_1$ and $\tilde{E}_2$). $\tilde{E}_1 \equiv E_1$, $\tilde{E}_2 \equiv E_2$ because $\tilde{E}_1$, $\tilde{E}_2$ contain no companion, so α can paramodulate E_1 into E_2 to get $C \equiv \tilde{C}$.

So we have defined α, and for $\square \in \tilde{\alpha}$, the corresponding clause of α is $\psi(\square) \equiv \square$, which proves the theorem.

6. Experimental results. Feasibility of the method was tested on examples from Alan et al. (1972) and from Huet (1971).

First we describe Alan's examples. The theorems are taken from the announcement entitled, *Some new axiomatizations in group theory*, on page A-547 of the Notices of the American Mathematical Society, vol. 19, 1972.

"We give axiomatizations of group theory and abelian group theory in terms of the binary group operation $f(a, b) = ab'$. Suppose G is a first order theory with equality with a binary operation symbol f, whose nonlogical axioms are: (G1) $f(x, x) = f(y, y)$, (G2) $f(x, f(y, y)) = x$, (G3) $f(f(x, z), f(y, z)) = f(x, y)$. Then G is an axiomatization of group theory (1 is defined as the (unique) element $f(x, x)$ and xy as $f(x, f(1, y))$). For abelian groups, G1, G2, G3 are replaced by: (A1)

$f(x,f(y,z))=f(z,f(y,x))$, (A2) $f(x,f(x,y))=y$, or, by the single axiom (A), $f(f(x,y),f(f(x,z),y))=z$. Some other single axioms for abelian groups are (A′) $f(f(x,f(y,z)),f(x,y))=z$ and (A″) $f(x,f(y,f(z,f(x,y))))=z$. (Received March 20, 1972.)"

In Alan et al. (1972) the following equations are introduced:

(L1) $f(x,x)=1$,

(L2) $f(x,1)=x$,

(L3) $f(1,f(x,y))=f(y,x)$,

(L4) $f(f(x,y),f(1,f(y,x)))=1$.

For axioms of elementary group theory (EGT),

(Axiom 1) $f(x,f(1,e))=x$ [e is a right identity],

(Axiom 2) $f(x,f(1,f(y,f(1,z))))=f(f(x,f(1,y)),f(1,z))$ [associativity],

(Axiom 3) $f(x,f(1,g(x)))=e$ [right inverse],

(inverse) $\forall x \exists y\, f(y,f(1,x))=1$.

After e is replaced by 1 (EGT′),

(Axiom 3′) $f(x,f(1,g(x)))=1$,

(L3′) $f(1,f(1,x))=x$ [left identity],

(Comm) $f(x,f(1,y))=f(y,f(1,x))$ [commutativity],

(X1) $f(x,f(f(y,z),f(y,x)))=z$,

(X2) $f(f(x,f(f(x,y),z)),z)=y$.

Statements of theorems using these axioms are in the table below.

Huet's examples. Those are examples 1–13. In example 12a negation of the theorem to be proven is $f(a,a,b)\neq a$. Huet introduces a constant c and proves (12a′) $f(a,a,b)=c\ \&\ a\neq c$.

In the experiments, we first found the proofs, as one would proceed in practice. Not all the clauses were included; obvious simplifications, such as $f(x,x)=1$ or $f(1,x)=x$ were left unmodified, etc. The proofs were usually obtained without any limit parameters. In almost all cases, the human intuition on how to modify the formula works well. After we found the proofs, the examples were run with the full modification, and with the optimal limits on the proof. Negation of the theorem was always used as a set of support.

The difficult examples taken from Alan were those involving Axiom 2 as a hypothesis of the theorem. It is very complex, so many substitutions can be made. It does not generate too many deductions; on the contrary, the strategy forces the theorem prover not to generate anything until it finds the right substitutions. And it may take many unifications to find the right combination. Therefore the order of the input clauses and their literals determine the speed of the search.

Among Huet's difficult examples were those that did not have a unit proof (with negation of the theorem being set of support)—you can tell them immediately from the table of performance (Table 1) by the large numbers of unifications performed.

In general, if the hypotheses are unit equations and negation of the theorem is a unit inequality (and they have no E-model), then it is possible to refute the STE-modification $\tilde{S}$ by a unit proof, because each clause of $\tilde{S}$ has at most one positive literal. It is not true, however, with the set of support strategy (for example, there is no unit proof of $\{a=b', b'=b, c=d', d'=d, f(a,c)\neq f(b,d)\}$ if only $f(a,c)\neq f(b,d)$ has support). Nevertheless, all of Alan's examples do have a unit proof, with a set of support being negation of the theorem.

TABLE 1

Huet's example	successful unifica-tions	limit		time [sec.]	unifica-tion attempts	stack entries generated	Huet	
		proof depth	func-tional nesting				time [sec.]	clauses generated
1	4 5	no no	no no	0.1 0.1	6 13	0 1	5.9 5.9	8 8
2	194 333	yes yes	yes yes	2.5 4.7	855 1615	13 20	18.9	18
3	4 6	no no	no no	0.1 0.1	4 9	0 1	3.9	5
4	4 9	no no	no no	0.1 0.1	7 28	0 1	6.0 5.3	15 12
5	10 75	no yes	no no	0.1 0.8	28 315	2 5	12.3	13
6	7 6	no no	no no	0.1 0.1	12 8	2 1	1.9 8.0	6 24
9	226 245	yes yes	yes yes	2.7 3.0	836 967	15 16	37.4 30.2	19 17
10	221 321	yes yes	yes yes	2.9 3.7	862 1134	11 13	20.1 44.2	12 40
11	>2943	yes	yes	>29.7	>8496	>61	376.2	161
12a	16 10	yes yes	no no	0.2 0.1	42 23	0 1		
12a′	17 11	yes yes	no no	0.2 0.1	47 25	1 1	40.6	68
12b	12 20	yes yes	no yes	0.2 0.2	41 52	3 4	49.4	19
13	>2814	yes	yes	>29.9	>8190	>42	208.6	150

Tables 1 and 2 give the performance of the theorem prover (described in Brand (1973)) on the theorems. It is given in terms of the number of literal pairs unified (successful unifications), actual running time in seconds on IBM 370, and the number of calls to the unification procedure (unification attempts). This last parameter does include tests for one literal being an instance of another (subsumption), and is supposed to give an idea of time requirements of the method, independent of computing environment. The number of successful unifications does not include these subsumption tests and is supposed to correspond to the number of clauses generated, which is usually used for a measure.

TABLE 2

Theorem	successful unifications	proof depth limited	time	unification attempts	stack entries generated
L1, L2, G3 ⊃ L3	15	no	0.3	91	1
	8	yes	0.1	13	1
L1, L2, L3, G3 ⊃ L4	4	no	0.1	15	0
	8	yes	0.1	13	0
L1, L2, G3 ⊃ Axiom 2	17	yes	0.2	53	2
	1001	yes	12.6	4048	39
L1, L2, G3 ⊃ inverse	77	no	0.7	257	3
A1, A2 ⊃ G1	14	no	0.2	42	2
	8	yes	0.1	14	2
A1, A2 ⊃ G2	14	no	0.2	40	2
	7	yes	0.1	10	1
L1, L2, A1, A2 ⊃ L3	144	no	2.1	773	6
	9	yes	0.1	16	1
L1, L2, L3, A1, A2 ⊃ L4	121	no	2.0	766	13
	614	yes	6.0	1654	14
A1, A2 ⊃ G3	42	no	0.5	154	3
	140	yes	1.0	302	3
A ⊃ G1	39	yes	0.8	256	16
	296	yes	2.6	447	5
L1, A ⊃ L2	4	no	0.1	7	0
	7	yes	0.1	13	0
A ⊃ A2	225	no	2.9	826	7
	503	yes	7.5	2038	7
A2, A ⊃ A1	48	no	0.7	252	10
	15	yes	0.1	23	2
A′ ⊃ G1	1000	yes	0.9	189	7
	>2585	yes	>29.5	>3982	>28
L1, A′ ⊃ L2	10	no	0.1	39	1
	59	yes	0.5	140	3
L1, L2, A′ ⊃ A2	11	no	0.2	65	4
	9	yes	0.1	17	1
L1, L2, A2, A′ ⊃ A1	79	no	1.4	611	15
	11	yes	0.1	16	1

TABLE 2—*continued*

Theorem	successful unifications	proof depth limited	time	unification attempts	stack entries generated
$A'' \supset G1$	553 >2904	yes yes	6.9 >29.4	1296 >4285	31 >28
$L1, A'' \supset L2$	21 591	no yes	0.3 4.2	76 1165	4 10
$L1, L2, A'' \supset A2$	7 15	no yes	0.1 0.1	19 34	1 1
$A2, A'' \supset A1$	23 3091	yes yes	0.2 19.5	66 4754	3 28
$A1, A2 \supset A$	25 12	no yes	0.3 0.1	67 18	2 2
$A1, A2 \supset A'$	17 184	no yes	0.2 1.3	36 350	2 6
$A1, A2 \supset A''$	146 260	no yes	1.7 1.9	557 515	11 7
L1, Axiom $1 \supset G2$	8 59	no yes	0.1 0.3	35 135	4 4
L1, Axiom 1, Axiom 2, Axiom $3 \supset e = 1$	73 164	yes yes	0.8 1.4	321 444	1 1
L2, Axiom 2, Axiom $3 \supset L3'$	35 44	yes yes	0.5 0.6	174 213	1 2
L1, L2, L3′, Axiom $2 \supset L3$	>4490	yes	>29.4	>9754	>3
L1, L2, L3, L3′, Axiom $2 \supset G3$	2053 >3940	yes yes	25.8 >29.4	8622 >9891	2 >3
L2, G3, comm., Axiom 3′, Axiom $2 \supset A2$	75 13	no yes	1.0 0.1	326 23	4 2
A2, G3, Axiom 3′, Axiom $2 \supset A$	11 23	no yes	0.1 0.2	39 53	2 2
$L1, L2, A2, G3 \supset X1$	63 83	yes yes	0.8 0.6	249 158	3 6
$L1, X1 \supset A$	193 228	yes yes	2.7 1.6	1154 409	23 6
$A1, A2 \supset X2$	15 57	yes yes	0.2 0.4	33 120	1 5

For each proof we also indicate whether an (optimal) limit on proof depth and functional nesting was applied (functional nesting was never limited for Alan's examples). The number of stack entries generated (for better explanation see Brand (1973)) is the number of deductions that were allowed to enter the proof process. It is a rather psychological measure—how many clauses a person would have to examine to follow the steps of the prover.

For every theorem, the first number gives the proof without fully modifying the formula. The second is with full modification. Never was a shorter proof extracted using the fully modified formula, but the generated proof may be shorter, due to the limit on proof depth and different ordering of the clauses.

If no proof was obtained, it is indicated by the "greater than" sign, e.g., >1000 successful unifications.

We do not have enough information to make any reasonable comparison with results in the literature. Alan et al. indicate only how fast proofs were obtained on the average. This ranges from "immediate" to 50 seconds (which was apparently their time limit).

Huet gives execution times for each example and number of clauses generated. It is not, however, quite clear how much work is done before a clause is generated. The number of clauses generated, given in the table below, does not include the input clauses and will correspond to our "successful unifications" in the case when Huet generates a clause from every successful unification.

Both Alan's and Huet's theorem provers are interactive and written in Lisp, whereas our program works off-line and is written in Spitbol (Snobol). Therefore comparing execution times is meaningless. (I understand that all Alan's and Huet's results are from runs without essential human interaction, except Huet's example 13.)

Huet gives results from examples 4, 9, 10 from two runs with different parameters. He proves example 12a′ rather than 12a because introducing the constant c makes his prover perform better.

It is interesting to note that even though Huet's program is designed for equality he apparently finds that "pulling terms out" works better than paramodulation. This is probably why he chose to write in example 3 ($x \cdot I(y) \neq z \vee \bar{S}x \vee \bar{S}y \vee \bar{S}z$) instead of ($\bar{S}x \vee \bar{S}y \vee S(x \cdot I(y))$), and similarly in example 6. Also he found example 6 harder when using equality in comparison with expressing $x \cdot y = z$ as a predicate and thus avoiding equality.

7. Conclusions. There is no doubt about the theoretical value of the modification method and the proof technique presented—it allowed us to prove completeness of paramodulation without the functionally reflexive axioms. Also experimental results are encouraging. The method's main advantages are its weakness (hence less chance of irrelevant inferences), and completeness with set-of-support strategy (paramodulation with set-of-support needs the functionally reflexive axioms for completeness). The main weaknesses are the presented treatment of symmetry (usually it is not necessary to consider both sides of every equation), and the oblivion of handling a well understood predicate (some obvious simplifications should be performed in the course of a proof).

Appendix A. Example where the modification method is weaker than paramodulation. From

$$S = \{a = b, Pfa, (\bar{P}x \vee Q(x, x)), x = x\}$$

it is possible to derive $Q(fa, fb)$ using paramodulation. From the modified set

$$\tilde{S} = \{b \neq w \vee a = w), (a \neq w \vee b = w), (a \neq w \vee Pfw), (\bar{P}x \vee Q(x, x)), x = x\},$$

resolution can derive $Q(fa, fa)$ or $Q(fb, fb)$, but not $Q(fa, fb)$.

Remark. It is not necessary to pull fw out of Pfw because $fw \neq u$ could be resolved only with $x = x$.

The reason for this weaker deductive power is the following. Paramodulation says: If $a = b$, replace *any* occurrance of a by b. The modification method amounts to a rule saying: If $a = b$, replace *all* occurrances of a by b, and we can escape this fact neither by generating companions for each occurrance of a (rather than one companion for all of them) nor by pulling out variables to any finite depth.

Acknowledgment. I would like to thank my advisor, Professor S. A. Cook, for his valuable guidance.

REFERENCES

J. ALAN, D. LUCKHAM AND G. MORELLI (1972), DSK: Report (1, JJM), Nov., unpublished.

D. BRAND (1973), *Resolution and equality in theorem proving*, Tech. Rep. No. 58, Department of Computer Science, University of Toronto.

C. L. CHANG AND R. C. T. LEE (1973), *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York.

J. L. DARLINGTON (1968), *Automatic theorem proving with equality substitution and mathematical induction*, Machine Intelligence, 3, pp. 113–127.

H. B. ENDERTON (1972), *A Mathematical Introduction to Logic*, Academic Press, New York.

G. P. HUET (1971), *Experiments with an interactive prover for Logic with Equality*, Rep. 1106, Jenning Computing Center, Case Western Reserve University, Cleveland, Ohio.

D. C. JENSEN AND T. PIETRZYKOWSKI (1972), *A complete mechanization of ω-order type theory*, Report CSRR 2060, Dept. of Applied Analysis and Computer Science, University of Waterloo, Waterloo, Ontario, Canada.

J. A. ROBINSON (1965a), *A machine oriented logic based on the resolution principle*, J. Assoc. Comput. Mach., 12, pp. 23–41.

——— (1965b), *Automatic deduction with hyper-resolution*, International J. Comput. Math., 1, pp. 227–234.

L. WOS AND G. A. ROBINSON (1969), *Paramodulation and theorem proving in first order theories with equality*, Machine Intelligence, 4, pp. 135–150.

SIAM J. Comput.
Vol. 4, No. 4, December 1975

RELATIVIZATIONS OF THE $\mathscr{P} =? \mathscr{NP}$ QUESTION*

THEODORE BAKER†, JOHN GILL‡ AND ROBERT SOLOVAY¶

Abstract. We investigate relativized versions of the open question of whether every language accepted nondeterministically in polynomial time can be recognized deterministically in polynomial time. For any set X, let $\mathscr{P}^X$ (resp. $\mathscr{NP}^X$) be the class of languages accepted in polynomial time by deterministic (resp. nondeterministic) query machines with oracle X. We construct a recursive set A such that $\mathscr{P}^A = \mathscr{NP}^A$. On the other hand, we construct a recursive set B such that $\mathscr{P}^B \neq \mathscr{NP}^B$. Oracles X are constructed to realize all consistent set inclusion relations between the relativized classes $\mathscr{P}^X$, $\mathscr{NP}^X$, and co $\mathscr{NP}^X$, the family of complements of languages in $\mathscr{NP}^X$. Several related open problems are described.

Key words. computational complexity, nondeterministic computation, query machines, polynomial-bounded computation

1. Introduction. An important problem in the theory of computation is to characterize the power of nondeterministic computation. A fundamental open question is whether $\mathscr{NP}$ properly contains $\mathscr{P}$. Here $\mathscr{P}$ is the class of languages recognized in polynomial time by deterministic Turing machines, and $\mathscr{NP}$ is the class of languages accepted in polynomial time by nondeterministic Turing machines. One reason for the importance of the $\mathscr{P} =? \mathscr{NP}$ question is that $\mathscr{P}$ and $\mathscr{NP}$ are very natural classes of languages, invariant under reasonable changes of machine model. $\mathscr{P}$ or $\mathscr{NP}$ is the same class whether defined by computations by one-tape Turing machines, multitape Turing machines, or random-access machines. The $\mathscr{P} =? \mathscr{NP}$ question thus deals with the basic nature of computation and not merely with minor aspects of our models of computers.

We can formulate a question similar to $\mathscr{P} =? \mathscr{NP}$ for other models of mathematical computers. In particular, we can relativize the $\mathscr{P} =? \mathscr{NP}$ question to the case of machines which compute with the aid of an oracle. When the oracle answers membership questions about sets of binary strings, the resulting machine class is formally quite similar to the class of Turing machines without oracle. The main result of this paper is that the relativized $\mathscr{P} =? \mathscr{NP}$ question has an affirmative answer for some oracles but a negative answer for other oracles.

We feel that this is further evidence of the difficulty of the $\mathscr{P} =? \mathscr{NP}$ question. By slightly altering the machine model, we can obtain differing answers to the relativized question. This suggests that resolving the original question requires careful analysis of the computational power of machines. It seems unlikely that ordinary diagonalization methods are adequate for producing an example of a language in $\mathscr{NP}$ but not in $\mathscr{P}$; such diagonalizations, we would expect, would apply equally well to the relativized classes, implying a negative answer to all relativized $\mathscr{P} =? \mathscr{NP}$ questions, a contradiction. On the other hand, we do not

* Received by the editors July 16, 1973, and in final revised form October 21, 1974. This work was supported by JSEP Contract N-00014-67-A-0112-0044 and NSF Grant GK-43121 at Stanford University, and NSF Grant GP-38753X at the University of California, Berkeley.

† Department of Mathematics, Florida State University, Tallahassee, Florida 32306.

‡ Department of Electrical Engineering, Stanford University, Stanford, California 94305.

¶ Department of Mathematics, University of California, Berkeley, California 94720.

feel that one can give a general method for simulating nondeterministic machines by deterministic machines in polynomial time, since such a method should apply as well to relativized machines and therefore imply affirmative answers to all relativized $\mathscr{P} =? \mathscr{NP}$ questions, also a contradiction. Our results suggest that the study of natural, specific decision problems offers a greater chance of success in showing $\mathscr{P} \neq \mathscr{NP}$ than constructions of a more general nature.

Our model for computation with the aid of an oracle is the query machine, which is an extension of the multitape Turing machine as described in Hopcroft and Ullman [2]. A *multitape Turing machine* consists of a finite-state control unit, a read-only input tape, a finite number of worktapes, and optionally a write-only output tape. A *query machine*, described by Cook [1], is a multitape Turing machine with a distinguished worktape, called the *query tape*, and three distinguished states, called the *query* state, the *yes* state, and the *no* state.

The action of a query machine is similar to that of a Turing machine with the following extension. When a query machine enters its query state, the next operation of the machine is determined by an oracle. An oracle for a set X will place the query machine into its *yes* state if the binary string written on the query tape is an element of X; otherwise the oracle places the machine into the *no* state. Since there is no chance for confusion, we will identify an oracle for a set X with the set X itself.

A query machine is *deterministic* if its finite control specifies at most one possible operation for each configuration of the machine; otherwise the machine is *nondeterministic*. Certain states of a query machine's finite control are designated as *accepting* states. A language is *recognized* by a deterministic query machine with oracle X if the machine halts on all inputs and halts in an accepting state just when the input string belongs to the language. The language *accepted* by a nondeterministic query machine with oracle X is the set of input strings for which some possible computation of the machine halts in an accepting state.

A query machine is *polynomial-bounded* if there is a polynomial $p(n)$ such that every computation of the machine on every input of length n halts within $p(n)$ steps, whatever oracle X is used. For any oracle X, we denote by $\mathscr{P}^X$ the class of languages recognized by polynomial-bounded deterministic query machines with oracle X, and we denote by $\mathscr{NP}^X$ the class of languages accepted by polynomial-bounded nondeterministic query machines with oracle X. The class of languages whose complements are in $\mathscr{NP}^X$ is denoted by co $\mathscr{NP}^X$.

Every query machine can be converted to a polynomial-bounded machine by attaching a clock which terminates every computation of the machine that exceeds some predetermined polynomial time bound. We can thereby produce a list of polynomial-bounded query machines which perform all possible polynomial-bounded computations. We will denote by P_i the ith deterministic polynomial-bounded machine in this list and by NP_i the ith nondeterministic polynomial-bounded query machine. Without loss of generality, we can assume that $p_i(n)$ is a strict upper bound on the length of any computation by P_i or NP_i with oracle X on an input of length n, where $p_i(n) = i + n^i$. We indicate by use of a superscript when an oracle has been specified for a query machine; thus NP_i^X and P_i^X denote query machines using oracle X.

We encode each finite sequence of binary strings $x_1, x_2, \cdots, x_m$ into the binary string $\langle x_1, x_2, \cdots, x_m \rangle$ that is obtained from the string $x_1 * x_2 * \cdots * x_m$ (over the alphabet $\{0, 1, *\}$) by replacing each occurrence of 0, 1, and $*$ by 00, 01, and 11, respectively. Both the encoding and decoding can be performed in time bounded above by a linear function of $|x_1| + |x_2| + \cdots + |x_m|$. Note that $|x_i| \leqq |\langle x_1, x_2, \cdots, x_m \rangle|$ for every $i \leqq m$.

Consider the canonical enumeration of binary strings: Λ, 0, 1, 00, 01, 10, 11, 000, $\cdots$. When convenient, we will identify the natural number i with the ith string in this enumeration. As usual, if x is a binary string, then x^n denotes x concatenated with itself n times.

In § 2 we shall construct an oracle A such that $\mathscr{P}^A = \mathscr{NP}^A$. By contrast, in § 3 we shall construct an oracle B such that $\mathscr{P}^B \neq \mathscr{NP}^B$. We shall also show that many other relations between the relativized $\mathscr{P}$, $\mathscr{NP}$, and co $\mathscr{NP}$ classes can be shown to hold for certain oracles. In § 4, we describe open problems concerned with the relativization of the Meyer–Stockmeyer $\mathscr{P}$-hierarchy [6].

Some of the results of this paper have been discovered independently by M. J. Fischer, R. Ladner, A. R. Meyer, and H. B. Hunt III. We have acknowledged these independent contributions in the body of the paper.

2. $\mathscr{P} = \mathscr{NP}$, relativized.

In this section we construct a recursive oracle A such that $\mathscr{P}^A = \mathscr{NP}^A$. We also prove that if A is any polynomial-space complete language, then $\mathscr{P}^A = \mathscr{NP}^A$. We conclude the section by showing that whenever $\mathscr{P}^A = \mathscr{NP}^A$, there is a deterministic procedure using oracle A which finds an accepting computation of NP_i^A on input x, provided NP_i^A accepts x; the running time of this procedure is bounded above by a polynomial of the maximum computation time of NP_i^A on input x.

Our first observation is that for every oracle X, the class $\mathscr{NP}^X$ contains polynomial-complete sets.

DEFINITION. Let $\mathscr{S}$ be a class of languages. A set K in $\mathscr{S}$ is *Cook-polynomial-complete* (or simply Cook-complete) in $\mathscr{S}$ if every language in $\mathscr{S}$ can be recognized by a polynomial-bounded deterministic query machine using oracle K. A set K in $\mathscr{S}$ is *Karp-polynomial-complete* (or Karp-complete) if for every set S in $\mathscr{S}$ there is a function $f(x)$ computable in polynomial time such that $x \in S \Leftrightarrow f(x) \in K$.

Every Karp-complete set is Cook-complete. Ladner, Lynch, and Selmon [4] compare the polynomial-bounded reducibilities of Cook and Karp. Note that if K is Cook-complete in $\mathscr{S}$, then $\mathscr{S} \subseteq \mathscr{P}^K$.

LEMMA 1. *For any oracle X, define the language $K(X)$ to be $\{\langle i, x, 0^n \rangle$: some computation of NP_i^X accepts x in fewer than n steps$\}$. Then $K(X)$ is Karp-complete in $\mathscr{NP}^X$. In particular, $\mathscr{P}^X = \mathscr{NP}^X$ if and only if $K(X) \in \mathscr{P}^X$.*

Proof. Clearly $K(X) \in \mathscr{NP}^X$. Now suppose $S \in \mathscr{NP}^X$, say S is accepted by NP_i^X. If we let $f(x) = \langle i, x, 0^{p_i(|x|)} \rangle$, then $f(x)$ is computable in polynomial time. Now $x \in S \Leftrightarrow NP_i^X$ accepts $x \Leftrightarrow NP_i^X$ accepts x in $< p_i(|x|)$ steps $\Leftrightarrow \langle i, x, 0^{p_i(|x|)} \rangle \in K(X)$. Since S was arbitrary, we conclude that $K(X)$ is Karp-complete in $\mathscr{NP}^X$.

Clearly, if $\mathscr{NP}^X = \mathscr{P}^X$, then $K(X) \in \mathscr{P}^X$. Conversely, suppose $K(X) \in \mathscr{P}^X$. Then $\mathscr{P}^{K(X)} \subseteq \mathscr{P}^X$, and $\mathscr{NP}^X \subseteq \mathscr{P}^{K(X)}$ because $K(X)$ is Cook-complete in $\mathscr{NP}^X$. Therefore $\mathscr{NP}^X \subseteq \mathscr{P}^X$ Q.E.D.

Remark. If $\tilde{K}(X)$ is defined by $\{\langle i, x \rangle : NP_i^X$ accepts $x\}$, then $\tilde{K}(x)$ does not belong to $\mathscr{NP}^X$, since there is no uniform polynomial bound on running time of machines NP_i. Thus, although every language in $\mathcal{NP}^X$ can be reduced to $\tilde{K}(X)$ in polynomial time, $\tilde{K}(X)$ is not polynomial-complete.

The following result was also discovered, independently, by Albert Meyer with Michael Fischer and by H. B. Hunt III.

THEOREM 1. *There is an oracle A such that $\mathcal{P}^A = \mathcal{NP}^A$.*

Proof. We construct an oracle A such that $A = K(A)$. Let $A = \{\langle i, x, 0^n\rangle : NP_i^A$ accepts x in $< n$ steps$\}$. This is a valid inductive definition of a set. In a computation of length $< n$, no string of length $\geqq n$ can be queried. To simulate NP_i^A on input x for $< n$ steps, we need know only which elements of length $< n \leqq |\langle i, x, 0^n\rangle|$ belong to A. Therefore A is well-defined, and by definition $A = K(A)$. Since $K(A) = A \in \mathcal{P}^A$, we conclude $\mathcal{P}^A = \mathcal{NP}^A$ by Lemma 1. Q.E.D.

Remarks. Kleene's recursion theorem can also be used to produce a recursive oracle A such that $A = K(A)$. The oracle A constructed in Theorem 1 can be recognized deterministically in exponential time.

We shall next show that there are naturally occurring languages A such that $\mathcal{P}^A = \mathcal{NP}^A$.

A language is said to be *recognizable in polynomial space* if there is a polynomial $p(n)$ and a deterministic Turing machine which recognizes the language and uses no more than $p(n)$ worktape squares on any input of length n. We denote by $\mathcal{PS}$ the family of languages recognizable in polynomial space. We could define $\mathcal{NPS}$ to be the family of languages accepted in polynomial space by nondeterministic Turing machines. But this definition is unnecessary because Savitch [9] has shown that $\mathcal{NPS} = \mathcal{PS}$; every language accepted nondeterministically in space $p(n)$ can be recognized deterministically in space $p(n)^2$.

A language S is *log-space reducible* to A if there is a function $f(x)$ computable in space $\log(|x|)$ such that $x \in S \Leftrightarrow f(x) \in A$. The language A is *polynomial-space complete* if every language in $\mathcal{PS}$ is log-space reducible to A. Since every function computable in log space is computable in polynomial time, every polynomial-space complete language is Karp-complete in $\mathcal{PS}$. If A is polynomial-space complete, then $\mathcal{PS} = \mathcal{P}^A$.

One example of a polynomial-space complete language is $1EQ$, the set of valid sentences in the first-order theory of equality. Other natural examples of word problems which represent polynomial-space complete languages are given by Stockmeyer and Meyer [10]. An artificial example of a polynomial-space complete language is $A = \{\langle i, x, 0^n\rangle$: deterministic Turing machine P_i recognizes x in space $< n\}$.

THEOREM 2. *If A is polynomial-space complete, then $\mathcal{P}^A = \mathcal{NP}^A$.*

Proof. Suppose A is polynomial-space complete. Then $A \in \mathcal{PS}$ and $\mathcal{PS} \subseteq \mathcal{P}^A$. Also $\mathcal{NP}^A \subseteq \mathcal{NPS}$, since every query made of the oracle for A can be answered in polynomial space without recourse to the oracle for A. But $\mathcal{NPS} = \mathcal{PS}$. Therefore $\mathcal{NP}^A \subseteq \mathcal{NPS} = \mathcal{PS} \subseteq \mathcal{P}^A$. Q.E.D.

When $\mathcal{P}^A = \mathcal{NP}^A$, then every language S in $\mathcal{NP}^A$ can be recognized in polynomial time by some deterministic query machine with oracle A. In fact, this deterministic machine P_j^A can be constructed so that it "simulates" a nondeterministic machine NP_i^A that accepts S; whenever NP_i^A on input x reaches a nondeterministic branch point in its computation, the simulating machine P_j^A correctly decides which branch to follow.

LEMMA 2. *Suppose A is an oracle such that $\mathscr{P}^A = \mathscr{NP}^A$. Then for each nondeterministic query machine NP_i there is a deterministic query machine P_j such that P_j^A on input x produces as output an accepting computation of NP_i^A on input x, whenever NP_i^A accepts x.*

Proof. A computation of a query machine NP_i^X is a sequence of instantaneous descriptions of the machine. An instantaneous description is an encoding of the total configuration of the machine, including the state of the finite control, contents of the tapes, and locations of tape heads. A sequence $I_0, I_1, \cdots, I_m$ of instantaneous descriptions of NP_i^X represents a computation of NP_i^X if I_k encodes a configuration of NP_i^X which can be reached in a single step from the configuration encoded by I_{k-1}.

We define COMP (X) to be the set of accepting computations of query machines with oracle X. Specifically, COMP $(X) = \{\langle i, x, 0^{p_i(|x|)}, I_0, I_1, \cdots, I_m\rangle :$ $I_0, I_1, \cdots, I_m$ is an accepting computation of NP_i^X on input $x\}$. Note that COMP $(X) \in \mathscr{P}^X$, and $m < p_i(|x|)$.

Let INIT (X) be the set of partial computations which are the initial parts of accepting computations by machines with oracle X; that is,

$$\text{INIT}(X) = \{\langle i, x, 0^{p_i(|x|)}, I_0, I_1, \cdots, I_k\rangle : \text{there exist } I_{k-1}, \cdots, I_m \text{ such that } m < p_i(|x|) \text{ and } \langle i, x, 0^{p_i(|x|)}, I_0, I_1, \cdots, I_m\rangle \in \text{COMP}(X)\}.$$

Then INIT $(X) \in \mathscr{NP}^X$.

If $\mathscr{P}^A = \mathscr{NP}^A$, then INIT$(A)$ can be recognized by some deterministic polynomial-bounded query machine with oracle A. Now suppose NP_i^A accepts x. Then $\langle i, x, 0^{p_i(|x|)}, I_0\rangle \in$ INIT (A), where I_0 is the instantaneous description of the initial configuration of NP_i^A with input x. We can find an accepting computation of NP_i^A on x as follows:

We wish to determine a computation $I_0, I_1, \cdots, I_m$ such that $m < p_i(|x|)$ and $\langle i, x, 0^{p_i(|x|)}, I_0, I_1, \cdots, I_k\rangle \in$ INIT (A) for each $k \leqq m$. To find I_k, suppose we have already found $I_0, I_1, \cdots, I_{k-1}$ with $\langle i, x, 0^{p_i(|x|)}, I_0, I_1, \cdots, I_{k-1}\rangle$ $\in$ INIT (A).

Determine in polynomial time an instantaneous description I_k such that $\langle i, x, 0^{p_i(|x|)}, I_0, \cdots, I_{k-1}, I_k\rangle \in$ INIT (A). (There are only finitely many possibilities for I_k, since I_k must be the instantaneous description of a configuration of NP_i^A reachable in one step from the configuration described by I_{k-1}.) Since NP_i^A accepts x, we have $\langle i, x, 0^{p_i(|x|)}, I_0, I_1, \cdots, I_m\rangle \in$ COMP (A) for some $m < p_i(|x|)$. Therefore we will find an accepting computation of NP_i^A on input x in a number of steps at most a polynomial of $p_i(|x|)$. Q.E.D.

We shall use the method of the proof of Lemma 2 in the proof of Theorem 6.

3. $\mathscr{P} \neq \mathscr{NP}$, relativized. In this section we show that there exist recursive oracles X such that $\mathscr{P}^X \subsetneq \mathscr{NP}^X$. ($\subsetneq$ denotes proper containment.) We shall construct recursive sets B, C, D, E, and F such that

(i) $\mathscr{P}^B \neq \mathscr{NP}^B$;

(ii) $\mathscr{NP}^C$ is not closed under complementation;

(iii) $\mathscr{P}^D \neq \mathscr{NP}^D$ but $\mathscr{NP}^D$ is closed under complementation;
(iv) $\mathscr{P}^E \neq \mathscr{NP}^E$ and $\mathscr{P}^E = \mathscr{NP}^E \cap \text{co}\,\mathscr{NP}^E$;
(v) $\mathscr{P}^F \subsetneq \mathscr{NP}^F \cap \text{co}\,\mathscr{NP}^F$ and $\mathscr{NP}^F \neq \text{co}\,\mathscr{NP}^F$.

(Recall that co $\mathscr{NP}^X$ is the class of languages whose complements belong to $\mathscr{NP}^X$.) By suitable choices of oracles, all consistent set inclusion relations among the relativized classes $\mathscr{P}$, $\mathscr{NP}$, and co $\mathscr{NP}$ can be realized.

For any oracle X, define the language $L(X) = \{x\colon \text{there is } y \in X \text{ such that } |y| = |x|\}$. Clearly $L(X) \in \mathscr{NP}^X$. In fact, $L(X)$ can be accepted in linear time by a query machine that writes on its query tape a nondeterministically chosen string y of the same length as input x, then accepts x if and only if y belongs to X.

The following result was obtained independently by Richard Ladner. Although the next theorem follows immediately from Theorems 4–7, we include a direct proof in order to illustrate the basic techniques of this section.

THEOREM 3. *There is an oracle B such that $\mathscr{P}^B \neq \mathscr{NP}^B$.*

Proof. We construct a set B such that $L(B)$ does not belong to $\mathscr{P}^B$. The construction of B proceeds in stages. We denote by $B(i)$ the finite set of strings placed into B prior to stage i. Recall that $p_i(n)$ is an upper bound on the lengths of computations by P_i^X and NP_i^X for all oracles X and all inputs of length n. Let $n_0 = 0$.

Stage i. Choose $n > n_i$ so large that $p_i(n) < 2^n$. Run query machine P_i with oracle $B(i)$ on input $x_i = 0^n$. If $P_i^{B(i)}$ accepts 0^n, then place no strings into B at this stage. Otherwise, if $P_i^{B(i)}$ rejects 0^n, then add to B the least string (that is, the earliest occurring string in the canonical enumeration of binary strings) of length n not queried during the computation of $P_i^{B(i)}$ on input 0^n. (Such a string exists because not every string of length n can be queried by $P_i^{B(i)}$ on 0^n; in $p_i(n)$ steps, $P_i^{B(i)}$ can ask at most $p_i(n)$ questions, and we have chosen n so that $p_i(n) < 2^n =$ the number of strings of length n.) Finally, let $n_{i+1} = 2^n$. (This will ensure that no string of length $\leqq 2^n$ is added to B at a later stage.) Go to the next stage.

The computation of P_i on input x_i is the same whether B or $B(i)$ is used as oracle, because no string queried by $P_i^{B(i)}$ on input x_i is later added to or deleted from B. At stage i, we ensure that P_i^B does not recognize $L(B)$; by construction, $P_i^{B(i)}$ and hence P_i^B rejects x_i iff some string of length $|x_i|$ belongs to B, that is, iff $x_i \in L(B)$. Therefore $L(B)$ does not belong to $\mathscr{P}^B$. Q.E.D.

The set B constructed in the proof of Theorem 3 is sparse; for every n, there is at most one string x in B such that $n \leqq |x| < 2^n$. An obvious modification of the proof of Theorem 3 yields the following result: there are arbitrarily sparse recursive sets such that $\mathscr{P}^B \neq \mathscr{NP}^B$.

Richard Ladner has shown that there are oracles B recognizable deterministically in exponential time such that $\mathscr{P}^B \neq \mathscr{NP}^B$.

The proof of Theorem 3 makes use of the fact that we can query an oracle about any number of length n in approximately n steps. We can therefore conclude that $L(B) \in \mathscr{NP}^B$. Alternate models for query machines involve the notion of *oracle tapes.* In one model, a query machine is supplied with a tape on which are written the strings in the oracle set, separated by asterisks and listed in increasing order. A variation of this model is obtained by supplying the query machine with an oracle tape on which is written the characteristic function of the oracle set. With these models of query machines, our proofs of Theorems 1–3 are no longer valid. Paul Morris [7] has pointed out that with the oracle-tape models for query machines, if $\mathscr{P} = \mathscr{NP}$, then $\mathscr{P}^X = \mathscr{NP}^X$ for every oracle X.

From Theorems 1 and 3 we see that the answer to the question $\mathscr{P}^X =? \mathscr{NP}^X$ depends on the oracle X. Kurt Mehlhorn [5] has shown that the family of oracles X for which $\mathscr{P}^X = \mathscr{NP}^X$ is a "meagre" set in a space of recursive oracles. In this sense, "most" oracles satisfy $\mathscr{P}^X \neq \mathscr{NP}^X$.

The next result follows from Theorem 6, but again the direct proof is much simpler.

THEOREM 4. *There is an oracle C such that $\mathscr{NP}^C$ is not closed under complementation.*

Proof. By a construction very similar to that of Theorem 3, we generate an oracle C such that $\mathscr{NP}^C$ does not contain $\bar{L}(c)$, the complement of $L(C)$. Let $C(i)$ be the set of string placed into C before stage i, and let $n_0 = 0$.

Stage i. Choose $n > n_i$ so that $p_i(n) < 2^n$. (Thus n is greater than the length of every string queried earlier in this construction.) Run nondeterministic query machine NP_i with oracle $C(i)$ on input $x_i = 0^n$. If $NP_i^{C(i)}$ accepts 0^n, then choose any accepting computation and place into C some string of length n not queried during this computation. Otherwise, place no string into C at this stage. Let $n_{i+1} = 2^n$, and go to next stage.

By construction, NP_i^C accepts x_i iff some string of length $|x_i|$ belongs to C, that is, iff $x_i \in L(C)$. Consequently NP_i^C does not accept $\bar{L}(C)$. Therefore $\bar{L}(C)$ does not belong to $\mathscr{NP}^C$. Q.E.D.

The next result is due to Albert Meyer with Michael Fischer and, independently, to Richard Ladner.

THEOREM 5. *There is an oracle D such that $\mathscr{P}^D \neq \mathscr{NP}^D$ but $\mathscr{NP}^D$ is closed under complementation.*

Proof. It is easily seen that $\mathscr{NP}^X$ is closed under complementation if and only if $\bar{K}(X) \in \mathscr{NP}^X$, where $\bar{K}(X)$ is the complement of the Karp-complete language $K(X)$. We shall construct an oracle D such that (i) $L(D) \in \mathscr{NP}^D - \mathscr{P}^D$ and (ii) $u \in \bar{K}^D$ iff u is a prefix of some string v in D such that $|v| = 2|u|$. Then $\mathscr{P}^D \neq \mathscr{NP}^D$ from (i); and $\bar{K}(D) \in \mathscr{NP}^D$ from (ii) and so $\mathscr{NP}^D$ is closed under complementation.

At stage n in the construction, we decide the membership in D of all strings of length n. In the course of the construction, some strings will be *reserved* for $\bar{D}$, that is, designated as nonmembers of D. An index i will be *cancelled* at some stage when we ensure that P_i^D does not recognize $L(D)$. As usual, $D(n)$ denotes those strings placed into D prior to stage n.

Stage $n = 2m$. For every string z of length $n = 2m$ not reserved for $\bar{D}$ at an earlier stage, determine the prefix u of z of length m. If u encodes a triple $\langle i, x, 0^l \rangle$, then place z into D iff $NP_i^{D(n)}$ does not accept x in fewer than l steps.

Stage $n = 2m + 1$. Let i be the least uncancelled index. If any string of length $\geqq n$ has been reserved for $\bar{D}$, or if $p_i(n) \geqq 2^m$, then add no elements to D at this stage. Otherwise, run P_i with oracle $D(n)$ on input 0^n and reserve for $\bar{D}$ all strings of length $\geqq n$ queried during this computation. If $P_i^{D(n)}$ accepts 0^n, then add no elements to D. But if $P_i^{D(n)}$ rejects 0^n, then add to D the least string of length n not queried (and so not reserved for $\bar{D}$). Finally cancel index i.

Every index is eventually cancelled, and when index i is cancelled at some stage, we have guaranteed that P_i^D does not recognize $L(D)$. Therefore $L(D) \in \mathcal{NP}^D - \mathcal{P}^D$.

At any odd stage $2m + 1$, at most $p_i(n) < 2^m$ strings are reserved for $\bar{D}$, and so fewer than $2^0 + 2^1 + \cdots + 2^{m-1} < 2^m$ strings of length $2m$ can be reserved for $\bar{D}$ at odd stages before stage $2m$. Therefore every string u of length m is the prefix of at least one string v of length $2m$ which is never reserved for $\bar{D}$. By construction, $u \in \bar{K}(D)$ iff u prefixes a number $v \in D$ of length $2|u|$, and so $\bar{K}(D) \in \mathcal{NP}^D$. Q.E.D.

The next theorem gives an answer to a question posed by Albert Meyer.

THEOREM 6. *There is an oracle E such that* $\mathcal{P}^E \neq \mathcal{NP}^E$ *and* $\mathcal{P}^E = \mathcal{NP}^E \cap \text{co}\,\mathcal{NP}^E$.

Proof. The construction of E is considerably more complicated than that of the preceding oracles. As usual, we guarantee that $L(E)$ is not in $\mathcal{P}^E$ and so $\mathcal{P}^E \neq \mathcal{NP}^E$. But we must also ensure that whenever both S and its complement $\bar{S}$ belong to $\mathcal{NP}^E$, then S in fact is in $\mathcal{P}^E$.

The rough idea of the proof is the following. By adding infinitely many new elements to an oracle A such that $\mathcal{P}^A = \mathcal{NP}^A$, we obtain an oracle E such that $\mathcal{P}^E \neq \mathcal{NP}^E$. Now if we could quickly recognize the set $E - A$, then $\mathcal{P}^E = \mathcal{NP}^E$. Although we cannot quickly recognize the entire set $E - A$, we can arrange the construction so that whenever both S and its complement $\bar{S}$ are accepted in polynomial time by nondeterministic query machines with oracle E, then we can quickly recognize relevant portions of the set $E - A$, so that we can then combine the machines accepting S and $\bar{S}$ into a deterministic machine to recognize S.

Define $e(n)$ inductively by $e(0) = 0$ and $e(n + 1) = 2^{2^{e(n)}}$. Choose any oracle A such that $\mathcal{P}^A = \mathcal{NP}^A$. Without any loss of generality, we can assume that A contains no elements of length $e(n)$ for any $n \geqq 0$. The oracle E is obtained by adding to A at stage n at most one string of length $e(n)$. Let $E(0) = A$ and let $E(n)$ denote the set of numbers placed into E before stage n. There are two types of requirements to be satisfied in the construction of E. An unsatisfied requirement $\langle i, i\rangle$ is *vulnerable* at stage n if $p_i(e(n)) < 2^{e(n)}$. An unsatisfied requirement $\langle j, k\rangle$ with $j \neq k$ is *vulnerable* at stage n if there is a string x such that $e(n - 1) < \log_2(|x|) \leqq e(n) \leqq \max\{p_j(|x|), p_k(|x|)\} < e(n + 1)$ and neither NP_j nor NP_k with oracle $E(n)$ accepts x. We agree that requirement r_1 has higher priority than requirement r_2 just when $r_1 < r_2$.

Stage n. We satisfy the requirement of highest priority that is vulnerable at stage n. To satisfy requirement $\langle j, k\rangle$ with $j \neq k$, we simply add no string to E at this stage. To satisfy requirement $\langle i, i\rangle$, we run P_i with oracle $E(n)$ on input $0^{e(n)}$. If $P_i^{E(n)}$ rejects $0^{e(n)}$, then we add to E the least string of length $e(n)$ not queried by $P_i^{E(n)}$ on input $0^{e(n)}$; otherwise we add no new element to E.

Every requirement $\langle i, i\rangle$ is eventually satisfied, since there are only finitely many requirements of higher priority. Suppose requirement $\langle i, i\rangle$ is satisfied at stage n. Then P_i^E rejects $0^{e(n)}$ iff there is a string in E of length $e(n)$, and so P_i^E does not recognize $L(E)$. Therefore $L(E) \in \mathcal{NP}^E - \mathcal{P}^E$.

Now suppose S and $\bar{S}$ belong to $\mathcal{NP}^E$. We must show that $S \in \mathcal{P}^E$. Assume NP_j^E accepts S and NP_k^E accepts $\bar{S}$. Note that requirement $\langle j, k\rangle$ is never satisfied, else there would be a string x accepted by neither NP_j^E nor NP_k^E. Let m be so large that

(i) for every x such that $|x| \geqq e(m)$ there is at most one n such that $\log_2 |x| \leqq e(n) \leqq \max \{p_j(|x|), p_k(|x|)\}$;

(ii) if a requirement of higher priority than requirement $\langle j, k\rangle$ is ever satisfied, then it is satisfied before stage m.

We can decide S deterministically in polynomial time using oracle E by the following procedure:

With input x, if $|x| < e(m)$, then use a finite table to decide if $x \in S$. Otherwise $|x| \geqq e(m)$. Calculate the least n such that $e(n) \geqq \log_2 |x|$. Determine which elements were added to E before stage n by querying the oracle E about *all* strings of lengths $e(0), e(1), \cdots, e(n-1)$. Only $O(|x|)$ strings need be queried here, since $e(n-1) \leqq \log_2 |x|$.

Now there are two cases. If $e(n) > \max \{p_j(|x|), p_k(|x|)\}$, then no computation of either NP_j^E or NP_k^E can query oracle E about any string of length $\geqq e(n)$. Therefore the computation of NP_j on input x is the same with oracle $E(n)$ as with oracle E. Since we have already calculated $E(n) - A$, we can produce a query machine $NP_{i(n)}^{E(n)}$ which simulates $NP_j^{E(n)}$ but makes no queries about strings of length $e(m)$ for any $m < n$; these queries are answered without recourse to the oracle $E(n)$ by using the finite table of elements of $E(n) - A$. Clearly $NP_{i(n)}$ on input x gives the same result with oracle A as with oracle $E(n)$. Therefore NP_j^E accepts x if and only if $NP_{i(n)}^A$ accepts x, that is, iff $\langle i(n), x, 0^{p_j(|x|)}\rangle \in K(A)$. (We can easily make sure that $NP_{i(n)}$ has the same running time bound as NP_j, and that the length of the index $i(n)$ is at most a polynomial of $|x|$.) Since $K(A)$ belongs to $\mathscr{P}^A$, we can now determine in polynomial time if NP_j^E accepts x.

In the other case, $e(n) \leqq \max \{p_j(|x|), p_k(|x|)\}$. If neither NP_j nor NP_k with oracle $E(n)$ accepted x, then requirement $\langle j, k\rangle$ would be satisfied at stage n. But requirement $\langle j, k\rangle$ can never be satisfied, for then both NP_j^E and NP_k^E would reject x. Therefore at least one of $NP_j^{E(n)}$ and $NP_k^{E(n)}$ accepts x. As in the first case, since we know $E(n) - A$, we can discover in polynomial time which machine accepts x.

Suppose, to be definite, $NP_j^{E(n)}$ accepts x. (In case $NP_k^{E(n)}$ accepts x, the argument is similar.) We must now determine if NP_j^E accepts x, where E might contain a string of length $e(n)$ not in $E(n)$.

Since $\mathscr{P}^A = \mathscr{NP}^A$ and since we have already calculated $E(n) - A$, we can use the method of Lemma 2 to find an accepting computation of $NP_j^{E(n)}$ on input x. Now we examine this computation to see if it represents a valid computation when oracle E is used instead of $E(n)$. Whenever a string y of length $e(n)$ is queried by $NP_j^{E(n)}$ on x, we consult oracle E about y. There are two subcases.

If no such y belongs to E, then the computation of $NP_j^{E(n)}$ on x is the same as the computation of NP_j^E on x. Now $NP_j^{E(n)}$ accepts x, and so NP_j^E also accepts x. We conclude that $x \in S$.

In the other subcase, $NP_j^{E(n)}$ on input x queries some string y of length $e(n)$ which belongs to E. By construction, there is at most one number in E of length $e(n)$. Thus we have correctly calculated $E(n+1) - E(n)$. Earlier we found $E(n) - A$, so we now know $E(n+1) - A$. Using the method of the first case above, we can finally determine which of machines NP_j and NP_k with oracle $E(n+1)$ accepts x. Since $e(n+1) > \max \{p_j(|x|), p_k(|x|)\}$, no number of length $\geqq e(n+1)$ can be queried by NP_j^E or NP_k^E on x, so NP_j^E accepts x iff $NP_j^{E(n+1)}$ accepts x. Q.E.D.

The next result accounts for the only remaining relation between relativized classes $\mathscr{P}$ and $\mathscr{NP}$.

THEOREM 7. *There is an oracle F such that* $\mathscr{P}^F \subsetneq \mathscr{NP}^F \cap \mathrm{co}\,\mathscr{NP}^F \subsetneq \mathscr{NP}$.

Proof. We outline the construction. Let $L_1(F) = \{x : |x| \text{ is even and there is } y \in F \text{ with } |y| = |x|\}$, and let $L_2(F) = \{x : |x| \text{ is odd and there is a string } 0y \in F \text{ with } |0y| = |x|\}$. Modify the construction of Theorem 4 so that $L_1(F) \in \mathscr{NP}^F - \mathrm{co}\,\mathscr{NP}^F$ and $L_2(F) \in \mathscr{NP}^F \cap \mathrm{co}\,\mathscr{NP}^F - \mathscr{P}^F$. To force $L_2(F) \in \mathscr{NP}^F \cap \mathrm{co}\,\mathscr{NP}^F$, we require that for every odd n there is a string $0y$ of length n in F iff there is no string $1y$ of length n in F. We omit the details. Q.E.D.

In this section we have constructed oracles X such that $\mathscr{P}^X \neq \mathscr{NP}^X$. The principal method for showing that $\mathscr{NP}^X$ properly contains $\mathscr{P}^X$ is to ensure that $L(X) \in \mathscr{NP}^X - \mathscr{P}^X$. It is easy to modify the proof of Theorem 3 to obtain an oracle X for which $L(X) \in \mathscr{NP}^X$, but every deterministic query machine with oracle X that recognizes $L(X)$ requires exponential time for all but finitely many inputs.

4. Open problems. We shall describe several open problems suggested by Albert Meyer. First we recall another characterization of $\mathscr{NP}$ [6].

LEMMA 3. *A language L belongs to* $\mathscr{NP}$ *iff there is a polynomial* $p(n)$ *and a predicate* $R(x, y)$ *in* $\mathscr{P}$ *such that* $x \in L \Leftrightarrow (\exists y)[|y| \leqq p(|x|) \,\&\, R(x, y)]$.

Proof. ($\Leftarrow$) Suppose $x \in L \Leftrightarrow (\exists y)[|y| \leqq p(|x|) \,\&\, R(x, y)]$. Then we can accept L in polynomial time by nondeterministically selecting a string y such that $|y| \leqq p(|x|)$ and accepting x if $R(x, y)$ is true.

($\Rightarrow$) If $L \in \mathscr{NP}$, then L is accepted by some query machine NP_i. Define predicate $R(x, y)$ to hold iff y encodes an accepting computation of NP_i on input x. Clearly $R(x, y)$ belongs to $\mathscr{P}$. We then choose a polynomial $p(n)$ large enough that $|y| < p(|x|)$ whenever y encodes a computation of length $\leqq p_i(|x|)$. Q.E.D.

One can draw analogies between the class $\mathscr{P}$ of languages recognizable in polynomial time and the class $\mathscr{R}$ of recursive (decidable) languages. We may consider a language "practically" decidable if it can be decided by some deterministic polynomial-bounded procedure. From Lemma 3, $\mathscr{NP}$ contains exactly those languages definable by polynomial-bounded existential quantification over predicates in $\mathscr{P}$. Similarly, co $\mathscr{NP}$ contains those languages definable by polynomial-bounded universal quantification over predicates in $\mathscr{P}$. Thus $\mathscr{NP}$ corresponds to Σ_1 in Kleene's arithmetic hierarchy [8], while co $\mathscr{NP}$ corresponds to Π_1.

Meyer and Stockmeyer [6] have defined a polynomial-bounded analogue of the arithmetic hierarchy, the $\mathscr{P}$-hierarchy. They define $\Sigma_0^{\mathscr{P}} = \Pi_0^{\mathscr{P}} = \Delta_0^{\mathscr{P}}$ to be the class $\mathscr{P}$. Then $\Sigma_{i+1}^{\mathscr{P}}$ is the class of languages definable by polynomial-bounded existential quantification over predicates in $\Pi_i^{\mathscr{P}}$; that is, $L \in \Sigma_{i+1}^{\mathscr{P}}$ iff there is a polynomial $p(n)$ and a predicate $R(x, y)$ in $\Pi_i^{\mathscr{P}}$ such that $x \in L \Leftrightarrow (\exists y)[|y| \leqq p(|x|) \,\&\, R(x, y)]$. Similarly, $\Pi_{i+1}^{\mathscr{P}}$ contains exactly those languages definable by polynomial-bounded universal quantification over predicates in $\Sigma_i^{\mathscr{P}}$. (Equivalently, $L \in \Sigma_i^{\mathscr{P}}$ iff there is a polynomial $p(n)$ and a predicate $R(x, y_1, \cdots, y_i)$ in $\mathscr{P}$ such that $x \in L \Leftrightarrow (\exists y_1)(\forall y_2) \cdots (Qy_i)[|y_1|, |y_2|, \cdots, |y_i| \leqq p(|x|) \,\&\, R(x, y_1, \cdots, y_i)]$, where there are i alternations of polynomial-bounded quantifiers.) $\Delta_{i+1}^{\mathscr{P}}$ is defined to be the class of languages recognizable in polynomial time with the aid of an oracle for some language in $\Sigma_i^{\mathscr{P}}$; that is, $L \in \Delta_{i+1}^{\mathscr{P}}$ iff $L \in \mathscr{P}^S$ for some S in $\Sigma_i^{\mathscr{P}}$. The $\mathscr{P}$-

hierarchy is $\{\Sigma_i^{\mathscr{P}}, \Pi_i^{\mathscr{P}}, \Delta_i^{\mathscr{P}} : i \geqq 0\}$.

The $\mathscr{P}$-hierarchy shares several of the properties of the arithmetic hierarchy:

(i) $L \in \Sigma_i^{\mathscr{P}} \Leftrightarrow \bar{L} \in \Pi_i^{\mathscr{P}}$;

(ii) $\Sigma_i^{\mathscr{P}} \cup \Pi_i^{\mathscr{P}} \subseteq \Delta_{i+1}^{\mathscr{P}} \subseteq \Sigma_{i+1}^{\mathscr{P}} \cap \Pi_{i+1}^{\mathscr{P}}$.

However, it is not known if any of the inclusions in (ii) are proper; the $\mathscr{P}$-hierarchy may consist of only a single class, namely, $\Sigma_0^{\mathscr{P}} = \Pi_0^{\mathscr{P}} = \Delta_0^{\mathscr{P}} = \mathscr{P}$. One of the results of [6] implies the following.

LEMMA 4 (Meyer–Stockmeyer). *If $\Sigma_i^{\mathscr{P}} = \Pi_i^{\mathscr{P}}$ for any $i \geqq 1$, then $\Sigma_j^{\mathscr{P}} = \Pi_j^{\mathscr{P}} = \Sigma_i^{\mathscr{P}}$ for every $j \geqq i$. In particular, if $\mathscr{P} = \mathscr{NP}$, then $\Sigma_i^{\mathscr{P}} = \Pi_i^{\mathscr{P}} = \Delta_i^{\mathscr{P}} = \mathscr{P}$ for every i.*

From Lemma 4 we see that proving $\Sigma_i^{\mathscr{P}} \neq \Pi_i^{\mathscr{P}}$ for any $i \geqq 1$ is a difficult problem since $\mathscr{P} \neq \mathscr{NP}$ is an immediate consequence.

It is easy to relativize the $\mathscr{P}$-hierarchy. For any oracle X, let $\Sigma_0^{\mathscr{P},X} = \Pi_0^{\mathscr{P},X} = \Delta_0^{\mathscr{P},X} = \mathscr{P}^X$. Then $\Sigma_{i+1}^{\mathscr{P},X}$ is the class of languages definable by polynomial-bounded existential quantification over predicates in $\Pi_i^{\mathscr{P},X}$; also $\Pi_{i+1}^{\mathscr{P},X}$ is the class of languages definable by polynomial-bounded universal quantification over predicates in $\Sigma_i^{\mathscr{P},X}$; and $\Delta_{i+1}^{\mathscr{P},X}$ contains languages L such that $L \in \mathscr{P}^S$ for some $S \in \Sigma_i^{\mathscr{P},X}$. The $\mathscr{P}$, X-hierarchy is $\{\Sigma_i^{\mathscr{P},X}, \Pi_i^{\mathscr{P},X}, \Delta_i^{\mathscr{P},X} : i \geqq 0\}$.

Properties (i) and (ii) hold for relativized $\mathscr{P}$-hierarchies, as does Lemma 4. For the oracle A of Theorem 1, $\mathscr{NP}^A = \mathscr{P}^A$, hence the $\mathscr{P}$, A-hierarchy collapses entirely. If D is the oracle of Theorem 5, then $\Sigma_0^{\mathscr{P},D} \subsetneqq \Sigma_1^{\mathscr{P},D} = \Pi_1^{\mathscr{P},D}$; the $\mathscr{P}$, D-hierarchy consists of only two levels. If E is the oracle of Theorem 6, then $\Sigma_0^{\mathscr{P},E} \subsetneqq \Sigma_1^{\mathscr{P},E} \subsetneqq \Sigma_2^{\mathscr{P},E}$ (since $\Pi_1^{\mathscr{P},E} \neq \Sigma_1^{\mathscr{P},E}$) and $\Sigma_0^{\mathscr{P},E} = \Sigma_1^{\mathscr{P},E} \cap \Pi_1^{\mathscr{P},E}$.

Several questions can be asked about relativized $\mathscr{P}$-hierarchies.

(i) Does there exist an oracle X such that $\Sigma_i^{\mathscr{P},X} \subsetneqq \Sigma_{i+1}^{\mathscr{P},X}$ for all i, that is, such that the $\mathscr{P}$, X-hierarchy contains infinitely many distinct classes?

(ii) Does there exist an oracle X such that $\Sigma_{i-1}^{\mathscr{P},X} \subsetneqq \Sigma_i^{\mathscr{P},X} = \Sigma_{i+1}^{\mathscr{P},X}$; that is, the $\mathscr{P}$, X-hierarchy extends exactly i levels?

(iii) Does there exist an oracle X such that the $\mathscr{P}$, X-hierarchy is not trivial but $\Sigma_i^{\mathscr{P},X} \subsetneqq \Delta_{i+1}^{\mathscr{P},X} = \Sigma_{i+1}^{\mathscr{P},X} \cap \Pi_{i+1}^{\mathscr{P},X}$ for all i?

(Only question (i) has an affirmative answer for the arithmetic hierarchy.) An interesting open question, less general than (i)–(iii) is

(iv) Does there exist an oracle X such that $\Sigma_2^{\mathscr{P},X} \neq \Pi_2^{\mathscr{P},X}$?

We were unable to settle (iv) by the methods of this paper.

Acknowledgments. We wish to thank Richard Ladner, Albert Meyer, and Michael Fischer for allowing us to report their results in this paper. We are especially grateful to Albert Meyer for suggesting Theorem 6 and the open problems of § 4. We thank the referees for their corrections and suggestions.

REFERENCES

[1] S. A. COOK, *The complexity of theorem-proving procedures*, Proc. Third Annual ACM Symposium on the Theory of Computing, Shaker Heights, Ohio, 1971, pp. 151–158.

[2] J. HOPCROFT AND J. ULLMAN, *Formal Languages and their Relation to Automata*, Addison-Wesley, Reading, Mass., 1969.

[3] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.

[4] R. LADNER, N. LYNCH AND A. SELMAN, *Comparison of polynomial-time reducibilities*, Proc. Sixth Annual ACM Symposium on the Theory of Computing, Seattle, Wash., 1974, pp. 110–121.

[5] K. Mehlhorn, *On the size of computable functions*, Proc. 14th IEEE Symposium on Switching and Automata Theory, Iowa City, Iowa, 1973, pp. 190–196.

[6] 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.

[7] P. H. Morris, personal communication.

[8] H. Rogers, Jr., *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York, 1967.

[9] W. J. Savitch, *Relationships between nondeterministic and deterministic tape complexities*, J. Comput. System Sci., 4 (1970), pp. 177–192.

[10] L. J. Stockmeyer and A. R. Meyer, *Word problems requiring exponential time: preliminary report*, Proc. Fifth Annual ACM Symposium on the Theory of Computing, Austin, Texas, 1973, pp. 1–9.

SIAM J. COMPUT.
Vol. 4, No. 4, December 1975

PRESERVING PROXIMITY IN ARRAYS*

ARNOLD L. ROSENBERG†

Abstract. Efficiency of storage management in algorithms which use arrays is often enhanced if the arrays are stored in a proximity-preserving manner; that is, array positions which are close to one another in the array are stored close to one another. This paper is devoted to studying certain qualitative and quantitative questions concerning preservation of proximity by array storage schemes (or, realizations). It is shown that fully extendible array realizations cannot preserve proximity in any global sense, not even proximity along a single direction (say, along rows). They can, however, preserve proximity in certain local senses; and realizations which are optimal in various senses of local preservation are exhibited. Partially extendible array storage schemes can preserve proximity in a global way; bounds on their "diameters of preservation" are derived, and optimal schemes are exhibited.

Key words. array, array realization, extendible array, storage allocation

1. Introduction. Algorithms which operate on arrays usually access an array by local groups of positions; that is, the array position accessed after position π usually lies within a small neighborhood of π. For instance, the conventional matrix multiplication scheme traverses matrices along rows and columns, as do many table-searching algorithms. Strassen's matrix multiplication scheme [8] does not proceed along rows or columns, but it does (at the bottom of the recursions) access matrices in blocks of four positions. The overhead for storage management or bookkeeping in such algorithms can be decreased materially if the arrays to be manipulated are stored so as to preserve proximity, that is, if positions which are close to one another in the array are stored close to one another. Preservation of proximity is especially important when dealing with arrays which are very large or which, by means of a series of extensions, can become very large. When dealing with such a (potentially) large array, one must plan for the contingency that the array cannot be accommodated, in its entirety, in main memory; that is, the array might have to be segmented, relegated to some backup store, and processed in main memory in pieces. Since transfers in and out of auxiliary memory tend to be costly, efficiency in such a paging environment is enhanced if the array segments which are brought into main memory admit significant processing before a new segment is needed. To the extent that the above-mentioned tendency to access arrays in local groups of positions is accurate, efficiency in a paging environment is thus enhanced by storing arrays in a proximity-preserving manner. Our purpose in this paper is to investigate certain qualitative and quantitative questions concerning the preservation of proximity by computed-access schemes[1] for allocating storage for arrays.

We investigate three classes of arrays, finite arrays, *prism* arrays which are finite in all but one direction, and *orthant* arrays which are infinite in every direction.

* Received by the editors June 14, 1974, and in revised form October 25, 1974.

† Mathematical Sciences Department, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.

[1] We say that an array storage scheme uses *computed access* if it determines the address assigned to array position π from π's coordinates, as a displacement from the address assigned to position $\langle 1, 1, \cdots, 1\rangle$.

We consider infinite as well as finite arrays to accommodate our notion of extendibility in array realizations [5]: We say that an array allocation scheme is *extendible* in a given direction if it realizes an array which is infinite in that direction; thus, full extendibility resides in realizations of orthant arrays. We prove that fully extendible array realizations cannot preserve proximity globally, not even in a single direction (say along rows in two dimensions). They can, however, preserve proximity locally, according to various criteria; and we derive realizations which are optimal in certain senses of local preservation. We show that realizations of prism and finite arrays can preserve proximity in a global sense; and we derive lower bounds on the *diameters of preservation* of such realizations. We give a recipe for constructing prism realizations which achieve the prism array lower bound. We do not know if the finite array lower bound is achievable, but we note that it is approachable.

The global results on orthant and prism arrays are obtained by studying functions which linearize the *shift chains* introduced in [5]. One benefit of using this intermediary notion is that our results have application to questions concerning preservation of proximity when linearizing data structures other than arrays, for example, infinite trees, lists of infinite lists, and other infinite "filamentous" structures. (These linearizations need not, of course, be for the purpose of assigning addresses. They might represent a schedule for "dovetailing" processes, or a route for threading the filamentous structure, for instance.)

Rather diverse problems concerning computation can be formulated as questions about the integer lattice points in Euclidean d-space. In addition to our current study of preservation of proximity in arrays, Stockmeyer [7] and the author [5], [6] have studied questions concerning the cost of extendibility in array realizations in terms of questions about the discrete positive orthant of d-space. Mylopoulos and Pavlidis [4] were led by consideration of picture processing and pattern recognition to study certain topological properties of discrete spaces. Wong and Maddocks [9] were led by problems concerning multimodule computer memory organizations to study discrete d-dimensional spheres. Karp, McKellar, and Wong [2] were concerned with distance-minimizing arrangements of records on a two-dimensional grid. This variety of applications lends hope that results obtained in this area may have broader implications than the motivating problems suggest.

2. Basic notions

2.1. Background. Let N denote the positive integers; for $n \in N$, let $N_n = \{1, \cdots, n\}$; for $d \in N$, let N^d denote the set of d-tuples of positive integers. We let ε ambiguously denote the tuples $\langle 1, \cdots, 1\rangle$, relying on context to remove ambiguity.

For each $\pi \in N^d$, we denote by π_i the ith coordinate of π (so $\varepsilon_i = 1$ for each i). Four functions on tuples will be useful in the sequel: Say $\pi \in N^d$. Then $\Sigma(\pi) = \sum_{i=1}^{d} \pi_i$; $\Pi(\pi) = \prod_{i=1}^{d} \pi_i$; $m(\pi) = \min\{\pi_i\}$; $M(\pi) = \max\{\pi_i\}$.

For each $d \in N$, we have use for the following set of functions from N^d into N^d:

$$\mathbf{M}_d = \{\mathbf{s}_{id} | i \in N_d\} \cup \{\mathbf{s}'_{id} | i \in N_d\}.$$

For each $\pi \in N^d$, and each $i \in N_d$,

$$(\mathbf{s}_{id}(\pi))_j = \begin{cases} \pi_j + 1 & \text{if } j = i, \\ \pi_j & \text{if } j \neq i; \end{cases}$$

$$(\mathbf{s}'_{id}(\pi))_j = \begin{cases} \left.\begin{matrix} \pi_j - 1 & \text{if } j = i \\ \pi_j & \text{if } j \neq i \end{matrix}\right\} & \text{if } \pi_i > 1, \\ \text{undefined otherwise.} \end{cases}$$

Thus $\mathbf{s}_{id}$ (resp., $\mathbf{s}'_{id}$) can be viewed as the successor (resp., predecessor) in the axis i direction in the d-dimensional positive orthant.

2.2. Arrays and their realizations. We are concerned with *computed-access* schemes for storing arrays. (See footnote .) Accordingly, we study the formal notion of *array scheme*, which is basically the set of positions of an array (i.e., an array with no data items); for mathematical simplicity, we view an array scheme as being imbedded in the positive orthant of Euclidean space (of appropriate dimensionality) with its positions at the integer lattice points. A *realization* of an array scheme is then just a one-to-one map of the positions of the array scheme into N, normalized so that position ε is assigned address 1. This normalization is not necessary for our development, but it does somewhat simplify certain computations later (specifically, the proofs of Theorems 5.2 and 5.3); it also has the aesthetic merit of making our realizations "begin" at the origin position.

(2.1) A *d-dimensional array scheme* (*array* for short) is a set $A = C_1 \times \cdots \times C_d$ of *positions*. Each *coordinate* set C_i is either the set N or the set N_n for some $n \in N$.
When $C_1 = C_2 = \cdots = C_d = N$ (i.e., $A = N^d$), we call A the d-dimensional *orthant* array, and we denote A by Ω_d.
When one $C_i = N$, and all others are finite, we call A a *prism* array.

Our array schemes are rectangular—that is, A is the cross product of its coordinate sets—in conformity with convention. We allow our arrays to be infinite in some or all coordinates in order to accommodate our formal notion of extendible array realization.

(2.2) A *realization* of the array scheme $A = C_1 \times \cdots \times C_d$ is a total one-to-one map $\mathbf{r}: A \to N$ such that $\mathbf{r}(\varepsilon) = 1$.
If $C_i = N$, then we say that $\mathbf{r}$ is *extendible* in direction i.
If $A = \Omega_d$, then we say that $\mathbf{r}$ is *fully extendible*, and we call $\mathbf{r}$ a *d-dimensional extendible array realization.*

Very informally, we associate extendibility in an array realization $\mathbf{r}$ with the realization's assigning addresses for an array which is infinite in directions of "easy extendibility." Expansions along these directions of a finite array stored by $\mathbf{r}$ can be accommodated without changing the (computational) form of $\mathbf{r}$ and without moving the already stored positions. We equate this stability with easy extendibility. We refer the reader to [5], [6] for further discussion of the rationale behind our notions of "array scheme" and "extendible array realization."

3. General results. In this section we establish certain general results which we specialize later to study preservation of proximity in arrays.

3.1. Neighborhoods in the positive orthant. Position ρ of an array is within distance r of position π if ρ lies in the radius r *neighborhood* of π. Many of our results concerning preservation of proximity can be deduced from knowledge of the volumes (in number of integer lattice points) of these discrete neighborhoods. We envisage arrays as being traversed, in general, along the axes, i.e., with sequences of moves from $\mathbf{M}_d$; accordingly, we study only neighborhoods defined in terms of the *rectilinear* (or L_1) metric, $\text{dist}_1(\pi, \rho) = \Sigma_i|\pi_i - \rho_i|$. Had we chosen another standard metric—say the *Euclidean* (or L_2) metric, $\text{dist}_2(\pi, \rho) = (\Sigma_i(\pi_i - \rho_i)^2)^{1/2}$, or the *maximum* (or L_∞) metric, $\text{dist}_\infty(\pi, \rho) = \max\{|\pi_i - \rho_i|\}$—our investigation would have changed in detail but not in concept. (We refer the reader who is curious about the changes involved in switching metrics to [2] and to § 5.4.)

(3.1) Let π be in N^d. The *radius r neighborhood* of π, denoted $\mathbf{N}(\pi; r)$ ($r \in N \cup \{0\}$), is defined as follows.

$$\mathbf{N}(\pi; 0) = \{\pi\};$$

$$\mathbf{N}(\pi; r + 1) = \mathbf{N}(\pi; r) \cup \{\mathbf{s}(\xi) \mid \mathbf{s} \in \mathbf{M}_d \text{ and } \xi \in \mathbf{N}(\pi; r) \cap \text{domain}(\mathbf{s})\}.$$

The *volume* of $\mathbf{N}(\pi; r)$ is $\mathcal{N}(\pi; r) = \#\mathbf{N}(\pi; r)$.

$\mathbf{N}(\pi; r)$ is thus the set of all lattice points in N^d reachable from π in r or fewer moves. (The use of the rectilinear metric is reflected in the fact that only "moves" from $\mathbf{M}_d$ are allowed.)

We now estimate the volume of neighborhoods.

THEOREM 3.1. *Let π be in N^d.*

(a) *For all* $r \in N \cup \{0\}$, $\binom{r+d}{d} \leqq \mathcal{N}(\pi; r) \leqq \sum_{k=0}^{d} \binom{d}{k}\binom{r}{k} 2^k$.

(b) *For all* $r \in N_{M(\pi)-1} \cup \{0\}$, $\mathcal{N}(\pi; r) \geqq \binom{r+d}{d} + \binom{r+d-1}{d}$.

In any case, $\mathcal{N}(\pi; r) = O(r^d)$.[2]

Proof. Fix on an arbitrary $r \in N \cup \{0\}$.

(a) *Lower bound.* It is obvious that, for all $\pi \in N^d$, $\mathcal{N}(\pi; r) \geqq \mathcal{N}(\varepsilon; r)$. One easily verifies that $\mathbf{N}(\varepsilon; r) = \{\pi \in N^d | d \leqq \Sigma(\pi) \leqq r + d\}$. The cardinality of this set is seen to be $\binom{r+d}{d}$ as follows: Lay out $r + d$ markers in a line. Insert, in an arbitrary pattern, d separators, each immediately to the right of some marker (so at most one separator separates adjacent markers). Discard all those markers which are to the right of the rightmost separator. The remaining configuration specifies a unique element of $\mathbf{N}(\varepsilon; r)$, and every element is so specifiable.

Upper bound. It is not hard to see that, for all $\pi \in N^d$,

$$\mathcal{N}(\pi; r) \leqq \mathcal{N}((r + 1)\cdot\varepsilon; r).$$

[Note that $\mathbf{N}((r + 1)\cdot\varepsilon; r)$ is an entire radius r d-sphere; that is, one can proceed r steps in any direction from $(r + 1)\cdot\varepsilon = \langle r + 1, \cdots, r + 1\rangle$ without being interrupted by a boundary of the orthant.] Indeed, equality obtains precisely

[2] By "$g(n) = O(f(n))$" we mean that there exist positive constants K_1 and K_2 such that $K_1 \cdot f(n) < g(n) < K_2 \cdot f(n)$ for almost all n.

when $r < m(\pi)$. Wong and Maddocks [9] show that the volume of a radius r d-sphere is $\sum_{k=0}^{d} \binom{d}{k}\binom{r}{k} 2^k$.

(b) Let $M = \{\pi \in N^d | M(\pi) > r\}$. Let $\pi^{(k)}$ $(k \in N_d)$ be that element of M such that $\pi_k^{(k)} = r + 1$, and $\pi_i^{(k)} = 1$ for $i \neq k$. It is not hard to verify that, for all $\pi \in M$, $\mathcal{N}(\pi; r) \geqq \mathcal{N}(\pi^{(k)}; r)$. (Intuitively, $\pi^{(k)}$ has the least space to grow in, among elements of M.) Now, $\mathbf{N}(\pi^{(k)}; r)$ can be partitioned into the two sets

$$A = \{\pi \in \mathbf{N}(\pi^{(k)}; r) | \pi_k \geqq r + 1\},$$

$$B = \{\pi \in \mathbf{N}(\pi^{(k)}; r) | \pi_k < r + 1\}.$$

The first set, A, is isomorphic to $\mathbf{N}(\varepsilon; r)$ under the correspondence $\pi \in \mathbf{N}(\varepsilon; r) \leftrightarrow \mathbf{s}_{kd}^{(r)}(\pi) \in A$.[3] The second set, B, is isomorphic to $\mathbf{N}(\varepsilon; r - 1)$ under the correspondence[4] $\pi \in \mathbf{N}(\varepsilon; r - 1) \leftrightarrow \mathbf{s}_{kd}^{(r-2\pi_k+1)}(\pi) \in B$. Part (b) now follows from the lower bound argument of part (a). □

Remark. The interested reader can easily verify the following extensions to Theorem 3.1. (i) The lower bound of part (a) is achieved precisely when $\pi = \varepsilon$. (ii) The upper bound of part (a) is achieved precisely when $m(\pi) > r$. (iii) The bound of part (b) is exact (i.e., equality holds) whenever $d - 1$ of π's coordinates are 1.

Remark. The fact that $\mathbf{N}(\pi; r)$ has volume $O(r^d)$ is insensitive to the choice of the L_1 or L_2 or L_∞ metric. Thus, at least in a gross sense, our results are independent of the metric employed. Of course, those results which depend on the actual growth rate as opposed to just its order are not insensitive to the chosen metric.

3.2. Preserving proximity in shift chains. Certain results concerning preservation of proximity in infinite arrays do not depend on the cross-hatched move structure of arrays (as exposed by $\mathbf{M}_d$), but rather on the presence of long "filaments" or chains in the arrays. We now abstract the arguments needed to prove these results, and we consider the problem of preserving proximity in *shift chains*. This abstraction serves two purposes. First (and foremost) it exposes the essence of certain of our arguments about arrays. Second, it establishes results about linearizations of infinite filamentous data structures other than arrays, for example, infinite trees or infinite lists of lists. As we mentioned in the Introduction, the linearizations of these data structures need not be for the purpose of assigning addresses; they may be for the purpose of threading the structures, of "dovetailing" processes, etc. The reader should have no problem in transporting our results into a variety of settings related to such linearizations.

Let $\mathbf{s}$ be a transformation of a set S, i.e., a (possibly nontotal) function from S into S. For arbitrary $\sigma, \tau \in S$, we say that σ $\mathbf{s}$-precedes τ—written $\sigma <_{\mathbf{s}} \tau$—if $\mathbf{s}^{(k)}(\sigma) = \tau$ for some $k \in N$ (see footnote 3). Clearly $<_{\mathbf{s}}$ is a transitive relation.

(3.2) The transformation $\mathbf{s}$ of S is a *shift* (of S) if the following conditions hold.

(a) $\mathbf{s}$ is one-to-one.

(b) $<_{\mathbf{s}}$ is cycle-free—for no $k \in N$ does $\mathbf{s}^{(k)}$ have a fixed point.

(c) Each $\sigma \in \mathbf{s}(S)$ (the range of $\mathbf{s}$) has an $\mathbf{s}$-predecessor which is not in $\mathbf{s}(S)$.

[3] Generally, $\mathbf{s}^{(r)}$ is the r-fold composition of the function $\mathbf{s}$. Obviously, $\mathbf{s}_{kd}^{(r)}$ is one-to-one.

[4] $\mathbf{N}(\varepsilon; k)$ is empty for $k < 0$; $\mathbf{s}_{kd}^{(-a)} = \mathbf{s}_{kd}^{\prime(a)}$.

The relation $<_{\mathbf{s}}$ yields a set $A(\mathbf{s}) \subseteq S$ of *atomic* elements which are not in the range of $\mathbf{s}$. When $\mathbf{s}$ is a shift, S is partitioned in a natural way into *shift chains*: For each $\alpha \in A(\mathbf{s})$, there is a shift chain $C(\alpha) = \{\alpha\} \cup \{\mathbf{s}^{(k)}(\alpha) | k \in N\}$ which is linearly ordered by $<_{\mathbf{s}}$. (The proofs of linearity and of partition are left to the reader.)

Remark. Each axis successor $\mathbf{s}_{id}$ is a shift of every d-dimensional array. In two dimensions, the shift chains resulting from $\mathbf{s}_{12}$ and $\mathbf{s}_{22}$ are called *columns* and *rows*, respectively.

The decomposition of S into shift chains affords us the desired analogue of neighborhood.

(3.3) Let $\mathbf{s}$ be a shift of the set S. For $\sigma \in S$ and $n \in N \cup \{0\}$, the *n-slice of S generated by* $\mathbf{s}$ *based at* σ is denoted $\mathrm{S}(\sigma; n)$ ($\mathbf{s}$ will always be clear from context) and is defined inductively as follows.

$$\mathrm{S}(\sigma; 0) = \{\sigma\};$$

$$\mathrm{S}(\sigma; k+1) = \begin{cases} \mathrm{S}(\sigma; k) \cup \{\mathbf{s}^{(k+1)}(\sigma)\} & \text{if } \sigma \in \text{domain } \mathbf{s}^{(k+1)}, \\ \varnothing & \text{if } \sigma \notin \text{domain } \mathbf{s}^{(k+1)}. \end{cases}$$

An n-slice is thus a one-sided one-dimensional neighborhood.

(3.4) Let $\mathbf{s}$ be a shift of the set S, and let $\mathbf{f}: S \to N$ be a total function. We say that $\mathbf{f}$ *preserves the n-slices generated by* $\mathbf{s}$ ($n \in N$) if there is a $d_n \in N$ such that, for all $\sigma \in S$ and all $\pi, \rho \in \mathrm{S}(\sigma; n)$, $|\mathbf{f}(\pi) - \mathbf{f}(\rho)| < d_n$. The smallest such d_n is called the *diameter* (of preservation).

The diameter of preservation is the size of the smallest interval capable of holding the images under $\mathbf{f}$ of $\mathbf{s}$'s n-slices; i.e., d_n measures the extent to which $\mathbf{f}$ spreads out the slices. (3.4) formalizes and quantifies our notion of preserving proximity in shift chains.

(3.5) The shift $\mathbf{s}$ is *l-limited* [$l \in N \cup \{0\}$] (resp., *unlimited*) if the relation $<_{\mathbf{s}}$ gives rise to precisely l (resp., infinitely many) infinite shift chains.

Remark. With a prism array, one axis successor is ($l > 0$)-limited and all others are 0-limited. With an orthant array, all successors are unlimited.

THEOREM 3.2. *Let* $\mathbf{s}$ *be an unlimited shift of the set S. If there exists* $n \in N$ *such that the total function* $\mathbf{f}: S \to N$ *preserves the n-slices generated by* $\mathbf{s}$, *then* $\mathbf{f}$ *is not one-to-one.*

Proof. Assume that $\mathbf{f}$ is one-to-one and preserves n-slices with diameter d_n. Let $A \subseteq A(\mathbf{s})$ comprise the atomic elements of the infinite shift chains of $<_{\mathbf{s}}$. Since A is infinite and $\mathbf{f}$ is total, there exists $b \in N$ such that the set $B = \{\alpha \in A | \mathbf{f}(\alpha) \leqq b\}$ contains at least d_n elements. Since $\mathbf{f}$ is one-to-one and all the shift chains associated with elements of B are infinite, each chain $C(\alpha)$ with $\alpha \in B$ must contain elements σ for which $\mathbf{f}(\sigma) \geqq b + d_n$. For $\alpha \in B$, let $\mu(\alpha)$ be that element $\mathbf{s}^{(k)}(\alpha)$ of $C(\alpha)$ for which (i) $\mathbf{f}(\mu(\alpha)) \geqq b + d_n$, and (ii) $l < k$ implies $\mathbf{f}(\mathbf{s}^{(l)}(\alpha)) < b + d_n$. Let $\nu(\alpha) = \mathbf{s}^{(k-1)}(\alpha)$.[5] By choice of b, $\nu(\alpha)$ exists; by (ii), $\mathbf{f}(\nu(\alpha)) < b + d_n$. Now, $\mathbf{f}$ preserves

[5] For any function $\mathbf{s}: S \to S$, $\mathbf{s}^{(0)}$ is the identity transformation on S.

n-slices with diameter d_n; therefore, $\mathbf{f}(u(\alpha)) - \mathbf{f}(v(\alpha)) < d_n$, so that $\mathbf{f}(v(\alpha))$ must lie in the interval $I = \{b + 1, b + 2, \cdots, b + d_n - 1\}$. Since α was chosen arbitrarily from B, the interval I must contain the images $\mathbf{f}(\sigma)$ of at least d_n elements $\sigma \in S$. We must conclude either that $\mathbf{f}$ is not one-to-one, or (since d_n was arbitrary) that $\mathbf{f}$ does not preserve n-slices. □

Remark. If one views $\mathbf{f}$ as assigning addresses to S, then Theorem 3.2 says that distance in storage is not functionally related to distance along shift chains. If one views $\mathbf{f}$ as specifying a schedule for visiting the elements of S, then the theorem says that one cannot plan to visit the shift chains within bounded intervals (i.e., one cannot plan to visit each individual chain at intervals of k or fewer time units).

When the shift $\mathbf{s}$ is limited, one can preserve n-slices for all n. The technique used in the proof of Theorem 3.2 permits us to bound tightly the diameter of preservation in the limited case.

THEOREM 3.3. *Let* $\mathbf{s}$ *be an l-limited shift of the set S. Say that the one-to-one total function* $\mathbf{f}: S \to N$ *preserves n-slices generated by* $\mathbf{s}$ *with diameter* d_n. *Then* $d_n \geqq l \cdot n + 1$.

Proof. Let $A \subseteq A(\mathbf{s})$ comprise the atomic elements of the l infinite shift chains of $<_{\mathbf{s}}$; let $m = \max \{\mathbf{f}(\alpha) | \alpha \in A\}$. ($m$ exists since $\mathbf{f}$ is total.)

Assume, for contradiction, that $\mathbf{f}$ preserves n-slices with diameter $d_n \leqq l \cdot n$. Then, since $\mathbf{f}$ is one-to-one, the interval $I = \{m + 1, \cdots, m + d_n - 1\}$ must contain images of fewer than n elements from some shift chain $C(\alpha_0)$ with $\alpha_0 \in A$. (I contains fewer than $l \cdot$ n integers, hence fewer than $l \cdot n$ images.)

For $k \in N \cup \{0\}$, denote by α_k the element $\alpha_k = \mathbf{s}^{(k)}(\alpha_0) \in C(\alpha_0)$. Let $u \geqq 0$ be the largest integer for which $\mathbf{f}(\alpha_u) \leqq m$. Let $v > u$ be the smallest integer for which $\mathbf{f}(\alpha_v) \geqq m + d_n$. Clearly $v - u \leqq n$, or else the interval I would contain the images of (at least) n elements of $C(\alpha_0)$, contrary to assumption. But, if $v - u \leqq n$, then $\alpha_v \in \mathbf{S}(\alpha_u; n)$; and then the fact that $|\mathbf{f}(\alpha_v) - \mathbf{f}(\alpha_u)| \geqq d_n$ would contradict either the fact that $\mathbf{f}$ is one-to-one, or the assumption that $\mathbf{f}$ preserves n-slices with diameter $d_n < l \cdot n + 1$. □

Remark. The lower bound of Theorem 3.3 is best possible since it is achievable whenever $A = A(\mathbf{s})$. To wit, if $A = \{\alpha_1, \cdots, \alpha_l\}$ (ignoring the notation of the preceding proof), define $\mathbf{f}: \mathbf{S} \to N$ by: $\mathbf{f}(\mathbf{s}^{(k)}(\alpha_j)) = k \cdot l + j$ for $k \in N \cup \{0\}$ and $j \in N_l$. One verifies easily that $\mathbf{f}$ is one-to-one and total and that $\mathbf{f}$ preserves the n-slices generated by $\mathbf{s}$ with diameter $d_n = l \cdot n + 1$.

4. Global preservation of proximity. The results of § 3 allow us to settle completely the question of globally preserving proximity when realizing arrays: Global preservation is possible with finite and prism arrays and is impossible for arrays which are extendible in more than one direction; in the former cases, tight lower bounds on diameter of preservation can be derived.

Our formal notion of global preservation of proximity follows (3.4).

(4.1) Let $\mathbf{r}$ be a realization of the array scheme A. We say that $\mathbf{r}$ *globally preserves n-neighborhoods* ($n \in N$) if there is a $d_n \in N$ such that, for all $\pi \in A$ and all $\rho, \sigma \in \mathbf{N}(\pi; n) \cap A$, $|\mathbf{r}(\rho) - \mathbf{r}(\sigma)| < d_n$. The smallest such d_n is called the *diameter* (of preservation).

As with (3.4), the diameter d_n is the size of the smallest interval capable of holding the image under **r** of any n-neighborhood within A (i.e., the portion of the neighborhood that lies within A). Since we shall be concerned only with global preservation in this section, we shall omit the qualifier "global" in what follows.

4.1. Finite arrays. Every realization of a finite array preserves neighborhoods. Thus the question of preserving proximity in finite arrays can be resolved by bounding from below the diameter of preservation and determining whether or not the derived bound is achievable.

Let $A = N_{l_1} \times \cdots \times N_{l_d}$ be a d-dimensional finite array. We call $\lambda_A = \langle l_1, \cdots, l_d \rangle$ the *size vector* of A.

THEOREM 4.1. *Let* **r** *realize the finite array* $A \subset N^d$. *There is a constant* $c > 0$, *depending only on* d, *such that, if* **r** *preserves* n*-neighborhoods with diameter* d_n, *then* $d_n > c \cdot n \cdot m(\lambda_A)^{d-1}$ *whenever* $n < m(\lambda_A)$.

Remark. The growth rate of d_n can be exposed in more detail, but only at the cost of investigating cases. In an attempt to enhance readability, we leave the theorem in its gross form and relegate the casewise analysis to the proof. The interested reader can easily refine the statement of the theorem.

Proof. For $\pi \in A$ and $n \in N$, let $\mathbf{A}(\pi; n) = \mathbf{N}(\pi; n) \cap A$ denote that portion of π's n-neighborhood which lies within A. We proceed in two steps to bound $\#\mathbf{A}(\pi; n)$ and, thereby, bound d_n.

LEMMA 4.1. *For all* $\pi \in A$ *and all* $k, n \in N$, $\#\mathbf{A}(\pi; kn) < 2kd_n$.

Proof. (Lemma 4.1). We show by induction (on k) that the image $\mathbf{r}(\mathbf{A}(\pi; kn))$ of $\mathbf{A}(\pi; kn)$ lies within an interval of size less than $2kd_n$. (An *interval* is a set of integers of the form $\{a, a+1, a+2, \cdots, b\}$.) Since **r** is one-to-one, the lemma will follow.

$k = 1$. Since **r** preserves n-neighborhoods with diameter d_n, it follows that $\mathbf{r}(\mathbf{A}(\pi; n))$ lies within an interval of size not exceeding $d_n < 2d_n$.

$k + 1$. Assume now that $\mathbf{r}(\mathbf{A}(\pi; kn))$ lies within an interval of size less than $2kd_n$. Note that each point $\xi \in \mathbf{A}(\pi; (k+1)n)$ resides in a neighborhood $\mathbf{A}(\eta; n)$ for some $\eta \in \mathbf{A}(\pi; kn)$. Therefore, **r**'s preservation of n-neighborhoods with diameter d_n forces us to conclude that the smallest interval containing $\mathbf{r}(\mathbf{A}(\pi; (k+1)n))$ contains fewer than $2d_n$ points more than the corresponding interval for $\mathbf{r}(\mathbf{A}(\pi; kn))$; that is, $\mathbf{r}(\mathbf{A}(\pi; (k+1)n))$ must lie within an interval of size $< 2d_n + 2kd_n$. □

Notation. For any real number x, $[x]$ denotes the integer part of x, and $\lceil x \rceil$ denotes the smallest integer not less than x.

LEMMA 4.2. *For each* $n \in N$, *let* $\alpha(n) = \max\{\#\mathbf{A}(\pi; n) | \pi \in A\}$.

(a) *For* $n < m(\lambda_A)$, $\alpha(n) \geqq \binom{n+d}{d}$.

(b) *For* $n < \min\{m(\lambda_A), \lceil M(\lambda_A)/2 \rceil\}$, $\alpha(n) \geqq \binom{n+d}{d} + \binom{n+d-1}{d}$.

(c) *For* $n < \lceil m(\lambda_A)/2 \rceil$, $\alpha(n) \geqq \sum_{i=0}^{d} \binom{d}{i}\binom{n}{i} 2^i$.

In all three cases, there is a constant b *for which* $\alpha(n) > b \cdot n^d$.

Proof. (Lemma 4.2). We invoke Theorem 3.1. When n is as specified, then:
(a) $\mathbf{N}(\varepsilon; n) \subset A$;
(b) if $M(\lambda_A) = l_i$, then $\mathbf{N}(\pi; n) \subset A$ where $\pi_i = \lceil l_i/2 \rceil$, and $\pi_j = 1$ for $j \neq i$;
(c) $\mathbf{N}(\pi; n) \subset A$, where $\pi = (\lceil m(\lambda_A)/2 \rceil) \cdot \varepsilon$. □

Return to proof. Lemmas 4.1 and 4.2 combine to yield: There is a constant b such that, for all $k, n \in N$ with $k \cdot n < m(\lambda_A)$, $b \cdot (kn)^d < \alpha(kn) < 2kd_n$. Letting $k = [m(\lambda_A)/n] - 1$, one obtains the gross bound of the theorem. □

We do not know if the bounds of Theorem 4.1 are achievable. Certainly the schemes in common use do not attain these bounds. The diameter of preservation of a conventional d-dimensional scheme (see [3, § 2.2.6]) is of the form $d_n = 2n \cdot \prod_{i \in S} l_i + 1$, where the set S is either N_{d-1} or $N_d - \{1\}$. Such diameters of preservation arise naturally since, as we noted in [5], these schemes do not store just a finite array; they actually store a prism array with either $C_d = N$ or $C_1 = N$ (which alternatives lead, respectively, to the choices for S). We now turn our attention to preservation of proximity by schemes for storing prism arrays.

4.2. Prism arrays. Since prism arrays are infinite in one direction, their realizations need not preserve neighborhoods. For those prism realizations which do preserve n-neighborhoods, we can derive a precise lower bound on the diameter of preservation. This lower bound is achievable and, in fact, is the diameter of preservation of the "sequential allocation" schemes described in [3, § 2.2.6]. Thus the schemes used by conventional compilers for ALGOL, FORTRAN, etc., are optimal in their preservation of proximity among all schemes for realizing prism arrays.

Let $A = C_1 \times \cdots \times C_d$ be a prism array with $C_i = N$ and $C_j = N_{l_j}$ ($l_j \in N$) for $j \neq i$. The *base* (area) of A is $\mathbf{B}(A) = \prod_{j \neq i} l_j$.

THEOREM 4.2. *Let* $\mathbf{r}$ *realize the prism array* A. *If* $\mathbf{r}$ *preserves n-neighborhoods with diameter* d_n, *then* $d_n \geqq 2n \cdot \mathbf{B}(A) + 1$. *Moreover, this lower bound on* d_n *is achievable.*

Proof. The lower bound on d_n follows directly from Theorem 3.3: If $A = C_1 \times \cdots \times C_d$ and $C_i = N$, then the axis successor $\mathbf{s}_{id}$ (cf. § 2.1) is a $\mathbf{B}(A)$-limited shift of A; and the diameter of preservation for n-neighborhoods can obviously be no less than that for $2n$-slices generated by $\mathbf{s}_{id}$, since $\mathbf{S}(\mathbf{s}_{id}^{(-n)}(\pi); 2n) \subset \mathbf{N}(\pi; n)$. (See footnote 4.)

Using the prescription in the remark following Theorem 3.3, we can construct prism realizations which preserve proximity optimally. Consider any realization $\mathbf{r}$ of A which, for each $c \in N$, assigns the addresses $(c - 1) \cdot \mathbf{B}(A) + 1$ through $c \cdot \mathbf{B}(A)$ to positions $\{\pi \in A | \pi_i = c\}$ in such a way that $\mathbf{r}(\mathbf{s}_{id}(\pi)) = \mathbf{r}(\pi) + \mathbf{B}(A)$ for all $\pi \in A$. (The "sequential allocation" schemes of [3] operate in this way.) Consider the diameter of preservation of such an $\mathbf{r}$: Let $\pi \in A$ be arbitrary, and let ρ be in $\mathbf{N}(\pi; n) \cap A$. Let $\sigma = \langle \pi_1, \cdots, \pi_{i-1}, \rho_i, \pi_{i+1}, \cdots, \pi_d \rangle$. We distinguish two cases.

(a) $\rho = \sigma$ (so one of π, ρ is in the n-slice generated by $\mathbf{s}_{id}$, based at the other). In this case $|\mathbf{r}(\pi) - \mathbf{r}(\rho)| = |\pi_i - \rho_i| \cdot \mathbf{B}(A) \leqq n \cdot \mathbf{B}(A)$.

(b) $\rho \neq \sigma$. In this case (i) $|\pi_i - \rho_i| < n$, or else ρ would not be in $\mathbf{N}(\pi; n)$—note the use of the L_1 metric; (ii) $|\mathbf{r}(\rho) - \mathbf{r}(\sigma)| < \mathbf{B}(A)$ by construction of $\mathbf{r}$. Therefore,

$$|\mathbf{r}(\pi) - \mathbf{r}(\rho)| \leqq |\mathbf{r}(\pi) - \mathbf{r}(\sigma)| + |\mathbf{r}(\rho) - \mathbf{r}(\sigma)| < (|\pi_i - \rho_i| + 1) \cdot \mathbf{B}(A) \leqq n \cdot \mathbf{B}(A).$$

In either case, $|\mathbf{r}(\pi) - \mathbf{r}(\rho)| \leqq n \cdot \mathbf{B}(A)$. It follows that, if we choose two arbitrary points $\xi, \eta \in \mathbf{N}(\pi; n) \cap A$, we find that

$$|\mathbf{r}(\xi) - \mathbf{r}(\eta)| \leqq |\mathbf{r}(\pi) - \mathbf{r}(\xi)| + |\mathbf{r}(\pi) - \mathbf{r}(\eta)| \leqq 2n \cdot \mathbf{B}(A) < 2n \cdot \mathbf{B}(A) + 1.$$

By definition then, **r** preserves n-neighborhoods with diameter $2n \cdot \mathbf{B}(A) + 1$, whence the lower bound is achievable. □

4.3. Doubly extendible arrays. If the array A is extendible in two or more directions, then no realization of A can preserve neighborhoods.

THEOREM 4.3. *Let $A = C_1 \times \cdots \times C_d$, and say that $C_i = C_j = N$ $(i \neq j)$. Let $n \in N$ be arbitrary. No realization of A preserves n-neighborhoods.*

Proof. Both $\mathbf{s}_{id}$ and $\mathbf{s}_{jd}$ are unlimited shifts of A. The theorem thus follows from Theorem 3.2 since $\mathbf{S}(\pi; n) \subset \mathbf{N}(\pi; n)$ for slices generated by any shift of A.

The theorem can also be derived from Theorem 4.1, where the lower bound on the diameter of neighborhood preservation is shown to grow with the size of the array. □

COROLLARY. *Let $d > 1$ and $n \in N$ be arbitrary. No realization of Ω_d preserves n-neighborhoods.*

In fact the nonpreservability of n-slices generated by unlimited shifts (Theorem 3.2) can be interpreted as saying that realizations of orthant arrays cannot preserve even one-sided unidirectional neighborhoods.

5. Local preservation of proximity. The corollary to Theorem 4.3 asserts that there is no function $D: N \to N$ which yields, for each $n \in N$, the diameter of the intervals into which radius n neighborhoods are mapped by extendible array realizations. However, if one allows the diameter-specifying function to operate locally—that is, let D map $N \times N^d$ into N, $D(n, \pi)$ being the size of the smallest interval containing the image of $\mathbf{N}(\pi; n)$—then obviously such a function D exists for every realization. In this section we consider the rate of growth of these local diameters of preservation. We show that realizations can be found which cause D to grow optimally slowly for an individual point π, but that no realization can have ideally slow growth for more than one π. We then establish a lower bound on the rate of growth of local diameters of preservation, but we do so in terms of a "cumulative" version of the function D, which tends to smooth out the possible erratic behavior of D; many realizations of Ω_d achieve this growth rate. Finally, we discuss the effect on our results of replacing the L_1 metric by some other metric.

Throughout this section, when discussing Ω_d, we shall understand $d > 1$.

5.1. Measures of local preservation.

(5.1) Let **r** be a realization of Ω_d. Define the functions $D_{\mathbf{r}}: N \times N^d \to N$ and $D^*_{\mathbf{r}}: N \times N^d \to N$ as follows:

(a) For each $n \in N$ and $\pi \in N^d$,

$$D_{\mathbf{r}}(n, \pi) = \max(\mathbf{r}(\mathbf{N}(\pi; n))) - \min(\mathbf{r}(\mathbf{N}(\pi : n))) + 1.$$

$D_{\mathbf{r}}$ is called **r**'s *local diameter of preservation*; for each n, π, $D_{\mathbf{r}}(n, \pi)$ is the size of the smallest interval which contains the image of $\mathbf{N}(\pi; n)$.

(b) For each $n \in N$ and $\pi \in N^d$,

$$D^*_{\mathbf{r}}(n, \pi) = \max\{D_{\mathbf{r}}(n, \xi) | M(\xi) \leqq M(\pi)\}.$$

$D^*_{\mathbf{r}}$ is called **r**'s *cumulative local diameter of preservation.*

The qualifier "local" will be understood in what follows; and the subscript "**r**" will be elided whenever **r** is clear from context.

The role of $D_{\mathbf{r}}^*$ is to average out the possible erratic behavior of $D_{\mathbf{r}}$ which results from our considering all possible realizations of Ω_d. This erratic behavior is discernible in the possibility of a realization's centering its layout of storage about a single point in the orthant.

5.2. Central points of realizations. Clearly, $D(n, \pi)$ can never be smaller than $\mathcal{N}(\pi; n)$ since realizations are one-to-one functions; moreover, $D(n, \pi)$ can be that small only if an entire interval is dedicated to the image of $\mathbf{N}(\pi; n)$. Given any single $\pi \in N^d$, one can construct a realization of Ω_d which is *centered* at π in the sense that it does dedicate entire intervals to the neighborhoods of π. However, these centered realizations can be monotone only when $\pi = \varepsilon$. Moreover, no realization $\mathbf{r}$ can so optimize the function $D_{\mathbf{r}}$ for more than one π.

(5.2) Let $\mathbf{r}$ be a realization of Ω_d. We say that $\mathbf{r}$ is *centered* at $\pi \in N^d$ (or that π is the *center* of $\mathbf{r}$) if, for all $n \in N$ and all $\rho, \sigma \in \mathbf{N}(\pi; n)$, $|\mathbf{r}(\rho) - \mathbf{r}(\sigma)| < \mathcal{N}(\pi; n)$.

Remark. π is the center of $\mathbf{r}$ iff $D(n, \pi) = \mathcal{N}(\pi; n)$.

Remark. That centers need not accompany even onto realizations is immediate; for instance, the onto realization $\mathbf{r}(\pi) = 2^{\pi_1 - 1}(2\pi_2 - 1)$ of Ω_2 is easily seen to have no center.

Definition (5.2) leaves us with two obligations. First, we must show that the definition is well-founded; that is, we must show that centers may exist. Then we must show that the article "the" is not misleading; that is, we must show that a realization cannot have more than one center.

THEOREM 5.1. *For every $\pi \in N^d$, there are realizations* $\mathbf{r}$ *of Ω_d which are centered at π.*

Proof sketch. We describe in broad terms a realization $\mathbf{r}$ which is centered at π. We discuss how $\mathbf{r}$ lays out storage rather than how $\mathbf{r}$ is computed.

(a) $\mathbf{r}(\varepsilon) = 1$ by definition (2.2) of realization.

(b) $\mathbf{r}(\pi) = 1 + \mathcal{N}(\varepsilon; \Sigma(\pi) - d - 1) = 1 + \binom{\Sigma(\pi) - 1}{d}$.

At several points in what follows, we shall wish to refer to the set of points which are exactly distance n from point π, so we establish the following notation.

(5.3) For $n \in N$ and $\pi \in N^d$, $\Delta(\pi; n) = \mathbf{N}(\pi; n) - \mathbf{N}(\pi; n - 1)$.

(c) Returning to the description of $\mathbf{r}$, let $s = \Sigma(\pi) - d - 1$, let $U = \{2, \cdots, \mathcal{N}(\varepsilon; s)\}$, and let $V = N - U - \{1, \mathbf{r}(\pi)\} = \{2 + \mathcal{N}(\varepsilon; s), \cdots\}$. For each $n \in N$, $\mathbf{r}$ assigns addresses to the points in $\Delta(\pi; n)$ as follows. The points in $\Delta(\pi; n) \cap \mathbf{N}(\varepsilon; s) - \{\varepsilon\}$ get assigned the *largest* hitherto unassigned integers from set U – recall that ε has already been assigned an address; the points in $\Delta(\pi; n) - \mathbf{N}(\varepsilon; s)$ get assigned the *smallest* unassigned integers from set V. The details of these assignments are immaterial as long as they are one-to-one. By design, no two points in $\mathbf{N}(\pi; n)$ are allocated addresses which are more than $\mathcal{N}(\pi; n) - 1$ apart, so by induction on n, $\mathbf{r}$ is shown to be centered at π. Figure 1 may be useful in visualizing the allocation scheme just described. □

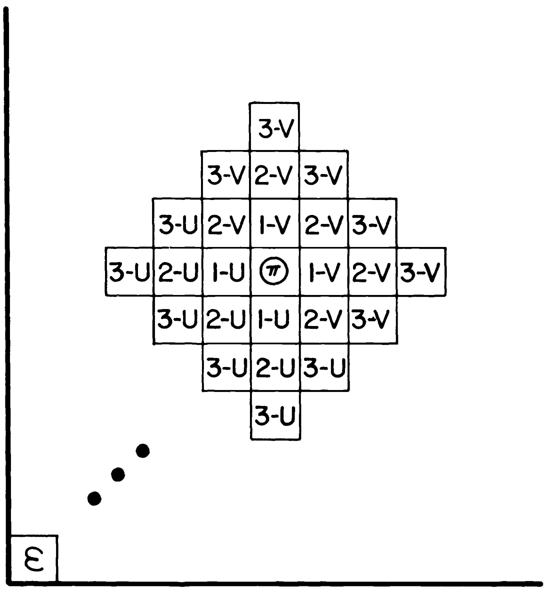


FIG. 1. *The partitioned shells used to center a realization at π. Each position is labeled "p − X," where p is the position's distance from π (= shell number), and X = U or V according as the position is closer than π to ε or not.*

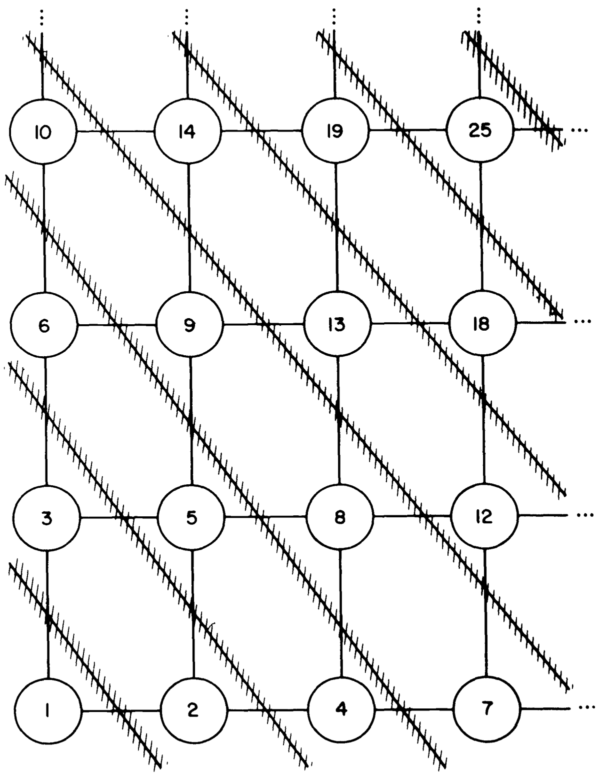


FIG. 2. *A monotonic realization of* Ω_2 *which is centered at* ε.

The most familiar (and, perhaps, the most useful) centered realization is probably the two-dimensional diagonal realization,

$$\mathbf{r}_d(\pi) = (\Sigma(\pi) - 1)(\Sigma(\pi) - 2)/2 + \pi_1,$$

which is centered at ε (see Fig. 2).

The scheme $\mathbf{r}$ described in the proof sketch is not the simplest (to describe) realization centered at π: One could simply treat the sets $\Delta(\pi; n)$ as units rather than partition them as we did. Our choice of layout was dictated by an attempt to construct an $\mathbf{r}$ which, at least for small arrays, was close to being monotonic.[6] However, any attempt to discover a centered $\mathbf{r}$ which is actually monotonic is bound to fail unless $\pi = \varepsilon$. (The diagonal realization $\mathbf{r}_d$ demonstrates that realizations centered at ε can be monotonic.) A preliminary lemma is useful in establishing this claim.

LEMMA 5.1. *Let* $\mathbf{r}$ *be a realization of* Ω_d *which is centered at* $\pi \in N^d$. *For all* $n \geqq \Sigma(\pi) - d$, $\mathbf{r}(\mathbf{N}(\pi; n)) = \{1, 2, \cdots, \mathcal{N}(\pi; n)\}$. *Therefore, for all such n,*

$$\min \mathbf{r}(\Delta(\pi; n+1)) > \mathcal{N}(\pi; n).$$

Proof. Since $\varepsilon \in \mathbf{N}(\pi; \Sigma(\pi) - d)$, $\min \mathbf{r}(\mathbf{N}(\pi; n)) = \mathbf{r}(\varepsilon) = 1$ for all $n \geqq \Sigma(\pi) - d$. By definition of center, then, it follows that $\max \mathbf{r}(\mathbf{N}(\pi; n)) \leqq \mathcal{N}(\pi; n)$; inequality is impossible since $\mathbf{r}$ is one-to-one. □

Note that Lemma 5.1 contains our first use of the assumption that $\mathbf{r}(\varepsilon) = 1$. The reader may find it instructive to derive a version of this lemma which does not depend on this assumption and to note how this more general lemma complicates the proofs of Theorems 5.2 and 5.3 (which both remain true).

THEOREM 5.2. *Let the realization* $\mathbf{r}$ *of* Ω_d *be centered at* $\pi \in N^d$. *If* $\pi \neq \varepsilon$, *then* $\mathbf{r}$ *is not monotonic.*

Proof. Say, with no loss of generality, that $\pi_1 > 1$. We exhibit specific pairs of points where $\mathbf{r}$'s nonmonotonicity is discernible. For each $n \geqq \Sigma(\pi) - d$, let $\rho_n = \mathbf{s}_{2d}^{(n)}(\pi) \in \Delta(\pi; n)$, and let $\sigma_n = \mathbf{s}'_{1d}(\rho_n) \in \Delta(\pi; n+1)$. By Lemma 5.1, $\mathbf{r}(\rho_n) \leqq \mathcal{N}(\pi; n) < \mathbf{r}(\sigma_n)$. Since $\rho_n = \mathbf{s}_{1d}(\sigma_n)$, these inequalities demonstrate that $\mathbf{r}$ is not monotonic. Figure 3 may aid the reader in visualizing the described layout. □

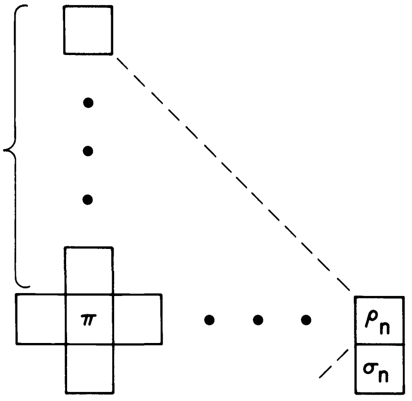


FIG. 3. *The relative positions of* π, ρ_n *and* σ_n *in the proof of nonmonotonicity of realizations centered at* $\pi \neq \varepsilon$.

[6] $\mathbf{f}: N^d \to N$ is *monotonic* if, for all $\pi \in N^d$ and all $\mathbf{s}_{id} \in \mathbf{M}_d$, $\mathbf{f}(\mathbf{s}_{id}(\pi)) \geqq \mathbf{f}(\pi)$.

A proof not unlike the preceding one shows that a realization can be centered at no more than one point.

THEOREM 5.3. *A realization of* Ω_d *can have at most one center.*

Proof. Assume, for contradiction, that the realization $\mathbf{r}$ of Ω_d is centered at both π and $\rho \neq \pi$. We distinguish two cases.

(a) $\pi_k = \rho_k$ for some k. Then, for all $n \in N$, $\sigma_n = \mathbf{s}_{kd}^{(n)}(\pi) \in \mathbf{N}(\pi; n) - \mathbf{N}(\rho; n)$, while $\tau_n = \mathbf{s}_{kd}^{(n)}(\rho) \in \mathbf{N}(\rho; n) - \mathbf{N}(\pi; n)$. See Figure 4(a).

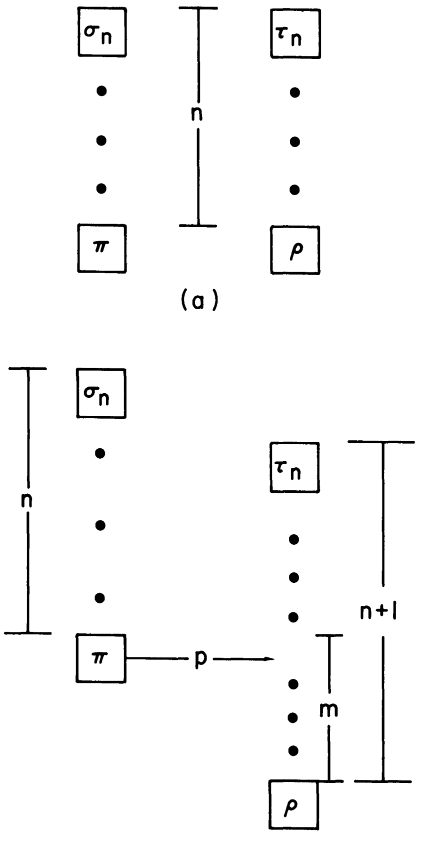


FIG. 4. *The relative positions of the alleged centers* π *and* ρ, *and the contradiction-lending points* σ_n *and* τ_n: (a) *when* π *and* ρ *share some coordinate*; (b) *when* π *and* ρ *differ in all coordinates*.

(b) $\pi_k \neq \rho_k$ for any k. Let $m = \min\{|\pi_i - \rho_i|\} > 0$. With no loss of generality, say that $m = \pi_l - \rho_l$ so that $\pi_l > \rho_l$. Let $p = \sum_{i \neq l} |\pi_i - \rho_i| \geqq (d-1)m \geqq m$. Now, for all $n \in N$, $\sigma_n = \mathbf{s}_{ld}^{(n)}(\pi) \in \mathbf{N}(\pi; n) - \mathbf{N}(\rho; n+1)$. (In fact, $\sigma_n \in \Delta(\rho; n + p + m)$.) By similar reasoning, for all $n \in N$, $\tau_n = \mathbf{s}_{ld}^{(n+1)}(\rho) \in \mathbf{N}(\rho; n+1) - \mathbf{N}(\pi; n)$. (Specifically, $\tau_n \in \Delta(\pi; n + p - m + 1)$.) See Fig. 4(b).

Coalescing cases (a), (b), for all n, there exist $\sigma_n \in \mathbf{N}(\pi; n) - \mathbf{N}(\rho; n')$ and $\tau_n \in \mathbf{N}(\rho; n') - \mathbf{N}(\pi; n)$; $n' = n$ in case (a), and $n' = n + 1$ in case (b). By Lemma 5.1, then, it follows that, for all $n \geqq \max\{\Sigma(\pi), \Sigma(\rho)\} - d$, the following inequalities hold. On the one hand, $\mathcal{N}(\rho; n') < \mathbf{r}(\sigma_n) \leqq \mathcal{N}(\pi; n)$, while on the other hand,

$\mathcal{N}(\pi\,; n) < \mathbf{r}(\tau_n) \leqq \mathcal{N}(\rho\,; n')$. These inequalities are clearly absurd, so we conclude that $\mathbf{r}$ cannot be centered at both π and ρ. □

5.3. Growth rate of local diameters of preservation. In this section we establish an upper bound on the rate of growth of local diameters of preservation and a lower bound on the rate of growth of cumulative local diameters. Both bounds are achievable by a large family of realizations.

Let $\delta(\pi, \rho)$ be the (L_1) distance between π and ρ; that is, $\rho \in \Delta(\pi\,; \delta(\pi, \rho))$.

LEMMA 5.2. *Let* $\mathbf{r}$ *realize* Ω_d. *For all* $\pi, \rho \in N^d$ *and all* $n \in N$,

$$D_{\mathbf{r}}(n, \pi) \leqq D_{\mathbf{r}}(n + \delta(\pi, \rho), \rho).$$

Proof. $\mathbf{N}(\pi\,; n) \subset \mathbf{N}(\rho\,; n + \delta(\pi, \rho))$. □

THEOREM 5.4. *For all* $d \in N$, *there is a realization* $\mathbf{r}$ *of* Ω_d *for which*

$$D_{\mathbf{r}}(n, \pi) \leqq \binom{\Sigma(\pi) + n}{d}.$$

Proof. Let $\mathbf{r}$ be centered at ε. Invocation of Theorem 3.1 and Lemma 5.2 yields the theorem since $\delta(\varepsilon, \pi) = \Sigma(\pi) - d$. □

Our lower bound on local diameter of preservation is in terms of D^* rather than D. A reasonable (almost everywhere) lower bound on the growth rate of D has eluded us and remains an inviting challenge. In certain cases, the growth rate of $D^*_{\mathbf{r}}$ reflects more accurately the behavior of $\mathbf{r}$ than does that of $D_{\mathbf{r}}$, since the former rate ignores the perturbations possible in unconstrained realizations. This rationalization is not intended to downplay the desirability of determining $D_{\mathbf{r}}$'s growth rate, but only to suggest that the rectangularity of arrays might well render $D^*_{\mathbf{r}}$ the more important measure of the efficiency of $\mathbf{r}$. Moreover, our universal lower bound for $D^*_{\mathbf{r}}$ does yield an infinitely-often lower bound on the growth of $D_{\mathbf{r}}$.

THEOREM 5.5. *Let* $d \in N$ *be arbitrary. There is a constant* $c > 0$, *depending only on d, such that, for all realizations* $\mathbf{r}$ *of* Ω_d, *and for all* $n \in N$ *and* $\pi \in N^d$, $D^*_{\mathbf{r}}(n, \pi) > c \cdot n \cdot (\max\{M(\pi), n\})^{d-1}$.

Proof. (a). If $n \geqq M(\pi)$, then the bound on D^* follows from Theorem 3.1 and the obvious fact that D^* majorizes D.

(b) Let $\pi \in N^d$ be arbitrary, and let $n \in N$ be such that $M(\pi) > n$. Let $\mathbf{r}$ be any realization of Ω_d. By definition of D^*, it follows easily that the (functional) restriction $\mathbf{r}'$ of $\mathbf{r}$ to the finite array $A = (N_{M(\pi)})^d$ has the following property: $\mathbf{r}'$ realizes A and *globally* preserves n-neighborhoods ($n < M(\pi) = m(\lambda_A)$) with diameter $D^*_{\mathbf{r}}(n, \pi)$. By Theorem 4.1, then, $D^*_{\mathbf{r}}(n, \pi) > c \cdot n \cdot (M(\pi))^{d-1}$ for some constant c depending only on d.

Parts (a) and (b) combine to yield the theorem. □

COROLLARY. *For all* $d \in N$ *there is a* $c > 0$ *such that, for all realizations* $\mathbf{r}$ *of* Ω_d *and all* $n \in N$, $D_{\mathbf{r}}(n, \pi) > c \cdot n \cdot M(\pi)^{d-1}$ *for infinitely many points* $\pi \in N^d$.

Many of the "nice" realizations of Ω_d discussed in [5], [6] come within a constant factor of attaining the lower bound of Theorem 5.5 and are, in this sense, optimal in preserving proximity among extendible realizations. Notable among these are the diagonal realizations which linearize the partial order [$\pi \leqq \rho$ iff

$\Sigma(\pi) \leqq \Sigma(\rho)$]—a two-dimensional version appears in Fig. 2—and the cubic shell realizations which linearize the partial order [$\pi \leqq \rho$ iff $M(\pi) \leqq M(\rho)$]. In two dimensions one cubic shell realization is given by $\mathbf{r}(\pi) = (M(\pi) - 1)^2 + M(\pi) + \pi_2 - \pi_1$; see Fig. 5.

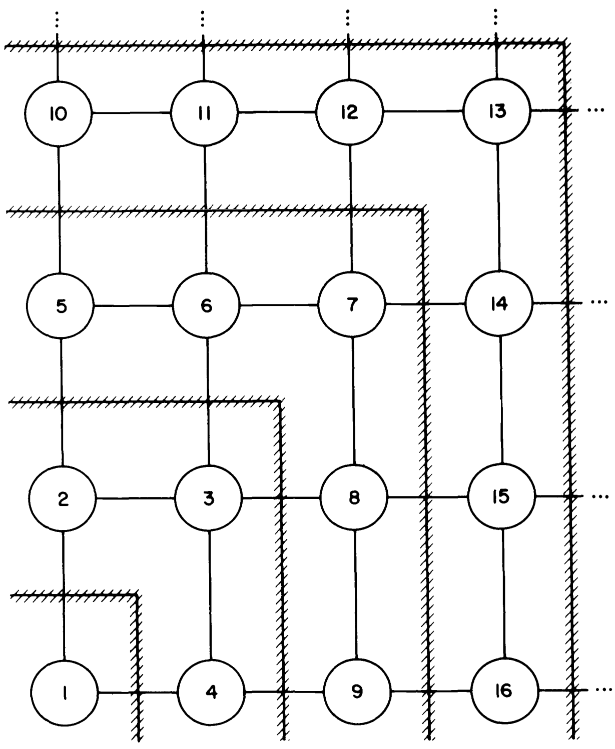


FIG. 5. *A realization of Ω_2 which is centered at ε under the L_∞ metric.*

5.4. Converting to other metrics. The results and proofs of § 5.3 depend, in their gross form, only on the rate of growth of d-dimensional neighborhoods (as determined in Theorem 3.1). Therefore, these results and proofs would be unaffected—except in the determination of specific constants—by a change to any other metric whose radius n d-dimensional neighborhoods enclose $O(n^d)$ lattice points; specifically, they would remain (grossly) valid for the L_2 and L_∞ metrics. The results and proofs in § 5.2, on the other hand, depend heavily on the specific metric used to define "neighborhood." We now indicate very informally how § 5.2 would change with adoption of a new metric.

1. Theorem 5.1 and its proof depend only on the finiteness of neighborhoods. Thus, realizations centered at an arbitrary point exist under both the L_2 and L_∞ metrics also; and their construction might proceed as described in the proof of Theorem 5.1.

2. The nonmonotonicity of realizations centered about points other than ε (Theorem 5.2) results from the "shape" of d-spheres. The result as stated holds when the spheres have no "flats" along the axes; that is, every line tangent to a sphere in the direction of an axis passes through at most one lattice point belonging

to the sphere. Thus, for example, the result, and at least the skeleton of the proof, remain valid under the L_2 metric, but they fail for the L_∞ metric. In fact, for the L_∞ metric, monotonic realizations centered at any point can be found.

3. Finally, our proof of the impossibility of centering a realization about two points (Theorem 5.3) depends on the way spheres overlap. Specifically, for any pair of points π and ρ, there must be "large" integers a and b—the analogue to Lemma 5.1 will tell just how large—such that the radius a sphere about π and the radius b sphere about ρ each contain points not contained by the other. Such conflict-exposing radii can be found for the L_2 metric as well as for the L_1 metric: When π and ρ share a coordinate, the proof for L_2 proceeds as does part (a) of the proof of Theorem 5.3; when π and ρ differ in all coordinates, the proof is similar to part (b) of the proof of Theorem 5.3, with "$n + m$" replacing "$n + 1$" everywhere; details are left to the reader. (The interested reader can verify that the proof for L_2 extends directly to any L_p metric.) An analogue of Theorem 5.3 cannot, however, be proved for the L_∞ metric. In fact, the cubic shell realization of Fig. 5 is centered at both $\langle 1, 1\rangle$ and $\langle 2, 1\rangle$.

6. Directions for further research.

The development in this paper has left unresolved a number of interesting questions about realizations of orthant arrays. We mention only a few of the important ones.

1. *Higher-dimensional storage.* All of our attention in this paper and in [5], [6] has been focussed on computer memories which are linear in structure; that is, our realizations map array schemes into N. Certain memory devices, disks being a prime example, are, to within a reasonable approximation, two-dimensional in structure; and one can easily envisage memory devices which, at least in certain locales, are e-dimensional for arbitrary fixed e. Thus, there is some motivation for reexamining the question of how to store arrays, in the context of memory with the structure of N^e for $e > 1$. With respect to the current paper, it is certain that analogues of Theorem 4.3 and 5.1 can be proved for arbitrary "realizations" of Ω_d in Ω_e, with appropriately adjusted notions of "neighborhood preservation" and "center." (In fact, the proof of Theorem 4.3 which is based on Theorem 4.1 goes through intact for arbitrary $e < d$.) Can one establish analogues of Theorems 5.2–5.5 in this more general setting? Certain extensions are immediate, but others appear quite challenging. More generally, it would be interesting and valuable to see how (and if) one could generalize the results in [5], [6] to arbitrary e-dimensional storage.[7]

2. *Centered realizations.* An extendible realization can have one center, but none can have two centers. In response to an "open problem" in an earlier version of this paper (IBM Rep. RC-4874, 1974), D. Bollman [1] has shown that a realization can have any finite number of quasi-centers: Points $\pi_1, \cdots, \pi_k \in N^d$ are *quasi-centers* of the realization $\mathbf{r}$ of Ω_d if, given any points ρ, σ, $|\mathbf{r}(\rho) - \mathbf{r}(\sigma)| < k \cdot \min \mathcal{N}(\pi_i; \max\{\delta(\pi_i, \rho), \delta(\pi_i, \sigma)\})$. Bollman's result indicates that "approximate" centers can exist in extendible realizations and raises the interesting question of whether such points can be present in realizations which afford one efficient traversal of arrays [5], [7] or which approach optimality in utilization of

[7] Some related questions are considered in S. AMOROSO AND I. J. EPSTEIN, *Maps preserving the uniformity of neighborhood interconnection patterns in tessellation structures*, Inf. Contr., 25 (1974), pp. 1–9.

storage [6]. Obviously this question requires further formulation before it can be answered. Investigation of this issue would advance the effort in [7] to determine to what extent various types of optimality can coexist in extendible array realizations.

3. *Growth rate of* $D(n, \pi)$. The local diameter of proximity preservation $D(n, \pi)$ is a quantity fundamental to the understanding of injections of N^d into N. It would be interesting to determine if the rate of growth of $D(n, \pi)$ can be bounded below tightly. Almost assuredly, our present lower bounds (the obvious universal bound $D(n, \pi) \geqq \mathcal{N}(\pi; n)$ and the infinitely-often lower bound of the corollary to Theorem 5.5) can be strengthened. Our initial conjecture that the strengthening of the bound could take the form of an almost-everywhere version of the corollary has been refuted by D. Bollman [1]. Specifically, she has found, for each dimensionality d, a realization $\mathbf{r}$ of Ω_d, and an infinite sequence of points $\pi_1, \pi_2, \cdots$ in N^d such that, for all $m \in N$ and all $n \leqq m$, $D_{\mathbf{r}}(n, \pi_m) = \mathcal{N}(\pi_m; n)$. Her results about and discussion of this problem merit the attention of anyone interested in this area of investigation.

4. *Complexity of realizations*. As a final note, we raise the persistent problem of how one might gauge the computational complexity inherent in realizations of Ω_d. Of course, should one be able to bound this complexity from below, he then faces the problem of relating this measure of complexity to other measures such as efficiency of traversal [5], [7], efficiency of storage utilization [6], and efficiency of local preservation of proximity (as measured, say, by the rate of growth of D or of D^*).

Acknowledgment. The suggestions, criticisms, and encouragement of Nicholas Pippenger, Larry Stockmeyer, and Shmuel Winograd are gratefully acknowledged.

REFERENCES

[1] D. Bollman, *Some tailor-made extendible array realizations*, IBM Rep. RC-5121, Yorktown Heights, N.Y., 1974.

[2] R. M. Karp, A. C. McKellar and C. K. Wong, *Near-optimal solutions to a two-dimensional placement problem*, IBM Rep. RC-4740, Yorktown Heights, N.Y., 1974.

[3] D. E. Knuth, *The Art of Computer Programming I: Fundamental Algorithms*, Addison-Wesley, Reading, Mass., 1968.

[4] J. P. Mylopoulos and T. Pavlidis, *On the topological properties of quantized spaces I, II*, J. Assoc. Comput. Mach., 18 (1971), pp. 239–246, 247–254.

[5] A. L. Rosenberg, *Allocating storage for extendible arrays*, Ibid., 21 (1974), pp. 652–670.

[6] ———, *Managing storage for extendible arrays*, this Journal, 4 (1975), pp. 287–306.

[7] L. J. Stockmeyer, *Extendible array realizations with additive traversal*, IBM Rep. RC-4578, Yorktown Heights, N.Y., 1973.

[8] V. Strassen, *Gaussian elimination is not optimal*, Numer. Math., 13 (1969), pp. 354–356.

[9] C. K. Wong and T. W. Maddocks, *A Generalized Pascal's triangle*, Fibonacci Quart., 13 (1975), pp. 134–136.

SIAM J. COMPUT.
Vol. 4, No. 4, December 1975

RESPONSE TIME OF A FIXED-HEAD DISK TO TRANSFERS OF VARIABLE LENGTH*

EROL GELENBE†, JACQUES LENFANT‡ AND DOMINIQUE POTIER¶

Abstract. Due to the practical complexity of addressing variable length records placed in arbitrary locations of a fixed-head disk (or drum), and because of difficulty of managing secondary memory space in such cases, variable length records are often stored with their first address at a fixed location of the magnetic support. We present a queuing model of such a scheme, assuming a Poisson arrival stream and arbitrary distributed record lengths. The stationary probability distribution of the number of transfer requests in queue and the expected response time are obtained. Numerical examples illustrating the results are presented.

Key words. imbedded Markov chains, disk units, input–output, memory management

1. Introduction. Even though the trend in modern computer systems is to replace rotating secondary memory devices by core or semiconductor memories, drums and more particularly fixed-head disks remain in wide usage while assuring important input-output functions in newer systems. The analysis of their performance remains of interest also because newer "rotating" memory devices such as magnetic bubble memories retain some of their information accessing and transfer characteristics.

The purpose of this paper is to analyze the response time of a fixed-head (as opposed to moveable arm) rotating secondary memory device when the length of the records to be transferred obeys some arbitrary distribution function and when the records are transferred in first come, first served (FCFS) order. The method of analysis of such devices, and the results obtained, will depend to a large extent on the scheduling algorithm used as well as on the nature of the transfers. Thus Coffman [1] has examined the behavior of a drum whose circumference is divided into equal sized sectors each containing a page, and for which page transfers are requested singly (as opposed to batched requests). In his model, the drum (or fixed-head disk) is scheduled using the Eschenbach scheme [2] with one queue per sector. In [3], an algorithm for optimally scheduling variable length transfers for a drum has been given and it has been shown that it is a solution of a special case of the traveling salesman problem. A comparison of the algorithm in [3] with the classical SLTF (shortest-latency-time-first) algorithm has been presented by Stone and Fuller [5].

* Received by the editors August 13, 1973, and in final revised form October 4, 1974. This work was performed in cooperation with Compagnie Internationale pour l'Informatique.

† Laboratoire de Recherche en Informatique et Automatique, Institut de Recherche d'Informatique et d'Automatique, 78150 Rocquenfort, France, and Chaire d'Informatique de l'Université de Liège, Liège, Belgium.

‡ Laboratoire de Recherche en Informatique et Automatique, Institut de Recherche d'Informatique et d'Automatique, 78150 Rocquencourt, France, and Departement de Mathématiques et Informatique, Université de Rennes, Rennes, France.

¶ Laboratoire de Recherche en Informatique et Automatique, Institut de Recherche d'Informatique et d'Automatique, 78150 Rocquencourt, France.

In this paper, in addition to our assumption of arbitrarily distributed length of records to be transferred, we also suppose that the starting address on the drum is at a fixed angular position on the circumference for *all* records. This is a realistic assumption in some operating systems (e.g. UNIX [13]) which use a similar policy because of the great ease and simplicity it implies. The records to be transferred in such a system might be segments, file records or groups of pages belonging to the same program or system module for which "pre-paging" is used.

In the sequel, we shall obtain the long run probability distribution for the number of requests waiting for transfer in the system we have described and use it to compute the expected response time of the disk with the assumptions we have made regarding its operation; the arrival process of requests for transfers is assumed to be Poisson and the length of each transfer obeys an arbitrary distribution with finite first and second moments.

Assuming that the starting addresses of records are drawn from a uniform distribution around the circumference of the disk, Fuller [11] obtains the mean response time with the shortest-latency-time-first and other "optimal" schedules for given probability distribution functions of the size of records to be transferred using simulation experiments. We provide numerical results to compare the performance of our policy with that of a first in, first out (FIFO) schedule (with which it seems to compare favorably) and with these optimal policies.

2. The model. The system we wish to analyze is shown in Fig. 1. A fixed starting address at a given angular position is established on the surface of the fixed-head disk (or drum, henceforth called the *disk*) for all records. For the cylindrical drum this address would correspond to a line, along the axis of rotation, on its surface. Concentric circles on the disk correspond to tracks. Even though the physical length of the tracks diminishes as the center of the disk is approached, they all contain the same number of words. A record beginning on one track may span several of them.

The lengths of records to be transferred are independent and identically distributed random variables with an arbitrary distribution function $F(x)$ having finite first and second moments. We shall define the probability r_n that a record is transferred in n rotations of the disk, beginning from its starting address:

$$\begin{aligned} r_n &= \text{Prob}\,[(n-1)T < x \leqq nT] \\ &= F(Tn) - F(T(n-1)), \qquad n = 1, 2, \cdots, \end{aligned} \tag{1}$$

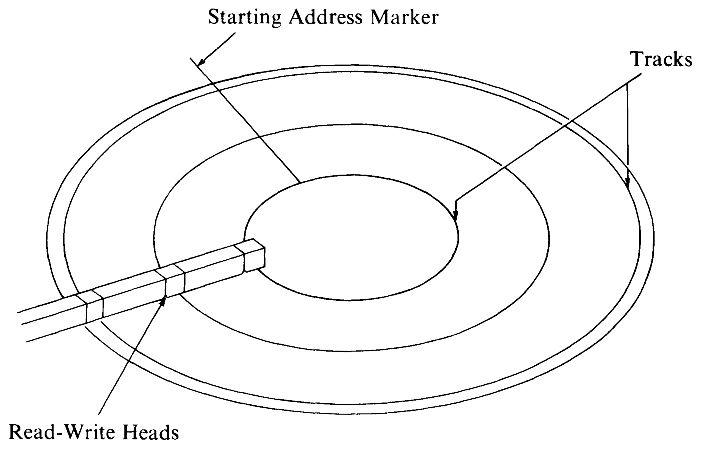
where T is the time necessary for one complete disk rotation. We have implicitly assumed that the length x of a record is given in units of time, and that it is equal to the time necessary for the disk heads to move from its starting address to its ending address.

We assume that the interarrival times of transfer requests are independent and identically distributed random variables with distribution function

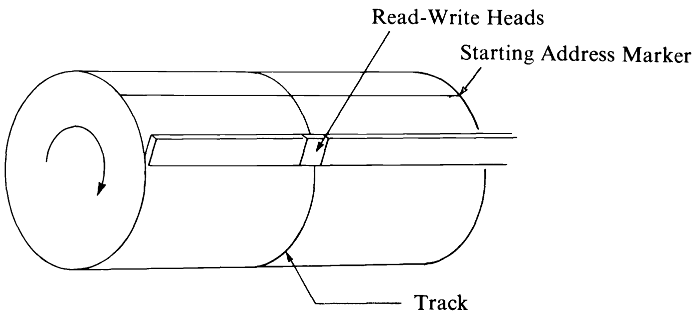
$$G(t) = 1 - e^{-\lambda t}$$

so that the probability of k arrivals in time T is

$$a_k = e^{-\lambda T}\frac{(\lambda T)^k}{k!}, \qquad k = 0, 1, 2, \cdots. \tag{2}$$



(a) Fixed-Head Disk



(b) Drum

FIG. 1

At an instant of time, t, let $M(t)$ be the number of requests awaiting transfer (including the record being transferred). The process $\{M(t)\}$ is not a Markov chain, unless $F(x)$ is an exponential distribution function. Since we cannot compute directly the distribution of $M(t)$, let us adjoin the supplementary variable $N(t)$ to $M(t)$ and consider the process $\{M(t), N(t)\}$, where the disk revolution for the record being transferred is the $N(t)$th. $\{M(t), N(t)\}$ is not, in the general case, a Markov chain either. We may consider, however, the Markov chain C imbedded in $\{M(t), N(t)\}$ at time $t = qT, q = 0, 1, \cdots$, just *after* the heads pass over the starting address marker of the disk. For convenience, we say that $N(t)$ is undefined when $M(t) = 0$.

The state of C takes the following values:

0 if $M(qT) = 0$,

(m, n) if $M(qT) = m > 0$ and the disk resolution starting at qT is the nth for the record being transferred,

for $q = 0, 1, 2, \cdots$ and $n = 1, 2, \cdots$.

That C is a Markov chain is easily verified. Its transition probabilities are obtained as follows:

$$\Pr[0|0] = a_0, \tag{3}$$

$$\Pr[(m, 1)|0] = a_m, \qquad m \geqq 1, \tag{4}$$

where $\Pr[v|u]$ is the probability of entering state v of C at time $(q + 1)T$ given that the chain is in state u at qT. We also have

$$\Pr[(m,n)|(m-j, n-1)] = a_j c_{n-1} \quad \text{for } m \geqq 1,\ n \geqq 2,\ 0 \leqq j \leqq m, \tag{5}$$

$$\Pr[(m,1)|(m-j+1, n)] = a_j(1 - c_n) \quad \text{for } m \geqq 1,\ n \geqq 1,\ 0 \leqq j \leqq m. \tag{6}$$

$$\Pr[0|(1,n)] = a_0(1 - c_n), \tag{7}$$

where

$$c_n = \Pr[x > Tn | x > T(n-1)] = \frac{1 - F(Tn)}{1 - F(T(n-1))}. \tag{8}$$

Notice that we also have

$$c_n = \frac{\sum_{j=n+1}^{\infty} r_j}{\sum_{j=n}^{\infty} r_j}. \tag{9}$$

The transition probabilities are zero for all cases not covered by equations (3) to (7).

If i, j are states of C ($i, j \in \{0\} \cup \{(m, n)\}$ where $m \geqq 1$, $n \geqq 1$), let $\Pi(i, j, q)$ be the q-step transition probability of C from i to j. From [4] we know that

$$\Pi_j = \lim_{q \to \infty} \Pi(i, j, q)$$

exists and is independent of i if

(a) C is aperiodic and irreducible, and

(b) there exist real numbers $\Pi_i > 0$ satisfying the equations

$$\sum_{\text{all } i} \Pi_i = 1,$$

$$\Pi_j = \sum_{\text{all } i} \Pi_i \Pi(i, j, 1) \quad \text{for each } j.$$

That C is aperiodic and irreducible is easily verified: it suffices to show, using (3) to (7), that there exists an integer $k > 0$ such that

$$\Pi(i, j, q) > 0$$

for all $q \geqq k$ [4], for all i, j. If (a) and (b) are satisfied, we say that C is ergodic.

THEOREM 1. *Denote Π_i by $\gamma(m, n)$ if $i = (m, n)$ and by $\gamma(0)$ if $i = 0$. If $E\{n\}\lambda T < 1$, where*

$$E\{n\} = \sum_{n=1}^{\infty} n r_n,$$

then C is ergodic and

$$\gamma(0) = 1 - \lambda T E\{n\}.$$

Furthermore, $P_n(z)$ the generating function

$$P_n(z) = \sum_{m=1}^{\infty} \gamma(m, n) z^m$$

is given by

$$P_n(z) = [1 - F((n-1)T)] P_1(z) [A(z)]^{n-1}$$

for $n \geqq 1$, where

$$A(z) = \sum_{k=0}^{\infty} a_k z^k = e^{-\lambda T(1-z)},$$

a_k being given by (2). *Also*

$$P_1(z) = \gamma(0) z \left[\frac{A(z) - 1}{z - R(A(z))} \right],$$

where

$$R(z) = \sum_{n=1}^{\infty} r_n z^n.$$

Theorem 1, as well as the next result we present, will be proved in later sections.

The result we give above yields the joint probability distribution, at the stationary state, for the number of records awaiting transfer and for the index of the current revolution for the record being transferred, at instants of time right after the disk heads pass over the starting address. We would also like to have information regarding the number of records awaiting transfer at *any* instant of time. For this purpose, we proceed as follows.

Let $p_m(u)$ be the stationary probability of finding m records waiting to be transferred at instants of time $qT + uT$, $q = 0, 1, 2, \cdots$, where $0 < u < 1$. Let $P(z, u)$ be the generating function

$$P(z, u) = \sum_{m=0}^{\infty} p_m(u) z^m. \tag{10}$$

Consider the function $P(z)$ defined as the mean of $P(z, u)$ over the interval $[0, 1]$:

$$P(z) = \int_0^1 P(z, u)\, du.$$

The expected number of requests waiting for transfer (including the one being transferred) at an arbitrary instant of time is then $E\{M\}$, given by

$$E\{M\} = \lim_{z \to 1} \frac{d}{dz} P(z).$$

THEOREM 2. *If* $\lambda T E\{n\} < 1$, *then*

$$E\{M\} = \lambda T[E\{l\} + \tfrac{1}{2}] + \frac{(\lambda T)^2}{2}\frac{E\{n^2\}}{[1 - \lambda T E\{n\}]},$$

where, x being the length of a transfer, l is given by

$$l = x/T$$

and

$$E\{n^2\} = \sum_{n=1}^{\infty} n^2 r_n.$$

Remark. It is known that certain results on the queuing analysis of input–output devices can be obtained as corollaries of a theorem of Skinner [7]. Theorem 1 of this paper *cannot* be obtained from Skinner's results, but Theorem 2 can be deduced as a special case from [7].

3. Proof of Theorem 1. We omit the proof that C is aperiodic and irreducible. We shall only show that there exist numbers $\Pi_i > 0$ satisfying the equations

$$\sum_{\text{all } i} \Pi_i = 1$$

and

$$\Pi_j = \sum_{\text{all } i} \Pi_i \Pi(i, j, 1)$$

for each j under the conditions stated in Theorem 1. Adopting the notations of Theorem 1, we have

$$\gamma(0) = a_0\gamma(0) + \sum_{n=1}^{\infty} a_0(1 - c_n)\gamma(1, n) \tag{11}$$

from (3) and (7). Equations (4) and (6) yield for $m \geqq 1$,

$$\gamma(m, 1) = a_m\gamma(0) + \sum_{j=0}^{m}\sum_{n=1}^{\infty} a_j(1 - c_n)\gamma(m - j + 1, n). \tag{12}$$

Finally, from (5) we have

$$\gamma(m, n) = \sum_{j=0}^{m-1} a_j c_{n-1}\gamma(m - j, n - 1) \tag{13}$$

for $m \geqq 1, n > 1$.

We now use these equations to compute the generating functions $P_n(z)$:

$$P_n(z) = \sum_{m=1}^{\infty} \gamma(m, n) z^m.$$

From (11) and (12) we have

$$
\begin{aligned}
z[\gamma(0) + P_1(z)] &= a_0\gamma(0)z + \sum_{n=1}^{\infty} a_0(1 - c_n)\gamma(1, n) \\
&\quad + \sum_{m=1}^{\infty} z^{m+1}[a_m\gamma(0) \\
&\quad + \sum_{j=0}^{m} \sum_{n=1}^{\infty} a_j(1 - c_n)\gamma(m - j + 1, n)] \\
&= z\gamma(0)A(z) + \sum_{n=1}^{\infty} (1 - c_n)A(z)P_n(z).
\end{aligned} \tag{14}
$$

Using (13), we obtain for $n > 1$,

$$P_n(z) = \sum_{m=1}^{\infty} \sum_{j=0}^{m-1} a_j c_{n-1}\gamma(m - j, n - 1)z^m \tag{15}$$

yielding

$$P_n(z) = c_{n-1}A(z)P_{n-1}(z), \qquad n = 2, 3, \cdots,$$

which when solved gives us for $n > 1$,

$$P_n(z) = \left[\prod_{j=1}^{n-1} c_j\right] P_1(z)[A(z)]^{n-1}. \tag{16}$$

Notice that by (8) we have

$$\prod_{j=1}^{n-1} c_j = 1 - F(T(n - 1))$$

so that $P_n(z)$ is of the form

$$P_n(z) = [1 - F(T(n - 1))]P_1(z)[A(z)]^{n-1} \tag{17}$$

which was to be shown.

From (14) we obtain

$$P_1(z) = A(z)\left[\gamma(0) + \frac{1}{z}\sum_{n=1}^{\infty} (1 - c_n)P_n(z)\right] - \gamma(0)$$

and using (17), we have

$$\sum_{n=1}^{\infty} (1 - c_n)P_n(z) = P_1(z) \sum_{n=1}^{\infty} [1 - F(T(n - 1))](1 - c_n)[A(z)]^{n-1}.$$

But (1) and (8) yield

$$[1 - F(T(n - 1))](1 - c_n) = r_n$$

and

$$R(A(z)) = \sum_{n=1}^{\infty} r_n[A(z)]^n.$$

Therefore $P_1(z)$ is of the form

$$(18)\quad \begin{aligned} P_1(z) &= \gamma(0)A(z) + \frac{1}{z}P_1(z)R(A(z)) - \gamma(0) \\ &= z\gamma(0)\left[\frac{A(z) - 1}{z - R(A(z))}\right] \end{aligned}$$

which was to be shown. To obtain $\gamma(0)$ we use the equality

$$\gamma(0) + \lim_{z\to 1} \sum_{n=1}^{\infty} P_n(z) = 1$$

which means that the sum of stationary probabilities over all states is 1. Using (17) we have

$$\begin{aligned} \lim_{z\to 1} \sum_{n=1}^{\infty} P_n(z) &= P_1(1) \sum_{n=1}^{\infty} [1 - F(T(n-1))] \\ &= P_1(1) \sum_{n=1}^{\infty} \left[1 - \sum_{j=1}^{n-1} r_j\right] \\ &= P_1(1) \sum_{n=1}^{\infty} \sum_{j=n}^{\infty} r_j \\ &= P_1(1) \sum_{n=1}^{\infty} nr_n. \end{aligned}$$

Therefore

$$\gamma(0) = 1 - E\{n\}P_1(1).$$

We compute $P_1(1)$ from (18) applying l'Hôpital's rule to obtain

$$P_1(1) = \gamma(0)\frac{\lambda T}{1 - \lambda TE\{n\}}.$$

Finally

$$\gamma(0) = 1 - \lambda TE\{n\}.$$

To insure egodicity we must have $\gamma(0) > 0$, therefore the condition

$$\lambda TE\{n\} < 1$$

must be satisfied.

4. Proof of Theorem 2. Consider the generating function $P(z, u)$ defined in (10). $p_m(u)$ is given by the following equation, in which $a_j(u)$ is the probability of j arrivals in time uT:

$$(19)\quad \begin{aligned} p_m(u) = \gamma(0)a_m(u) &+ \sum_{n=1}^{\infty} \sum_{k=1}^{m} (1 - g_n(u))\gamma(k, n)a_{m-k}(u) \\ &+ \sum_{n=1}^{\infty} \sum_{k=1}^{n} g_n(u)\gamma(k, n)a_{m-k+1}(u), \end{aligned}$$

where $g_n(u)$, the probability that a transfer is no longer than $T(n-1+u)$ given that it is longer than $T(n-1)$, is given by

$$g_n(u) = \frac{F(T(n-1+u)) - F(T(n-1))}{1 - F(T(n-1))}. \tag{20}$$

Let $A_u(z)$ be the generating function

$$A_u(z) = \sum_{j=0}^{\infty} a_j(u) z^j = e^{\lambda T u(z-1)}. \tag{21}$$

Then $P(z,u)$ is obtained from (19) and (21) as

$$P(z,u) = \gamma(0)A_u(z) + \sum_{n=1}^{\infty} (1 - g_n(u))A_u(z)P_n(z) + \sum_{n=1}^{\infty} g_n(u)A_u(z)\frac{P_n(u)}{z} \tag{22}$$

and

$$E\{M\} = \lim_{z\to 1} \int_0^1 \frac{\partial}{\partial z} P(z,u)\, du. \tag{23}$$

Notice that

$$\lim_{z\to 1} \frac{\partial}{dz} A_u(z) = \lambda T u$$

and that

$$\sum_{n=1}^{\infty} P_n(1) = 1 - \gamma(0).$$

Using these relations, we obtain after some algebra that

$$E\{M\} = \frac{\lambda T}{2} + \sum_{n=1}^{\infty} \frac{d}{dz} p_n(z)|_{z=1} - \int_0^1 du \sum_{n=1}^{\infty} g_n(u) P_n(1).$$

Notice from Theorem 1 and (21) that

$$P_n(1) g_n(u) = [F(T(n-1+u)) - F(T(n-1))]P_1(1)$$

and

$$\int_0^1 du\, P_n(1) g_n = P_1(1) \int_{n-1}^{n} [F(yT) - F((n-1)T)]\, dy.$$

We also have

$$\sum_{n=1}^{\infty} \int_0^1 du\, P_n(1) g_n(u) = P_1(1)\left[- \sum_{n=1}^{\infty} \int_{n-1}^{n} (1 - F(yT))\, dy + \sum_{n=1}^{\infty} (1 - F((n-1)T)) \right]$$

$$= \lambda T \left[- \int_0^{\infty} (1 - F(yT))\, dy + \sum_{n=1}^{\infty} nG(n) \right]$$

where we have used $P_1(1) = \lambda T$ and where

$$G(n) = F(nT) - F((n-1)T)$$

is the probability that a transfer will be complete during its nth rotation, and we have used the fact that

$$1 - F((n-1)T) = \sum_{m=n}^{\infty} G(m).$$

Notice that

$$\int_0^{\infty} [1 - F(yT)]\,dy = \int_0^{\infty} \frac{du}{T}[1 - F(u)]$$
$$= \frac{u}{T}[1 - F(u)]|_0^{\infty} + \int_0^{\infty} \frac{u}{T}\,dF(u)$$

is obtained by a change of variables and an integration by parts. This yields

$$\int_0^{\infty} [1 - F(yT)]\,dy = E\{l\},$$

where x being the length of a transfer, l is given by $l = x/T$. We now have

$$E\{M\} = \frac{\lambda T}{2} + \lambda T E\{l\} - \lambda T E\{n\} + \sum_{n=1}^{\infty} \frac{d}{dz} P_n(z)|_{z=1}.$$

The remaining algebra to complete the proof of Theorem 2 is straightforward, and we omit presenting it here.

5. Numerical examples and conclusions. In this paper, we have analyzed the behavior of a fixed-head disk or drum used for storing variable length records. The initial address of each record corresponds to a fixed angular position on the disk surface (or drum circumference). This placement method has considerable advantages over "optimal" scheduling methods because it simplifies the addressing problems of the secondary memory device.

It is also known that SLTF or optimal schedules for variable length records cannot be implemented efficiently unless special hardware features, which are often nonexistent on commercially available equipment, are installed. The major disadvantage of the policy we have analyzed seems to be the wastage in disk space which may be incurred, although it avoids the complicated placement algorithms one would use if an attempt were made to minimize wasted space on the disk.

The policy we propose has to be compared with other placement policies or scheduling methods with respect to three points:

(a) the response time,

(b) the wastage (or internal fragmentation) of disk space,

(c) the complexity of the addressing and scheduling policy.

It is clear that with respect to (c) it is difficult to find policies more economical than ours. For (a) we have compared on Figs. 2 and 3 our results with Fuller's [11] simulations of various "optimal" scheduling strategies. In [11] the initial addresses of the records are assumed to be uniformly distributed along a disk track. On the same figures the performance of a FCFS scheduling algorithm with the same assumptions regarding the distribution of records lengths and the distribution

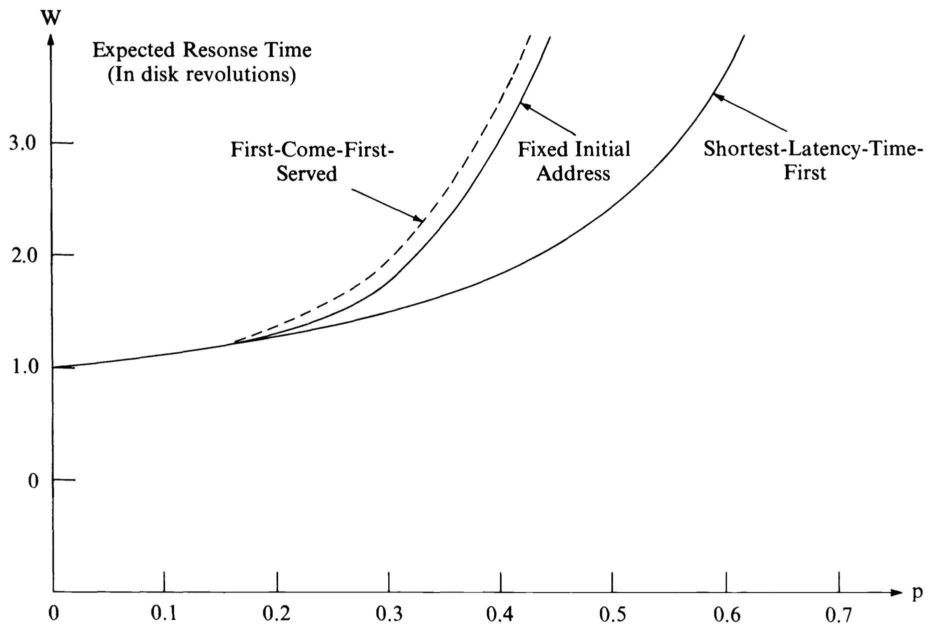


FIG. 2. *The expected response time when the records are uniformly distributed from zero to a full disk revolution*

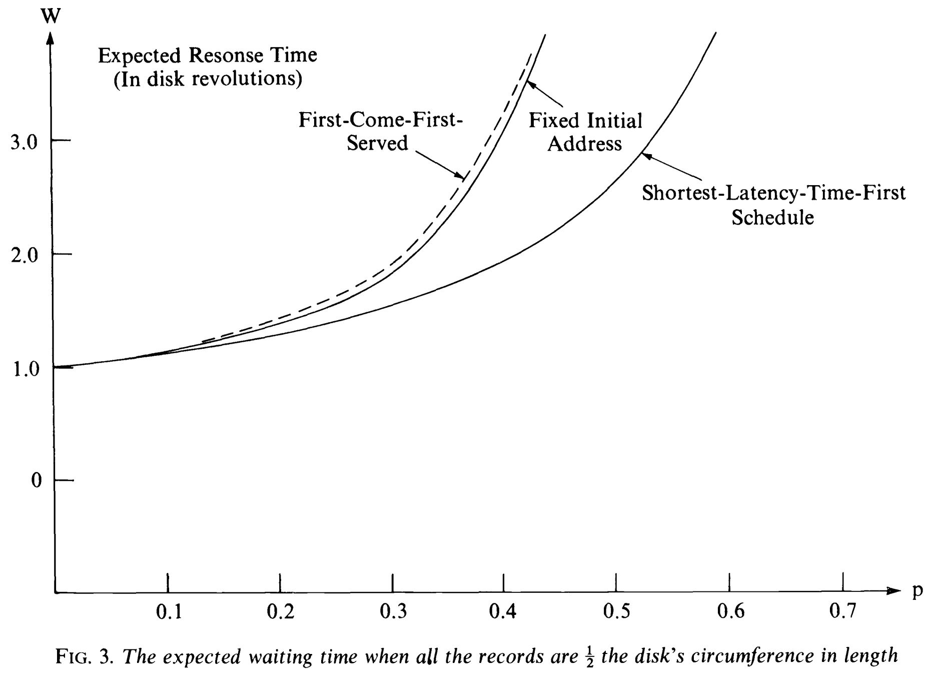


FIG. 3. *The expected waiting time when all the records are $\frac{1}{2}$ the disk's circumference in length*

of the initial address of records is shown. For all of the cases considered we see that our policy yields mean response times which are slightly better than with FCFS but considerably worse than "optimal" policies with the same distribution function of record lengths.

In [11] results for large record lengths are not given: in this case one might expect that optimal policies will yield response times comparable to that of FCFS scheduling.

Suppose that records are transferred on a FCFS basis from the disk, and that they are located at random on the disk surface. The appropriate model in this case is the $M/G/1$ queue [4] with a service time which is the sum of two independent random variables: an access time uniformly distributed between 0 and T, and a transfer time x with distribution function $F(x)$. By a straightforward application of the formula of Pollaczek and Khintchine we obtain the disk response time as:

$$W_R = T\left(E\{l\} + \frac{1}{2} + \lambda T \frac{E\{l\} + \frac{1}{3} + E\{l^2\}}{2(1 - \lambda T(\frac{1}{2} + E\{l\}))}\right). \tag{24}$$

Using Little's formula and Theorem 2 we obtain the expected response time for the algorithm we have analyzed:

$$W_F = T\left[E\{l\} + \frac{1}{2} + \lambda T \frac{E\{n^2\}}{2(1 - \lambda T E\{N\})}\right]. \tag{25}$$

Let us compare W_R and W_F for exponentially distributed record lengths. Normalizing to $T = 1$ we set $F(x) = 1 - e^{-\mu x}$, so that $E\{l\} = \mu^{-1}$, $E\{l^2\} = 2\mu^{-2}$, and we derive

$$E\{n\} = (1 - e^{-\mu})^{-1}, \qquad E\{n^2\} = (1 + e^{-\mu})(1 - e^{-\mu})^{-2}. \tag{26}$$

For large average record lengths, that is $\mu \ll 1$, it is easily seen that

$$E\{n\} \cong \mu^{-1} + 1/2, \qquad E\{n^2\} \cong 2\mu^{-2} + \mu^{-1} - 1$$

correct to $O(\mu)$ so that $W_F \cong W_R$. The approximation is very good even for moderate values of average record length (μ^{-1} of the order of 4 or 5).

As far as point (b) is concerned, first note that for small average record sizes with random placement, the "2/3 rule" of Knuth [9], [10] will come into effect so that on the average one third of disk space will be wasted due to the phenomenon of external fragmentation. An exact analysis of this is unavailable for primary storage since it is a very difficult problem and we do not attempt to solve here the still more complex case for disk space. For small average record size (smaller than T) our policy will waste an amount of space which is large compared to the utilized area. However, for large average record lengths, the wasted space will be approximately one half sector per record: one of the authors has shown elsewhere [12] that this result is exact for probability density functions of record lengths which have rational Laplace transforms (or equivalently which can be expressed as convex combinations of Erlang densities).

REFERENCES

[1] E. G. Coffman, *Analysis of a drum input/output queue under scheduled operation in a paged computer system*, J. Assoc. Comput. Mach., 16, 1 (1969), pp. 73–90.

[2] A. Weingarten, *The Eschenbach drum scheme*, Comm. ACM, 9 (1966), pp. 509–512.

[3] S. Fuller, *An optimal drum scheduling algorithm*, IEEE Trans. Electronic Computers, C-21 (1972), pp. 1153–1165.

[4] W. Feller, *An Introduction to Probability Theory and its Applications*, vol. 1, 3rd ed., John Wiley, New York, 1968.

[5] H. S. Stone and S. H. Fuller, *On the near-optimality of the shortest-latency-time-first drum scheduling discipline*, Comm. ACM, 16 (1973), pp. 352–353.

[6] S. H. Fuller and F. Baskett, *An analysis of drum storage units*, Tech. Rep. 26, Digital Systems Laboratory, Stanford Univ., Stanford, Calif., 1972.

[7] C. E. Skinner, *A priority queuing system with server-walking type*, Operations Res., 15 (1967), pp. 278–285.

[8] J. Abate and H. Dubner, *Optimizing the performance of a drum-like storage*, IEEE Trans. Electronic Computers, C-18 (1969), pp. 992–996.

[9] D. E. Knuth, *The Art of Computer Programming, Volume I: Fundamental Algorithms*, John Wiley, New York, 1969.

[10] E. Gelenbe, *The two-thirds rule for dynamic storage allocation under equilibrium*, Information Processing Letters, 1 (1971), pp. 59–60.

[11] S. H. Fuller, *Random arrivals and MTPT disk scheduling disciplines*, Tech. Rep. 29, Digital Systems Laboratory, Stanford Univ., Stanford, Calif., 1972.

[12] E. Gelenbe, J. C. A. Boekhorst and J. L. W. Kessels, *Minimizing wasted space in partitioned segmentation*, Comm. ACM, 16 (1973), pp. 343–349.

[13] D. M. Ritchie and K. Thompson, *The UNIX Time-sharing System*, Operating Systems Review (ACM-SIGOPS), 7 (1973), p. 27.

SIAM J. COMPUT.
Vol. 4, No. 4, December 1975

THE ENUMERATION OF GENERALIZED DOUBLE STOCHASTIC NONNEGATIVE INTEGER SQUARE MATRICES*

D. M. JACKSON† AND G. H. J. VAN REES‡

Abstract. The problem of enumerating generalized double stochastic integer square matrices is considered. The superposition theorem is used in conjunction with Schur functions to obtain the counting series for the 5×5 and 6×6 cases.

Key words. enumeration, stochastic matrices, generating functions, partitions, Schur functions, symmetric group

1. Introduction. Let $H_m(n)$ be the number of $m \times m$ matrices over $\mathbb{N} \equiv \{0, 1, 2, \cdots\}$ with line sum $n \in \mathbb{N}$. The purpose of this paper is to give the ordinary generating functions $\Phi_5(t)$, $\Phi_6(t)$ for the sequences $\{H_5(n)\}$ and $\{H_6(n)\}$ respectively with the anticipation that these additional sequences may be of service in determining the relationship among the $\Phi_i(t)$. The sequences for $\{H_3(n)\}$, $\{H_4(n)\}$ have been given by Sloane [1], together with references to the papers in which they first appear. The theorems supporting the computational method are given in § 2, while § 3 contains examples and a special case. The initial segments of sequences $\{H_i(n)\}$ and the generating functions $\Phi_i(t)$ are tabulated in § 4 for $i = 2, 3, 4$ (for completeness) and for $i = 5, 6$. Use has been made of Theorem 2, which is amenable to automatic computation.

2. Main theorem. The following result permits the computation of $H_m(n)$ by polynomial interpolation. The symmetry relation reduces the number of values of n at which $H_m(n)$ need be computed for interpolatory purposes.

CONJECTURE (Anand–Dumir–Gupta [2]).

(i) *$H_m(n)$ is a polynomial in n of degree $(m - 1)^2$.*

(ii) $H_m(n) = (-1)^{m-1} H_m(-m-n)$.

Proof. See Stanley [3].

Values of $H_m(n)$ may be computed according to the following theorem.

THEOREM 1. *$H_m(n) = N(h_n^m * h_n^m)$, where*

(i) *h_n is the cycle index polynomial in the indeterminates $x_1 \cdots x_n$ for the symmetric group S_n;*

(ii) *if $A(\mathbf{x})$, $B(\mathbf{x})$ are two multivariate polynomials in $x_1 \cdots x_n$ such that*

$$A(\mathbf{x}) = \sum_{(\mathbf{i})} a_{\mathbf{i}} \mathbf{x}^{\mathbf{i}},$$

$$B(\mathbf{x}) = \sum_{(\mathbf{i})} b_{\mathbf{i}} \mathbf{x}^{\mathbf{i}},$$

where the summation is over all partitions $(\mathbf{i}) = 1^{i_1} 2^{i_2} \cdots n^{i_n}$ of n, then the

* Received by the editors June 17, 1974. This work was supported by the National Research Council of Canada.

† Department of Combinatorics and Optimization, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1. This work was done while the author was visiting at Department of Pure Mathematics and Applied Statistics, University of Cambridge, Cambridge, England.

‡ Department of Combinatorics and Optimization, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1.

inner product $A * B$ is defined by

$$A(\mathbf{x}) * B(\mathbf{x}) = \sum_{(\mathbf{i})} a_{\mathbf{i}} b_{\mathbf{i}} g(\mathbf{i}) \mathbf{x}^{\mathbf{i}},$$

where $g(\mathbf{i}) = (1^{i_1} 2^{i_2} \cdots n^{i_n}) i_1 ! i_2 ! \cdots i_n !$;

(iii) $N(A(\mathbf{x})) = A(\mathbf{x})|_{\mathbf{x}=(1,1,\ldots,1)}$.

Proof. See Read [4].

The use of Theorem 1 is illustrated by Example 1 of § 3.

Since h_n^m is a symmetric function, the computation of $H_m(n)$ may be simplified by expressing h_n^m as a linear combination of Schur functions and by using the orthogonality relation for Schur functions.

Let $\{\lambda\}$ be the Schur function associated with the partition (λ) of n. Then $h_n = \{n\}$ gives the representation of the cycle index polynomial for S_n in terms of Schur functions.

Lemma 1. $h_n^m = \sum_{(\lambda)} \alpha_\lambda \{\lambda\}$, *where the summation is over all partitions of mn.*

Proof. h_n^m is a symmetric function.

Lemma 2 (Orthogonality relation).

$$N(\{\lambda\} * \{\mu\}) = \begin{cases} 1 & \textit{if } (\lambda) \equiv (\mu), \\ 0 & \textit{otherwise}. \end{cases}$$

Proof. See Littlewood [5].

Theorem 2. $H_m(n) = \Sigma_{(\lambda)} \alpha_\lambda^2$.

Proof. The proof is direct from Theorem 1 and Lemmas 1 and 2.

The remaining problem of expressing a symmetric polynomial as a linear combination of Schur functions may be carried out by a method given by Littlewood [5].

Let $\{\lambda\}$ be the Schur function corresponding to a partition (λ) of r. To evaluate $\{\lambda\}\{n\}$, construct the Young diagram for (λ) using "asterisks". Add to this diagram n "dots" in all possible ways, subject to the conditions

(i) resulting diagram is a Young diagram in the two symbols,

(ii) no two "dots" lie in the same vertical line.

The expansion of $\{\lambda\}\{n\}$ is the sum of the Schur functions corresponding to the partitions of $r + n$ generated in this manner. Example 2 of the following section illustrates this procedure.

3. Examples.

Example 1. Theorem 1 may be usefully employed for small values of n, for which the exponentiation of the polynomial h_n is readily constructed. We shall compute $H_m(2)$. Now $h_2 = \frac{1}{2}(x_1^2 + x_2)$, the cycle index polynomial for S_2. Therefore

$$H_m(2) = N(h_2^m * h_2^m) = N\left(\frac{1}{2^m}(x_1^2 + x_2)^m * \frac{1}{2^m}(x_1^2 + x_2)^m\right)$$

$$= \frac{1}{4^m} \sum_{\substack{i_1, i_2 \geqq 0 \\ i_1 + i_2 = m}} \binom{m}{i_1}^2 1^{2i_1} 2^{i_2} (2i_1)! (i_2)!$$

$$= \frac{(m!)^2}{2^m} \sum_{i_1=0}^{m} \frac{2^{-i_1}}{(m - i_1)!} \binom{2i_1}{i_1}.$$

This is in agreement with known results.

Example 2. We shall compute $H_3(2)$ using Theorem 2.

We shall represent $\{\mu\}$ by $[G(\mu)]$, where $G(\mu)$ is the Young diagram for (μ), and write $\{\mu\} \equiv [G(\mu)]$.

Then $\{2\} \equiv [* *]$, so

$$\{2\}^2 = [* * \cdot \cdot] + \left[\begin{smallmatrix} * & * & \cdot \\ \cdot & & \end{smallmatrix}\right] + \left[\begin{smallmatrix} * & * \\ \cdot & \cdot \end{smallmatrix}\right],$$

whence

$$\{2\}^2 = \{4\} + \{3,1\} + \{2^2\}.$$

Finally

$$\begin{aligned}\{2\}^3 &= \{2\}^2\{2\} = \{4\}\{2\} + \{3,1\}\{2\} + \{2^2\}\{2\} \\ &= \{6\} + 2\{5,1\} + 3\{4,2\} + 2\{3,2,1\} + \{4,1^2\} + \{3^2\} + \{2^3\}\end{aligned}$$

Thus, from Theorem (2),

$$H_3(2) = N(\{2\}^3 * \{2\}^3) = 1^2 + 2^2 + 3^2 + 2^2 + 1^2 + 1^2 + 1^2 = 21.$$

4. Tabulation of $\{H_m(n)\}$, $\Phi_m(t)$. The generating function for $\{H_m(n)\}$ is given by

$$\begin{aligned}\Phi_m(t) &= \sum_{n=0}^{\infty} t^n H_m(n) \\ &= \frac{f_m(t)}{(1-t)^{(m-1)^2+1}},\end{aligned}$$

TABLE 1
Tabulation of $a_i^{(m)}$ for $m = 2, 3, 4, 5, 6$

m	2	3	4	5	6
i 0	1	1	1	1	1
1		1	14	103	694
2		1	87	4306	184015
3			148	63110	15902580
4			87	388615	567296265
5			14	1115068	9816969306
6			1	1575669	91422589980
7				1115068	490333468494
8				388615	1583419977390
9				63110	3166404385990
10				4306	3982599815746
11				103	3166404385990
12				1	1583419977390
13					490333468494
14					91422589980
15					9816969306
16					567296265
17					15902580
18					184015
19					694
20					1

where $f_m(t) = \sum_{i=0}^{(m-1)(m-2)} a_i^{(m)} t^i$ and all $a_i^{(m)}$ are integers.

Note that

$$f_m(t) = t^{(m-1)(m-2)} f_m(t^{-1})$$

is a consequence of

$$H_m(n) = (-1)^{m-1} H_m(-n-m).$$

The values of $a_i^{(m)}$ are given in Table 1, while Table 2 contains the initial segments of $\{H_m(n)\}$.

TABLE 2

Initial segments of $\{H_m(n)\}$ for $m = 2, 3, 4, 5, 6$

m	$H_m(n)$
2	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
3	1, 6, 21, 55, 120, 231, 406, 665, 1035, 1540, 2211
4	1, 24, 282, 2008, 10147, 40176, 132724, 381424, 981541, 2309384, 5045326
5	1, 120, 6210, 153040, 2224955, 22069251, 164176640, 976395820, 4855258305, 2085679285, 79315936751
6	1, 720, 202410, 20933840, 1047649905, 30767936616, 602351808741, 8575979362560, 94459713879600, 842286559093240, 6292583664553881

REFERENCES

[1] N. J. A. SLOANE, *A Handbook of Integer Sequences*, Academic Press, New York, 1973.

[2] H. ANAND, V. C. DUMIR AND H. GUPTA, *A combinatorial distribution problem*, Duke Math. J., 33 (1966), pp. 757–770.

[3] R. P. STANLEY, *Linear homogeneous diophantine equations and magic labelings of graphs*, Ibid., 40 (1973), pp. 607–632.

[4] R. C. READ, *The enumeration of locally restricted graphs I*, J. London Math. Soc., 38 (1963), pp. 433–455.

[5] D. LITTLEWOOD, *The Theory of Group Characters*, Clarendon Press, Oxford, 1950.

SIAM J. COMPUT.
Vol. 4, No. 4, December 1975

ON SCHEDULING CHAINS OF JOBS ON ONE PROCESSOR WITH LIMITED PREEMPTION*

JOHN BRUNO† AND MICHA HOFRI‡

Abstract. A scheduling rule is given for determining the processing order of tasks which have the precedence structure of chains. It is assumed that the service times follow known distributions, that they are all independent, that costs are accrued by tasks at a constant rate until their service requirements are satisfied, that all the tasks are available at time 0 and that the service is interruptible at task-specific sets of points. The rule consists of computing for each chain an "optimal assignment" for its tasks and a rank function which depends on this assignment. Choosing at each point in time the chain with the smallest rank produces an optimal schedule. It is proved that the "optimal assignments" have the desirable property that as long as a task does not exceed its allotted service time, no preemption should take place.

Key words. preemptive scheduling, priority scheduling, precedence relations, stochastic service, rank schedule, mean flow time, linear costs

1. Introduction.

1. In this paper we consider a collection of n chains of tasks where the service requirement of each task is distributed according to some known distribution law and accrues cost at some constant rate until it completes its service requirement. There is a single server, and a finite number of preemptions are allowed. Our objective is to construct a scheduling policy which obtains, over all allowable policies, the minimum expected total cost.

2. Sevcik [6] and Schrage [5] have considered the case where all the tasks are independent and defined a "rank" function which they used to determine the optimal service assignments. The addition of order constraints in chains alters the cost structure of the problem in the sense that, if we view a chain of tasks as a single unit, the cost accrual rate of the chain varies with time; i.e., it decreases as individual tasks in the chain complete and depart. We have been able to extend the use of a rank function and the interchange argument used by Sevcik to handle chains of tasks.

3. A nonpreemptive, deterministic version of our problem has been studied [3], [7], and it is known that one must minimize a rank function over all prefixes of chains in order to determine an optimal assignment. Thus we were led to try to extend the idea of rank to prefixes of chains in the nondeterministic case with the criterion that this rank reduce to the known rank function for the deterministic case. More general precedence structures, such as trees, can be handled in the deterministic case, but we have been unable to extend the notion of rank to these. We shall number paragraphs and equations continuously through the paper and separately in the appendices. P1 is paragraph number 1 and (2) is equation number 2.

2. Description of the problem.

4. We consider a scheduling problem with the following characteristics: a single server; a set of n chains of tasks; the ith chain holds k_i tasks (tasks in a chain

* Received by the editors April 8, 1974, and in final revised form November 25, 1974.

† Computer Science Department, Pennsylvania State University, University Park, Pennsylvania 16802.

‡ Computer Science Department, Technion-Israel Institute of Technology, Haifa, Israel.

must be executed in a given order; one task may not start until all the tasks that precede it in the chain are completed). All the tasks are available at $t = 0$, and no further arrivals are taken into account. Cost is incurred by any task from time 0 until its service requirement is satisfied, at a constant rate w_{ij} for task T_{ij}, $i = 1(1)n$, $j = 1(1)k_i$. The service requirement of a task, S_{ij}, is distributed according to a known law, $F_{ij}(\cdot)$. We also assume that tasks may be interrupted and preempted at a set of instants $\{t\}_{ij}$, with t (the attained service) measured only when T_{ij} is being serviced. Suppose $\{t\}_{ij} = \{1, 3, 4, 9\}$. When the processor is assigned to T_{ij} for the first time, we may preempt T_{ij} (assuming T_{ij} does not finish) only after it has attained exactly 1, 3, 4 or 9 time units of service. If T_{ij} is preempted at, say, time 3, then the next time the processor is assigned to T_{ij}, the service may be preempted only after T_{ij} has obtained a *total* of exactly 4 or 9 time units of service. If at any time a task completes service, the processor may be assigned to another task even though we are not at a preemption point for T_{ij}.

5. We have found it necessary to choose between two classes of preemption epochs. One can imagine these as determined by the system (the server), where these epochs could be end of a shift, regeneration points of some internal cycle, and other examples where the status of the serviced task is not consulted. It is also possible to imagine these epochs as task-controlled; preemption may occur only at some opportunities during the service, and are defined by the stage of service of that task. In particular, in this approach a task service termination is always a preemptible epoch, whereas in the former it may not be. The analysis we pursue will always assume the latter structure.

6. We shall also assume that once a chain is preempted, it is immediately a candidate for assignment. No cost is incurred by preemption. If preemption occurs while a task (i, j) is being served, that task will be considered as the first in the remaining chain i, and to require a service, having already been served for a duration t, which is distributed according to the law $F_{ij|S_{ij}>t}(\cdot)$.

7. Our aim in scheduling these tasks is to minimize the expected total cost incurred, $C = \sum_{i=1}^{n} \sum_{j=1}^{k_i} w_{ij}E(d_{ij})$, when d_{ij} is the time of departure of the (i, j)th task. When $w_{ij} = 1$, this minimizes the expected mean flow time of the tasks.

8. We propose to show the optimal scheduling policy. We shall do it, however, under conditions which limit the *total number* of possible schedules to a *finite* number. Under this restriction, we shall prove the existence of a ranking method which at any occasion specifies which task is to be processed first, in order to minimize C.

9. The restriction to a finite policy space is done through the following two assumptions.

(a). Time is "discrete"—by which we mean that when a task is assigned, a finite time τ must elapse before it can be preempted.

(b). There is a finite M such that $F_{ij}(M) = 1$ for all i and j. Note however that since the only requirement is that n, k_i, τ and M be finite, with no explicit restriction imposed, the proposed method will handle correctly arbitrarily large system with *any* service distributions and preemption times that can be specified in practice. Note that we would obtain exactly the same result if nothing is said about M, but we only require that the sets $\{t\}_{ij}$ be all finite, and allow any task which has exhausted its preemptible epochs to be served to completion. We still require, though, $E(S_{ij})<\infty$.

10. An *assignment* consists of selecting a chain i, letting its first task get served up to a maximum time $q_1(q_1 \in \{t\}_{i1})$. If the task is not concluded, the chain is preempted and a new assignment has to be made. If the first task is concluded before time q_1 elapsed, the second task is immediately initiated and given a "quantum" of service not longer than $q_2(q_2 \in \{t\}_{i2} \cup \{0\})$, whence the same procedure is recursively followed. A chain is preempted perforce if a quantum in the assignment is given the value 0. Otherwise it is either serviced to completion of all the tasks or is preempted when any task is not concluded within its allotted quantum.

11. A *schedule* consists of an initial assignment and subsequent assignments. Since we must prescribe an assignment for any possible realization, a schedule can be described as a tree. In P20 we shall elaborate on this specification.

3. The rank of a chain and its properties.

12. The schedule we advocate is done on the basis of a rank defined for each chain. Define the random variable L as the number of tasks completed during an assignment, as described in P10. L can range for the ith chain from 0 to k_i, initially. We define rank as

$$(1)\qquad r(i)=\min_{\mathbf{q}} r(i,\mathbf{q}),\qquad q_j\in\{t\}_{ij}\cup\{0\},\qquad \mathbf{q}=(q_1,q_2,\cdots,q_{k_i}),$$

$$(2)\qquad r(i,\mathbf{q})=E_L\left\{\sum_{j=1}^{L}\tau_{ij}+q_{L+1}\right\}\Big/\{\bar{W}_{i,1}-E_L[\bar{W}_{i,L+1}]\},$$

$$(3)\qquad \bar{W}_{i,r}=\sum_{j=r}^{k_i}w_{ij},\qquad \tau_{ij}=E(S_{ij}|S_{ij}\leqq q_j)\quad\text{and}\quad q_{k_i+1}\triangleq 0.$$

13. Note that $P(L=l)=P(S_{i1}\leqq q_1)P(S_{i2}\leqq q_2)\cdots P(S_{il}\leqq q_l)P(S_{il+1}>q_{l+1})$, for $l=1(1)k_i-1$. Then $P(L=0)=P(S_{i1}>q_1)$ and $P(L=k_i)=P(S_{i1}\leqq q_1)\cdots P(S_{ik_i}\leqq q_{k_i})$. A representation of $r(i,\mathbf{q})$ we shall find convenient is obtained by writing $P(l)$ for $P(L=l)$ and $\bar{P}(l)$ for $P(L\geqq l)$. Then

$$(4)\qquad r(i,\mathbf{q})=\left\{\sum_{l=0}^{k_i-1}P(l)q_{l+1}+\sum_{l=1}^{k_i}\bar{P}(l)\tau_{il}\right\}\Big/\sum_{l=1}^{k_i}\bar{P}(l)w_{il}$$

14. When the discussion concerns a single chain—as it does, e.g., when we discuss the properties of the function $r(i,\mathbf{q})$ and in Appendices A and C—we shall omit the chain index.

15. There is no direct intuitive interpretation for $r(i,\mathbf{q})$, although it arises in a very natural fashion when one examines the relative merit of assigning two chains in alternative sequences (this will be clarified in P33). We note that for the case of single-task "chains", Sevcik [6] does impute to $r(i,\mathbf{q})$ some intuitive meaning, which may be carried through to our situation if we interpret the sum of the cost accrual rates of the tasks as the cost accrual rate of the chain. Then the denominator of (4) represents the decrease in the cost accrual rate of the chain due to the assignment $\mathbf{q}$. The numerator can be interpreted as the expected time the chain will occupy the server when assigned with quanta $\mathbf{q}$.

16. We present in the form of theorems the properties of the rank that we shall require for our main argument. First we note that the minimization indicated in (1) is done, by P9, over a finite set of finite intervals (as isolated points), so that a minimum exists. Its evaluation, however, may be a lengthy procedure. (See also P39–P41). A little help can be offered by the following.

THEOREM 1. *The value of q_1 which minimizes the rank of a chain with q_2 forced to be zero will be also the optimal value when the restriction is relaxed.*

A proof and a discussion of the theorem are given in Appendix A.

Some reflection of the meaning of optimal quanta—taking into consideration the stochastic independence of services to different tasks—may convince the reader that Theorem 1 is valid.

17. Another interesting property of $r(i)$—a property we shall find essential for our main argument—is given by Theorem 2.

THEOREM 2. *The rank of an assigned chain cannot increase during the service time, as long as the rank-quanta, $\mathbf{q}^*$, are not exceeded, where $\mathbf{q}^*$ are the quanta on which $r(i, \mathbf{q})$ obtains its minimum value.*

18. The intuitive idea of a rank which is possible in the degenerate cases of P15 would lead one to expect this result. It was gratifying to find it holds also in the more general case. A proof of Theorem 2 is given in Appendix B.

19. Finally we observe the following property which we conjecture to be true.

PROPERTY 3. *Let a chain be preempted (either because it exceeded a rank quantum or a task was finished and the next rank quantum has length* 0); *its present rank is no lower than it was when last assigned.*

Again, this is natural if we allow for the intuitive ordering property of a rank. There is more about it in Appendix C.

4. Optimality of smallest rank (SR) schedules.

20. A *schedule* is a labeled binary tree[1] where the labels correspond to assignments. If v is a vertex, than $\alpha(v)$ denotes the chain assigned at that vertex; i.e., $1 \leqq \alpha(v) \leqq n$, and $\beta(v)$ is the service allotment to be given the first task in chain $\alpha(v)$. If the task is not completed within time $\beta(v)$, then it is preempted and subsequent assignments are made according to the left subtree of v, called the *no-completion* (sub)schedule of v. Otherwise, as soon as the task completes its service requirement, the remaining assignments are made according to the right subtree of v, called the *completion* (sub)schedule of v.

Not every schedule is valid for a given problem—it is natural to require that there are no parts of a schedule which are useless and that it provides for all contingencies. When we use the word schedule we mean a valid schedule.

21. Two schedules will be called *equivalent* if they give rise to precisely the same performance under all circumstances. Let S be a schedule and v a vertex of S. We say that v is a *unit assignment* if $\beta(v)$ is as small as possible; S is a *unit schedule* if every vertex in S is a unit assignment. Clearly, any schedule can be transformed into an equivalent unit schedule (Fig. 1).

[1] A *binary tree* consists of a finite set of elements called *vertices* and is either empty or consists of a distinguished element called the *root* and a partition of the remaining elements into two binary trees, called the left and right subtrees of the root.

22. Let S be a unit schedule and v a vertex of S. We say that v is an SR vertex if $r(\alpha(v)) \leqq r(i)$ for all nonempty chains i; a unit schedule S is SR if every vertex in S is SR. A schedule is said to be an SR *schedule* if its corresponding unit schedule is SR.

23. The following theorem is the main result of this paper.

THEOREM 4. *Let P be a scheduling problem. A schedule for P is optimal if and only if it is an* SR *schedule.*

Proof. The number of possible assignment intervals for a task is called the index of a task. For example, a task which cannot be preempted until it obtains its service requirement has an index of 1, a task with one preemption point allowed has index 2, etc. The index of a scheduling problem is the sum of the indices of its tasks.

The proof is by induction on m, the index of a scheduling problem.

24. *Basis.* Clearly, the theorem is true for all scheduling problems with index 1.

25. *Induction hypothesis.* Let $m > 1$ and assume the theorem is true for any scheduling problem whose index is less than m.

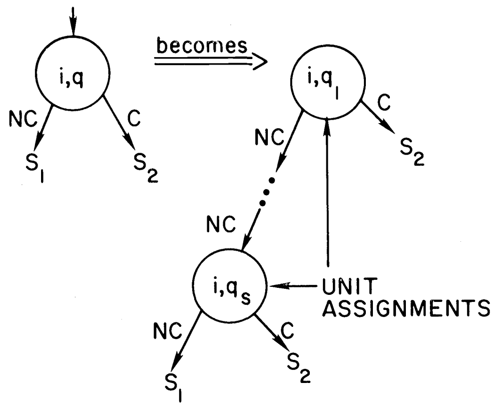
Induction Step. Let P be a scheduling problem with index m. If P consists of a single chain, we are finished. Accordingly, we assume there are at least two nonempty chains. The rest of the proof will work as follows:

(A) We show that if S is an optimal schedule for P, then S is an SR schedule, and

(B) All SR schedules for P have the same cost.

Arguments for (A) and (B) suffice to prove the theorem.

26. Let S be an optimal schedule and assume S is not SR. We shall consider the schedule decomposed to its equivalent unit schedule. By the assumption on S, not all the vertices can be SR. Working our way up from the leaves of the tree, we find a vertex v in S such that v is not SR and the completion and no-completion schedules of v are both SR schedules.



$q = q_1 + \cdots + q_s$

C = Completion subschedule

NC = No-completion subschedule

FIG. 1

27. The idea of the proof is to show that since S is not SR, it can be modified in a manner that will decrease the expected cost associated with it. We know how to evaluate the change in cost between two schedules only when they differ by a single exchange. An exchange is the operation that transforms a schedule which assigns at a particular situation chain i for quanta vector $\mathbf{q}$ and starts its completion *and* no-completion schedules by assigning chain j for quanta $\mathbf{p}$ into a schedule that at the same situation starts by assigning j for $\mathbf{p}$, and continues, both upon completion or no-completion, by assigning i for $\mathbf{q}$. A representation of an exchange is given in Fig. 5.

P28–P32 present the groundwork necessary for this exchange.

28. The *no-completion sequence* of v is a sequence of vertices $v_1, \cdots, v_k$ such that $v_1 = v$, v_{i+1} is the root of the no-completion subschedule of v_i, $\alpha(v_i) = \alpha(v)$ and k is maximal. By the induction hypothesis, all the nonempty completion subschedules of $v_1, \cdots, v_k$ can be taken to be equivalent (see P21) to a fixed nonempty SR schedule S', since they all operate on the same set of tasks, at the same stages of execution, and are of the same index. If all the completion subschedules of $v_1, \cdots, v_k$ are empty, then we take S' to be empty. Let S_1 denote the no-completion subschedule of v_k. We can collapse the sequence $v_1, \cdots, v_k$ into a single vertex with assignment specifying chain $i = \alpha(v)$ for a service interval not to exceed $q_1 = \beta(v_1) + \cdots + \beta(v_k)$ (Fig. 2).

29. The process just described is applied repeatedly to S' as long as α(root of $S') = i = \alpha(v)$. The result of this transformation is shown in Fig. 3.

30. Since i was not an SR assignment at vertex v, there exists a chain j, $j \neq i$, which has the lowest rank value $r(j)$ at vertex v. By the induction hypothesis, we can arrange to have all the nonempty schedules among $S_1, \cdots, S_{a+1}$ begin by assigning chain j. Let $p_1, \cdots, p_b$ be its corresponding rank quanta, where $p_l > 0$, $1 \leqq l \leqq b$, $1 \leqq b \leqq k_j$.

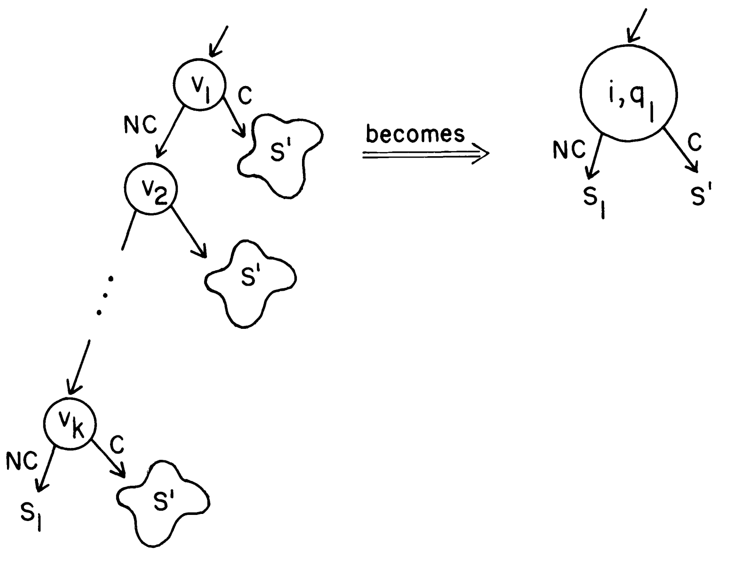


FIG. 2

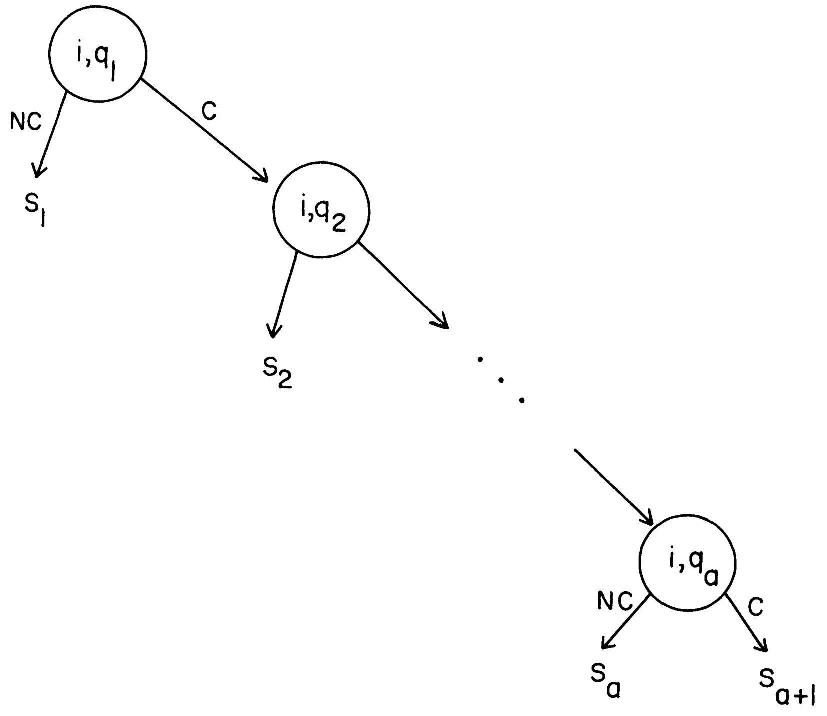


FIG. 3

31. Let S_l be a nonempty schedule $1 \leqq l \leqq a+1$ (there must be at least one because of P25). By the induction hypothesis and Theorem 2, the no-completion sequence of the root of S_l must assign at least the rank quantum p_1 to the first task in chain j. We can restructure S_l, if necessary, such that the root assigns chain j for the service interval p_1, where the no-completion subschedule of the root is denoted $S_{l,1}$ and the completion subschedule S_l'. Using Theorem 2, we can apply this restructuring to S_l' $(b-1)$ additional times, resulting in the schedule shown in Fig. 4.

32. In Fig. 5(a) we show the complete restructuring of S in which, for the sake of simplicity, we have neglected the possibility of empty subschedules. Our plan is to examine the effect of an interchange as explained in P27. This interchange is shown in Fig. 5(b) and, as indicated, it is always possible to resume the appropriate schedule $S_{x,y}$ such that $E(\Delta)$, the difference between the expected cost before and after the interchange depends only on the interchanged service intervals and no subsequent modifications result.

33. In order to calculate Δ, we define the random variables L and M as the number of tasks of chains i and j, respectively, that are completed during the assignments $\mathbf{q}$ and $\mathbf{p}$. Since the only difference in the costs of the schedules represented by Fig. 5(a) and Fig. 5(b) results from the change in residence time of the tasks of chains i and j, we have, using notations of P12, with Δ conditioned on

L and M only,

$$\Delta = \text{cost before exchange} - \text{cost after exchange}$$

$$(5)\quad \begin{aligned} &= \bar{W}_{j,1}\left(\sum_{l=1}^{L} \tau_{il} + q_{L+1}\right) + \sum_{l=1}^{L} \bar{W}_{i,l}\tau_{il} + \bar{W}_{i,L+1}q_{L+1} \\ &\quad + \bar{W}_{i,L+1}\left(\sum_{m=1}^{M} \tau_{jm} + p_{M+1}\right) + \sum_{m=1}^{M} \bar{W}_{j,m}\tau_{jm} + \bar{W}_{j,M+1}p_{M+1} \\ &\quad - \left\{ \bar{W}_{i,1}\left(\sum_{m=1}^{M} \tau_{jm} + p_{M+1}\right) + \sum_{m=1}^{M} \bar{W}_{j,m}\tau_{jm} + \bar{W}_{j,M+1}p_{M+1} \right. \\ &\quad \left. + \bar{W}_{j,M+1}\left(\sum_{l=1}^{L} \tau_{il} + q_{L+1}\right) + \sum_{l=1}^{L} \bar{W}_{i,l}\tau_{il} + \bar{W}_{i,L+1}q_{L+1} \right\} \end{aligned}$$

$$(6)\quad = \left(\sum_{l=1}^{L} \tau_{il} + q_{L+1}\right)(\bar{W}_{j,1} - \bar{W}_{j,M+1}) - \left(\sum_{m=1}^{M} \tau_{jm} + p_{M+1}\right)(\bar{W}_{i,1} - \bar{W}_{i,L+1}).$$

From (6), the independence of L and M, and (2), we obtain that

$$(7)\qquad E(\Delta) > 0 \Leftrightarrow r(i, \mathbf{q}) > r(j, \mathbf{p}).$$

Hence, if the original S was not SR, it could not have been optimal. This completes the proof of (A).

34. (B) Let S_1 and S_2 be unit SR schedules for P. Let v_1 and v_2 be the roots of S_1 and S_2, respectively. If $\alpha(v_1) = \alpha(v_2)$, then, by the inductive hypothesis, the completion and no-completion subschedules of both v_1 and v_2 are optimal and incur indentical costs, and therefore the expected cost of S_1 is equal to the expected cost of S_2.

35. Assume $i = \alpha(v_1) \neq j = \alpha(v_2)$. Obviously $r(i) = r(j)$, since both schedules are SR. Our goal is to transform S_1 into S_2 using an interchange in such a way that at each stage the cost decreases. Since by symmetry we could have just as easily transformed S_2 into S_1, we can conclude that the cost of S_1 is equal to the cost of S_2.

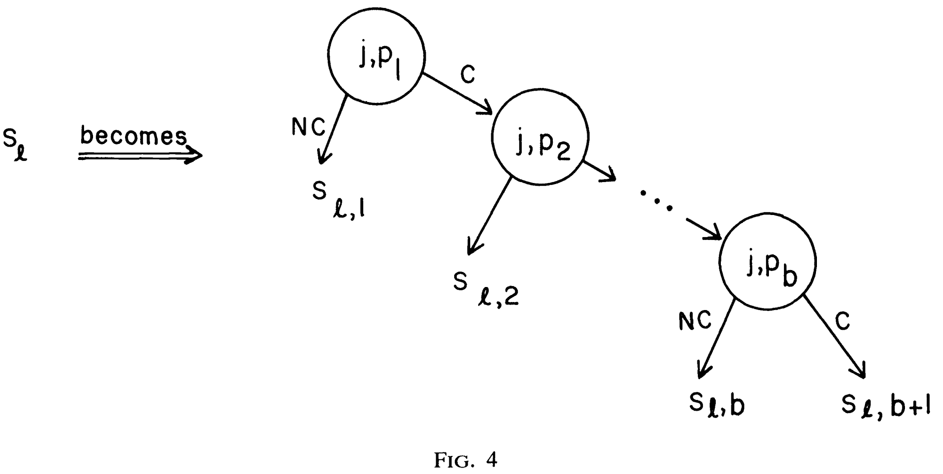


FIG. 4

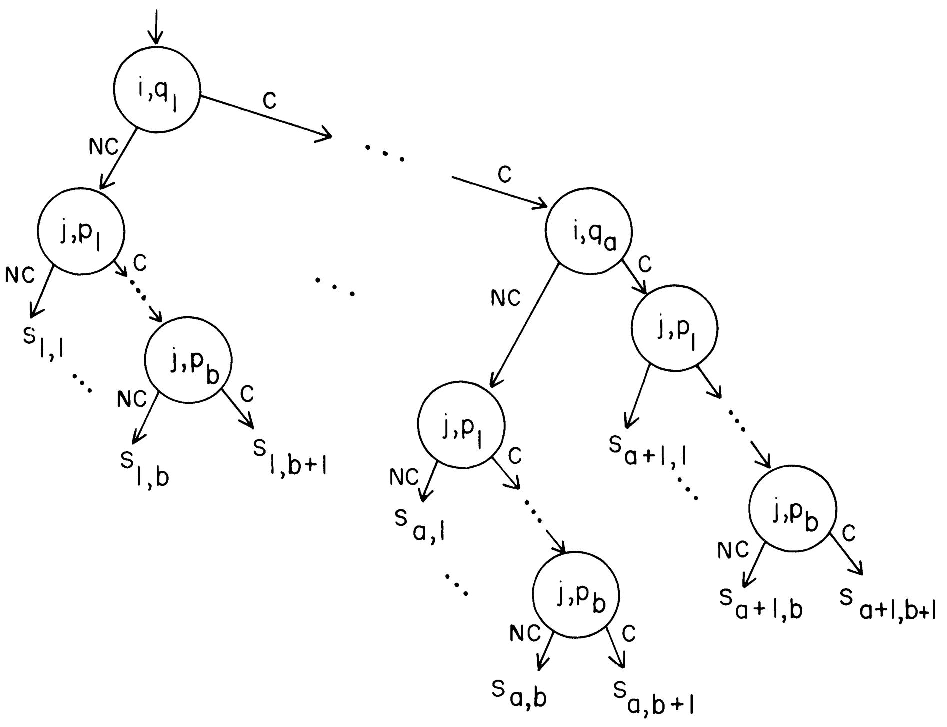


FIG. 5(a). *Before interchange*

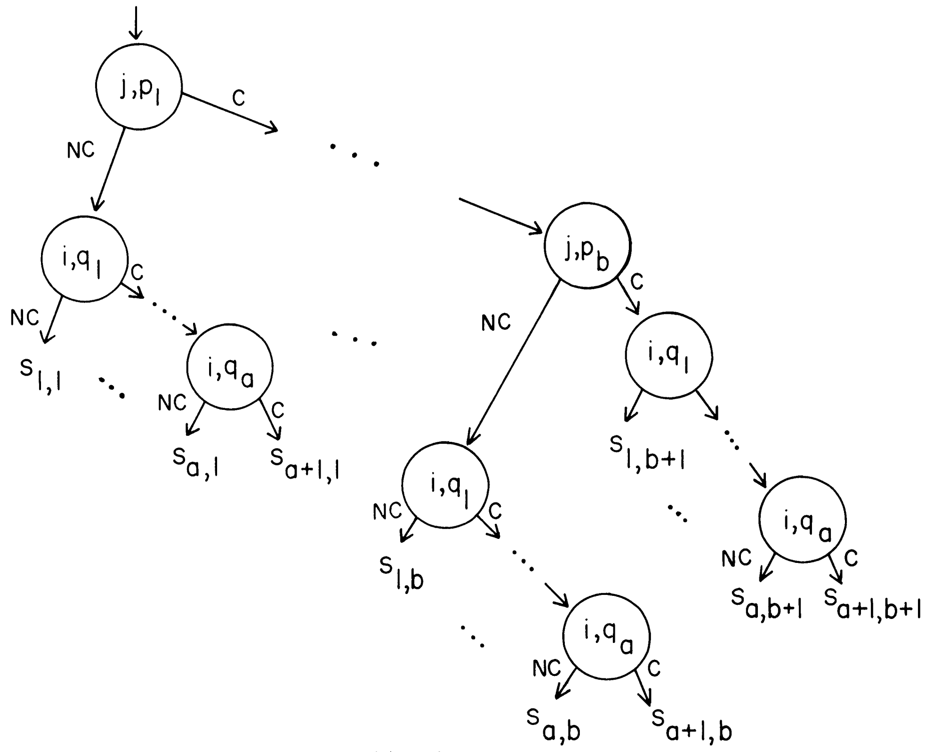


FIG. 5(b). *After interchange*

36. To transform S_1 into S_2, we exhibit S_1 as in Fig. 5(a), where $p_1, \cdots, p_b$ are the rank quanta for chain j. The induction hypothesis enables us to have chain j at the root of all the no-completion schedules of the tasks of i. Thus the interchange (Fig. 5(b)) yields a schedule S' with cost no larger than S_1. The no-completion and completion subschedules of the root of S' can be taken to be SR schedules without increasing the cost, and using the induction hypothesis, we can go to S_2, again, without an increase in cost; hence the costs of S_1 and S_2 are equal. □

5. Discussion.

37. We have extended a result for optimal scheduling of independent jobs to the case where they are arranged in a number of chains. To facilitate the proofs, time was considered discrete and the service requirements were bounded. These two restrictions seem to us inessential, in the sense that the result carries over to cases where they are not satisfied. We would require, though, finite service expectations (the main difficulty would then be the existence of service contributions that would specify infinitesimal quanta as the optimal ones. That would be usually the case with DFR distributions and, more generally, in situations where the expected remaining service time is an increasing function of the attained service time). In these cases, "processor sharing" will have to be introduced. Permitting this, the arbitrariness allowed in the time scale we used indicates that a schedule based on the same rank will be optimal.

38. A more interesting extension will be to cases where arrivals are expected, whether of individual tasks to existent chains, or of whole new chains. Here a completely different approach is necessary. We note in passing that a *finite* number of expected arrivals is more likely to produce an unmanageable problem than a stationary input process. It can be shown that the determination of the optimal schedule in a deterministic context, when one arrival is expected, is "polynomial-complete" [4].

39. The minimization implied in the definition of a rank in (1) can prove to be a major problem, and the value of Theorem 1 is quite limited in the case of a long chain. This problem can be shown to be a special case of a problem treated in [1], and the efficient algorithm presented there can be applied in the present case.

Appendix A. Proof of Theorem 1. The value of q_1 that minimizes $r(\mathbf{q})$ when $q_2 = 0$ is the value that minimizes

$$r(q_1) = [(1-P_1)q_1 + P_1\tau_1]/P_1 w_1, \tag{A.1}$$

where $P_i = P(S_i \leqq q_i)$, $\tau_i = E(S_i | S_i \leqq q_i)$. Using the modification of (2) as in P13, we can write for the expression for the rank $r(\mathbf{q})$, with a chain of k tasks,

$$r(\mathbf{q}) = \frac{\sum_{j=0}^{k-1} P(L=j)\left(\sum_{l=1}^{j} \tau_l + q_{j+1}\right) + P(L=k)\sum_{l=1}^{k} \tau_l}{\sum_{j=1}^{k} P(L \geqq j) w_j}. \tag{A.2}$$

Substitute $P(L=j) = P_1 P_2 \cdots P_j(1-P_{j+1})$ for $j = 1(1)k-1$, $P(L=0) = 1-P_1$, $P(L=k) = P_1 P_2 \cdots P_k$; $P(L \geqq j) = P_1 \cdots P_j$, and we obtain, after we separate τ_1

from the rest and note that $\sum_{j=1}^{k} P(L=j)=P_1$,

$$r(\mathbf{q})=\frac{q_1(1-P_1)+P_1\tau_1}{P_1 W}+\frac{Q+T}{W}, \tag{A.3}$$

where Q, T and W arise from q_i and τ_i, respectively, for $i\geqq 2$, do not depend in any way on q_1, and are nonnegative. This assures that the same value of q_1 that would minimize $r(q_1)$ would also be the first component of the $\mathbf{q}$ vector that will minimize $r(\mathbf{q})$ in (A.3).

Unfortunately, this manipulation cannot be extended any further. As a demonstration, we show that the value of q_2 that will minimize $r(\mathbf{q})$ may depend on whether or not q_3 is forced to be zero. Suppose the first three tasks had deterministic service requirements a, b and c, and cost accrual rates w_1, w_2 and w_3. Then the ranks when one, two or three quanta may differ from zero are

$$\begin{aligned} &r(q_1)=a/w_1, \qquad r(q_1,q_2)=\min\{a/w_1,(a+b)/(w_1+w_2)\},\\ &r(q_1,q_2,q_3)=\min\{a/w_1,(a+b)/(w_1+w_2),(a+b+c)/(w_1+w_2+w_3)\}.\end{aligned} \tag{A.4}$$

It is a simple matter to find values for these parameters so that we shall have $\mathbf{q}_1^*=a$, $\mathbf{q}_2^*=(a,0)$ $\mathbf{q}_3^*=(a,b,c)$, as asserted. The values $a=c=1$, $b=2$, $w_1=w_2=1$, $w_3=4$ satisfy this.

Appendix B. Proof of Theorem 2.

1. Consider a chain having at time 0 a rank r_0, obtained with the rank-quanta assignment $\mathbf{q}$. Let q_i, some rank-quantum, be equal to $a+b$, $a\geqq 0$, $b>0$. Theorem 2 states that at time t_a, after the chain has been serviced and the $i-1$ first tasks finished within their allotted quanta, and task i was processed for time a and not terminated, the rank of the chain, r_a, is no greater than r_0.

2. We shall consider assignments with the following quanta: at time 0, the assignment $\mathbf{q}$ gives rise to the rank r_0. The assignment $\mathbf{q}'$, with $q_i=a$ only and otherwise equal to $\mathbf{q}$, gives the rank-value $r_0'\geqq r_0$. At time t_a there is an optimal assignment $\hat{\mathbf{q}}$ which gives rise to the rank r_a, and we shall also consider the assignment then of $(b, q_{i+1},\cdots,q_k)$ which gives rise to the rank-value $r_b\geqq r_a$.

3. Using the interpretation of the rank given in P15, we write

$$r_0=T/W, \qquad r_0'=T'/W', \qquad r_b=T_b/W_b. \tag{B.1}$$

We use the independence between the service durations to different tasks to further write

$$T=\tau_1+P_1\tau_{a+b}+P_1P_{a+b}\tau_2, \tag{B.2}$$

where

τ_1 is the expected time spent on first $i-1$ tasks,
P_1 is the probability task i is reached,
τ_{a+b} is the expected time spent on task i,
P_{a+b} is the probability that task i terminates within $a+b$,

and

τ_2 is the expected time spent on the last $k-i$ tasks.

Similarly,

$$W = W_1 + P_1 P_{a+b} w_i + P_1 P_{a+b} W_2, \tag{B.3}$$

where

W_1 is the expected decrease in accrual rate from processing the first $i-1$ tasks,

and

W_2 is the expected decrease in accrual rate from processing the last $k-i$ tasks.

And likewise,

$$T' = \tau_1 + P_1 \tau_a + P_1 P_a \tau_2, \tag{B.4}$$

with obvious meanings for τ_a and P_a,

$$W' = W_1 + P_1 P_a w_i + P_1 P_a W_2, \tag{B.5}$$

$$T_b = \tau_b + P_b \tau_2, \qquad W_b = P_b w_i + P_b W_2, \tag{B.6}$$

where

$$\begin{aligned} \tau_b &= E(S_i - a \mid a < S_i \leqq a+b) + b \cdot P(S_i > a+b \mid S_i > a) \\ &= (\tau_{a+b} - \tau_a)/(1-P_a), \end{aligned} \tag{B.7}$$

$$P_b = P(S_i \leq a+b \mid S_i > a) = (P_{a+b} - P_a)/(1-P_a). \tag{B.8}$$

4. Using these values, we obtain

$$\begin{aligned} r_0 &= (\tau_1 + P_1 \tau_{a+b} + P_1 P_{a+b} \tau_2)/(W_1 + P_1 P_{a+b} w_i + P_1 P_{a+b} W_2) \\ &= \alpha r_0' + (1-\alpha) r_b, \end{aligned} \tag{B.9}$$

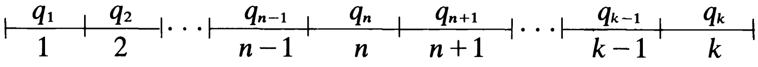
with $\alpha = (W_1 + P_1 P_a (w_i + W_2))/(W_1 + P_1 P_{a+b}(w_i + W_2))$, as substitution readily shows.

Since obviously $P_{a+b} \geqq P_a \geqq 0$, we have that $0 \leqq \alpha \leqq 1$, and thus in (B.9), r_0 is a convex combination of r_0' and r_b. Since by definition $r_0 \leqq r_0'$, we must have $r_0 \geqq r_b \geqq r_a$, as required.

Note that $a = 0$ corresponds to the special case of rank after a service termination.

Appendix C. Discussion of Property 3.

1. In spite of its intuitive content, we have not been able to fully prove Property 3 of the rank as it is stated in P19. We can show part of it: if the rank of the chain



is obtained at $\mathbf{q} = \mathbf{q}^*$, with $q_{n+1}^* = 0$ and $r = r_1^*$, and when the chain was assigned for the first task it carried through and completed the first n tasks within their allotted quanta, then the rank of the remaining part is larger than r_1^*. Suppose the rank of the remaining part, r_n, is obtained at $\hat{\mathbf{q}}$, which is $(\hat{q}_{n+1}, \cdots, \hat{q}_k)$, with $\hat{q}_{n+1} > 0$. Define $r(\mathbf{q})$ by (4) as

$$(C.1) \qquad r(\mathbf{q}) = \frac{\sum_{l=0}^{n-1} P(l)q_{l+1} + \sum_{l=1}^{n} \bar{P}(l)\tau_l + \sum_{l=n}^{k-1} P(l)q_{l+1} + \sum_{l=n+1}^{k} \bar{P}(l)\tau_l}{\sum_{l=1}^{n} \bar{P}(l)w_l + \sum_{l=n+1}^{k} \bar{P}(l)w_l},$$

where the first n components of $\mathbf{q}$ are those of $\mathbf{q}^*$, and the last $k-n$ are $\hat{\mathbf{q}}$, we can write then, denoting the value of r_1^* from (4) as the ratio A/B:

$$(C.2) \qquad r(\mathbf{q}) = \frac{A + \bar{P}(n)\left\{\sum_{l=n}^{k-1} P_{n+1}(l)\hat{q}_{l+1} + \sum_{l=n}^{k} \bar{P}_{n+1}(l)\hat{\tau}_l\right\}}{B + \bar{P}(n) \sum_{l=n+1}^{k} \bar{P}_{n+1}(l)w_l},$$

where $P_{n+1}(l) = P$ (task l is the last one completed within its quantum, starting with the $(n+1)$st task). $\bar{P}_{n+1}(l)$ is defined accordingly, as in P13. Since with this notation r_n can be written as

$$(C.3) \qquad r_n = \frac{\sum_{l=n}^{k-1} P_{n+1}(l)\hat{q}_{l+1} + \sum_{l=n}^{k} \bar{P}_{n+1}(l)\hat{\tau}_l}{\sum_{l=n+1}^{k} \bar{P}_{n+1}(l)w_l},$$

we have

$$(C.4) \qquad r(\mathbf{q}) = \frac{A + Dr_n}{B+D} = \frac{B}{B+D} r_1^* + \frac{D}{B+D} r_n,$$

which is a convex combination. Since by definition $r(\mathbf{q}) \geqq r_1^*$, we must have $r_n \geqq r_1^*$, as claimed.

REFERENCES

[1] J. BRUNO, E. COFFMAN AND D. JOHNSON, *On batch scheduling of jobs with stochastic service and cost structures on a single server*, Tech. Rep. 159, Dept. of Computer Sci., Pennsylvania State Univ., University Park, 1974.

[2] R. W. CONWAY, W. L. MAXWELL AND L. W. MILLER, *Theory of Scheduling*, Addison-Wesley, Reading, Mass., 1967.

[3] W. A. HORN, *Single-machine job sequencing with treelike precedence ordering and linear delay penalties*, SIAM J. Appl. Math., 23 (1972), pp. 189–202.

[4] R. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.

[5] L. SCHRAGE, *Optimal scheduling disciplines for a single machine under various degrees of information*, 44th ORSA Conf., Boston, April 1974.

[6] KENNETH C. SEVCIK, *Scheduling for minimum total loss using service time distribution*, J. Assoc. Comput. Mach., 21, pp. 66–75 (1974).

[7] JEFFREY B. SIDNEY, *Decomposition algorithms for single-machine sequencing with precedence relations and deferral costs*, Operations Res., 23 (1975), pp. 283–298.

SIAM J. COMPUT.
Vol. 4, No. 4, December, 1975

A CONVERGENCE THEOREM FOR HIERARCHIES OF MODEL NEURONES*

M. D. ALDER†

Abstract. The threshold logic unit (T.L.U.) has been proposed as a model for a single neurone; other substantially cognate terms are "perceptron" and "adaline". Networks of these elements have been advanced as tentative models of some aspects of brain functioning. In particular, hierarchical nets appear to exhibit a sufficient flexibility to make them interesting both as plausible models of learning in the central nervous system and also as general objects of study in connection with pattern recognition and artificial intelligence.

In this paper, we discuss the well-known "perceptron convergence theorem" in a fairly general setting, and consider variations appropriate to nets of such units. A certain familiarity with the relevant chapters of Nilsson's *Learning Machines* [1] and also with current mathematical formalism is presupposed.

Key words. model neurones, perceptron, learning machine

1. Introduction. There are two prerequisites for reading this paper: the first is Nilsson's *Learning Machines* [1] and the second is some familiarity with current mathematical formalism. Some care has been taken in what follows to explain the formalism and mitigate its abstraction, but there are limits to how far one can go in this direction (if there were not, the formalism would be pointless).

The reason for citing [1] as prerequisite is mainly motivational. This paper is, it is hoped, self-contained logically, but this introduction can only very briefly recapitulate the development. Together, [1] and the review [6] by H. D. Block of Minsky and Papert's "perceptrons" indicate why perceptrons or threshold logic units (T.L.U.'s) are of interest in machine learning and brain modeling.

To date, a great deal has been written on the shortcomings of the single unit both as a trainable pattern classifier [3] and as a model of perception [4]. The virtues of the T.L.U. are first that it is strongly reminiscent of various models of adaptive behavior in the central nervous system, and second that it constitutes a simple, well-defined "trainable" object. It might reasonably be hoped that nets of such units would overcome the objections of inflexibility that are leveled against the single unit, but little appears to have been done in this direction. Although there is no hope of rescuing the perceptron as a model for perception, nets of perceptron like elements would appear to be interesting objects of study in artificial intelligence and brain modeling. The central problems appear to be: first, what configurations of net are capable of adaptive behavior, and second, how does one train such a net? Given a procedure for training a single unit and a net of such units, how does one select a particular unit from the net to be corrected, when the net as a whole is in error?

In this paper, two types of nets are considered, committee nets and hierarchy nets. A simple algorithm is specified and a convergence theorem proved. In § 2, a careful formalization of the problem is given. Section 3 contains a new proof of the

* Received by the editors October 6, 1972, and in revised form October 12, 1974.

† Department of Mathematics, University of Western Australia, Nedlands, Western Australia 6009.

perceptron convergence theorem, mainly to indicate the terminology and methods of the net theorem, which appears in § 4, as Theorem 4.24.

2. Terminology, notation, conventions.

Remark 2.1. I shall rely heavily on modern pure mathematical formalism; it is powerful and convenient and leaves no serious alternative. Since this paper is not written for pure mathematicians, however, it seems appropriate to explain some of the definitions verbally and informally, and I shall do this. Initially we set up the language.

Notation 2.2. S^0 denotes the two point set $\{-1, 1\}$, $\mathbb{R}^n$ the space of n-tuples of real numbers, $\mathbb{N}$ the set of natural numbers.

DEFINITION 2.3. A *data set* on $\mathbb{R}^n$ is a map $f : D \to S^0$ where D is some *finite* subset of $\mathbb{R}^n$.

Remark 2.3.1. This simply means we pick out a finite set of points in $\mathbb{R}^n$, calling some negative (-1) and some positive.

Remark 2.4. It is a triviality that any affine map from $\mathbb{R}^n$ to $\mathbb{R}$ can be described by $n+1$ numbers: for $n=2$ the general affine map sends (x, y) to $ax+by+c$, where a, b, c, three constants, determine the map. In general, any affine map from $\mathbb{R}^n$ to $\mathbb{R}$ sends the vector $(x_1, x_2, \cdots x_n)$ to $\sum_{i=1,n} a_i x_i + c$, where the $n+1$ numbers $a_1, a_2, \cdots, a_n, c$, determine the map. The *kernel* of this map is the subset of $\mathbb{R}^n$ that gets sent to zero, i.e.,

$$\left\{(x_1, x_2, \cdots, x_n) \in \mathbb{R}^n : \sum_{i=1,n} a_i x_i + c = 0\right\},$$

provided not all the a_i are zero, this is a hyperplane in $\mathbb{R}^n$ of dimension $n-1$; in particular it is a line in $\mathbb{R}^2$. We shall work in the space of affine maps from $\mathbb{R}^n$ to $\mathbb{R}$. Specifying each of them by $n+1$ numbers in the above manner, we see that we can think of this space as "being" $\mathbb{R}^{n+1}$, and we shall henceforward make this identification without further comment. More formally we have the next definition.

DEFINITION 2.4.1. Let X_n^* denote the space of all affine maps from $\mathbb{R}^n$ to $\mathbb{R}$. Clearly X_n^* is isomorphic to $\mathbb{R}^{n+1}$. It is called the "weight space" in [1].

DEFINITION 2.5. Let $\operatorname{sgn} : \mathbb{R}-\{0\} \to S^0$ be the map sending x to $x/|x|$, i.e., $\operatorname{sgn}(x)$ is $+1$ or -1 according to the sign of x.

DEFINITION 2.6. Given a data set f and $x \in D = \operatorname{dom} f$, we shall say that $L \in X_n^*$ *correctly classifies* x if $f(x) = \operatorname{sgn} \circ \mathrm{L}(x)$.

DEFINITION 2.7. If $\exists L \in X_n^* : f(x) = \operatorname{sgn} \circ L(x)\ \forall x \in \operatorname{dom} f$, we shall say that f is *linearly separable.*

Remark 2.7.1. In English, if there is any affine map which correctly classifies all points of a data set, then it (the data set) is linearly separable. Clearly the affine map has kernel coming "between the two types of point".

DEFINITION 2.8. A *training sequence* for a data set f is a map $\sigma : \mathbb{N} \to D$, where $D = \operatorname{dom} f$, such that for all $x \in D$, $\sigma^{-1}(x)$ is infinite.

Remark 2.8.1. We merely wish to have a sequence of all points in the data set, each one "called" as often as is necessary. Hopefully, convergence of the training procedure means that after some number, no correction is necessary; the unit has converged. Since we cannot say in advance when this point is to be reached, we allow an infinite set of possible "calls" of each data point.

DEFINITION 2.9. Let $\tilde{X}_n^*$ denote X_n^* with the constant affine maps removed. Alternatively $\tilde{X}_n^*$ contains only those affine maps from $\mathbb{R}^n$ to $\mathbb{R}$ that are surjective. $\tilde{X}_n^*$ is isomorphic to a subset of $\mathbb{R}^{n+1}$, and we shall make this identification without further comment. These are the maps having kernels precisely the hyperplanes of dimension $n-1$ in $\mathbb{R}^n$. We shall drop the n in X_n^* and $\tilde{X}_n^*$ from now on; it is excess baggage. We shall use $A \times B$ to denote the Cartesian (set) product, and the notation

$$\prod_{j=1,n} A_j$$

for the Cartesian product of a set of sets.

DEFINITION 2.10. A *training procedure* or *correction procedure* or *correction algorithm* for a data set f, is a map

$$\Phi : D \times \tilde{X}^* \to \tilde{X}^*$$

satisfying the relation

$$\operatorname{sgn} \circ (\Phi(x, L))(x) = f(x) \quad \forall x \in D,$$

where $D = \operatorname{dom} f$.

Remark 2.10.1. This is stronger than is necessary and requires that given a point of the data set x and any affine map L in $\tilde{X}^*$, we have provided a (possibly the same, possibly new) map $\Phi(x, L)$ which now correctly classifies the point. We might have demanded only that some iterate of Φ (using the same x) should correctly classify the point. We shall in fact work with a standard procedure described immediately below which is a correction procedure in our sense. The results in this paper do not in general depend strongly on the correction procedure.

Remark 2.11. A point $x \in \operatorname{dom} f$, f a data set, determines in X^* a hyperplane passing through the origin and dividing X^* into two halves, the sets X_+^* and X_-^* defined by:

$$L \in X_+^* \quad \text{iff} \quad \operatorname{sgn} \circ L(x) = f(x).$$

The hyperplane determined by x we shall call $\tilde{x}$. Now we define the standard correction procedure.

DEFINITION 2.12. The *standard correction procedure* or *standard correction algorithm* is the map

$$\Phi : D \times X^* \to X^*$$

given by

$$\Phi(x, L) = L \quad \text{if } \operatorname{sgn} \circ L(x) = f(x)$$

and

$$\Phi(x, L) = L - \frac{[(1+\alpha) \cdot \langle \hat{x}, L \rangle]}{\langle \hat{x}, \hat{x} \rangle} \hat{x} \quad \text{otherwise,}$$

where α is any real number such that $0 < \alpha < 1$, and $\hat{x}$ is the normal vector in $X^* \approx \mathbb{R}^{n+1}$ to the hyperplane $\tilde{x}$ determined by x (obtained by augmenting the vector x with a 1 in the $(n+1)$th place, in the obvious representation of X^*). $\langle \hat{x}, L \rangle$ is the usual inner product in $\mathbb{R}^{n+1}$.

Remark 2.12.1. The definition of Φ is perhaps not very natural at first sight. The point $x, f(x)$ determines in X^* the oriented hyperplane $\tilde{x}$. L is on the "wrong side" of $\tilde{x}$ and is just a point in X^*, so we proceed simply to "reflect" the point L in the hyperplane $\tilde{x}$ to obtain a new L. A full reflection would correspond to $\alpha = 1$; it is convenient to "dilute" the reflection so that the point changes sides of the hyperplane but finishes up closer to it. It is easy to verify that Φ accomplishes precisely this.

DEFINITION 2.13. Let $\sigma : \mathbb{N} \to D$ be a T.L.U. training sequence. Then σ generates an action on $\tilde{X}^*$, i.e., a map Σ that assigns to each number $n \in \mathbb{N}$, that map from $\tilde{X}^*$ to itself sending L to $\Phi(\sigma(n), L)$ in $\tilde{X}^*$.

Formally we have: $\Sigma : \mathbb{N} \to (\tilde{X}^*, \tilde{X}^*)$. (where (X, Y) is the set of maps from X to Y), given by $\Sigma(n) : \tilde{X}^* \to \tilde{X}^*, \forall L \in \tilde{X}^*$, sends $L \rightsquigarrow \Phi(\sigma(n), L, \forall L \in \tilde{X}^*$. If Φ is the standard correction procedure, Σ is called the *standard action for* σ.

Remark 2.13.1. We "call out" the points of the data set in some sequence, and corresponding to each number we have a data point. This data point now tells us how to "move" any given affine map until it correctly classifies the point, i.e., the data point determines a map from $\tilde{X}^*$ to $\tilde{X}^*$.

If the sequence σ and data set f are chosen, and if f is linearly separable, we might reasonably hope that for each L there is some natural number N such that convergence occurs and every point in dom f is correctly classified. In this event, with the standard correction procedure, L would never be moved again.

DEFINITION 2.14. A T.L.U. training sequence *σ converges under the standard action* (or *converges*, for short) if $\forall L \in \tilde{X}^* \exists N \in \mathbb{N} : n > N \Rightarrow (\Sigma(n))(L) = (\Sigma(N))(L)$.

THEOREM 2.15. *Let f be any linearly separable data set in $\mathbb{R}^n$, σ any training sequence. Then σ converges under the standard action.*

Remark 2.15.1. We give no proof of Theorem 2.15 at this stage; see [1] for a somewhat different version.

The least $N \in \mathbb{N}$ such that the implication of Definition 2.14 holds is the stage or iteration at which the T.L.U. has converged. We say informally that the T.L.U. has "learned" the data set at this point.

Remark 2.16. This concludes our formalization of the "single unit" case. Apart from our choice of a standard correction procedure and hence a standard action of training sequences, this merely writes out in an abstract setting what has been well known. In fine, the long string of definitions to date is no more than a mild tightening up of the language of [1], and nothing has actually been said yet. It should not be necessary to point out that the economy of a terse formal language is a considerable aid to thought when matters get complicated. We proceed to discuss the nets we shall be concerned with in later sections.

Remark 2.17. We suppose we have q T.L.U.'S, each "state" in $\tilde{X}^*$. Assume q is odd.

DEFINITION 2.18. The *committee-net* of order q is the map

$$\underbrace{C : \tilde{X}^* \times \tilde{X}^* \times \cdots \times \tilde{X}^*}_{q \text{ copies}} \times \mathbb{R}^n \to S^0$$

defined by

$$C(L_1, L_2, \cdots L_q, x) = \operatorname{sgn}\left[\left(\sum_{j=1}^{q} \operatorname{sgn}(L_j(x))\right)\right].$$

This map, strictly speaking, has domain the subset of

$$\tilde{X}^* \times \tilde{X}^* \times \cdots \times \tilde{X}^* \times \mathbb{R}^n$$

where points of $\mathbb{R}^n$ do not occur in the kernels of the L_j. The difference is a set of measure zero and will be ignored.

Remark 2.18.1. We have not one but q affine maps, each giving a verdict ± 1 on x. C takes the majority vote of the committee as the output. Writing $\prod_{j=1,q} \tilde{X}_j^*$ for the above Cartesian product (j *not* indicating the dimension!), $L = (L_1, L_2, \cdots L_q) \in \prod_{j=1,q} \tilde{X}_j^*$.

DEFINITION 2.19. A data set f is *committee q-separable* iff

$$\exists L \in \prod_{j=1,q} \tilde{X}_j^* : C(L, x) = f(x) \quad \forall x \in \mathit{dom}\, f.$$

Remark 2.19.1. That is to say, a data set f is committee q-separable if there is a set of committee "members", each member a T.L.U. such that the majority vote on each data point is always right. We note two things: first that since the data set is finite, if any *one* solution exists, then an open neighborhood of solutions exist consisting of L's "close enough" to the given L. Since $\prod_{j=1,q} \tilde{X}_j^*$ is a subset of $\mathbb{R}^{(n+1)q}$, we may take the Euclidean metric in $\prod_{j=1,q} \tilde{X}_j^*$ and talk of an open "ball" on a given L. Second, permuting committee members gives a generally distinct solution that is just as good. Effectively we are observing that there is a high degree of symmetry in the "solution space", where the solution space is that subset of $\prod_{j=1,q} \tilde{X}_j^*$ containing exclusively points L such that

$$C(L, x) = f(x) \quad \forall x \in \operatorname{dom} f.$$

We turn to the second kind of net we shall consider.

DEFINITION 2.20. The *simple hierarchy net* is the map

$$H : \tilde{X}^* \times \tilde{X}^* \times \tilde{X}^* \times \mathbb{R}^n \to S^0$$

given by

$$H(L_1, L_2, L_3, x) = (L_3(x)) \cdot \frac{(1 + L_1(x))}{2} + (L_2(x)) \cdot \frac{(1 - L_1(x))}{2},$$

where $L_j(x)$ is an abbreviation for $(\operatorname{sgn} \circ L_j)(x)$. Again strictly speaking, we take a subset of $\tilde{X}^* \times \tilde{X}^* \times \tilde{X}^* \times \mathbb{R}^n$, the map not being defined on (L_1, L_2, L_3, x) if x is in $\ker L_1$ or $\ker L_2$ or $\ker L_3$. This set has measure zero.

Remark 2.20.1. The effect of this is that if L_1 "decides" that $x \in \mathbb{R}^n$ is of type $+1$, then L_3 is "allocated" the job of deciding on the nets view of the true category of x. If sgn $(L_1(x)) = -1$, then L_2 gets the job of deciding. We have a decision hierarchy, where the "top layer" ($L_1 \in X_1^*$) decides not on the category of the point, but on which of the other two T.L.U.'s decides the matter.

To indicate the force of this process, consider a data set $f: D \to S^0$ for $D \in \mathbb{R}^2$, defined by $f(x) = +1$ if x is strictly in the first or third quadrant of the plane, $f(x) = -1$ if x is strictly in the second or fourth quadrant and D any finite set of points.

It is easy to see that this is a serious problem for a committee net: roughly speaking, the number of committee members required increases monotonically with the number of points. However, the simple hierarchy can correctly classify any finite number of points; we leave the reader to find a solution.

The problem arises: how does one correct such a machine, for the correction process can occur at either level in general? If the hierarchy is in error, we can suppose either that the executive (lower) unit selected was in error or that the policy (upper) unit erred in its allocation. If we have no evidence to prefer one hypothesis to the other, it seems reasonable to choose at random between layers, to choose at random a T.L.U. in the lower layer if that is selected and to correct it if it is in error. We show later that this somewhat haphazard procedure yields convergence.

Remark 2.21. There is for any net the following problem: which T.L.U. does one select for "correction" by the standard procedure, when it is the net as a whole that is in error? As remarked above, for the simple hierarchy, this becomes a matter of choosing between layers; for the committee net we have a similar problem: which committee member gets corrected?

We shall show in both cases that the "net correction procedure" of choosing at *random* some unit to correct will guarantee convergence, provided that the data points are also called at random. The method of argument appears to hold for quite general net configurations, but we restrict ourselves to the particular nets discussed. The convergence result for both nets is established in § 4; in § 3 as a preamble we give a proof of the perceptron convergence theorem for the standard correction procedure.

3. The perceptron convergence theorem.

Remark 3.1. For $x \in \operatorname{dom} f$, the action of Definition 2.12 defines a map $\Phi(x) : X^* \to X^*$ which is a map of $\mathbb{R}^{n+1}$ onto a half-space. It is easy to see that the map $\Phi(x) : X^* \to X_+^*$ is a retraction (i.e., it leaves points already in the half-space X_+^* fixed) satisfying the following Lipschitz condition:

$$(1) \qquad \|\Phi(x)(y) - \Phi(x)(z)\| \leqq \|y - z\| \quad \forall y, z \in X^*, \quad \forall x \in \operatorname{dom} f.$$

where $\| \cdot \|$ denotes the usual Euclidean norm on X^*, taken isomorphic to $\mathbb{R}^{n+1}$. Let A be a subset of $\mathbb{R}^n$ and $r : \mathbb{R}^n \to A$ a retraction satisfying

$$(2) \qquad \|r(y) - r(z)\| \leqq \|y - z\| \quad \forall y, z \in \mathbb{R}^n$$

(and $r(a) = a, \forall a \in A$, since r is a retraction); then r is continuous and A is necessarily convex.

The first part of this remark, that $\Phi(x)$ satisfies the Lipschitz condition, follows from the "diluted reflection in a plane mirror" nature of $\Phi(x)$. Obviously, no two points can be moved further apart by the map; it is "nonexpansive", which is what (1) says. That a nonexpansive map is continuous is a triviality; to see that A is necessarily convex, take any two points a, b in A. Then the line segment joining a and b lies wholly in A, for take c on the line segment between a and b. c cannot be moved without increasing its distance from a or b. r is nonexpansive, so c is not moved by r; hence $c \in A$. We deal therefore with a set of nonexpansive retractions onto convex subsets and with composites of them. It is clear that if we have two nonexpansive retractions onto sets which intersect, then the composite of the retractions "winds $\mathbb{R}^n$" around the intersection. If the intersection has nonzero measure, a "bite" of at least this size is taken out by each iteration of the composite of the retractions. For example, take the retraction of $\mathbb{R}$ to $\mathbb{R}$ which sends x to $|x|$ and the second retraction sending x to $-|x|$. Iterating the composed map "winds" $\mathbb{R}$ around the origin, which has measure zero.

Compare with the maps $r_1 : \mathbb{R} \to \mathbb{R}$ with $r_1(x) = x$ if $x > -1$, $r_1(x) = -x-2$ if $x \leqq -1$; and $r_2 : \mathbb{R} \to \mathbb{R}$ with $r_2(x) = 2 - x$ if $x > 1$, $r_2(x) = x$ if $x \leqq 1$. Then r_1 is a nonexpansive retraction onto $[-1, \infty)$ and r_2 is a nonexpansive retraction onto $(-\infty, 1]$ with a "bite" between $[-1, 1]$ taken out of the composite $r_1 \circ r_2$. The composite is obviously a contraction mapping outside $[-1, 1]$ (on which it is fixed). Hence for any $x \in \mathbb{R}$, there is a number n such that $(r_1 \circ r_2)^n$ takes x into $[-1, 1]$. In fact $n = [x/2]$ clearly suffices. The following proposition states that this situation holds quite generally.

PROPOSITION 3.2. *Let* $\{A_j\} 1 \leqq j \leqq J$ *be subsets of* $\mathbb{R}^n$ *and* $\{r_j : \mathbb{R}^n \to A_j,\ 1 \leqq j \leqq J\}$ *a family of retractions satisfying* $\|r_j(x) - r_j(y)\| \leqq \|x - y\| \forall x,\ y \in \mathbb{R}^n$, $\forall j : 1 \leqq j \leqq J$. *Suppose that* $\bigcap_{1 \leqq j \leqq J} A_j$ *contains an open set* $\neq \emptyset$. *Let* $g = r_J \circ r_{J-1} \circ \cdots \circ r_2 \circ r_1$. *Then* $\forall x \in \mathbb{R}^n\ \exists k \in \mathbb{N} : g^k(x) \in \bigcap_{1 \leqq j \leqq J} A_j$.

Proof. We shall deal with the case $J = 2$; the result follows by an easy induction. The conventions of point set topology are employed: if A is a set, $\bar{A}$ denotes the closure of A.

First we note that if W is the interior of $A_1 \cap A_2$, then $A_1 \cap A_2 = \bar{W}$, for A_1 and A_2 are closed by continuity; hence $\bar{W} \subseteq A_1 \cap A_2$. It suffices to show that if y is a boundary point of $A_1 \cap A_2$, then any neighborhood of it intersects W.

This follows by taking the cone on W with vertex y, i.e., the set of line segments in $\mathbb{R}^n$ joining y to a point of W. Since y and W are in $A_1 \cap A_2$ and $A_1 \cap A_2$ is convex, the cone is also in $A_1 \cap A_2$. Now any neighborhood of y intersects a "contracted" copy of W which is itself open and in $A_1 \cap A_2$.

Now suppose $x \in \mathbb{R}^n$ given; we seek an m such that $g^m x \in A_1 \cap A_2$. Put $W_1 = (g^{-1} W) \cup (A_1 \cap A_2)$. It is easy to see that W_1 is an open set containing $A_1 \cap A_2$. Generally put $W_m = g^{-1} W_{m-1}$ and $V = \bigcup_{m \in \mathbb{N}} W_m$. V covers $\mathbb{R}^n$, for if not let w be the point of $\mathbb{R}^n - V$ closest to $A_1 \cap A_2$ (which exists by an easy compactness argument). Either $g(w) = w$ when $w \in A_1 \cap A_2$ and hence $w \in W_1$, or g moves w into V, when $\exists m : g(w) \in W_m$ whence $w \in W_{m+1}$, contradiction. It is easy to see that g cannot move w to a new point; it would have to be at the same distance as w from every point in an open set. So V covers $\mathbb{R}^n$ and for any $x \in \mathbb{R}^n\ \exists m$ such that $x \in W_m$, i.e., $g^m(x) \in A_1 \cap A_2$. The result follows.

Remark 3.3. For simplicity we have defined g as the composite of the retractions in some order. It is clear that the argument does not depend on the order. Moreover, replacing g by any sequence of the retractions r_1 through r_J in which each retraction occurs will not affect the result. Thus we have the following corollary.

COROLLARY 3.4. *Taking the A_j to be half-spaces of X^* bounded by hyperplanes $\tilde{x}_j$, corresponding to the points x_j in some data set f, and interpreting the retractions $r_j : X^* \to A_j$ as the standard correction procedure for x_j, we have Theorem* 2.15, *the perceptron convergence theorem for the standard correction algorithm.*

4. Committee and hierarchy nets.

Remark 4.1. The committee net, cognate with the MADALINE, is formally defined as in Definition 2.18 as follows:

DEFINITION 4.2. The committee net of order q is the map

$$C : \prod_{j=1,q} \tilde{X}_j^* \times \mathbb{R}^n \to S^0$$

given by $C(L_1, L_2, \cdots, L_q, x) = \text{sgn}\,[\sum_{j=1,q} \text{sgn}\,(L_j(x))]$.

Remark 4.2.1. Informally we recall that this means that q distinct T.L.U.'s each give a verdict of ± 1, and the majority of the committee determines the output. We shall suppose q odd to avoid ties; strictly speaking C is not defined everywhere on the nominal domain above; for a given set $(L_1, L_2, \cdots, L_q)$, then $C(L_1, L_2, \cdots, L_q, x)$ is not defined for x in the kernel of any of the L_j. This is a set of measure zero, and we shall ignore it; arbitrary conventions are possible that will cope with this. We shall frequently let the map sgn be understood, and write $L_j(x)$ where we mean sgn $(L_j(x))$ in what follows.

Remark 4.3. The simple hierarchy net has not, to my knowledge been treated in the literature. We recall that it consists of a master and two slave units; the master on receipt of a data point allocates the decision to one or other of the two slaves. The output of the chosen slave is the output of the net. Formally we have, as in Definition 2.20, Definition 4.4.

DEFINITION 4.4. The simple hierarchy net is the map

$$H : \tilde{X}^* \times \tilde{X}^* \times \tilde{X}^* \times \mathbb{R}^n \to S^0$$

given by

$$H(L_1, L_2, L_3, x) = (L_3(x)) \cdot \left(\frac{1 + L_1(x)}{2}\right) + (L_2(x))\left(\frac{1 - L_1(x)}{2}\right)$$

(where $L_j(x)$ is an abbreviation for sgn $(L_j(x))$).

Remark 4.5. Much of what follows does not depend on whether we are discussing the committee net or the hierarchy net: the term "net" will in general mean some configuration of T.L.U.'s having the property that a data point of $\mathbb{R}^n$ may be presented to some subset of the set of T.L.U.'s (possibly all of them) and a unique output, ± 1 is allocated to the data point by the net. We shall not inquire how this is accomplished in general, but we observe that the simple hierarchy net and the committee net are nets in this sense. More formally, we have the following definition.

DEFINITION 4.5.1. A *net of order q* is a set of at least q T.L.U.'s and a map

$$N : \prod_{j=1,q} \tilde{X}_j^* \times \mathbb{R}^n \to S^0.$$

Remark 4.6. The *net problem* is as follows: given that the net has delivered its verdict on a particular data point, we require an algorithm (which term is used somewhat loosely to include random processes) for selecting a unit in the net. Then this unit is deemed to be wrong and is corrected by the standard correction procedure. We might reasonably suppose that if the net verdict is correct, that we do not need to select a unit for correction; this supposition, however, is inconvenient, and we do not make it.

Obviously, we require if possible an algorithm having the property that under its action the net will *converge*. In general, we ask for a specific net whether such algorithms exist: first we formalize the problem.

DEFINITION 4.7. The symbol $\underline{\underline{q}}$ denotes the set of

$$q+1 \text{ numbers } \{0, 1, 2, \ldots, q\}.$$

DEFINITION 4.8. A choice process for a net of order q is a map

$$\psi : \mathbb{N} \to \underline{\underline{q}}.$$

Remark 4.8.1. We have then two sequences, σ which picks out the data points from a finite set and "presents" them to the machine, and ψ which, when the machine/net verdict is in error (but even perhaps when it is not in error) will choose a unit in the net and then "correct" it. We have only q T.L.U.'s in the net; we have $q+1$ possible values for $\psi(n)$. We suppose that if $\psi(n) = j$ for $1 \leqq j \leqq q$, we correct the jth unit, and if $\psi(n) = 0$, we make no correction at all.

Remark 4.9. $\psi(n)$ will depend on, in general, all the verdicts of the individual units in the net on the occasion when $\sigma(n)$ is presented. It may also depend on the point $\sigma(n)$ itself, but it should not depend on the whole data set and should certainly not depend upon $\sigma(\mathrm{N})$ for $\mathrm{N} > n$. Otherwise we expect ψ to be a computable function, and we would prefer it to be a simple one that can be computed quickly. In the case of the committee net, for instance, we might, when the net is in error, choose for correction among those units which are wrong, that unit having the absolute value of $L_j(\sigma(n))$ a minimum, i.e., that unit having kernel hyperplane closest to the data point. Or we might choose among the wrong units at random. From our point of view this is a simpler, computable function. (Obviously a uniform probability distribution is appropriate).

We can only deal with convergence of nets when the problem is solvable.

DEFINITION 4.10. Given a net N of order q and a data set $f : D \to S^0$, we say that the data set f is N-solvable iff

$$\exists (L_1, L_2, \cdots, L_q) \in \prod_{j=1,q} \tilde{X}_j^* :$$

$$N(L_1, L_2, \cdots, L_q, x) = f(x), \quad \forall x \in D.$$

Remark 4.11. We recall the standard correction procedure of Definition 2.12. Here, we have in general the difficulty that it does not make sense to say that a particular unit is "wrong". (It does in the committee net, but not in the simple hierarchy net) here we *always* correct the unit $\psi(n)$, so long as $\psi(n) \neq 0$. Formally we have the following.

DEFINITION 4.12. The standard correction procedure for a net unit is the map

$$\Phi : D \times \tilde{X}^* \to \tilde{X}^*,$$

where X^* is the weight space for the unit, and

$$\Phi(x, L) = L - \frac{[(1+\alpha)\cdot\langle\hat{x}, L\rangle]}{\langle\hat{x}, \hat{x}\rangle}\cdot\hat{x},$$

where α is any real number such that $0<\alpha<1$ and $\hat{x}$ is the normal vector in $X^* \approx \mathbb{R}^{n+1}$ to the hyperplane $\tilde{x}$ determined by x (obtained by augmenting the vector x with a 1 in the $(n+1)$th place, in the obvious representation of X^*) $\langle\hat{x}, L\rangle$ is the usual inner product in $\mathbb{R}^{n+1}$(cf. Definition 2.12).

DEFINITION 4.13. Given a net N of order q, a data set $f : D \to S^0$, a training sequence $\sigma : \mathbb{N} \to D$ and a choice process $\psi : \mathbb{N} \to \underline{\underline{q}}$, the *standard action* of (σ, ψ) is the map

$$\Sigma : \mathbb{N} \times \prod_{j=1,q} \tilde{X}_j^* \to \prod_{j=1,q} \tilde{X}_j^*$$

given by

$$n, (L_1, L_2, \cdots, L_q) \rightsquigarrow (L_1, \cdots, \hat{L}_j, \cdots, L_q),$$

where $(L_1, \cdots, L_j, \cdots, L_q)$ differs from $(L_1, \cdots, \hat{L}_j, \cdots, L_q)$ in the jth place for $j = \psi(n)$ when $\psi(n) \neq 0$, and where $\hat{L}_j = \Phi(\sigma(n), L_j)$ where Φ is the map of Definition 4.12.

If $\psi(n) = 0$, then $\Sigma(n, L_1, L_2, \cdots, L_q) = (L_1, L_2, \cdots, L_q)$, and there is no change in the state.

Remark 4.14. This expresses unambiguously the evolution of the net under the combined action of the choice of data points and the choice of unit to correct; now we observe that what is required is an algorithm for ψ that will, for any given initial state of the net and for a reasonably large class of plausible training sequences σ, guarantee that after a finite number of moves N, the net has converged, i.e., reached a solution, and that thereafter, for $n > \mathrm{N}$,

$$\Sigma_n : \prod_{j=1,q} \tilde{X}_j^* \to \prod_{j=1,q} \tilde{X}_j^*$$

is the identity.

We shall suppose that we do *not* have control of σ, but that it is random. For learning machines, i.e., nets modeled on a computer, this assumption is unnecessarily pessimistic; one can control the order of presentation of points, although it is not always clear how one should pick a "good" order, i.e., one that causes rapid convergence.

From the point of view of modeling neutral nets, this total control of input is an unrealistic premise, and the random ordering is a closer approximation.

Remark 4.15. At this point, it is possible to sketch out verbally the argument used and the result obtained. To this end, consider a committee of 3 units learning to classify a data set that is 3-separable in $\mathbb{R}^2$. Now we may draw noughts and crosses on a sheet of paper and the committee becomes three oriented lines that are moved about under the correction procedure. To make things as concrete as possible, we may take an equilateral triangle and put crosses at the vertices and a nought at the center of gravity. Then there is a disposition of the three lines which will correctly classify these four points, as a committee, i.e., by taking the majority verdict.

This solution has the property that for each line, the line is right about some points and generally wrong about others. Consider the maximal subset of points about which the point is right. If the points are called at random, sooner or later there will be called just this set of points in a sequence, and if we allocate corrections at random, then sooner or later such a concentration of just these points will be called when it is this unit's turn to be corrected. This also holds for the other units. Consequently, if we wait long enough and call all the points at random *and* allocate at random, the coincidence will occur of moving the lines into the right positions to give a correct classification.

It will be objected that waiting for a monkey to type out even a single line of Shakespeare demands more patience than seems reasonable. Quantitative examinations of convergence times, however, does something to alleviate the depression engendered by meditating on the proof. Two considerations are relevant, viz. (a) a good deal of symmetry in the solution space exists, and there may be many routes to many solutions; (b) a little redundancy of T.L.U.'s goes a long way. A third matter will be treated in a subsequent paper: there are ways of doing better than blind chance.

There is a complication. It is conceivable that the net as a whole will be right on some of the occasions when we wish to move a line, and hence if we only correct a unit when the net is wrong, we have no guarantee that we can always juggle lines into the correct position. On the other hand, if we correct the net independently of whether it is right or wrong, we may come to a solution state and then go straight back out of it again; we wander through all dispositions of the lines indiscriminately and convergence never occurs. In the one case, we may get locked in a part of the maze unable to get out; in the other case, we can go everywhere but have no way of recognizing the exit.

We want, then, a choice process ψ which usually calls 0 in q when the net is right about a point, but that sometimes doesn't, picking at random from the adaptive units. Moreover, this trick of correcting the net even when it is right must cease after some number of moves, being switched off by the net being consistently right, and being possibly switched on again should the net have some errors left. By means of this somewhat unlikely procedure, we can guarantee a probabilistic convergence: the probability that the net will *not* converge in a finite number of moves is zero.

Remark 4.16. The argument outlined above has to be formalized with some care; as the informal outline suggests, the complications present difficulties.

It is convenient to view the net as navigating a maze: when a unit is corrected with respect to a data point we imagine that a choice has been made that sends the net into a new state. By a state in this context, we mean an equivalence class of all those states which agree in their verdict on every point. One may imagine that the data point in the plane are little pegs; then a line may be moved about from one position to an equivalent position so long as it does not cross over a peg. It is the equivalence class of positions which is the important "state" of a single unit, and the list of these "states" for the net that is the important sense of the term "*net state*". What is important about it is (a) we don't actually care what the actual *position* of each unit is, only at most what its "state" is, and (b) there are only a finite number of distinct "net states".

The collection of net states in this sense can be thought of as a (finite) collection of vertices or nodes. When a choice of unit and of data point can throw a unit from a position in one state to some position in another state, we may draw a directed arc from the first state to the second. In this way we build up a *directed graph*, our maze, and the problem becomes one of taking a random walk in this maze. Formally we have Definition 4.17.

DEFINITION 4.17. Given a net N of order q and a data set $f: D \to S^0$, then two states $(L_1, \cdots, L_q)$ and $(L'_1, \cdots, L'_q)$ are *f-equivalent* iff

$$L_j(x) = L'_j(x) \qquad \forall j : 1 \leqq j \leqq q, \quad \forall x \in D.$$

Remark 4.17.1. It is clear that this is an equivalence relation in the technical sense.

DEFINITION 4.18. With the terminology of Definition 4.17, we shall write $[L_1, \cdots, L_q]$ to denote the set of all states equivalent to $(L_1, \cdots, L_q)$ with respect to a given data set.

Remark 4.18.1. In accordance with conventional mathematical usage, we shall abuse language by referring to the "state" $[L_1, \cdots, L_q]$ when we really mean the set. It is trivial that the set of states $[L_1, \cdots, L_q]$ is finite, since D is.

DEFINITION 4.19. Given a net N of order q and a data set $f: D \to S^0$, we say that the *derived graph* of the pair (N, f) is the collection of vertices or nodes the set of all different states $[L_1, \cdots, L_q]$, and where there is an arc (x, j) from $[L_1, \cdots, L_q]$ to $[L'_1, \cdots, L'_q]$ whenever there is an element

$$(\tilde{L}_1, \tilde{L}_2, \cdots, \tilde{L}_q) \in [L_1, \cdots, L_q]$$

such that

$$(\tilde{L}_1, \cdots, \hat{L}_j, \cdots, \tilde{L}_q) \in [L'_1, \cdots, L'_q],$$

where $\hat{L}_j = \Phi(x, \tilde{L}_j)$, Φ the map of Definition 4.12.

Remark 4.19.1. We recall that a directed graph is defined to be such a collection of vertices and directed arcs; other far more obscure definitions are available in the literature.

Here the arcs and vertices are both finite sets. We observe that there may be none or several arcs between any two vertices. Also, because $(L_1, \cdots, L_q)$ and $(\tilde{L}_1, \cdots, \tilde{L}_q)$ are equivalent (i.e., $[L_1, \cdots, L_q] = [\tilde{L}_1, \cdots, \tilde{L}_q]$), it by no means follows that it is possible to make the transition from $(L_1, \cdots, L_q)$ to anything in $[L'_1, L'_2, \cdots, L'_q]$ just because there is a transition from $(\tilde{L}_1, \cdots, \tilde{L}_q)$ to something in $[L'_1, L'_2, \cdots, L'_q]$. In other words, the arc (x, j) really acts between states $(L_1, \cdots, L_q)$, not between the quotient states $[L_1, \cdots, L_q]$.

Remark 4.20. If we call x and j at random, then we take a random walk on the derived graph. If this graph has closed subgraphs that one can enter but not exit, or if it is disconnected, then the random walk may not take us through all possible vertices. The following proposition is stated without proof.

PROPOSITION 4.21. *A finite directed graph has a distinguished vertex* $*$ *and the property that for every vertex* v *in the graph, there is a route, i.e., a sequence of arcs, starting at* v *and terminating at* $*$.

A route is generated by selecting a vertex at random and then choosing an arc leaving it (uniformly) at random, and iterating from the vertex arrived at. Then the probability is zero that the route generated from any vertex will not pass through $*$ *in a finite number of moves.*

Remark 4.22. $*$ is going to be a solution state; there may be a number of different solutions, and we certainly cannot guarantee terminating at any particular one of them.

Now we define the class of choice functions we shall use.

DEFINITION 4.23. Let N be a net of order q, f a data set, and σ a training sequence given as randomly generated. Then a choice function $\psi : \mathbb{N} \to \underline{\underline{q}}$ is said to be *planetic* if it is constructed as follows:

If $N(\sigma(n)) \neq f(\sigma(n))$ (i.e., if the net is wrong about the nth point called), then $\psi(n)$ is chosen at random from the elements of $\underline{\underline{q}} - \{0\} = \{1, 2, 3, \cdots, q\}$, with a uniform probability distribution.

If $N(\sigma(n)) = f(\sigma(n))$ (i.e., if the net is right), then $\psi(n) = 0$ *unless* $N(\sigma(n-1)) \neq f(\sigma(n-1))$ when $\psi(n)$ is chosen randomly from $\underline{\underline{q}}$ with a uniform probability distribution.

Remark 4.23.1. A choice function constructed according to this rule will select a unit to correct at random when the net is in error, and if the net is correct but was in error last move, then in $q/q+1$ cases a correction will still be made. This last, rather odd, condition is what guarantees that the choice function together with the training sequence is "planetic", i.e., wandering; that is to say, that the state of the net will wander over all possible states, all possible vertices of the derived graph, and not get locked in a closed subgraph.

THEOREM 4.24. *Let* N *be a net of order* q, f *an* N*-solvable data set,* σ *a randomly generated training sequence and* ψ *a planetic choice process. Then under the standard action of* (σ, ψ), *the probability is zero that the net will not converge in some finite number of moves.*

Proof. First we show that the net can actually get to a solution; then we show that it will stay at a solution. Let $*$ denote the class of net states f-equivalent to a solution state: we know there is at least one since f is N-solvable. Let the net start off in an arbitrary state $(L_1, \cdots, L_q)$, the quotient state of f-equivalent net states denoted by $[L_1, \cdots, L_q]$ as before. Since the choice of an arc to leave $[L_1, \cdots, L_q]$ is made at random, we may try to use Proposition 4.21; to do so, it suffices to establish that there is a route from $[L_1, \cdots, L_q]$ to $*$; more particularly, since not all arcs from $[L_1, \cdots, L_q]$ to $[L_1', \cdots, L_q']$ are traversible from the starting state $(L_1, \cdots, L_q)$, we show that there is a route of accessible arcs; i.e., we can choose σ and ψ so as to force convergence.

Take $(Z_1, Z_2, \cdots, Z_q) \in *$. Considering only L_1, we observe that there is some maximal subset $D_1 \subset D = \operatorname{dom} f$, such that Z_1 correctly classifies $f|D_1$. That is, D_1

is linearly separable, and Z_1 correctly separates it, and D_1 is maximal in this respect. Call under σ the points of D_1 in any order and correct under ψ the unit L_1. It may happen that the net as a whole is always right about the points of D_1, but it isn't right about every point unless it is already at a solution state. Hence we can choose a point about which N is wrong, then on the next move, pick a point of D_1 that we need to correct L_1. In this way we can force L_1, by the ordinary perceptron convergence theorem on D_1, until it is in a state equivalent to Z_1. Now repeat on L_2, and so on up to L_q.

The only obstruction to forcing the net from $(L_1, \cdots, L_q)$ to some state equivalent to $*$ is the possibility of arriving at another solution, which will do just as well. (I hope it is clear that I am not suggesting this as a practical convergence algorithm; I am merely establishing that there do exist specific routes from any given state to a solution state!)

Now by Proposition 4.21, it follows that a random walk will sooner or later bring the net to $*$, unless some other solution intervenes. Finally, having arrived at $*$, the last move having corrected an error, it follows that the net is going to make a "correction" on the next move, despite being right about all the points. This "correction" may take it out of $*$, but there is a $1/q+1$ probability of choosing 0 when no correction will be made. Subsequently, no point can move the net. Hence, providing the net passes through $*$ often enough (and it will by the first part of the argument), then we can expect that sooner or later it will stop there, and convergence will occur. Again, the only obstruction to converging at $*$ is converging somewhere else.

Remark 4.25. The theorem guarantees probabilistic convergence: it is less than efficient. We have exchanged generality for credibility to some extent here, and must put the matter right. For the committee net, it is obviously foolish to correct units which are "right"; it makes good sense here to say of a particular unit and a particular data point whether or not the unit is correct in its verdict, and we can accelerate convergence by changing the unit state only if it is wrong. In terms of the derived graph, it makes sense to excise some arcs because we know they lead away from a solution. It is eminently practical therefore to choose at random only among the units which are wrong.

In the case of the simple hierarchy net, it does not always make sense to talk of a unit being "wrong". Was it the master who erred in assigning the decision to the wrong slave? Or did the slave simply bungle the job? There simply is no way of meaningfully deciding the issue, which conclusion must be alarming to moralists.

If the data points are presented at random, then it is not necessary that the unit also be chosen randomly to ensure the walk through the net be a random walk. One might pick the units by strict rotation, or according to any scheme that will not "derandomize" the effect of choosing the data points randomly. Conversely, we can hope that the route through the maze will be sufficiently planetic to converge even if the data points are not called randomly, provided that the choice of units to be corrected is random. In short, it is possible to strengthen the result Theorem 4.24 somewhat; to do so is technically difficult and involves studying the properties of derived graphs in particular cases, and the nature of random walks, so we shall not pursue the matter here.

5. Conclusions. Theorem 4.24 gives a procedure for correcting two types of net of threshold logic units and establishes that if a data set is "learnable" by the net, then failure to converge in a finite number of moves has probability zero of occurrence. It is clear that the proof holds for more complicated nets—hierarchies having more than two layers, hierarchies having committees as elements, and so on. A general argument requires a characterization of such nets which will appear in a subsequent paper.

The practical significance of this result is that it becomes possible to model complicated nets and to train them to correctly classify complicated patterns. It is intuitively plausible that *any* finite data set can be learned by a net having order the "convex complexity" [7] of the data set. We can chop the space into blocks using hierarchies and learn within each block by committees. (The convex complexity of a data set is the minimum number of elements of a cover of the space by convex sets such that wherever two elements of the data set are in the same convex set they have the same sign.) In short, we can be reasonably economical in the number of units we use to learn a given data set.

The analysis of Minsky and Papert [3] of the perceptron and its limitations is aimed mainly at its role in perception; as a model of animal perception it is of course grossly unsatisfactory on a number of counts: as a device for machine classification of data points it has serious drawbacks, which are examined in [3] and [4]. The class of T.L.U. nets I have discussed here is not included in these strictures: the perceptron is essentially a single T.L.U. Instead of restricting ourselves to affine functions as in the case of the single unit, it is fairly clear that we can realize any finitely piecewise affine function by a net. Indeed, there is a sense in which a data set $f: D \to S^0$ is "approximated" by a piecewise affine function realized by a net, the net function being defined over almost all of $\mathbb{R}^n$.

The process by which a finite data set is learned can be viewed as the development of a theorizing machine, from a "random" theory to a theory which is compatible with the data set. One of the limitations of our understanding at the moment is that we cannot in general say anything about the capacity of a given net to learn a given data set; it would be useful to have a way of deciding whether a given data set is N-solvable for a given net N.

Another drawback is the problem of estimating the distribution of convergence times; the simplest procedure is to try it and see. A practical problem arises here; because of the stability of a net of this type under malfunction (an attractive feature from the brain-modeling point of view!) it is possible to have a net converge in a computer simulation despite serious programming errors. On two occasions I have been disappointed by the behavior of a simulated net, only to find on program modification that there were several bugs still in there. This kind of thing improves one's opinion of the net as a candidate for brain modeling, but destroys ones faith in the reliability of quantitative data.

It is, in principle, possible to actually compute the derived graph; but for even the simplest cases this is a big and messy problem. It might be hoped that some qualitative properties of the derived graph might be computable, perhaps by symmetry arguments on the net or on the data set. I do not know which finite graphs are derived graphs for various nets and data sets. Even the crudest estimates relating complexity of the data set, complexity of the net and mean convergence time would be of great interest. Another crucial point is the representation in $\mathbb{R}^n$ of real problems. If one wants to teach a net to recognize symbols on a grid for instance, how can one estimate the complexity of the corresponding data sets in $\mathbb{R}^n$ (one data set for each dichotomy, say).

Some interesting experiments can be done here. It is perfectly possible to use a figure or letter reading program, devised by human ingenuity, to train a net to do the same job. How many units are required for a net to be capable of mastering the task? What net configurations work? How long does it take? It is conceivable that one might dispose of net models of this type, or confirm their credibility as brain models by some estimates of net capacities based on such data.

Acknowledgment. I wish to thank some of my students—Robyn Owens and Jo Goodall—for various assistance connected with this work, and in particular Miss Jane Lake for providing me with a proof of Proposition 4.21.

REFERENCES

[1] NILS NILSSON, *Learning Machines*, McGraw-Hill, New York, 1965.
[2] M. FILLENZ, *Hypothesis for a neuronal mechanism involved in memory*, Nature, 238 (1962), pp. 41, 43.
[3] M. MINSKY AND S. PAPERT, *Perceptrons*, MIT Press, Cambridge, Mass., 1969.
[4] B. ROSENBERG, *A criticism of adaptive neutral nets as models of perception*, Internat. J. of Man–Machine Studies, 1 (1972), pp. 45–53.
[5] J. MYCIELSKI, *Review of 'Perceptrons'*, Bull. Amer. Math. Soc., 78 (1972), pp. 12–15.
[6] H. D. BLOCK, *Review of 'Perceptrons'*, Information and Control, 17 (1970), pp. 501–522.
[7] M. D. ALDER, *On Theories*, Philos. Sci., 40 (1973), pp. 213–226.

SIAM J. Comput.
Vol. 4, No. 4, December 1975

NETWORK FLOW AND TESTING GRAPH CONNECTIVITY*

SHIMON EVEN† and R. ENDRE TARJAN‡

Abstract. An algorithm of Dinic for finding the maximum flow in a network is described. It is then shown that if the vertex capacities are all equal to one, the algorithm requires at most $O(|V|^{1/2} \cdot |E|)$ time, and if the edge capacities are all equal to one, the algorithm requires at most $O(|V|^{2/3} \cdot |E|)$ time. Also, these bounds are tight for Dinic's algorithm.

These results are used to test the vertex connectivity of a graph in $O(|V|^{1/2} \cdot |E|^2)$ time and the edge connectivity in $O(|V|^{5/3} \cdot |E|)$ time.

Key words. Dinic's algorithm, maximum flow, connectivity, vertex connectivity, edge connectivity

1. Network flow. Let $G(V, E)$ be a finite directed graph, where V is the set of vertices and E is the set of edges. Each edge e is assigned a capacity $c(e) \geqq 0$. One of the vertices, s, is called the *source*, and another, t, is called the *sink*. We seek a flow function $f(e)$ on the edges such that for every e, $c(e) \geqq f(e) \geqq 0$ and such that the total flow which enters a vertex, other than s or t, will equal the total flow which leaves the vertex. Of all such flows, we want one for which the net total flow which emanates from s is maximum.

This well-known network flow problem [1] was recently reexamined. A solution in $O(n^5)$ steps, where n is the number of vertices, was produced by Edmonds and Karp [2] in 1969. A solution in $O(|V|^2 \cdot |E|)$ steps was published in Russian by Dinic [3] in 1970.

In this section we present a solution in $O(|V|^2 \cdot |E|)$, essentially the same as Dinic's. (This version was discovered independently by S. Even and J. Hopcroft.)

The algorithm runs in phases, at most $|V| - 1$ in number. We start with zero flow; that is, $f(e) = 0$ for every $e \in E$. In each phase, the flow is increased. New phases are applied until no increase is possible. At that point, the proof of maximality is the same as that of Ford and Fulkerson [1], and it will not be repeated here. However, the algorithm up to that point is not a restriction of the freedom allowed by the Ford and Fulkerson algorithm—as is the case with the Edmonds and Karp algorithm. The computation within each phase is through a different method of labeling and path finding.

Assume that we have a present flow $f(e)$. An edge is *usable* in the *forward direction* if $f(e) < c(e)$, and it is usable in the *backward direction* if $f(e) > 0$. Clearly, an edge may be usable in both directions.

Each phase starts with a breadth-first search from s. That is, we start by labeling s with 0; i.e., $\lambda(s) = 0$. Next, we label with 1 all unlabeled vertices which are reachable from s via a single usable edge, where the usable direction is from s to

* Received by the editors June 27, 1974, and in revised form November 15, 1974.

† Computer Science Department, Technion-Israel Institute of Technology, Haifa, Israel. On leave of absence from the Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel. Parts of this work were completed during the summers of 1972 and 1973 while he visited the Department of Computer Science, Cornell University, Ithaca, New York.

‡ Computer Science Division, University of California at Berkeley, Berkeley, California 94720. The work of this author was supported in part by the National Science Foundation under Grant NSF-GJ-35604X, and by a Miller Research Fellowship.

them. This action is called *scanning* s. We scan the vertices in the order they are labeled; that is, first-labeled, first-scanned. When v is scanned, all unlabeled vertices reachable from v are labeled $\lambda(v) + 1$. Once t is labeled while scanning, say, w, we continue scanning all labeled but unscanned vertices v for which $\lambda(v) = \lambda(w)$, and terminate the breadth-first search once we reach a vertex v, waiting to be scanned, for which $\lambda(v) > \lambda(w)$.

It is easy to see that this is nothing but the well-known algorithm [4] for finding a shortest path from s to t when length is measured by the number of edges on the path. Edges are used only in a usable direction, and every shortest path indicates an augmenting path for increasing the flow.

As we conduct the breadth-first search, we prepare a copy of all vertices and edges traced. For every vertex v, we keep a list of the edges which are usable from v to vertices which are labeled $\lambda(v) + 1$. The total number of steps required for this is $O(|E|)$ if the data structure of the graph is originally by lists of adjacent edges for each vertex. Let us call the newly prepared structure the *auxiliary graph*. All paths from s to t in the auxiliary graph are of length $\lambda(t)$ edges.

Now we use the auxiliary graph to trace flow augmenting paths, from s to t. These paths are found by depth-first search [5], [6]. We start tracing from s, move through a usable edge to a vertex labeled 1, move from there to a vertex labeled 2, etc. If we reach t, we have an augmenting path, and we push through it as much flow as is possible. All edges along the path which cease to be usable (in the direction used) due to this change in the flow are erased from the auxiliary graph, and a new depth-first search is started. Clearly, each time an augmenting path is used, at least one edge is removed from the auxiliary graph. (Such an edge is called a *bottleneck* of the path.) If the depth-first search ends in a dead-end, namely, a vertex v from which no usable edge leads to a vertex whose label is $\lambda(v) + 1$, then we retrace to the vertex preceding v on the path and erase the last edge from the path and from the auxiliary graph. We continue the search from there. If we cannot proceed from s the phase is over.

Finding one successful path takes $O(\lambda(t))$ steps, and in the case of failure (when we retrace), the number of steps cannot exceed $O(\lambda(t))$. In either case, at least one edge is erased. Thus the total number of steps in tracing paths during one phase is bounded by $O(|V| \cdot |E|)$. It follows that each phase cannot take more than $O(|V| \cdot |E|)$ steps. We shall show that the number of phases is bounded by $|V| - 1$, and therefore the whole algorithm does not take more than $O(|V|^2 \cdot |E|)$.

Each auxiliary graph, when first constructed, describes all shortest augmenting paths for the present $f(e)$. It has the property that there is no usable edge which leads, in a usable direction, from a vertex v to a vertex whose label is higher than $\lambda(v) + 1$. The changes in the flow performed by pushing flow through a shortest augmenting path may create a new usable direction for some of its edges, but these directions are from some v to a vertex labeled $\lambda(v) - 1$. Thus the property remains valid through all the changes during the phase. It follows that at the end of the phase, a shortest augmenting path is of length higher than $\lambda(t)$. Thus, from phase to phase, $\lambda(t)$ increases, and therefore the number of phases is bounded by $|V| - 1$.

In the last phase, the labeling does not reach t, and the proof of maximality (which brings up the max-flow min-cut theorem) is identical with that of Ford and Fulkerson [1].

2. Zero-one network flow. Consider now a network flow problem, as above, except that for all $e \in E$, $c(e) = 1$. (One should realize that even if for all $e \in E$, $c(e)$ is integral, not necessarily 1, the algorithm described above will never introduce fractions.[1]) It follows that for all flow functions along the way and for every e, $f(e)$ is either zero or one.

When we trace an augmenting path, the increase in the flow through it is exactly 1, and all edges used in it are erased from the auxiliary graph; that is, each edge, when used, is a bottleneck. Also, each time we backtrack one edge, it is erased. It follows that the number of steps per phase is at most $O(|E|)$, and the total number of steps the algorithm requires is bounded by $O(|V| \cdot |E|)$.

Another bound, $O(|E|^{3/2})$, which is better for sparse graphs can be proved, but we have to prepare a few tools first.

Let $G(V, E)$ be a network with integral edge capacities $c(e)$. Assume the maximum total flow from s to t is M. Also, assume that through Dinic's algorithm (or any other algorithm which does not introduce fractions), the flow has been increased from zero to a present flow function f, and the present total flow from s to t is F. Define now the network $\tilde{G}(V, \tilde{E})$ with capacities $\tilde{c}(e)$ as follows:

(i) If $e \in E$ and $f(e) < c(e)$, then $e \in \tilde{E}$ and $\tilde{c}(e) = c(e) - f(e)$.

(ii) If $e \in E$ and $f(e) > 0$ then $e' \in E$, where e' connects between the same two vertices as e, but in the reverse direction, and $\tilde{c}(e') = f(e)$.

Clearly, each edge of G generates at least one edge in $\tilde{G}$, and if $0 < f(e) < c(e)$, e generates two edges. However, in the case that $c(e) = 1$ for every edge e, each edge generates exactly one edge in G, since $f(e)$ can be either 0 or 1.

We shall use the following notation: $(S; \bar{S})_G$ is a *cut* separating s from t in G; that is, $S \cup \bar{S} = V$, $S \cap \bar{S} = \emptyset$, $s \in S$, $t \in \bar{S}$ and $(S; \bar{S})$ is the set of edges in G which lead from a vertex in S to a vertex in $\bar{S}$.

LEMMA 1. *The maximum flow in* $\tilde{G}$ *is* $M - F$.

Proof. The definition of $\tilde{G}$ implies that

$$\sum_{a \in (S;\bar{S})_{\tilde{G}}} \tilde{c}(a) = \sum_{a \in (S;\bar{S})_G} (c(a) - f(a)) + \sum_{e \in (\bar{S};S)_G} f(e),$$

However,

$$F = \sum_{a \in (S;\bar{S})_G} f(a) - \sum_{e \in (\bar{S};S)_G} f(e),$$

Thus

$$\sum_{a \in (S;\bar{S})_{\tilde{G}}} \tilde{c}(a) = \sum_{a \in (S;\bar{S})_G} c(a) - F,$$

This implies that a minimum cut of $\tilde{G}$ corresponds to a minimum cut of G (namely, is defined by the same S). By the max-flow min-cut theorem, the value of the minimum cut of G is M. Thus the value of a minimum cut in $\tilde{G}$ is $M - F$. Again, by the max-flow min-cut theorem, the maximum flow in $\tilde{G}$ is $M - F$. Q.E.D.

LEMMA 2. *Let $G(V, E)$ be a network in which $c(e) = 1$ for every $e \in E$. Assume the maximum flow from s to t is M. The distance from s to t when the flow is zero everywhere is at most $|E|/M$.*

[1] This holds for all "reasonable" algorithms for network flow problems.

Proof. Let $V_i = \{v | v \text{ is at distance } i \text{ from } s\}$. Here the distances are with zero flow and V_i corresponds to the set of vertices on the ith level of the first phase of Dinic's algorithm. Let l be the distance from s to t. The set of edges from V_i to V_{i+1} is a cut, and therefore the number of edges between V_i and V_{i+1} is at least M. Thus

$$l \cdot M \leqq |E|. \qquad \text{Q.E.D.}$$

THEOREM 1. *For networks with unit edge capacities, Dinic's algorithm requires at most* $O(|E|^{3/2})$ *steps.*

Proof. If $M \leqq |E|^{1/2}$, then the number of phases is bounded by $|E|^{1/2}$, and the result follows. Otherwise, consider the phase during which the flow reaches the value $M - |E|^{1/2}$. The value of the flow, F, when the auxiliary graph for this phase is constructed is less than $M - |E|^{1/2}$. However, this auxiliary graph is identical with the initial auxiliary graph for the network $\tilde{G}$. $\tilde{G}$ still has unit edge capacities, and by Lemma 1 its maximum flow is

$$\tilde{M} = M - F > M - (M - |E|^{1/2}) = |E|^{1/2}.$$

Thus, by Lemma 2, the length, l, of a shortest augmenting path satisfies

$$l \leqq \frac{|E|}{\tilde{M}} \leqq |E|^{1/2}.$$

Thus the number of phases up to this point is at most $|E|^{1/2} - 1$, and since the number of phases to completion is at most $|E|^{1/2}$, the total number of phases is at most $2|E|^{1/2}$. Q.E.D.

A network is of *type* 1 if it satisfies the following conditions:

(i) All (edge) capacities are equal to 1.

(ii) There are no parallel edges; that is, an edge is identified by its start and end vertices.

LEMMA 3. *Let* $G(V, E)$ *be a network of type* 1, *with maximum flow* M *from* s *to* t. *The distance from* s *to* t *when the flow is zero everywhere is at most* $2|V|/\sqrt{M}$.

Proof. Let V_i and l be as in the proof of Lemma 2. Since the network is of type 1, we have $|V_i| \cdot |V_{i+1}| \geqq M$. Thus, for all $0 \leqq i < l$, either

$$|V_i| \geqq \sqrt{M} \quad \text{or} \quad |V_{i+1}| \geqq \sqrt{M}.$$

Since $\sum_{i=0}^{l} |V_i| \leqq |V|$, we have

$$\left\lfloor \frac{l+1}{1} \right\rfloor \cdot \sqrt{M} \leqq |V|,$$

and $l \leqq 2|V|/\sqrt{M}$. Q.E.D.

THEOREM 2. *For networks of type* 1, *Dinic's algorithm requires at most* $O(|V|^{2/3} \cdot |E|)$ *steps.*

Proof. The proof is similar to that of Theorem 1: if $M \leqq |V|^{2/3}$, the result follows immediately. Let F be the flow when the auxiliary graph for the phase during which the flow reaches the value $M - |V|^{2/3}$ is constructed. Again, this auxiliary graph is identical to the initial auxiliary graph for the network $\tilde{G}$. $\tilde{G}$ may not be of type 1 since it may have parallel edges, but it can have at most two parallel edges from one vertex to another.[2] By Lemma 1, $\tilde{M} > |V|^{2/3}$. By a variation of Lemma 3, the length l of a shortest augmenting path satisfies

$$l \leqq \frac{2\sqrt{2}|V|}{\sqrt{|V|^{2/3}}} = 2\sqrt{2} \cdot |V|^{2/3}.$$

Thus the number of phases up to this point is at most $O(|V|^{2/3})$ and since the number of phases to completion is at most $|V|^{2/3}$, the total number of phases is at most $O(|V|^{2/3})$. Q.E.D.

In certain applications, we need upper bounds on the flow through vertices. This restriction can be translated to bounds on the flow through edges as follows: each vertex v which has a vertex capacity $c(v)$ is split into two vertices, v' and v'', which have no explicit upper bound on the flow through them; a new edge e connects from v' to v'' and $c(e) = c(v)$; all edges which formerly led to v now lead to v', and all edges which emanated from v now emanate from v''. Clearly, the new edge e and its capacity implicitly specify the upper bound on the flow through v.

A network is of *type* 2 if all (edge) capacities are equal to 1 and every vertex v other than s or t either has a single edge emanating from it or has a single edge entering it.

One important source of such networks is the case of networks with vertex unit capacities for all vertices other than the source and the sink, which were translated into edge capacities as above. Even if the original network had no edge capacities (∞), all edge flows (assuming no edges go directly from s to t) are implicitly bounded by 1.

LEMMA 4. *Let $G(V, E)$ be a network of type 2 with maximum flow M from s to t. The distance from s to t when the flow is zero everywhere is at most* $(|V| - 2)/M + 1$.

Proof. The structure of G implies that a flow in G can be decomposed into vertex-disjoint directed paths from s to t.[3] The number of these paths is equal to the value of the flow. Assume we have a flow function f which achieves M. Let l be the length of a shortest path among the paths implied by f. Thus each path uses at least $l - 1$ intermediate vertices. We have

$$M \cdot (l - 1) \leqq |V| - 2. \qquad \text{Q.E.D.}$$

LEMMA 5. *If G is a network of type 2 and the present flow function is f, then $\tilde{G}$ is also of type 2.*

Proof. If there is no flow through v (per f), then v still satisfies the condition that there is a single edge entering it or a single edge emanating from it. If the flow going through v is 1, assume it enters via e_1, and leaves via e_2. In $\tilde{G}$ both these edges do not appear, but each gives rise to an edge in the reverse direction. The other edges of G which are incident to v remain intact in $\tilde{G}$. Thus the number of incoming edges and the number of outgoing edges of v did not change. Q.E.D.

[2] In G, we may have antiparallel edges; that is e and e', where both connect between the same two vertices but in opposite directions. One of them may stay in $\tilde{G}$ while the other gives rise to an edge which is parallel to the first.

[3] Namely, no two paths share a vertex except s and t. In addition, the flow may imply directed cycles which are of no interest to us.

THEOREM 3. *For a network of type 2, Dinic's algorithm requires at most* $O(|V|^{1/2} \cdot |E|)$ *steps.*

Proof. If $M \leqq |V|^{1/2}$, then the number of phases is bounded by $|V|^{1/2}$, and the result follows. Otherwise, consider the phase during which the flow reaches the value $M - |V|^{1/2}$. Thus the value of the flow, F, when the auxiliary graph for this phase is constructed is less than $M - |V|^{1/2}$. However, this auxiliary graph is identical to the initial auxiliary graph for the network $\tilde{G}$. By Lemma 5, $\tilde{G}$ is also of type 2. By Lemma 1, the maximum flow in $\tilde{G}$ is greater than $|V|^{1/2}$. By Lemma 4, the length, l, of a shortest augmenting path satisfies

$$l \leqq \frac{|V| - 2}{|V|^{1/2}} + 1 = O(|V|^{1/2}).$$

Thus the number of phases up to this point is at most $O(|V|^{1/2})$. Since the number of phases to completion is at most $|V|^{1/2}$, the total number of phases is at most $O(|V|^{1/2})$. Q.E.D.

3. Applications. We want to point out two areas of applications of the results of the previous sections. They are:

(i) matching in the bipartite graph;

(ii) connectivity of a graph.

The best known algorithm for finding a maximum matching in a bipartite graph is that of Hopcroft and Karp [7]. Their algorithm takes at most $O(n^{2.5})$ steps; it is a variant of the Hungarian method and is very close to Dinic's algorithm, in spite of the fact that they do not use the network-flow formulation. In fact, we have borrowed the idea for the bounds of the previous section from them. However, their result can be viewed as a special case of Theorem 3. One can use the network-flow approach to solve the maximum matching in the bipartite graph [1], and the network is of type 2.

In the remainder of this section we shall discuss the testing of connectivity in a graph.

Let $G(V, E)$ be a finite undirected graph. We assume that G has no self-loops. A set of vertices, S, is called an (a, b) *vertex separator* if $\{a, b\} \subset V - S$ and every path connecting a and b passes through at least one vertex of S. Let $N(a, b)$ be the least cardinality of an (a, b) vertex separator, assuming one exists.[4] It is a theorem that $N(a, b)$ is equal to the maximum number of vertex disjoint paths connecting a with b. This theorem is well known and is one of the variations of Menger's theorem [8]. It is not only reminiscent of the max-flow min-cut theorem, but in fact can be proved by it. Dantzig and Fulkerson [9] pointed out this relationship, and their proof offers an algorithm to determine $N(a, b)$. This is done as follows:

Construct a directed network flow graph $\bar{G}(\bar{V}, \bar{E})$, where $\bar{V} = V$ and $\bar{E}$ is a set of directed edges; for each $e \in E$, we have e' and e'' in $\bar{E}$, where e' and e'' connect between the two end vertices of e and are directed in opposite directions. Each v, other than a and b, has vertex capacity 1. These vertex capacities can now be translated to edge capacities, as was pointed out in the previous section. The maximum flow in this network is equal to $N(a, b)$. This last network is of type 2, and therefore Dinic's algorithm achieves this result in at most $O(|V|^{1/2} \cdot |E|)$ steps. (See Theorem 3).

[4] Clearly, if a and b are connected by an edge, then no (a, b) vertex separator exists.

The *vertex-connectivity*, c, of G is defined in the following way:

(i) If G is completely connected,[5] then $c = |V| - 1$.

(ii) If G is not completely connected, then $c = \min_{a,b} N(a, b)$.

(If G is not completely connected, then the minimum value of $P(a, b)$, where $P(a, b)$ is the maximum number of vertex disjoint paths connecting a and b, will be equal to $\min_{a,b} N(a, b)$, in spite of the fact that $N(a, b)$ is only defined for pairs which are not connected by an edge.)

The obvious way, then, to find c if G is not completely connected is to compute $N(a, b)$ for all pairs a, b which are not connected by an edge. This leads to at most $O(|V|^2)$ computations, and each requires at most $O(|V|^{1/2} \cdot |E|)$ steps. Hence at most $O(|V|^{2.5} \cdot |E|)$ steps.

However, a slightly better bound can be proven.

LEMMA 6. *The (edge or vertex) connectivity, c, of an undirected graph $G(V, E)$ with no self-loops and no parallel edges satisfies $c \leqq 2|E|/|V|$.*

Proof. The connectivity cannot exceed $\min_v d(v)$, where $d(v)$ is the degree of vertex v.[6] Also,

$$\sum_V d(v) = 2 \cdot |E|,$$

Thus $c \leqq 2|E|/|V|$. Q.E.D.

Now let us conduct the procedure in the following manner: we choose a vertex v_1 and compute $N(v_1, v)$ for each v not connected to v_1 by an edge; there are at most $|V|$ such computations. We repeat the computation for v_2, v_3, etc. We terminate with v_k once k exceeds the minimum value of $N(a, b)$ observed so far, γ.

THEOREM 4. *The value γ resulting from the procedure above is equal to the connectivity, c.*

Proof. By Menger's theorem, there is a vertex separator S such that $|S| = c$. Thus at least one of the vertices $v_1, v_2, \cdots, v_{c+1}$ is not in the separator. Assume it is v_j. There is a vertex v such that $N(v_j, v) = c$. Clearly $\gamma \geqq c$. Also $k > \gamma$. Thus $k \geqq c + 1$. Therefore $\gamma = c$. Q.E.D.

Lemma 6 and Theorem 4 imply that $k \leqq 2|E|/|V| + 1$. Thus the total number of steps of our procedure is at most $O(|V|^{1/2} \cdot |E|^2)$.

In case G is a directed graph, similar definitions and approach lead to the same result, except that for each $v_1, v_2, \cdots, v_k$, we compute both $N(v_i, v)$ and $N(v, v_i)$ (which are now not necessarily the same) for each applicable v.[7]

A natural idea, in relation to these computations is to use the technique of Gomory and Hu [10]. They find the maximum flows between every two vertices in an undirected flow graph by solving only $|V| - 1$ flow problems. However, their technique is not applicable to directed graphs. Observe that the network flow problems we solve, even for the vertex connectivity of undirected graphs, are all directed. Thus this does not suggest an improvement.

Now let us consider the question of edge connectivity. Again, let $G(V, E)$ be a finite undirected graph. A set of edges, T, is called an (a, b) *edge separator* if

[5] Each pair of vertices is connected by an edge. In this case, there are no vertex separators.

[6] The degree of a vertex is the number of edges incident to it.

[7] If there is an edge from v_i to v, $N(v_i, v)$ is not computed, and if there is an edge from v to v_i, $N(v, v_i)$ is not computed.

$\{a, b\} \subset V$ and every path connecting a and b passes through at least one edge of T. Let $M(a, b)$ be the least cardinality of an (a, b) edge separator. It is a theorem that $M(a, b)$ is equal to the maximum number of edge disjoint paths connecting a with b. This is another variation of Menger's theorem, and again, one can use the network flow approach to determine $M(a, b)$. Here, too, we may construct $\bar{G}$, but one can use the undirected graph, with edge capacities all equal to 1. Since $\bar{G}$ is of type 1, both Theorem 1 and Theorem 2 provide upper bounds on the number of steps Dinic's algorithm will need. Thus

$$O(|E| \cdot \min \{|V|^{2/3}, |E|^{1/2}\})$$

is an upper bound on the number of steps for evaluating $M(a, b)$.

The *edge connectivity*, c', of G is defined by $c' = \min_{a,b} M(a, b)$.

Let T be a minimum edge separator in G; that is, $|T| = c'$. Let v be any vertex of G; then every vertex v' on the other side of T satisfies $M(v, v') = c'$. Thus in order to determine c', we can use

$$c' = \min_{v' \in V - \{v\}} M(v, v').$$

This takes at most $O(|V| \cdot |E| \cdot \min \{|V|^{2/3}, |E|^{1/2}\})$ steps.

The approach described above can be used to determine the edge connectivity for directed graphs, too, with the modification that for every v', both $M(v, v')$ and $M(v', v)$ have to be computed. The same bounds follow.

It is interesting to note that using the technique of Gomory and Hu would yield the same bound for edge connectivity in the undirected case, when one uses Dinic's algorithm to solve each of the $|V| - 1$ flow problems.[8] However, our observation is simpler and works for directed graphs as well.

4. Lower bounds. In this section we shall show that the upper bounds on Dinic's algorithm, discussed in §§ 1 and 2, are tight. Namely, in each case there are graphs for which the number of steps is as high as the upper bound.

N. Zadeh [11] showed a family of flow problems for which Edmonds and Karp's algorithm requires $O(n^3)$ augmenting paths and a total of $O(n^5)$ steps. The same family requires $O(|V|^2 \cdot |E|)$ steps when Dinic's algorithm is used, thus proving that the bound given in § 1 cannot be improved.

Let us now consider the problem of maximum matching for a family of bipartite graphs.

Let

$$X_m = \{a_{ij} | 1 \leqq j \leqq i \leqq m\}, \qquad Y_m = \{b_{ij} | 1 \leqq j \leqq i \leqq m\},$$
$$E'_m = \{(a_{ij}, b_{ij}) | 1 \leqq j \leqq i \leqq m\},$$
$$E''_m = \{(a_{ij}, b_{i,j+1}) | 1 \leqq j < i \leqq m\},$$
$$E_m = E'_m \cup E''_m.$$

The bipartite graph $G_m(X_m, Y_m, E_m)$ is drawn for $m = 4$ in Fig. 1. Clearly, the maximum matching in this case is unique and is given by E'_m. The value of the

[8] In the case of edge connectivity of undirected graphs their technique is applicable.

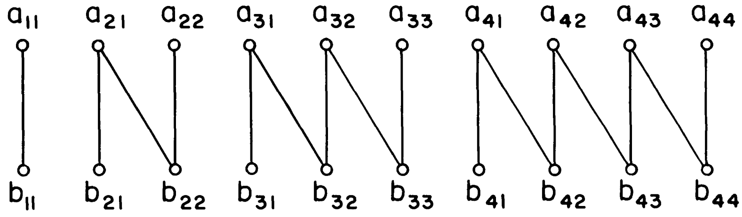


FIG. 1

matching is

$$\frac{m(m+1)}{2} = O(m^2),$$

and the number of vertices in the graph is

$$|V| = m(m+1) = O(m^2).$$

Assume now that we are looking for a maximum matching for G_m by using Dinic's algorithm, as suggested in § 3. The network is achieved by adding two new vertices: s, the source, and t, the sink. Also, connect s with each a_{ij} via a directed edge (s, a_{ij}) with capacity 1, direct all the edges of G_m from X_m to Y_m and assign each edge the capacity 1 (one may use here ∞ as well), and, finally, connect each b_{ij} to t via a directed edge (b_{ij}, t) again with capacity 1. The network is shown, for $m = 4$, in Fig. 2.

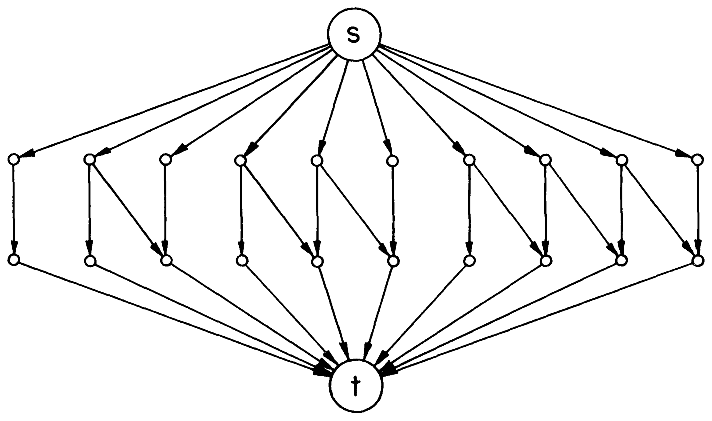


FIG. 2

The first phase of Dinic's algorithm may produce a unit flow in each of the edges of E''_m. The corresponding matching is shown in Fig. 3, for $m = 4$, where the edges in the matching are represented by wiggly lines. In this case, the second phase will add $\{(a_{21}, b_{21}), (a_{22}, b_{22})\}$ to the matching, and subtract $\{(a_{21}, b_{22})\}$. In general, the ith phase will add to the matching the set

$$\{(a_{i1}, b_{i1}), (a_{i2}, b_{i2}), \cdots, (a_{ii}, b_{ii})\}$$

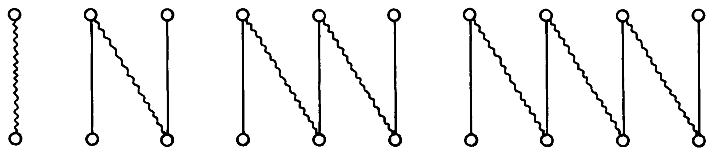


FIG. 3

and subtract the set

$$\{(a_{i1}, b_{i2}), (a_{i2}, b_{i3}), \cdots, (a_{i,i-1}, b_{ii})\}.$$

Therefore the number of phases will be m. It is not hard to see that the total number of steps is $O(m^3)$. Since $|E| = O(m^2)$, these examples show that the bounds given by Theorems 1 and 3 are tight. Namely, the bound $O(|E|^{3/2})$ is the best possible, for graphs with unit edge capacities, in terms of $|E|$, and the bound $O(|V|^{1/2} \cdot |E|)$ is the best possible, for graphs of type 2 in terms of $|V|$ and $|E|$.

Clearly, for dense graphs of type 2, the bound $O(|V|^{1/2} \cdot |E|)$ is more informative than $O(|E|^{3/2})$, and one may wonder if $O(|V|^{1/2} \cdot |E|)$ still remains tight there. A family of dense graphs $(O(|E|) = O(|V|^2))$ for which this bound is still tight is achieved by adding to G_m the following set of edges:

$$\{(a_{ij}, b_{kl}) | (1 \leqq j \leqq i < k) \wedge (j \leqq l \leqq k \leqq m)\}.$$

The steps of Dinic's algorithm can be chosen in such a way that none of these edges ever enters the matching. An examination reveals that

$$|E| = O(m^4), \qquad |V| = O(m^2),$$

and the number of steps is $O(m^5)$.

Next, we want to show that the bound given by Theorem 2 cannot be improved either. Let

$$\begin{aligned} V_m = \{s, t\} &\cup \{a_i | 1 \leqq i \leqq m^3\} \cup \{b_i | 1 \leqq i \leqq m^3\} \\ &\cup \{c_i | 1 \leqq i \leqq m^2\} \cup \{d_{ij} | (1 \leqq i \leqq m^2) \wedge (1 \leqq j \leqq m)\} \\ &\cup \{e_i | 1 \leqq i \leqq m^2\}, \end{aligned}$$

and

$$\begin{aligned} E_m = &\{(s, a_i) | 1 \leqq i \leqq m^3\} \cup \{(e_i, t) | 1 \leqq i \leqq m^2\} \\ &\cup \{(a_i, b_j) | 1 \leqq i, j \leqq m^3\} \\ &\cup \{(b_i, c_j) | (1 \leqq i \leqq m^3) \wedge (1 \leqq j \leqq m^2)\} \\ &\cup \{(c_i, d_{i1}) | 1 \leqq i \leqq m^2\} \\ &\cup \{(d_{m^2,i}, e_j) | (1 \leqq i \leqq m) \wedge (1 \leqq j \leqq m^2)\} \\ &\cup \{(d_{ij}, d_{i+1,k}) | (1 \leqq i < m^2) \wedge (1 \leqq j, k \leqq m)\}. \end{aligned}$$

The graph $G_m(V_m, E_m)$ is shown for the case $m = 2$ in Fig. 4. Now assume we want to find the maximum number of edge disjoint paths between s and t. If we use Dinic's algorithm for finding a maximum flow from s to t, where all edge capacities

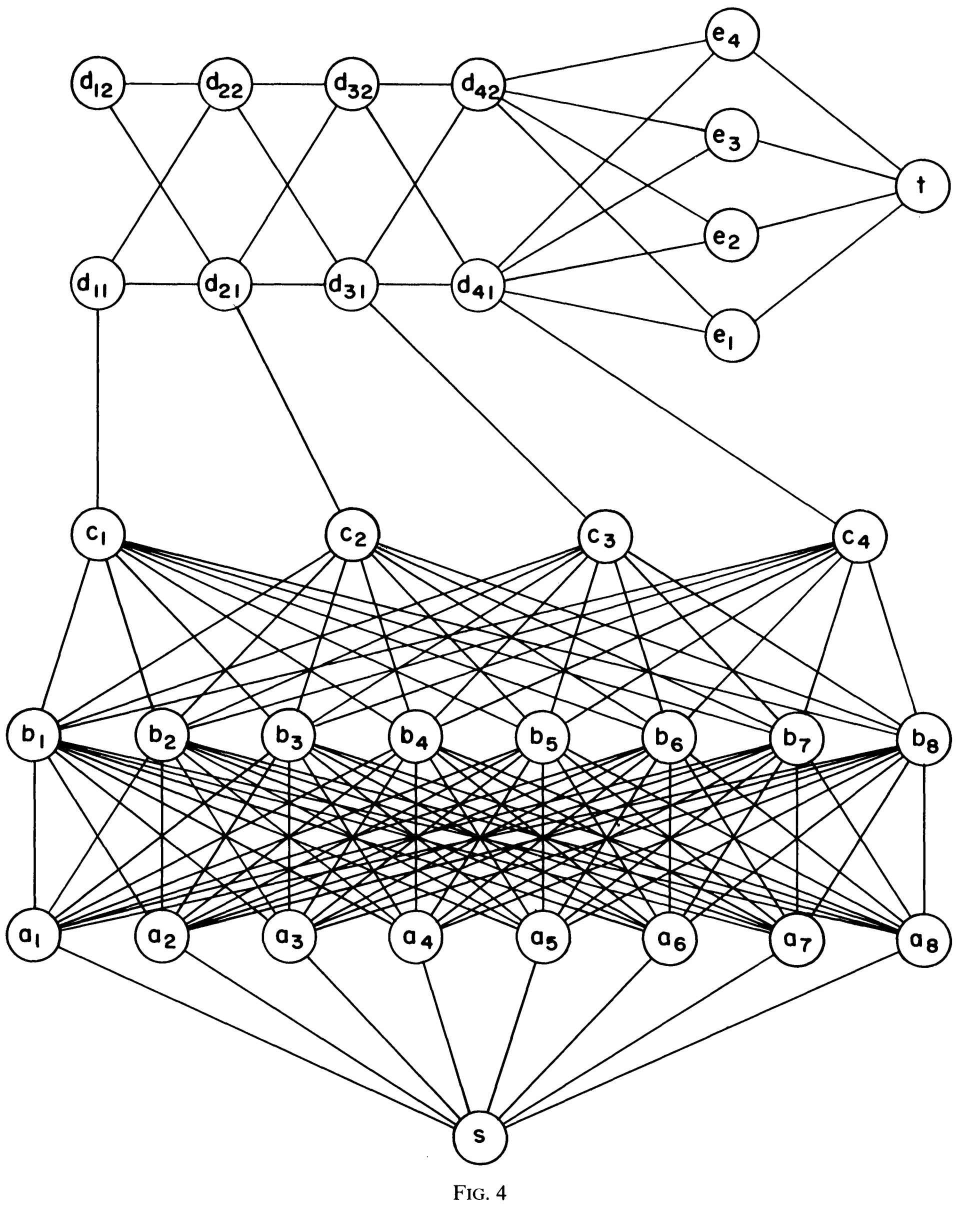


FIG. 4

are 1, we first find a path of length 6 (via $d_{m^2,1}$), then of length of 7, etc., up to a path of length $m^2 + 5$ (via d_{11}). The number of phases is therefore $O(m^2)$. The number of edges is $O(m^6)$, and so is the number of steps per phase. Thus the total number of steps is $O(m^8)$. Since $|E| = O(m^6)$ and $|V| = O(m^3)$, $O(m^8) = O(|V|^{2/3} \cdot |E|)$. This shows that the bound given by Theorem 2 is tight.

5. Remarks. One may make changes in Dinic's algorithm to produce an algorithm for which the lower bounds, as established by the examples of the previous section, are not equal to the upper bounds of § 2. However, for all the changes we have tried, we could find other examples which showed that the upper bounds of § 2 are tight. Yet, let us show some results which seem to indicate that a better algorithm exists.

THEOREM 5. *For a network of type* 2, *the total length of all augmenting paths in Dinic's algorithm is at most* $O(|V| \cdot \log |V|)$.

Proof. The last augmenting path, by Lemma 4, is of length at most $(|V| - 2)/1 + 1$, the next to last is at most $(|V| - 2)/2 + 1$ long, etc. Since the number of augmenting paths is at most $|V| - 1$, the total length of the augmenting paths L satisfies

$$L \leqq |V| - 1 + (|V| - 2) \cdot \sum_{i=1}^{|V|-1} \frac{1}{i} = O(|V| \cdot \log |V|).$$

THEOREM 6. *For a network of type* 1, *the total length of all augmenting paths in Dinic's algorithm is at most*

$$O(\min \{|V|^{3/2}, |E| \cdot \log |V|\}).$$

Proof. By a similar argument, following this time Lemma 3, and observing that the number of augmenting paths is again bounded by $|V| - 1$, we get

$$L \leqq |V| - 1 + 2|V| \sum_{i=1}^{|V|-1} \frac{1}{\sqrt{i}} = O(|V|^{3/2}).$$

On the other hand, if we use Lemma 2, we get

$$L \leqq |E| \cdot \sum_{i=1}^{|V|-1} \frac{1}{i} = O(|E| \cdot \log |V|) \qquad \text{Q.E.D.}$$

It remains to be shown that one could trace all these paths without spending more time than their total lengths.

REFERENCES

[1] L. R. FORD AND D. R. FULKERSON, *Flows in Networks*, Princeton University Press, Princeton, N.J., 1962.

[2] J. EDMONDS AND R. M. KARP, *Theoretical improvements in algorithmic efficiency for network flow problems*, J. Assoc. Comput. Mach., 19 (1972), pp. 248–264.

[3] E. A. DINIC, *Algorithm for solution of a problem of maximum flow in a network with power estimation*, Soviet Math. Dokl., 11 (1970), pp. 1277–1280.

[4] E. F. MOORE, *The shortest path through a maze*, Proc. of an Internat. Symp. on the Theory of Switching (April 1957), Harvard University Press, Cambridge, Mass., 1959, pp. 285–292.

[5] J. HOPCROFT AND R. TARJAN, *Algorith* 447: *Efficient algorithms for graph manipulation*, Comm., ACM, 16 (1973), pp. 372–378.

[6] R. TARJAN, *Depth-first search and linear graph algorithms*, this Journal, 2 (1972), pp. 146–160.

[7] J. E. HOPCROFT AND R. M. KARP, *An* $n^{5/2}$ *algorithm for maximum matching in bipartite graphs*, this Journal, pp. 225–231.

[8] K. MENGER, *Zur allgemeinen Kurventheorie*, Fund. Math., 10 (1927), pp. 96–115.

[9] G. B. DANTZIG AND D. R. FULKERSON, *On the max-flow min-cut theorem of networks*, Linear Inequalities and Related Systems, Annals of Math. Study 38, Princeton University Press, Princeton, N.J., 1956, pp. 215–221.

[10] R. E. GOMORY AND T. C. HU, *Multi-terminal network flows*, J. Soc. Indust. and Appl. Math., 9 (1961), pp. 551–570.

[11] N. ZADEH, *Theoretical efficiency of the Edmonds–Karp algorithm for computing maximal flows*, J. Assoc. Comput. Mach., 19 (1972), pp. 184–192.

SIAM J. COMPUT.
Vol. 4, No. 4, December 1975

A SIMPLE ALGORITHM FOR GLOBAL DATA FLOW ANALYSIS PROBLEMS*

MATTHEW S. HECHT† AND JEFFREY D. ULLMAN‡

Abstract. A simple, iterative bit propagation algorithm for solving global data flow analysis problems such as "available expressions" and "live variables" is presented and shown to be quite comparable in speed to the corresponding interval analysis algorithm. This comparison is facilitated by a result relating two parameters of a reducible flow graph (rfg). Namely, if G is an rfg, d is the largest number of back edges found in any cycle-free path in G, and k is the length of the interval derived sequence of G, then $k \geqq d$. (Intuitively, k is the maximum nesting depth of loops in a computer program, while d is a measure of the maximum loop-interconnectedness.) The node ordering employed by the simple algorithm is the reverse of the order in which a node is last visited while growing any depth-first spanning tree of the flow graph. In addition, a dominator algorithm for an rfg is presented which takes O(edges) bit vector steps.

Key words. global code improvement, flow graph, reducibility, interval analysis, dominance, depth-first spanning tree, data flow analysis, available expressions, live variables

1. Introduction. When analyzing computer programs for code improvement [2], [4], [19], [22], there is a class of problems, each of which can be solved in essentially the same manner. These problems, called "global data flow analysis problems", involve the local collection of information which is distributed throughout the program. Some examples of global data flow analysis problems are "available expressions" [8], [26], "live variables" [14], "reaching definitions" [5], [6] and "very busy variables" [22]. There are several general algorithms to solve such problems.

The "interval" approach [2], [5]–[8], [14], [22] collects relevant information by partitioning the flow graph of the program into subgraphs called intervals, replacing each interval by a single node containing the local information within that interval, and continuing to define such interval partitions until the graph becomes a single node itself, at which time global information is propagated locally by reversing the partition process.

Another approach[1] [16], [26], [27] propagates information in a simple iterative manner until all the required information is collected; that is, until the process converges. It is this approach which will be described and analyzed in this study. We shall show that this second approach (with a suitable node ordering) is time competitive with the interval approach!

Prior to presenting the main result and the algorithm, we review part of the theory of reducible flow graphs.

* Received by the editors May 21, 1973, and in revised form August 15, 1974. This work was supported by the National Science Foundation under Grant GJ-1052.

† Department of Computer Science, University of Maryland, College Park, Maryland 20742.

‡ Department of Electrical Engineering, Princeton University, Princeton, New Jersey 08540.

[1] In 1961, V. A. Vyssotsky [27] implemented this kind of flow analysis (and presumably the simple iterative algorithm) in a Bell Laboratories 7090 FORTRAN II compiler—for strictly diagnostic purposes.

2. Background.

2.1 Basic definitions. A *flow graph* [2], [5], [10], e.g., is a triple $G = (N, E, s)$, where:

(a) N is a finite set of *nodes*. Let $n = |N|$.

(b) E is a subset of $N \times N$ called the *edges*. Let $e = |E|$. The edge (x, y) *enters* node y and *leaves* node x. We say that x is a *predecessor* of y, and y is a *successor* of x.

A *path* from x_1 to x_k is a sequence of nodes $(x_1, \cdots, x_k)$ such that (x_i, x_{i+1}) is in E for $1 \leqq i \leqq k-1$. The *path length* of $(x_1, \cdots, x_k)$ is $k-1$. If $x_1 = x_k$ and $k > 1$, the path is a *cycle*.

(c) Node s (in N) is the *initial node*. There is a path from s to every node.

Let G be flow graph and let h be a node of G. The *interval with header h* [2], [5], [10], [22], e.g., denoted by $I(h)$, is constructed as follows:

(a) First, set $I(h)$ to $\{h\}$.

(b) **while** m is a node not yet in $I(h)$ **and** $m \neq s$ **and** all predecessors of m are in $I(h)$ **do** add m to $I(h)$ **end**

Observe that although m in the while-loop above may not be well-defined, $I(h)$ does not depend on the order in which candidates for m are chosen. A candidate at one iteration of the while-loop will, if it is not chosen, still be a candidate at the next iteration.

There is a certain way to choose interval headers so that a flow graph is uniquely partitioned into disjoint intervals. This process takes $O(e)$ steps [2], [5].

If G is a flow graph, then the *derived flow graph* of G [2], [5], [10], [22], denoted by $I(G)$, is defined as follows:

(a) The nodes of $I(G)$ are the intervals of G determined from the unique interval partitioning of G.

(b) There is an edge from the node representing interval J to that representing K if there is any edge from a node in J to the header of K, and $J \neq K$. Note that there cannot be an edge from J entering a node other than the header of K.

(c) The initial node of $I(G)$ is $I(s)$.

The sequence $G = G_0, G_1, \cdots, G_k$ is called the *derived sequence* for G [2], [5], [10], [22] if $G_{i+1} = I(G_i)$ for $0 \leqq i \leqq k-1$, $G_{k-1} \neq G_k$, and $I(G_k) = G_k$. G_k is called the *limit flow graph* of G [2], [10].

A flow graph is called *reducible* (an rfg) [2], [5], [10], [22] if and only if its limit flow graph is the single node with no edge (henceforth called the *trivial flow graph*). Otherwise it is called *irreducible* or *nonreducible*.

Let G be a flow graph and let (x, x) be an edge of G. Transformation $T1$ [10] is removal of this edge.

Let y not be the initial node and have a single predecessor, x. Transformation $T2$ [10] is the replacement of x, y and (x, y) by a single node z. Predecessors of x become predecessors of z. Successors of x or y become successors of z. There is an edge (z, z) if and only if there was formerly an edge (y, x) or (x, x). (Whenever $T2$ is applied as described here, we say that *x consumes y*.)

There are two results from [10] which interest us here. First, if $T1$ and $T2$ are applied to a flow graph until no longer possible, a unique flow graph results, independent of the sequence of applications of $T1$ and $T2$ actually chosen. Second, a flow graph is reducible by intervals if and only if repeated application of $T1$ and $T2$ yields the trivial flow graph.

If x and y are two distinct nodes in a flow graph G, then x *dominates* y [2], [5], [11], [19], [20] if every path in G from the initial node to y contains x.

Let $G=(N, E, s)$ be a flow graph, let $N_1 \subseteq N$, let $E_1 \subseteq E$, and let h be in N_1. We say $R=(N_1, E_1, h)$ is a *region* of G with *header* h [11], [26] if in every path $(x_1, \cdots, x_k)$, where $x_1=s$ and x_k is in N_1, there is some $i \leqq k$ such that

(a) $x_i = h$; and

(b) $x_{i+1}, \cdots, x_k$ are in N_1; and

(c) $(x_i, x_{i+1}), (x_{i+1}, x_{i+2}), \cdots, (x_{k-1}, x_i)$ are in E_1.

That is, access to every node in the region is through the header only.

As we proceed to apply $T1$ and $T2$ to a flow graph, each edge of an intermediate graph represents a set of edges and each node represents a set of nodes and edges in a natural way [26].

We say that each node and edge in the original flow graph *represents* itself. If $T1$ is applied to node x with edge (x, x), then the resulting node *represents* what node x and edge (x, x) represented. If $T2$ is applied to x and y, with edge (x, y) eliminated, then the resulting node z *represents* what x, y, and (x, y) represented. In addition, if two edges (x, u) and (y, u) are replaced by a single edge (z, u), then (z, u) *represents* what (x, u) and (y, u) represented.

In [26] it was established that a node at any stage of the reduction of an rfg represents a region.

Since $T1$ and $T2$ may be applied to an rfg in different sequences, it becomes necessary to discuss specific sequences of application of $T1$ and $T2$. Informally, a "parse" of an rfg is a list of the reductions made ($T1$ or $T2$) and the regions to which they apply.

Formally, a *parse* π of an rfg $G=(N, E, s)$ [26] is a sequence with elements of form $(T1, u, v, S)$ or $(T2, u, v, w, S)$, where u, v and w are names of nodes and S is a set of edges. We define the parse of an rfg recursively as follows:

(a) The trivial flow graph has only the empty sequence as its parse.

(b) If G' (which may not be the original flow graph in a sequence of reductions) is reduced to G'' by an application of $T1$ to node u, and the resulting node is named v in G'', then $(T1, u, v, S)$ followed by a parse of G'' is a parse of G', where S is the set of edges of G represented by the edge (u, u) eliminated from G'.

(c) If G' is reduced to G'' by an application of $T2$ to nodes u and v (with u consuming v), and the resulting node is called w, then $(T2, u, v, w, S)$ followed by a parse of G'' is a parse of G', where S is the set of edges of G represented by the edge (u, v) in G'.

If G is an rfg and π is a parse of G, we call an edge of G a *backward edge wrt* (with respect to) π [26] if for some u and v this edge appears in set S of an entry $(T1, u, v, S)$ of π.

The next two results appear in [11].

LEMMA 1. *The backward edges of an* rfg *are independent of the parse.*

LEMMA 2. *Edge (x, y) is a backward edge of an* rfg *if and only if $x=y$ or y dominates x.*

2.2. Depth-first search. A *depth-first spanning tree* (DFST) of a flow graph G is a directed, rooted, ordered spanning tree of G grown by Algorithm A [23]. This algorithm also defines an ordering on the nodes of G which we call[2] rPOSTORDER (reverse POSTORDER).

ALGORITHM A: Computes rPOSTORDER for the nodes of a flow graph G.

Input. $G=(N, E, s)$ is a flow graph with n nodes numbered from 1 to n in an arbitrary manner. G is represented by successor (adjacency) lists. That is, for each node x let $S(x)=\{y \mid (x, y) \in E\}$.

Output. A numbering of the nodes from 1 to n (rPOSTORDER) indicating the reverse of the order in which each node was last visited.

Method.

Initially all nodes are marked "new".

There is a global integer variable i with initial value n.

There is a global integer array rPOSTORDER[1 : n].

The algorithm consists of a call to DFS(s), where DFS is the procedure defined below.

```
recursive procedure DFS($x$)
    mark $x$ "old"
    while $S(x)$ is not empty
        do
            select and delete a node $y$ from $S(x)$
            if $y$ is marked "new"
                then
                    /* add $(x, y)$ to DFST */
                    call DFS($y$)
                end
        end
    rPOSTORDER[$x$] $\leftarrow i$
    $i \leftarrow i-1$
    end DFS □
```

If (u, v) is an edge in a DFST, then u is the *parent* of v, and v is a *child* of u. The *ancestor* and *descendant* relations are the respective transitive closures of the parent and child relations.

Let $G=(N, E, s)$ be a flow graph and let $T=(N, E')$ be a DFST of G. The edges in $E-E'$ fall into three categories:

(a) Edges which go from ancestors to decendants we call *forward edges wrt T.*

(b) Edges which go from descendants to ancestors or from a node to itself we call *back edges wrt T.*

(c) Edges which go between nodes that are unrelated by the ancestor-descendant relation we call *cross edges wrt T.*

Since a DFST is an ordered tree, the children of each node z are ordered from left-to-right so that "younger" children of z are to the right of "older" children of z. We extend the notion of "to the right" in a DFST by saying that if x is to the

[2] The reason why the name "rPOSTORDER" was chosen here is worthy of mention. In the 1974 edition of [17], Knuth has renamed the binary tree traversals which were formerly (preorder, postorder, endorder) in the 1968 edition to (preorder, inorder, postorder), respectively. A DFST T of a flow graph is an ordered tree. The order in which a node was last visited while growing T is the POSTORDER (à la 1974 edition of [17]) traversal of T when T is considered as a general ordered tree.

right of y, then all of x's descendants are to the right of all of y's descendants. Thus, if (u, v) is a cross edge wrt a DFST, then u is to the right of v [23].

LEMMA 3 [11]. *The backward edges of an* rfg *G are exactly the back edges of a* DFST for G.

In view of Lemmas 1 and 3, we can safely confuse backward edges and back edges in an rfg.

3. A dominator algorithm.

Developing an idea of [7] leads, when combined with depth-first search, to a linear bit vector algorithm to find dominators.

Let T be a DFST of a flow graph G with n nodes. We consider two orderings of the nodes of G:

(a) rPOSTORDER—as defined in Algorithm A. According to this order, x precedes y if rPOSTORDER[x] < rPOSTORDER[y].

(b) POSTORDER—where POSTORDER[x] $= n+1-$ rPOSTORDER[x], for each node x. Here x precedes y if POSTORDER[x] < POSTORDER[y], i.e., if rPOSTORDER[y] < rPOSTORDER[x].

We define the dag of an rfg G [11], [26] to be G minus all of its back edges.

LEMMA 4. rPOSTORDER *topologically sorts the* dag *of an* rfg. (That is, the partial order defined by the dag of an rfg is a subset of the total order defined by rPOSTORDER.)

Proof. Let G be an rfg, let G' be the dag of G, and let T be any DFST of G. By Lemma 3, G' is G without the back edges of T. It suffices to show that if there is a path in G' from the initial node to node y which includes node x, with $x \neq y$, then rPOSTORDER[x] < rPOSTORDER[y].

Suppose, in contradiction, that there are two distinct nodes x and y such that there is a path in G' from the initial node to y which includes x, and rPOSTORDER[x] > rPOSTORDER[y]. Then POSTORDER[x] < POSTORDER[y]. That is, y is last visited after x is last visited while growing T.

Either y is an ancestor of x, or y is to the right of x in T. If y is an ancestor of x, then G' contains a cycle. This is impossible. Consequently, y is to the right of x. The path from x to y must go through a common ancestor of x and y [23], so there would again be a cycle in G'. $\square$

If i is a predecessor of j in an rfg, then either (i, j) is a back edge or it is not. If it is a back edge, then either j dominates i or $i = j$ (Lemma 2), and thus i cannot dominate j. If (i, j) is not a back edge of an rfg, then rPOSTORDER[i] < rPOSTORDER[j]. These properties of rPOSTORDER are exploited in Algorithm B.

ALGORITHM B: Computes a set DOM(x), the dominators of x, for each node x.

Input. $G = (N, E, s)$ is an rfg with n nodes represented by predecessor lists. That is, for each node y, let $P(y) = \{x \mid (x, y) \in E\}$. We assume that the nodes are numbered from 1 to n by rPOSTORDER, and we refer to each node by this number.[3]

[3] This assumption can be implemented in one of two ways. One way is to replace the statement "rPOSTORDER[x] ← i" by "rPOSTORDER[i] ← x" in Algorithm A, and then use rPOSTORDER appropriately. Another way is to just use a "bucket sort" (e.g., see [3]) to renumber nodes.

Output. Sets DOM(j), $1 \leqq j \leqq n$, where $i \in$ DOM(j) iff i dominates j.
Method.

```
DOM(1) ← $\varnothing$
for $j \leftarrow 2$ to $n$
    do
        DOM($j$) ← $\bigcap_{k \in P(j), k<j}$ ($\{k\} \cup$ DOM($k$))
    end □
```

THEOREM 1. *Algorithm* B *is correct. That is, after Algorithm* B *terminates, i is in* DOM(j) *if and only if i dominates j.*

Proof. Let G be an rfg. We proceed by induction on j.

Inductive hypothesis. After processing node j, i is in DOM(j) if and only if i dominates j.

Basis ($j = 1$). This is trivially true.

Induction step ($j > 1$). Assume the inductive hypothesis for all $k < j$, and consider the case for j.

If i dominates j, then surely i dominates every predecessor of j except i itself (if $i \in P(j)$). Thus i is placed in DOM(j) by Algorithm B.

Now, suppose i is in DOM(j), but i does not dominate j. Then there is a cycle-free path from the initial node to j which does not pass through i. Let k be the node on the path immediately before j. By Lemma 2, (k, j) cannot be a back edge, else j would dominate k or $k = j$, and the path would have a cycle. Thus (k, j) is not a back edge and rPOSTORDER[k] < rPOSTORDER[j]. As $i \neq k$ and i not dominate k, we have by the induction hypothesis that i is not in $\{k\} \cup$ DOM(k) and hence not in DOM(j). □

If the DOM sets are implemented by bit vectors, then Algorithm B requires $O(e)$ bit vector steps. This follows because in a flow graph with e edges at most e bit vector intersections are computed in the for-loop of Algorithm B. Also, the node ordering (rPOSTORDER) assumed as input can be computed in $O(e)$ steps [23].

In [2], Aho and Ullman present an $O(ne)$ step algorithm to compute dominators. Purdom and Moore's algorithm [21] has the same time bound.

Allen and Cocke [7] suggest breadth-first ordering of the nodes to compute dominators of an arbitrary graph, but their algorithm (to which Algorithm B is similar) may require more than one pass through the nodes.

Earnest et al. [9] present an algorithm which establishes an "interval ordering" (similar to rPOSTORDER) but takes more than $O(e)$ steps to compute.

Aho, Hopcroft and Ullman [1] give an $O(e \log e)$ step algorithm to find immediate (closest) dominators in an rfg. In [24], Tarjan presents an algorithm for determining immediate dominators in $O(e + n \log n)$ steps for an arbitrary graph.

4. The central result.

Following several lemmas, we establish the central result of this paper.

DEFINITION. The *loop-interconnectedness parameter* of an rfg G, which we shall denote by $d(G)$ or simply d, is the largest number of back edges found in any cycle-free path in G.

DEFINITION. Let G be a flow graph, let $I(G)$ be the derived flow graph of G, and let G' be G minus all of its self-loops, where a *self-loop* is an edge from a node to itself. We define the *length of the derived sequence* of G to be 0 if G' is the trivial flow graph, and otherwise it is that $k \neq 0$ such that

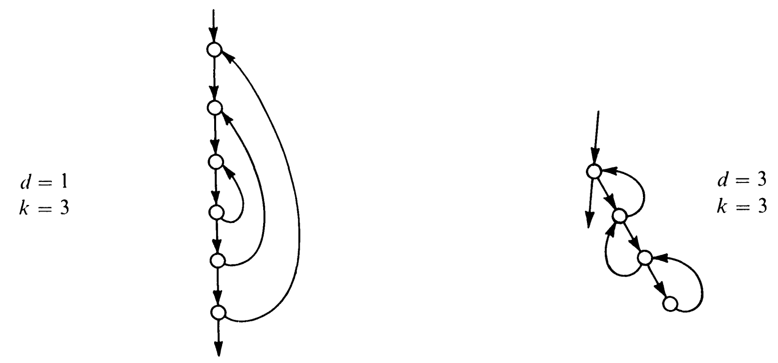
(a) $G_0 = G'$,

(b) $G_{i+1} = I(G_i)$, $i \geqq 0$,

(c) G_k is the limit flow graph of G, and

(d) $G_k \neq G_{k-1}$.

The length of the derived sequence of an rfg corresponds intuitively to the maximum "nesting depth" of the loops of a computer program. The "interconnectedness" of the loops of a computer program can be an entirely different thing. For example, Fig. 1 shows a possible configuration of three nested for-loops versus three nested while-loops with the corresponding k and d values.



(i) *Three for-loops* (ii) *Three while-loops*

FIG. 1. *"Maximum nesting depth of loops" (k) vs. "loop-interconnectedness" (d)*

LEMMA 5. *Let G be an* rfg *and let G' be G at some intermediate stage of its reduction by $T1$ and $T2$. If there is a path from node u to node v in G', then there exist nodes w and x in G such that w and x are respectively represented by nodes u and v in G' and there is a path from w to x in G.*

Proof. The lemma is an easy induction on the number of steps of π taken to reach G'. □

LEMMA 6. *Let G be an* rfg. *Nodes entered by back edges in G head intervals in G.*

Proof. The lemma is obvious for self-loops. So, let (m, h) be a back edge in G and suppose $m \neq h$. Thus h dominates m by Lemma 2. If h is the initial node, the lemma follows. Now consider the case where h is not the initial node.

Suppose h is in interval K but does not head K. Then by the method of constructing intervals, we must conclude that m is in K, since (m, h) is an edge.

As (m, h) is an edge, m must be added to K before h is. But then there is a path from the initial node to the header of K and thence to m which does not pass through h. This would contradict the assumption that h dominates m. □

LEMMA 7. *If u dominates v in an* rfg *G, u heads an interval in G, J is the interval containing v, and $I(u) \neq J$, then $I(u)$ dominates J in $I(G)$.*

Proof. Neither T1 nor T2 create any new paths between nodes. That is, edges of $I(G)$ are based precisely on edges of G. Thus, if $I(u)$ did not dominate J, then u would not dominate v. □

LEMMA 8. *Let d be the loop-interconnectedness parameter of an* rfg *G, let d' be the loop-interconnectedness parameter of $I(G)$, and suppose $G \neq I(G)$. Then $d' \geqq d-1$.*

Proof. Assume all the hypotheses, and let P be any circle-free path in G from p_1 to p_m containing d back edges. We write P as an ordered sequence of edges, $P=[(p_1, p_2), (p_2, p_3), \cdots, (p_{m-1}, p_m)]$, where the jth edge in P is (p_j, p_{j+1}). Let $[(x_1, r_1), (x_2, r_2), \cdots, (x_d, r_d)]$ be the subsequence of P consisting of all and only the back edges of P. (See Fig. 2.) Note that the r_i's are distinct, since P has no cycles.

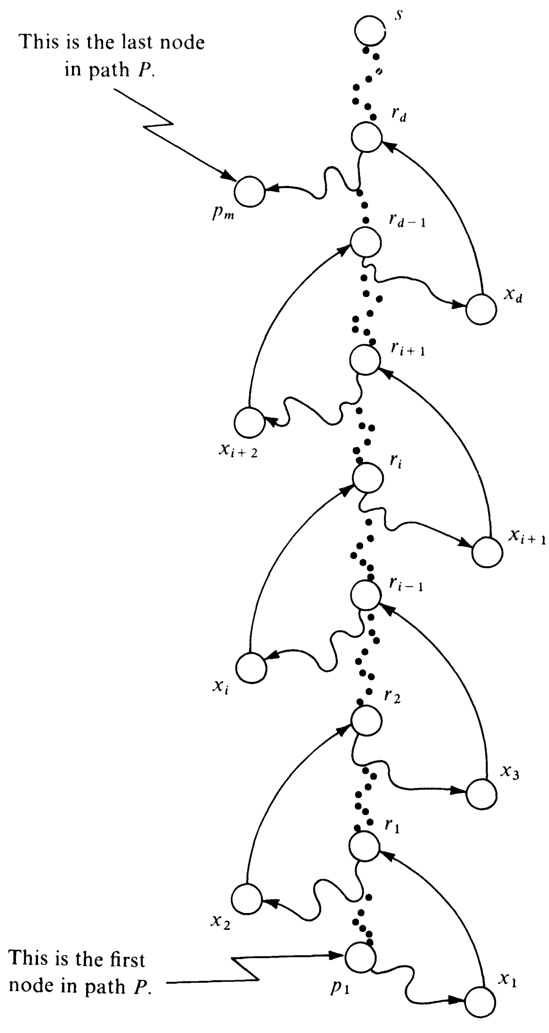


FIG. 2. *A cycle-free path P in an* rfg *from p_1 to p_m containing $d > 0$ back edges*

Let J_y denote the interval containing the node y. By Lemma 6, each r_i heads an interval J_{r_i}. Also, there is a path of nonback edges from each r_i to x_{i+1}, so x_{i+1} cannot be in the interval $J_{r_{i+1}}$. Thus we have $J_{x_{i+1}} \neq J_{r_{i+1}}$, and the edge $(J_{x_{i+1}}, J_{r_{i+1}})$ is in $I(G)$. Furthermore, we know that r_{i+1} dominates x_{i+1} by Lemma 2, and so by Lemma 7, $J_{r_{i+1}}$ dominates $J_{x_{i+1}}$ in $I(G)$. Now by Lemma 2 again, we may conclude that $(J_{x_{i+1}}, J_{r_{i+1}})$ is a back edge of $I(G)$. That is to say, each back edge of G, except possibly the first, is preserved in $I(G)$. If the first is also preserved, then $d' = d$. Otherwise, $d' = d - 1$. □

THEOREM 2 (Central Theorem). *If G is an* rfg *with loop-interconnectedness parameter d and derived sequence length k, then $k \geqq d$.*

Proof. Let $d = d_0, d_1, \cdots, d_k$ be the loop-interconnectedness parameters of the members of the derived sequence. By Lemma 8, we know that $d_{i-1} \leqq d_i + 1$ for $1 \leqq i \leqq k$. Also, $d_k = 0$, since the last graph in the derived sequence is trivial. An elementary induction on i shows that $d_{k-i} \leqq i$. Thus $d_0 \leqq k$, as was to be proved. □

We shall postpone explaining the significance of Theorem 2 until after the next two sections. In these sections we present and then analyze a simple, iterative, bit propagation algorithm for global data flow analysis problems.

5. Solution of two global data flow analysis problems.

5.1. Available expressions (as in, e.g., [8], [26]). An expression such as $A + B$ is *available* at point p in a flow graph if every sequence of branches which the program may take to p causes $A+B$ to have been computed after the last computation of A or B. If we can determine the set of available expressions at entrance to the nodes of a flow graph, then we know which expressions have already been computed prior to each node. Thus we may be able to eliminate the redundant computation of some expressions within each node.

Let $\mathscr{E}$ be the set of *expressions* computed in a flow graph $G = (N, E, s)$.

Let $\mathscr{K} : N \to 2^{\mathscr{E}}$. We interpret $\mathscr{K}[x]$ as the set of expressions which are *killed* in node x. Informally, expression $A\ \theta\ B$ is killed if either A or B is assigned within node x. (The symbol θ indicates a generic binary operator.)

Let $\mathscr{G} : N \to 2^{\mathscr{E}}$. If an expression $r = A\ \theta\ B$ is in $\mathscr{G}[x]$, then we imagine that r is *generated* within node x, and that neither A nor B is subsequently assigned within x.

Let AEIN[x] and AEOUT[x], for each node x, be, respectively, the set of expressions available at entrance to and at exit from node x.

The fundamental relationships which enable us to compute AEIN[x] for each node x are:

AE1. AEIN[s] $= \varnothing$.

AE2. For $x \neq s$, AEIN[x] is the intersection of AEOUT[y] over all predecessors y of x.

AE3. AEOUT[x] $=$ (AEIN[x] $-$ $\mathscr{K}[x]$) $\cup$ $\mathscr{G}[x]$ for each node x.

AE4. Since AE1–AE3 do not necessarily have a unique solution for AEIN[x], we want the largest solution.

The algorithm which follows is a bit vector algorithm and similar to those in [16], [26] and [27], except for the node ordering. We distinguish between sets and bits vectors by using AEIN for sets and AEin for bit vectors. Algorithm C is essentially AE3 "plugged into" AE2.

ALGORITHM C: Computes bit vectors AEin[x] for each node x.

Input. $G=(N, E, s)$ is a flow graph with n nodes represented by predecessor lists. That is, for each node y let $P(y)=\{x|(x, y)\in E\}$. The nodes are numbered from 1 to n by rPOSTORDER. We refer to each node by its rPOSTORDER number.

Bit vectors NOTKILL[j] and GEN[j], $1\leqq j\leqq n$, are input where the ith bit of NOTKILL[j] (resp. GEN[j]) is 1 if and only if the ith expression is not in $\mathcal{K}[j]$ (resp. in $\mathcal{G}[j]$). All bit vectors have length p, where p is the number of expressions.

Output. Bit vectors AEin[j], $1\leqq j\leqq n$.

Method.

```
AEin[1] ← all 0's
**for** $j\leftarrow 2$ **to** $n$ **do** AEin[$j$] ← all 1's **end**
change ← **true**
**while** change
    **do**
        change ← **false**
        **for** $j\leftarrow 2$ **to** $n$   /*rPOSTORDER */
            **do**
                previous ← AEin[$j$]
                AEin[$j$] ← $\bigwedge_{k\in P(j)}$ ((AEin[$k$] $\wedge$ NOTKILL [$k$]) $\vee$ GEN[$k$])[4]
                **if** previous $\neq$ AEin[$j$] **then** change ← **true** **end**
            **end**
    **end** □
```

5.2. Live variables (as in, e.g., [14]). A path in a flow graph is called *definition-clear* with respect to a variable v if there is no definition of v (by assignment or input) on that path. A variable v is *live* at a point p in a flow graph if there is a definition-clear path for v from p to a use of v. That is, v is live if its current value might be used before v is redefined. Having determined the set of live variables at exit from each node in a flow graph, we can use this information for register allocation—we can determine when a value should be kept in a register because of a subsequent use.

Let $\mathcal{V}$ be the set of *variables* occurring in a flow graph $G=(N, E, s)$.

Let $\mathcal{C}: N\rightarrow 2^{\mathcal{V}}$. $\mathcal{C}[x]$, the *clear* of x, is the set of variables which are not defined in node x.

Let $\mathcal{U}: N\rightarrow 2^{\mathcal{V}}$. $\mathcal{U}[x]$ is the set of variables which have *exposed uses* in node x, i.e., those variables with a definition-clear path from the entry of node x to a use within node x.

Let LVOUT[x] and LVIN[x], for each node x, be the set of variables live at exit from and on entrance to node x.

The fundamental relationships which enable us to compute LVOUT[x] for each node x are:

LV1. If x has no successors, then LVOUT[x] $=\emptyset$.

[4] Here, the symbols $\bigwedge$, $\bigvee$ and $\neg$ stand for the AND (bitwise product), OR (bitwise sum) and NOT (bitwise complement) functions, respectively. Note also that this expression can be evaluated for k once on each pass, after the new value of AEin[k] is computed, and then used subsequently without recomputation.

LV2. Otherwise, LVOUT[x] is the union of LVIN[y] over all successors y of x.

LV3. LVIN[x] = (LVOUT[x] $\cap$ $\mathscr{C}[x]$) $\cup$ $\mathscr{U}[x]$ for each node x.

LV4. Since LV1–LV3 do not necessarily have a unique solution for LVOUT[x], we want the smallest such solution.

Let LVout be the bit vector for set LVOUT.

Algorithm D which follows is just an iterative version of LV3 "plugged into" LV2, with a suitable initialization for LVOUT, and with a suitable node ordering.

Algorithms C and D are not exactly the same. Algorithm D propagates information opposite to the direction of control flow, while C propagates it in the same direction. In Algorithm D we visit the nodes in POSTORDER rather than the reverse. Also, here we are computing OUT's rather than IN's.

ALGORITHM D: Computes bit vectors LVout[x] for each node x.

Input. $G = (N, E, s)$ is a flow graph with n nodes represented by successor lists. That is, for each node x let $S(x) = \{y \mid (x, y) \in E\}$. The nodes are numbered from 1 to n by rPOSTORDER. We refer to each node by its rPOSTORDER number. It is assumed that each node has at least one successor. A node without any successors can be combined with its predecessors first, with a resulting gain in efficiency.

Bit vectors CLEAR[j] and XUSE[j], $1 \leqq j \leqq n$, are input where the ith bit of CLEAR[j] (resp. XUSE[j]) is 1 if and only if the ith variable is in $\mathscr{C}[j]$ (resp. $\mathscr{U}[j]$). All bit vectors have length q, where q is the number of variables.

Output. Bit vectors LVout[j], $1 \leqq j \leqq n$.

Method.

```
for $j \leftarrow 1$ to $n$ do LVout[$j$] $\leftarrow$ all 0's end
change $\leftarrow$ true
while change
    do
        change $\leftarrow$ false
        for $j \leftarrow n$ to 1 by $-1$    /*POSTORDER */
            do
                previous $\leftarrow$ LVout[$j$]
                LVout[$j$] $\leftarrow \bigvee_{k \in S(j)}$ ((LVout[$k$] $\wedge$ CLEAR[$k$]) $\vee$ XUSE[$k$])[5]
                if previous $\neq$ LVout[$j$] then change $\leftarrow$ true end
            end
    end □
```

6. Analysis. The termination and correctness of Algorithms C and D follow directly from [16] and [26]. We focus on the complexity.

THEOREM 3. *The while-loop of Algorithm* C *is executed at most* $d + 2$ *times for an* rfg G.

Proof. A 0 propagates from its point of "origin"—a "kill" or the initial node—to the place where it is needed in $d + 1$ iterations if it must propagate along

[5] Analogous to Algorithm C, note that this expression can be evaluated for k once on each pass, after the new value of LVout[k] is computed, and then used subsequently without recomputation. Algorithms C and D are only presented without this feature so that each algorithm will be more readable.

a path P of d back edges. It takes one iteration for a 0 to arrive at the tail of the first back edge of P. This follows since the numbers along the path must be in increasing sequence according to rPOSTORDER. After this point, it takes one iteration for a 0 to climb up each back edge in P to the tail of the next back edge, by the same argument. Hence we need at most $d+1$ iterations to propagate information, plus one more iteration to detect that there are no further changes. $\square$

THEOREM 4. *The while-loop of Algorithm* D *is executed at most $d+2$ times for an* rfg G.

Proof. A 1 indicating a use propagates backward along a cycle-free path to a given point in $d+1$ iterations if there are d back edges in the path from the point to the use. It takes one iteration for a 1 to reach the head of the dth back edge in such a path. The proof is then analogous to that of Theorem 3. $\square$

THEOREM 5. *If we ignore initialization, Algorithm* C (*or Algorithm* D) *takes at most $(d+2)(e+n)$ bit vector steps, where a bit vector step is the logical $\bigwedge$ or $\bigvee$ of bit vectors.*

Proof. Since Algorithms C and D are almost identical, we mostly refer to Algorithm D below. We assume that Algorithm D has been rewritten so that the expression ((LVout[k] $\wedge$ CLEAR[k]) $\vee$ XUSE[k]) is evaluated for k once on each pass. Also, we assume that each node has at least one successor, since any node without successors can be easily combined with its predecessors first, with a resulting gain in efficiency.

For each iteration of the while-loop of Algorithm D, $2n$ bit vector steps are used to evaluate ((LVout[k] $\wedge$ CLEAR[k]) $\vee$ XUSE[k]) for all k, that is, n nodes at 2 bit vector steps per node. (In Algorithm C this process uses $2(n-1)$ bit vector steps.) At most $e-n$ bit vector steps are aggregately used to perform $\bigvee$ over all successors of each node, because each node with m successors requires $m-1$ bit vector steps, and e successors less one for each of n nodes yields $e-n$. Since the while-loop is executed at most $d+2$ times (Theorem 3 and 4), we have at most $(d+2)(e-n+2n)=(d+2)(e+n)$ bit vector steps. $\square$

7. Discussion and conclusions.

The given iterative bit propagation algorithm for global data flow analysis problems is conceptually simple, it is very easy to program, and the length of the resulting program is quite short. It takes $O(e)$ steps to compute the rPOSTORDER numbering of the nodes, and at most $(d+2)(e+n)$ bit vector steps to propagate the required information in a reducible graph. Moreover, the bit propagation algorithm, with no modification whatsoever, works on nonreducible graphs.

The interval analysis algorithm for global data flow analysis problems is conceptually complicated, it is quite difficult to program, and the length of the resulting program is very long. Nonreducible graphs present an additional time and programming complication.

Kennedy [15] has done some detailed comparisons of our Algorithm D and his algorithm for live variable analysis [14]. While [15] bounds the number of bit vector steps for the algorithm of [14] more carefully than for our algorithm, it is basically correct and indicates that on practical problems our algorithm will run in time comparable to the interval analysis algorithm but may take twice as long as some flow graphs.

We can obtain a *lower* bound of $2(e+n)$ bit vector steps for any of the published interval analysis algorithms [2], [5], [6], [8], [14], [22]. In comparison, the *upper* bound for our algorithm in Theorem 5 is $(d+2)(e+n)$. Thus the ratio of our time to that of interval analysis is at most $d/2+1$, and may be less. Knuth [18] in a study of 50 FORTRAN programs found all to be reducible with derived sequence length $k \leqq 6$ and an average k of 2.75. Since $d \leqq k$ is Theorem 2, we can expect an average value of $d/2+1$ to be no more than 2.4. Thus, we may claim that on the average, propagation algorithms take no more than 2.4 times that of the corresponding interval analysis algorithms.

The above analysis has, however, been overly conservative. There are a number of other factors which tend both to argue that (a) the ratio 2.4 is higher than it should be, and (b) propagation algorithms have other virtues which make them preferable to the interval approach.

1. Propagation algorithms handle nonreducible flow graphs with no added effort, in fact, without even noting they are nonreducible.
2. The coding for a propagation algorithm is trivial in comparison with the coding necessary to implement an interval analysis algorithm.
3. The interval algorithms require additional work finding and manipulating intervals. This work requires no bit vector steps and so was not reflected in the calculation. In comparison, the cost of a depth-first search, the only significant part of our algorithm not reflected in the count of bit vector steps, is negligible.
4. In the above analysis, we assumed $k = d$. In fact, Fig. 1 indicates that $d < k$ is quite possible, especially in FORTRAN programs whose jumps are all caused by DO statements.
5. Without increasing the difficulty of coding our algorithm beyond that of the interval algorithms, we could check whether the predecessors of a node n had had their values changed since the last time we visited n. If not, no calculation at n is necessary for the current pass.

When all the above issues are taken into account, we believe a strong case can be made for using bit propagation algorithms for data flow analysis.

Acknowledgment. We appreciate the help of a referee who simplified our proofs of Lemma 8 and Theorem 2.

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *On finding lowest common ancestors in trees*, Proc. 5th Ann. ACM Symp. on Theory of Computing, Austin, Tex., May 1973, pp. 253–265.

[2] A. V. AHO AND J. D. ULLMAN, *The Theory of Parsing, Translation and Compiling: Vol. II—Compiling*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

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

[4] F. E. ALLEN, *Program optimization, Annual Review Automatic Programming*, vol. 5, Pergamon Press, New York, 1969.

[5] ———, *Control flow analysis*, SIGPLAN Notices, (1970), 5, pp. 1–19.

[6] ———, *A Basis for Program Optimization*, Proc. IFIP Congr. 71, North-Holland, Amsterdam, 1971.

[7] F. E. ALLEN AND J. COCKE, *Graph theoretic constructs for program control flow analysis*, IBM Res. Rep. RC 3923, IBM T. J. Watson Res. Center, Yorktown Heights, N.Y., 1972.

[8] J. COCKE, *Global common subexpression elimination*, SIGPLAN Notices, 5 (1970), pp. 20–24.

[9] C. P. EARNEST, K. G. BALKE AND J. ANDERSON, *Analysis of graphs by ordering of nodes*, J. Assoc. Comput. Mach., 19(1972), pp. 23–42.

[10] M. S. HECHT AND J. D. ULLMAN, *Flow graph reducibility*, this Journal, 1 (1972), pp. 188–202.

[11] ———, *Characterizations of reducible flow graphs*, J. Assoc. Comput. Mach., 21 (1974), pp. 367–375.

[12] M. S. HECHT, *Topological sorting and flow graphs*, Proc. IFIP Congr. 74, August 1974, pp. 494–499.

[13] J. E. HOPCROFT AND J. D. ULLMAN, *An n* log *n algorithm for detecting reducible graphs*, Proc. 6th Ann. Princeton Conf. on Information Sciences and Systems, March 1972, pp. 119–122.

[14] K. KENNEDY, *A global flow analysis algorithm*, Internat. J. Comput. Math., 3 (1971), pp. 5–15.

[15] ———, *A comparison of global data flow analysis programs*, Rice Tech. Rep. 476-093-1, Rice Univ., Houston, Tex., 1974.

[16] G. A. KILDALL, *A unified approach to global program optimization*, Conf. Rec. of ACM Symp. on Principles of Programming Languages, Boston, Oct. 1973, pp. 194–206.

[17] D. E. KNUTH, *The Art of Computer Programming*: *Vol. I—Fundamental Algorithms*, 2nd ed., Addison-Wesley, Reading, Mass., 1974.

[18] ———, *An Empirical Study of FORTRAN Programs*, Software Practice and Experience, April (1971), pp. 105–134.

[19] E. S. LOWRY AND C. W. MEDLOCK, *Object code optimization*, Comm. ACM, 12 (1969), pp. 13–22.

[20] R. T. PROSSER, *Applications of Boolean matrices to the analysis of flow diagrams*, Proc. Eastern Joint Computer Conf., Spartan Books, New York, 1959, pp. 133–138.

[21] P. W. PURDOM AND E. F. MOORE, *Immediate predominators in a directed graph*, Comm. ACM, 15 (1972), pp. 777–778.

[22] M. SCHAEFER, *A Mathematical Theory of Global Program Optimization*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[23] R. E. TARJAN, *Depth-first search and linear graph algorithms*, this Journal, 1 (1972), pp. 146–160.

[24] ———, *Finding dominators in directed graphs*, Proc. 7th Ann. Princeton Conf. on Information Sciences and Systems, March 1973.

[25] ———, *Testing flow graph reducibility*, Proc. 5th Ann. ACM Symp. on Theory of Computing, Austin, Tex., May 1973, pp. 96–107.

[26] J. D. ULLMAN, *Fast algorithms for the elimination of common subexpressions*, Acta Informatica, 2 (1973), pp. 191–213.

[27] V. A. VYSSOTSKY, Private communication, June 7, 1973.

SIAM J. COMPUT.
Vol. 4, No. 4, December 1975

EVALUATING POLYNOMIALS AT FIXED SETS OF POINTS*

A. V. AHO†, K. STEIGLITZ‡ AND J. D. ULLMAN‡

Abstract. We investigate the evaluation of an $(n-1)$st degree polynomial at a sequence of n points. It is shown that such an evaluation reduces directly to a simple convolution if and only if the sequence of points is of the form $b, ba, ba^2, \cdots, ba^{n-1}$ for complex numbers a and b (the so-called "chirp transform"). By more complex reductions we develop an $O(n \log n)$ evaluation algorithm for sequences of points of the form

$$b+c+d, \quad ba^2+ca+d, \quad ba^4+ca^2+d, \cdots$$

for complex numbers a, b, c and d. Finally we show that the evaluation of an $(n-1)$st-degree polynomial and all its derivatives at a single point requires at most $O(n \log n)$ steps.

Key words. polynomial evaluation, derivative, fast Fourier transform, chirp transform, straight-line code, computational complexity, Taylor series

1. Introduction. We consider the following problem. Given an infinite sequence of points $a_0, a_1, a_2, \cdots$, how long, as a function of n, does it take to evaluate an arbitrary dense univariate polynomial of degree $n-1$ at the first n of these points?

Our model of computation is the *straight-line code* model. For each n, we assume that the computation is performed by a sequence of assignment statements of the form $A \leftarrow B\ \theta\ C$, where A, B and C are variable names, constants, or the names of coefficients of the polynomial (input variables), and θ is one of the operators $+$, $-$, $\times$ or $/$. In addition, n variables are designated as output variables, and after execution of the sequence of assignment statements, these variables are to hold the values of the polynomial at the n points. Such a sequence of assignment statements will be called an *algorithm*, and the *complexity* of an algorithm is the number of assignment statements therein.

A straight-line algorithm that evaluates any $(n-1)$st-degree polynomial at n points is said to be an *evaluation algorithm*. The inputs to the algorithm are the coefficients of the polynomial. A sequence of points $a_0, a_1, a_2, \cdots$ is said to be of *complexity* at most $T(n)$, if for all positive n there is a straight-line algorithm with at most $T(n)$ statements that evaluates any $(n-1)$st-degree polynomial at the first n points of the sequence.

It is known that an arbitrary sequence of points is of complexity at most $O(n \log^2 n)$([1] modified by the treatment in [2], [3], [4]). Certain sequences of points, however, are of complexity at most $O(n \log n)$. The best known such sequence of points is the "chirp transform" [5], [6], a generalization of the "fast

* Received by the editors May 13, 1974, and in revised form November 15, 1974. The work of the second author was supported in part by the U.S. Army Research Office, Durham, under Contract DAHCO4-09-C-0012, and by The National Science Foundation under Grant GJ-965. The work of the third author was supported in part by The National Science Foundation under Grant GJ-35570.

† Bell Laboratories, Murray Hill, New Jersey 07974.

‡ School of Engineering and Applied Science, Princeton University, Princeton, New Jersey 08540.

Fourier transform" (FFT) [7]. In the chirp transform, the sequence of points z^0, z^1, $z^2, \cdots$ is used, where z is any complex number. A polynomial $\sum_{i=0}^{n-1} b_i x^i$ can be evaluated at the points $z^0, z^1, \cdots, z^{n-1}$ in $O(n \log n)$ time as follows. To compute

$$(1) \qquad c_j = \sum_{i=0}^{n-1} b_i z^{ij} \quad \text{for } 0 \leqq j \leqq n-1,$$

we rewrite (1) as:

$$(2) \qquad c_j = \sum_{i=0}^{n-1} b_i \, z^{-(j-i)^2/2} \, z^{i^2/2} \, z^{j^2/2} \quad \text{for } 0 \leqq j \leqq n-1.$$

Equation (2) is a convolution which can be evaluated in $O(n \log n)$ steps with the FFT.

Our goal is to increase the set of sequences which are known to have $O(n \log n)$ evaluation algorithms. While an interesting problem in its own right, digital signal processing provides the practical motivation for considering the evaluation of high-degree dense polynomials at more general sequences of points than that of the chirp transform. See [8] and [9], for example.

The approach we use is to consider *classes* of sequences. The $T(n)$ class of sequences is the set of all sequences which have an $O(T(n))$ evaluation algorithm. Thus for each complex number z, the sequence $z^0, z^1, \cdots$ is in the $n \log n$ class, and every sequence is in the $n \log^2 n$ class. We shall ultimately consider closure properties of classes, but first we shall consider to what extent the chirp transformation generalizes directly.

It should be noted that our definition of "class" smears the boundary between sequences of distinct degrees of difficulty. For example, it is by no means clear that there is speedup by constant factors in the straight-line code model, as there is for Turing machines [10]. It is likely that there are sequences that can be evaluated in time $(1+\varepsilon)T(n)$ but not in time $T(n)$ for any reasonable $T(n)$ and $\varepsilon > 0$. Nevertheless, the definition is a useful one to make, and we shall use it to advantage subsequently.

2. Uniqueness of the chirp transform. We have seen that the chirp transform reduces to a convolution of the form $\alpha(j) \sum_{i=0}^{n-1} b_i \beta(i) \gamma(j-i)$, where the b_i's are the coefficients of the polynomial to be evaluated, and α, β and γ are independent of the b_i's. It is interesting to consider what other transformations, if any, can be reduced to a convolution of this form. The following theorem shows that except for a constant factor, the chirp transform is the most we can obtain by this technique.

THEOREM 1. *For $n \geqq 3$, suppose that the evaluation of an arbitrary $(n-1)$-st-degree polynomial $\sum_{i=0}^{n-1} b_i x^i$ at the points $a_0, a_1, \cdots, a_{n-1}$ can be expressed as a convolution of the form*

$$(3) \qquad \sum_{i=0}^{n-1} b_i a_j^i = \alpha(j) \sum_{i=0}^{n-1} b_i \beta(i) \gamma(j-i) \quad \textit{for } 0 \leqq j \leqq n-1$$

for some functions α, β and γ independent of the b's. Then $a_j = z_1 (z_2)^j$ for some complex numbers z_1 and z_2.

Proof. Since the b_i's are arbitrary, the left and right sides of (3) must agree term by term. Thus

$$a_j^i = \alpha(j)\beta(i)\gamma(j-i) \tag{4}$$

for all i and j between 0 and $n-1$.

Suppose temporarily that $a_j \neq 0$ for any j. Taking logarithms of (4), we obtain

$$i \log a_j = \log \alpha(j) + \log \beta(i) + \log \gamma(j-i) \quad \text{for } 0 \leqq i, j \leqq n-1. \tag{5}$$

Taking finite differences of (5) with respect to $i+1$ and j gives

$$\log a_j = \log \frac{\beta(i+1)}{\beta(i)} + \log \frac{\gamma(j-i-1)}{\gamma(j-i)} \tag{6}$$

for $0 \leqq i < n-1$ and $0 \leqq j \leqq n-1$, and

$$i \log \frac{a_j}{a_{j-1}} = \log \frac{\alpha(j)}{\alpha(j-1)} + \log \frac{\gamma(j-i)}{\gamma(j-i-1)} \tag{7}$$

for $0 \leqq i \leqq n-1$ and $0 < j \leqq n-1$. Summing (6) and (7), we obtain

$$\log a_j + i \log \frac{a_j}{a_{j-1}} = \log \frac{\alpha(j)}{\alpha(j-1)} + \log \frac{\beta(i+1)}{\beta(i)} \tag{8}$$

for $0 \leqq i < n-1$ and $0 < j \leqq n-1$. Taking (8) at $i = 1$ and subtracting from it (8) at $i = 0$, we obtain

$$\log \frac{a_j}{a_{j-1}} = \log \frac{\beta(0)\beta(2)}{\beta(1)^2} \quad \text{for } 0 < j \leqq n-1. \tag{9}$$

(Note that $n \geqq 3$ is necessary for this step to make sense.) It follows from (9) that

$$\frac{a_j}{a_{j-1}} = \frac{\beta(0)\beta(2)}{\beta(1)^2} \quad \text{for } 0 < j \leqq n-1,$$

and therefore

$$a_j = a_0 \left[\frac{\beta(0)\beta(2)}{\beta(1)^2}\right]^j \quad \text{for } 0 \leqq j \leqq n-1.$$

Let $z_1 = a_0$ and $z_2 = \beta(0)\beta(2)/\beta(1)^2$ to prove the theorem.

The detail which remains is what happens when $a_j = 0$ for some j, say j_0. Referring back to (3), we see that the left side evaluated at a_{j_0} is just b_0. Thus, in place of equation (4) with $j = j_0$, we have

$$\alpha(j_0)\beta(i)\gamma(j_0 - i) = 0 \quad \text{for } 1 \leqq i \leqq n-1, \tag{10}$$

$$\alpha(j_0)\beta(0)\gamma(j_0) = 1. \tag{11}$$

From (11) we see $\alpha(j_0) \neq 0$. Thus by (10) we have $\beta(i)\gamma(j_0 - i) = 0$ for $1 \leqq i \leqq n-1$.

The theorem is easily seen to hold if $a_j = 0$ for all j. Thus assume the contrary. If $\beta(i) = 0$ for any i, then the right side of (3) is independent of b_i. This is impossible, since we assumed not all a_j's are zero. We may conclude that $\gamma(j_0 - i) = 0$ for $1 \leqq i \leqq n-1$. It is sufficient to consider two cases.

Case 1. $a_{j_0}=0$ and $a_{j_0+1}\neq 0$. Then, since $\gamma(j_0-1)=0$, it follows that $\gamma((j_0+1)-2)=0$. Thus, for $j=j_0+1$, the right side of (3) is independent of b_2. Since $n\geqq 3$ is assumed, we have a contradiction.

Case 2. $a_{j_0}=0$ and $a_{j_0-1}\neq 0$. Then the right side of (3) is independent of b_0 for $j=j_0-1$, again yielding a contradiction. □

Note that Theorem 1 is trivially true for $n=1$, but there is a counterexample with $a_0=0$ for the case $n=2$.

3. Closure properties of sequence classes. We see from Theorem 1 that the chirp transform is essentially all that we can obtain by a simple convolution. More extensive algebraic manipulations, however, do yield larger classes of sequences for the $n \log n$ class. Before looking at these more complex operations, we derive several "closure properties" that hold for the various sequence classes.

LEMMA 1. *If sequence $a_0, a_1, a_2, \cdots$ is in class $T(n)\geqq n$, and c is any complex number, then sequence $ca_0, ca_1, ca_2, \cdots$ is also in class $T(n)$.*

Proof. Let $\mathbf{A}_n$ be an algorithm that takes as input the coefficients $b_0, b_1, \cdots, b_{n-1}$ and produces $d_j=\sum_{i=0}^{n-1} b_i a_j^i$ for $0\leqq j\leqq n-1$ as outputs. Then we may construct algorithm $\mathbf{B}_n$ to compute $e_j=\sum_{i=0}^{n-1} b_i(ca_j)^i$ for $0\leqq j\leqq n-1$. $\mathbf{B}_n$ works as follows:

1. In $n-2$ steps, compute $f_i=c^i$ for $2\leqq i\leqq n-1$. Let $f_0=1$ and $f_1=c$.
2. In $n-1$ steps, compute $g_i=b_if_i$ for $0\leqq i\leqq n-1$.
3. Apply algorithm $\mathbf{A}_n$ to coefficients $g_0, g_1, \cdots, g_{n-1}$.
4. The outputs of $\mathbf{A}_n$ are the desired outputs for $\mathbf{B}_n$.

It should be clear that $\mathbf{B}_n$ works, and that the length of the straight-line algorithm $\mathbf{B}_n$ is $2n-3$ plus the length of $\mathbf{A}_n$. Thus, since $T(n)\geqq n$, we know that the length of $\mathbf{B}_n$ does not exceed $3T(n)$. □

LEMMA 2. *If $a_0, a_1, a_2, \cdots$ is in the $T(n)\geqq n \log n$ class, and k is any positive integer, then $a_0^k, a_1^k, a_2^k, \cdots$ is also in the $T(n)$ class.*

Proof. The proof is again straightforward. Given the coefficients $b_0, b_1, \cdots, b_{n-1}$, we construct a new sequence of coefficients of length kn by inserting $k-1$ 0's after each of the b's. Then we break this sequence into k subsequences of length n. We let the subsequences represent k polynomials $p_0, p_1, \cdots, p_{k-1}$. We now have

$$\sum_{i=0}^{n-1} b_i a_j^{ki}=\sum_{r=0}^{k-1} a_j^{rn}p_r(a_j) \tag{12}$$

for $0\leqq j\leqq n-1$.

We use the assumed $O(T(n))$ algorithm k times to evaluate the p_r's at $a_0, a_1, \cdots, a_{n-1}$. The terms a_j^{rn} for $0\leqq j\leqq n-1$ and $0\leqq r\leqq k-1$ can be evaluated in $O(kn+n\log n)$ steps, and the right side of (12) can be evaluated in $O(kn)$ steps given the a_j^{rn}'s and $p_r(a_j)$'s. Thus the entire algorithm requires $O(kT(n)) + O(kn+n\log n)$ steps. Since k is a constant and $T(n)\geqq n\log n$, this function is of the order of $T(n)$. □

LEMMA 3. *If $a_0, a_1, \cdots$ is in the $T(n)\geqq n \log n$ class and c is any complex number, then $a_0+c, a_1+c, \cdots$ is also in the $T(n)$ class.*

Proof. We wish to compute $\sum_{i=0}^{n-1} b_i(a_j+c)^i$ for $0 \leqq j \leqq n-1$. This can be done in the following manner:

$$\begin{aligned}\sum_{i=0}^{n-1} b_i(a_j+c)^i &= \sum_{i=0}^{n-1} b_i \sum_{r=0}^{i} \binom{i}{r} a_j^r c^{i-r} = \sum_{r=0}^{n-1}\sum_{i=r}^{n-1} b_i \binom{i}{r} a_j^r c^{i-r} \\ &= \sum_{r=0}^{n-1} \frac{a_j^r}{r!} \sum_{i=r}^{n-1} \frac{b_i i! c^{i-r}}{(i-r)!}.\end{aligned} \tag{13}$$

If we define $f(x) = b_x x!$ for $0 \leqq x \leqq n-1$ and

$$g(x) = \begin{cases} c^{-x}/(-x)! & \text{for } -(n-1) \leqq x \leqq 0, \\ 0 & \text{for } 1 \leqq x \leqq n-1, \end{cases}$$

then we can allow the inner summation of (13) to range from 0 to $n-1$ and write (13) as:

$$\sum_{i=0}^{n-1} b_i(a_j+c)^i = \sum_{r=0}^{n-1} \frac{a_j^r}{r!} \sum_{i=0}^{n-1} f(i)g(r-i). \tag{14}$$

It is easy to see how to compute the necessary values of $f(x)$ and $g(x)$ in $O(n)$ steps. Then the inner summation of (14) can be evaluated for $0 \leqq r \leqq n-1$ in $O(n \log n)$ steps, since it is a convolution. In $O(n)$ more steps, we can compute $r!$ for $0 \leqq r \leqq n-1$. Thus we can compute $d_r = (1/r!) \sum_{i=0}^{n-1} f(i)g(r-i)$ for $0 \leqq r \leqq n-1$ in $O(n \log n)$ steps.

Thus the problem of evaluating $\sum_{i=0}^{n-1} b_i x^i$ at points $a_0+c, a_1+c, \cdots$ has been reduced in $O(n \log n)$ steps to the problem of evaluating $\sum_{i=0}^{n-1} d_i x^i$ at points $a_0, a_1, \cdots$. The latter evaluation can be done in $T(n)$ steps. Since $T(n) \geqq n \log n$, the desired evaluation takes $O(T(n))$ steps. $\square$

We may now combine the three lemmas to obtain additional closure properties of sequence classes.

THEOREM 2. *If $a_0, a_1, \cdots$ is in class $T(n) \geqq n \log n$, b and c are complex numbers, and k is any positive integer, then $ba_0^k+c, ba_1^k+c, \cdots$ is in class $T(n)$.*

Proof. By Lemma 2, sequence $a_0^k, a_1^k, \cdots$ is in class $T(n)$. By Lemma 1, so is sequence $ba_0^k, ba_1^k, \cdots$, and by Lemma 3 we have the theorem. $\square$

THEOREM 3. *If sequence $a_0, a_1, \cdots$ is in class $T(n) \geqq n \log n$, and b, c and d are complex numbers, then $ba_0^2+ca_0+d, ba_1^2+ca_1+d, \cdots$ is in class $T(n)$.*

Proof. By completing the square, we can find complex numbers e and f such that for all x,

$$b(x+e)^2+f = bx^2+cx+d.$$

By Lemma 3, $a_0+e, a_1+e, \cdots$ is in class $T(n)$. Using Theorem 2 with $k=2$, we see that $b(a_0+e)^2+f, b(a_1+e)^2+f, \cdots$ is in class $T(n)$. This sequence is the desired one. $\square$

We have the following corollary to the theorems above and the chirp transform theorem.

THEOREM 4. *The following sequences are in class $n \log n$ for complex numbers a, b, c and d, and positive integer k:*

$$b+c, ba^k+c, ba^{2k}+c, \cdots \tag{15}$$

and

$$(16)\qquad b+c+d,\ ba^2+ca+d,\ ba^4+ca^2+d,\cdots.$$

Note that (15) is a special case of (16) with a, b and d set to a^k, 0 and b, respectively.

4. Evaluation of a polynomial and all its derivatives. There has been recent interest in the question of how fast one can evaluate a polynomial and all its derivatives at a single point. Shaw and Traub [11] show that one can reduce the number of multiplications to $O(n)$, although the algorithm given required $O(n^2)$ total operations. Kung [3] and Borodin and Munro [13] independently observed that evaluation of a polynomial and its derivatives reduces to evaluation and interpolation of polynomials, and thus could be done in $O(n \log^2 n)$ steps. Kung [12] gives another $O(n \log^2 n)$ algorithm without using evaluation and interpolation. In this section we show that problem can be done in $O(n \log n)$ steps. This result hinges upon the following definition and lemma.

The *Taylor series* of a polynomial $\sum_{i=0}^{n-1} b_i x^i$ at point a is that polynomial $\sum_{i=0}^{n-1} c_i x^i$ such that for all x,

$$\sum_{i=0}^{n-1} c_i(x-a)^i = \sum_{i=0}^{n-1} b_i x^i.$$

LEMMA 4. *The problems of evaluating a polynomial and all its derivatives at a point and of finding the Taylor series of a polynomial at a point require within* $2n$ *operations of each other for polynomials of degree* $n-1$.

Proof. Let $\sum_{i=0}^{n-1} b_i x^i = \sum_{i=0}^{n-1} c_i(x-a)^i$ for all x. Then

$$\frac{d^k}{dx^k}\left(\sum_{i=0}^{n-1} b^i x^i\right)\Bigg|_{x=a} = k!c_k.$$

Thus the kth derivative at point a and the kth coefficient of the Taylor expansion can be recovered from one another by multiplying or dividing by k! □

THEOREM 5. *An* $(n-1)$*-st-degree polynomial and all its derivatives can be evaluated at point c in* $O(n \log n)$ *steps.*

Proof. In Lemma 3 we showed how to compute from c and the b_i's in $O(n \log n)$ steps those numbers d_j, $0 \leqq j \leqq n-1$, such that for all x,

$$(17)\qquad \sum_{i=0}^{n-1} b_i(x+c)^i = \sum_{r=0}^{n-1} d_r x^r.$$

If (17) holds, then it is surely true that for any x,

$$\sum_{i=0}^{n-1} b_i x^i = \sum_{i=0}^{n-1} d_i(x-c)^i.$$

The theorem then follows from Lemma 4. □

Theorem 5 has been independently shown by Vari [14].

Acknowledgment. The authors would like to thank the referees for a number of perceptive comments.

REFERENCES

[1] R. MOENCK AND A. B. BORODIN, *Fast modular transforms via division*, Conf. Rec. IEEE 13th Ann. Symp. on Switching and Automata Theory, 1972, pp. 90–96.

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

[3] H. T. KUNG, *Fast evaluation and interpolation*, Tech. Rep., Dept. of Computer Sci., Carnegie-Mellon Univ., Pittsburgh, 1973.

[4] M. SIEVEKING, *An algorithm for the division of power series*, Computing, 10 (1972), pp. 153–156.

[5] L. I. BLUESTEIN, *A linear filtering approach to the computation of the discrete Fourier transform*, IEEE Trans. Electroacoustics, AU-18 (1970), pp. 451–455.

[6] L. R. RABINER, R. W. SCHAFER AND C. M. RADER, *The chirp z-transform and its applications*, Bell System Tech. J., 48 (1969), pp. 1249–1292.

[7] J. M. COOLEY AND J. W. TUKEY, *An algorithm for the calculation of Fourier series*, Math. Comp., 19 (1965), pp. 297–301.

[8] A. OPPENHEIM AND D. H. JOHNSON, *Discrete representation of signals*, Proc. IEEE, 60 (1972), pp. 681–691.

[9] A. OPPENHEIM, D. H. JOHNSON AND K. STEIGLITZ, *Computation of spectra with unequal resolution using the fast Fourier transform*, Proc. IEEE (Lett), 59 (1971), pp. 299–301.

[10] J. HARTMANIS AND R. E. STEARNS, *On the computational complexity of algorithms*, Trans. Amer. Math. Soc., 117 (1965), 285–306.

[11] M. SHAW AND J. F. TRAUB, *On the number of multiplications for the evaluation of a polynomial and some of its derivatives*, J. Assoc. Comput. Mach., 21 (1974), pp. 161–167.

[12] H. T. KUNG, *A new upper bound on the complexity of derivative evaluation*, Information Processing Letters, 2 (1973), pp. 146–147.

[13] A. B. BORODIN AND I. MUNRO, *Notes on Efficient and Optimal Algorithms*, American-Elsevier, New York, 1975.

[14] T. M. VARI, *Some complexity results for a class of Toeplitz matrices*, Tech. Rep., Dept. of Computer Sci. and Mathematics, York Univ., Toronto, 1974.

SIAM J. COMPUT.
Vol. 4, No. 4, December 1975

AN ELEMENTARY SOLUTION OF THE QUEUING SYSTEM G/G/1*

ALAN G. KONHEIM†

Abstract. In this note we give an elementary method for calculating the stationary distribution of waiting time in a G/G/1 queue.

Key words. queuing theory

Our goal is to provide an elementary method for calculating the stationary distribution of waiting time in a G/G/1 queue. The analysis of the waiting time of the nth customer has been given by several authors; in 1952 by both Pollaczek [4] and Lindley [3] and by Spitzer [5] in 1957. A good presentation of these methods may be found in the recent book by Cohen [2]. All of these solutions share the common use of complex analytic methods. Lindley's solution involves a Wiener–Hopf factorization, while both Pollaczek and Spitzer require the evaluation of contour integrals. Our point of view is somewhat different. We propose to examine the G/G/1 system within a discrete framework. This will avoid some of the complications in the aforementioned solutions. Our solution will only require the ability to factor polynomials. The discrete setting appears natural for applications in computer science in which the statistical data for the problem may be empirically derived.

Our assumptions are as follows:

1. Customers (service requests) enter the service station singly; their *arrival times* $\{T_i : 1 \leqq i < \infty\}$ $(T_0 = 0)$ form a renewal process with *inter-arrival times* $\{\tau_i = T_i - T_{i-1} : 1 \leqq i < \infty\}$. The inter-arrival times are independent identically distributed random variables with law

$$\Pr\{\tau_i = k\delta\} = p_k, \qquad 0 \leqq k \leqq K_1 < \infty.$$

2. *The service time process* $\{S_i : 1 \leqq i < \infty\}$, where S_i is the service time of the ith request, consists of independent identically distributed random variables with law

$$\Pr\{S_i = k\delta\} = q_k, \qquad 0 \leqq k \leqq K_2 < \infty.$$

3. The inter-arrival time and service processes are independent.
4. The queuing process is positive recurrent:

$$\sum_{k=0}^{K_1} kp_k > \sum_{k=0}^{K_2} kq_k.$$

* Received by the editors August 27, 1974, and in revised form December 7, 1974.

† IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.

Let W_n denote the *waiting time* of the nth customer. Then $W_1 = 0$ and

$$W_{n+1} = \begin{cases} W_n + S_n - \tau_{n+1} & \text{if } T_n + S_n + W_n \geqq T_{n+1}, \\ 0 & \text{otherwise,} \end{cases}$$

which we may write as

$$W_{n+1} = (W_n + (S_n - \tau_{n+1}))^+, \qquad 1 \leqq n < \infty, \quad W_1 = 0, \tag{1}$$

with $a^+ = \max(a, 0)$. If $F_n(x) = \Pr\{W_n \leqq x\}$ is the distribution function of W_n, then (1) yields

$$F_{n+1}(x) = \begin{cases} \int_{-\infty}^{x} F_n(x-y)G(dy) & \text{if } 0 \leqq x < \infty, \\ 0 & \text{otherwise,} \end{cases} \tag{2}$$

where G is the distribution function of $S_n - \tau_{n+1}$. Since the queueing process is positive recurrent, the sequence $\{F_n\}$ converges as $n \to \infty$ to a proper distribution function F which satisfies the so-called Wiener–Hopf equation

$$F(x) = \begin{cases} \int_{-\infty}^{x} F_n(x-y)G(dy) & \text{if } 0 \leqq x < \infty, \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

It is clear from our assumptions that the distribution function F has points of incr^ase only in the set $\{k\delta : k = 0, 1, 2, \cdots\}$. Setting $f_k = F(k\delta) - F(k\delta -)$, we observe that (3) is equivalent to the system of algebraic equations

$$\sum_{k=0}^{j} f_k = \sum_{k=0}^{\infty} f_k r_{j-k}, \qquad 0 \leqq j < \infty, \tag{4}$$

where

$$r_m = \sum_{\substack{(j,k) \\ 0 \leqq j \leqq K_1, 0 \leqq k \leqq K_2 \\ k-j \leqq m}} p_j q_k. \tag{5}$$

Note that r_m vanishes outside of the set of points $\{-K_1, \cdots, 0, 1, \cdots, K_2\}$. To the sequences $\{p_k\}$ and $\{q_k\}$ we associate the generating functions

$$P(z) = p_0 + p_1 z + \cdots + p_{K_1} z^{K_1},$$
$$Q(z) = q_0 + q_1 z + \cdots + q_{K_2} z^{K_2}$$

and to $\{r_k\}$ the series

$$R(z) = r_{-K_1} z^{-K_1} + \cdots + r_{K_2} z^{K_2}.$$

Note from (5) that

$$R(z) = \frac{P(1/z)Q(z)}{1-z},$$

valid for $0<|z|<\infty$. To solve the system of (4), we introduce the auxiliary sequence $\{\vec{f}_k: -K_1 \leqq k<\infty\}$ defined by

$$\text{(6)} \qquad \sum_{k=-K_1}^{j} \vec{f}_k = \sum_{k=0}^{\infty} f_k r_{j-k}, \qquad -K_1 \leqq j < \infty.$$

Observe that

$$f_0 = \sum_{k=-K_1}^{0} \vec{f}_k,$$

$$f_k = \vec{f}_k, \qquad 1 \leqq k < \infty.$$

Let us set

$$F(z) = f_0 + f_1 z + \cdots + f_k z^k + \cdots,$$

$$\vec{F}(z) = \vec{f}_{-K_1} z^{-K_1} + \cdots + \vec{f}_0 + \vec{f}_1 z + \cdots + \vec{f}_k z^k + \cdots,$$

the first series converging when $|z| \leqq 1$, the second when $0<|z|\leqq 1$. From (6) we have

$$\text{(7)} \qquad \frac{\vec{F}(z)}{1-z} = \frac{F(z)P(1/z)Q(z)}{1-z}.$$

We write $\vec{F}(z) = z^{-K_1}M(z)(1-z)+F(z)$, where M is a polynomial of degree at most K_1-1, and note from (7) that

$$F(z)\frac{1-P(1/z)Q(z)}{1-z} = z^{-K_1}M(z).$$

We assert that the analyticity of F determines M uniquely. To show this, set

$$\text{(8)} \qquad S(z) = \frac{1-P(1/z)Q(z)}{1-z} = \frac{z^{K_1}-P^*(z)Q(z)}{z^{K_1}(1-z)},$$

where $P^*(z) = p_{K_1} + p_{K_1-1}z + \cdots + p_0 z^{K_1}$. The numerator of (8) has a zero at $z=1$ since $P(1)=P^*(1)=Q(1)=1$. It is a simple zero since the derivative of the numerator evaluated at $z=1$ is

$$K_1 - \left[K_1 - \sum_{k=0}^{K_1} kp_k\right] - \sum_{k=0}^{K_2} kq_k > 0.$$

Both P and Q are polynomials with nonnegative coefficients. It follows that $z=1$ is the only zero of the numerator on the unit circle unless both P and Q are polynomials in z^s for some integer $s>1$. In this latter case, $\{p_k\}$ and $\{q_k\}$ vanish unless k is a multiple of s and we may reduce this case to $s=1$ by replacing δ by $s\delta$. Hence we may assume that the zeros of S do not lie on $\{z : z=1\}$. Factor $S(z)$:

$$S(z) = S^+(z)S^-(z),$$

$$S^+(z) = C_1 \prod (z-\eta_i^+)^{r_i^+}, \qquad 1<|\eta_i^+|,$$

$$S^-(z) = C_2 z^{-C_3} \prod (z-\eta_i^-)^{r_i^-}, \quad 0<|\eta_i^-|<1,$$

$$S^+(1) = 1.$$

LEMMA. *C_3 is equal to one plus the number of roots (counting multiplicity) of S in $0<|z|<1$.*

Proof. The proof is a consequence of Rouché's theorem [1]. Take $\varepsilon > 0$; then

$$\begin{aligned}\max_{|z|\leqq 1+\varepsilon} |P^*(z)Q(z)| &= P^*(1+\varepsilon)Q(1+\varepsilon)\\ &= 1+[K_1-P'(1)+Q'(1)]\varepsilon+O(\varepsilon^2)\\ &< 1+K_1\varepsilon \leqq (1+\varepsilon)^{K_1}.\end{aligned}$$

Thus by Rouché's theorem, z^{K_1} and $z^{K_1}-P^*(z)Q(z)$ have the same number of zeros in $\{z : |z|\leqq 1+\varepsilon\}$. One of these zeros is at $z=1$, and since ε may be taken arbitrarily small, we conclude that the numerator of S has K_1-1 zeros in $\{z : |z|<1\}$. This implies the lemma. Note that $C_3 \leqq K_1$.

We now have

$$F(z)S^+(z)=\frac{-M(z)}{C_2 z^{K_1-C_3}\prod(z-\eta_i^-)r_i^-}=\frac{-M(z)}{D(z)} \quad \text{(say)},$$

where D is a polynomial whose degree by the lemma is K_1-1. But F is analytic in $|z|<1$, and hence D must be a factor of M. Since the degree of M is K_1-1, we must have $D=-cM$. Finally, $F(1)S^+(1)=1$ and hence $c=1$. This yields our theorem.

THEOREM. $F(z)=1/S^+(z)$.

Examples. We take the service time to be uniformly distributed on the set $1, 2, \cdots, 10$:

$$q_k=\begin{cases}0.1 & \text{if } k=1,2,\cdots,10,\\ 0 & \text{otherwise.}\end{cases}$$

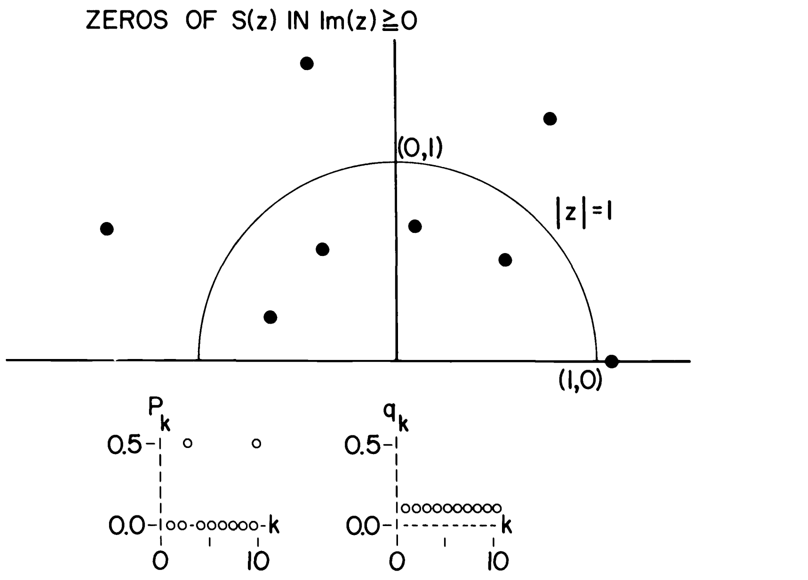
The expected service time is 5.5. The inter-arrival times have law

$$p_k=\begin{cases}0.5 & \text{if } k=a, 10,\\ 0 & \text{otherwise,}\end{cases}$$

with expectation $5+0.5a$. We consider the three cases $a=2, 3, 5$. The determination of $S^+(z)$ in this case requires a factorization of

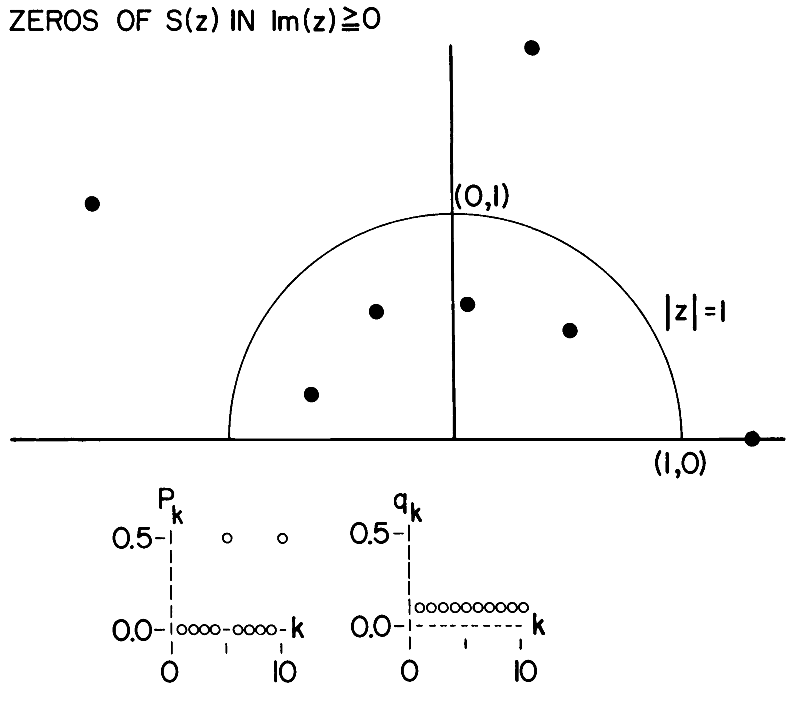
$$S(z)=\frac{z^9-0.05(1+z^{10-a})(1+z+\cdots+z^9)}{z^9(1-z)}.$$

The degree of S^+ is easily seen to be $10-a$. A standard root finding algorithm implemented in the APL/360 system was used. The location of the zeros in the three cases is shown in Figs. 1–3.



$S^+(z)=3.29-0.46z-0.41z^2-0.37z^3-0.32z^4-0.29z^5-0.24z^6-0.20z^7$

FIG. 1



$S^+(z)=1.70-0.18z-0.16z^2-0.14z^3-0.12z^4-0.10z^5$

FIG. 2

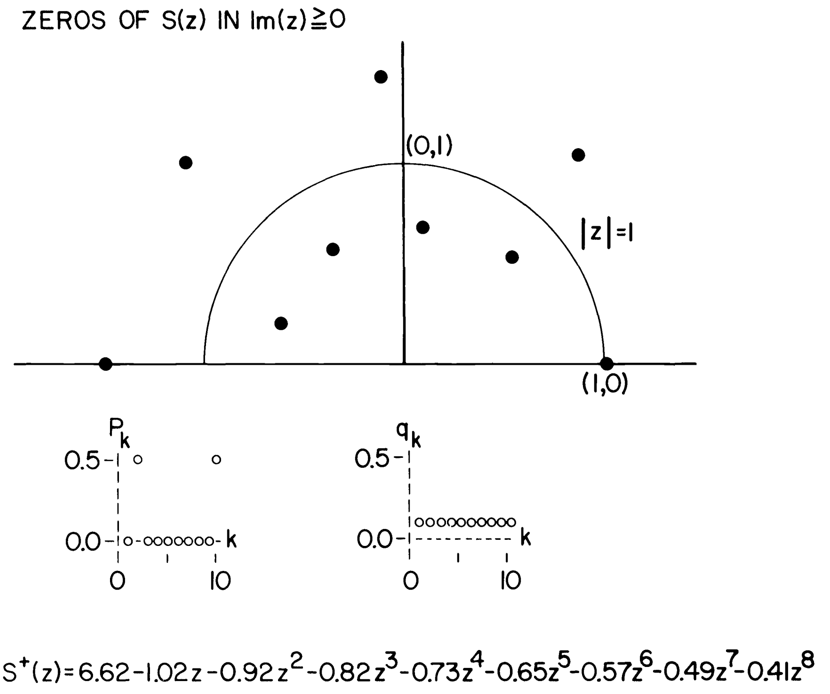


FIG. 3

REFERENCES

[1] LARS V. AHLFORS, *Complex Analysis*, McGraw-Hill, New York, 1953.

[2] J. W. COHEN, *The Single Server Queue*, Wiley–Interscience, New York, 1969.

[3] D. V. LINDLEY, *The theory of queues with a single server*, Proc. Cambridge Philos. Soc., 48 (1952), pp. 277–289.

[4] FELIX POLLACZEK, *Fonctions caractéristiques de certain es répartitions définies au moyen de la notion d'orde. Application à la théorie des attentes*, C.R. Acad. Sci. Paris, 234 (1952), pp. 2334–2336.

[5] FRANK SPITZER, *The Wiener–Hopf equation whose kernel is a probability density*, Duke Math. J., 24 (1957), pp. 327–343.