PREFACE

Any new journal begins as an idea in the minds of its editors. This idea is reflected in an initial editorial policy. Then, as the journal develops, this initial policy is expanded and modified by the interchange of ideas that occurs among the editors, authors, referees and readers. In this way the journal can transcend the original vision of its founders and increase its usefulness to the scientific community.

To encourage this interchange we present some of the concepts that have guided us thus far. We invite your comments on these ideas and on the contents of this and future issues.

We see several trends related to mathematics and computing:

(1) There is an increasing realization that computer science and computer applications have important and useful mathematical foundations.

(2) In many mathematical disciplines there is an increasing amount of work which is motivated by computational problems or in which computational issues are crucial. This has led to an increasing interchange of ideas across disciplines using computer oriented concepts as the unifying thread.

(3) The implications of these new mathematical results are important to both those interested in computers per se and those interested in using computers to solve problems in other application areas.

(4) This new mathematics is not only of theoretical interest, but is useful to computer people in their everyday work. The "art" of computers and computer usage is rapidly becoming a "science" and that science is based on mathematics.

These trends imply that results in a wide spectrum of mathematical disciplines are of increasing interest to a great many applied mathematicians who are involved with computers and computation. Thus it is particularly appropriate that SIAM undertake to publish a new journal on the mathematics of computers and computing.

SIAM Journal on Computing (SICOMP) is a mathematics journal serving the computing community. We shall attempt to bring together in one journal all the mathematical disciplines related to the nonnumerical aspects of computers and computing. We shall encourage our authors to explain fully the implications of their results for real computing problems.

In preparing this issue, several questions of editorial policy have arisen. We would like to share our current thoughts on these questions with you and invite your comments.

(1) What exactly is a "mathematical contribution to computers or computing"? This is not a trivial question since we are talking about a wide spectrum of mathematical disciplines and a range of application areas, some of which may be only indirectly associated with computers. We must confess, although we think we have a good intuitive feel for what we mean, we have not been able to write down a completely satisfactory answer to this question. Perhaps the best one sentence description of our ideas is: *The mathematics must either be directed towards computers or involve, in an essential way, computational issues such as computation*

time, storage space or data structures.

(2) What should be our policy with respect to the publication of algorithms? Our initial thought is that SICOMP is concerned with the *theory* of algorithms and is not a journal where a compendium of useful algorithms appears. This implies we publish only algorithms that involve interesting mathematics. The fact that an algorithm works is not necessarily an interesting mathematical result.

(3) What should be our policy with respect to the publication of descriptions of new programming languages? Our initial thought is that most new programming languages are not contributions to mathematics and hence would not be appropriate for SICOMP. However, a new language could conceivably be of sufficient interest mathematically, or of sufficient interest to mathematicians that we would want to publish its description.

We welcome your comments on any of these questions and on any other subject relevant to the journal. We hope you find our first issue interesting and useful. With your help we will make SICOMP even more interesting and useful in the years to come.

PHILIP M. LEWIS II

SIAM J. COMPUT.
Vol. 1, No. 1, March 1972

OPTIMIZATION OF STRAIGHT LINE PROGRAMS*

ALFRED V. AHO† AND JEFFREY D. ULLMAN‡

Abstract. We provide a set of transformations capable of transforming a straight line program into any other equivalent one assuming no algebraic laws hold. We then show that optimization of straight line code under "reasonable" cost criteria can always be accomplished by applying sequences of these transformations in a prescribed order.

Keywords. Programming theory, code optimization, straight line programs, optimizing transformations

1. Introduction. The problem we consider is the optimization of straight line segments of computer code. The general approach is to find a set of transformations which is complete, in the sense that any two equivalent programs can be transformed into one another using transformations from this set. We then characterize code optimization algorithms in terms of these transformations.

Our motivation for looking at straight line code is twofold. First, knowing how to optimize straight line code seems to be a necessary prerequisite for optimization of more general programs. Secondly, we wish to isolate problems inherent in code optimization without encountering too many undecidability results. When one considers anything more complex than straight line programs, the equivalence problem for programs is likely to be undecidable. A good example of this phenomenon occurs in [1], [2] where a model of programs similar to ours, but having loops, is considered.

Various practical approaches to code optimization have involved looking at transformations which preserve program equivalence. Some of these works are [3]–[8]. This group of papers considers programs with loops, and no attempt has been made to find complete sets of transformations.

Various works on program schemata following Ianov [9] have shown equivalence to be decidable for programs with loops. The seeming discrepancy between these schemata and our previous comments can be explained by the fact that our definition of when two programs are equivalent is considerably more liberal than that of [9]–[13]. The latter works require that programs apply the same operations in the same order, while we allow independent operations to be applied in any order.

Igarishi has characterized equivalent straight line programs by a set of transformations somewhat similar to ours [14]. However, two major differences between our approach and Igarishi's are:

(i) In [14], statements of the form $A \leftarrow B$ are permitted, while we do not permit these. These statements could be incorporated into our model, but they would disappear in the simple graphical representation of programs which we use.

* Received by the editors July 21, 1971.

† Bell Telephone Laboratories, Inc., Murray Hill, New Jersey 07974.

‡ Electrical Engineering Department, Princeton University, Princeton, New Jersey 08540. The work of this author was supported in part by the National Science Foundation under Grant GJ-1052.

(ii) We do not care how many variables have the same value at the end of a program or what names are given to a variable. Only the set of values computed is important. Our motivation is that if a straight line block is part of a larger program, there is never a need for two or more copies of the same value to be passed from block to block, provided we know for certain that the values will always be the same. In contrast, the name of the variable holding a given value is important in [14].

Thus, the models and notion of equivalence in [14] are not quite the same, and a direct derivation of our transformations from Igarishi's "axioms" would be difficult. Our model admits of a very simple characterization of equivalent programs as well as a simple graphical representation of programs. Both these features are useful in discussing optimization algorithms.

Another work [15] has done an analysis quite similar to ours for a much simpler model of a program, in which all statements are of the form $A \leftarrow B$ for variables A and B.

A preliminary version of some of the material in this paper appears in [16], and the latter paper also contains some consideration of how to extend these ideas in a straightforward manner to the case in which algebraic laws are permitted to transform expressions into algebraically equivalent expressions.

2. Basic concepts. We begin by defining our model of a straight line program together with other concepts we shall use.

2.1. Programs. Let Θ and Σ be disjoint sets; Θ, the *operator alphabet*, is finite, and Σ, the *variable names*, is a countable set. We assume that each operator in Θ has an associated positive integer, its *rank*, which indicates the number of arguments taken by that operator. A *statement over* Θ *and* Σ (or *statement* when Θ and Σ are understood) is a string of the form $A \leftarrow \theta B_1 \cdots B_n$, where θ is in Θ and has rank n, $A, B_1, \cdots, B_n$ are in Σ and $\leftarrow$ is a metasymbol, presumed to be in neither Θ nor Σ. We say the statement $A \leftarrow \theta B_1 \cdots B_n$ *sets* A and *references* $B_1, \cdots, B_n$.

A *program over* Θ *and* Σ (or *program*, where Θ and Σ are understood) is a triple $\pi = (P, I, U)$, where:

1. I and U are finite subsets of Σ, the *input* and *output variables*, respectively.
2. P is a list of statements $S_1; S_2; \cdots; S_m$, where $m \geqq 0$, and S_i is a statement over Θ and Σ, for $1 \leqq i \leqq m$.
3. If S_i references a variable B, then either B is in I or B is set by some previous S_j, $j < i$.
4. If B is in U, then either B is in I or B is set by some statement.

Thus, a program is a sequence of statements, with a known set of initially defined variables (I) and a known set of output variables (U). Conditions 3 and 4 ensure that any variable appearing in the program will have a well-defined value.

Let $\pi = (P, I, U)$ be a program over Θ and Σ, with $P = S_1; S_2; \cdots; S_m$. The *value of variable* A *at time* t *according to* π, denoted $v_t^\pi(A)$ (or $v_t(A)$, if π is understood), is defined to be an expression over $\Theta \cup \Sigma$ in prefix Polish notation. The definition is as follows.

1. $v_0(A) = A$ for all A in I.
2. Let S_t be $A \leftarrow \theta B_1 \cdots B_n$, $t \geqq 1$. Then $v_t(A) = \theta v_{t-1}(B_1) v_{t-1}(B_2) \cdots v_{t-1}(B_n)$.

3. Let S_t be $A \leftarrow \theta B_1 \cdots B_n$, and let $C \neq A$. If $v_{t-1}(C)$ is defined, then $v_t(C) = v_{t-1}(C)$.

4. $v_t(A)$ is undefined if not defined by (i)–(iii).

We observe that condition 3 in the definition of program ensures that whenever rule 2, above, is applied, $v_{t-1}(B_i)$ will be defined, for $1 \leqq i \leqq n$.

The *value of program* $\pi = (P, I, U)$, denoted $v(\pi)$, is $\{v_m(A) | A \text{ is in } U\}$, where m is the number of statements in P. We say π_1 is *equivalent* to π_2 (written $\pi_1 \equiv \pi_2$) if $v(\pi_1) = v(\pi_2)$.

Example 2.1. We shall consider a program $\pi = (P, I, U)$ over Θ and Σ, where Θ includes $+$ and $*$, and Σ includes $A, B, \cdots, Z$. Let P be the list of statements

$$F \leftarrow +AX;$$

$$T \leftarrow *XX;$$

$$G \leftarrow +FT;$$

$$T \leftarrow *TT;$$

$$G \leftarrow +GT,$$

and let $I = \{A, X\}, U = \{F, G\}$. Table 1 gives the values of each variable mentioned, where $y = +++AX*XX**XX*XX$. Thus $v(\pi) = \{+AX, y\}$. This program represents the evaluation of the polynomials $A + X$ and $A + X + X^2 + X^4$.

TABLE 1

t	$v_t(A)$	$v_t(X)$	$v_t(F)$	$v_t(T)$	$v_t(G)$
0	A	X	—	—	—
1	A	X	$+AX$	—	—
2	A	X	$+AX$	$*XX$	—
3	A	X	$+AX$	$*XX$	$++AX*XX$
4	A	X	$+AX$	$**XX*XX$	$++AX*XX$
5	A	X	$+AX$	$**XX*XX$	y

2.2. Directed acyclic graphs (DAGs). An *ordered directed graph* G is a pair (N, R), where N is a set of nodes and R is a set of lists of nodes such that for each a in N, there is at most one list of the form $((a, b_1), (a, b_2), \cdots, (a, b_n))$, where each b_i is in N. This element would indicate that node a has n *descendants*, the first being b_1, the second b_2 and so on. Node a is an *ancestor* of each b_i. The pàir (a, b_i) is called an *edge directed from a to* b_i.

A *path* in graph G is a sequence of nodes $n_1, n_2, \cdots, n_m, m > 1$, such that n_i is a descendant of n_{i-1}, for all $i, 1 < i \leqq m$. The *length* of the path is $m - 1$. The path is a *cycle* if $n_1 = n_m$. A *leaf* is a node with no descendants. A node which is not a leaf is called an *interior node*. A node with no ancestors is called a *root*.

We call an ordered directed graph with no cycles a DAG (for directed, acyclic graph). We shall display a DAG with descendants below their ancestor. Also we assume that the edges immediately leaving a node are in order from left to right, with the first edge leftmost.

Figure 2.1 shows a DAG in which node A has three descendants, node B being the first, C the second, and B again the third. Nodes A and D are roots and nodes B, C, and D are leaves.

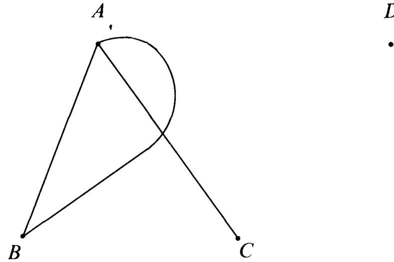


FIG. 2.1. *Example of a DAG*

Let Θ and Σ be disjoint sets. A Θ-Σ *labeling* (or labeling) of a DAG assigns to each leaf an element of Σ and to each interior node an element of Θ. In addition, each node is either "distinguished" or "not distinguished," a notion whose meaning will become clear shortly. When drawing a DAG, we shall put the labels directly on the nodes and circle distinguished nodes.

Let D be a DAG with a Θ-Σ labeling. We assign a *value* to the nodes of D and to D itself, as follows. Let $v^D(n)$, or $v(n)$ where D is understood, denote the value of node n; $v(D)$ represents the value of D.

1. If n is a leaf with label A in Σ, then $v(n) = A$.

2. If n is an interior node with label θ in Θ, and its descendants have values $v_1, v_2, \cdots, v_n$, in order from the left, then $v(n) = \theta v_1 v_2 \cdots v_m$.

3. $v(D) = \{v(n) | n \text{ is a distinguished node}\}$.

To every program π there corresponds naturally a labeled DAG, which we denote $D(\pi)$. Given a program $\pi = (P, I, U)$ we may construct $D(\pi)$ as follows.

1. For each element A of I, create a node n_A and label it A; n_A will be a leaf of $D(\pi)$.

2. Let $P = S_1; S_2; \cdots; S_m$. We create an interior node of $D(\pi)$ *corresponding to* each statement of P. Suppose nodes $n_1, n_2, \cdots, n_i$ have been created, corresponding to $S_1, S_2, \cdots, S_i$, respectively. (Initially, $i = 0$.) Let S_{i+1} be $A \leftarrow \theta B_1 \cdots B_r$. Create node n_{i+1} and label it θ. Node n_{i+1} has r descendants, $x_1, \cdots, x_r$, defined as follows, in order from the left.

(a) Suppose for some k, $1 \leqq k \leqq r$, that j is the largest integer less than i such that S_j sets B_k. Then x_k is the node n_j corresponding to S_j.

(b) If no such j exists, then B_k must be in I, and x_k is the leaf n_{B_k}.

3. For each element A of U, if j is the largest integer such that S_j sets A, then n_j is distinguished. If no such j exists, leaf n_A is distinguished.

From the construction of $D(\pi)$ we can show by induction on i that $v^D(n_i) = v_i^\pi(A)$ if S_i sets A. Consequently if n_i is a distinguished node, and S_i sets A, then $v_i^\pi(A) = v_m^\pi(A)$. Hence $v(D(\pi)) = v(\pi)$. That is to say, the values of the distinguished nodes of D are the same as the values of the output variables of π after the last statement of the program.

Example 2.2. Let π be as in Example 2.1. Then $D(\pi)$ is shown in Fig. 2.2.

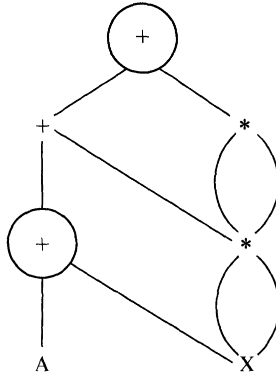


FIG. 2.2. *DAG for* π

2.3. The concept of scope. Let $\pi = (P, I, U)$ be a Θ-Σ program with $P = S_1; S_2; \cdots; S_m$. Suppose statement S_i is $A \leftarrow \theta B_1 \cdots B_r$. The *scope* of S_i is the sequence of statements $S_{i+1}, S_{i+2}, \cdots, S_j$, where j is the largest integer such that A is referenced by S_j but is not set by $S_{i+1}, \cdots, S_{j-1}$. In addition if A is in U, and none of $S_{i+1}, \cdots, S_m$ set A, then the scope of S_i is U and the statements $S_{i+1}, \cdots, S_m$. Otherwise, the scope of S_i is null.

We can also associate a scope with input variables. The *scope of A in I* is the sequence of statements $S_1, \cdots, S_j$, where j is the largest integer such that S_j references A and A is not set by $S_1, \cdots, S_{j-1}$. Also, if A is in U and is not set by any statement in P, then the scope of A in I is U and all statements. Otherwise, the scope of A in I is null. We say that a variable is *used* if it is referenced by some statement or is an output variable.

If a variable A is set by statement S_i, we say A is *active* over the scope of S_i. Likewise, if A is in I, then A is *active* over the scope of A in I.

In several situations we want to know what names can be used for a variable defined by a statement of a program. We say a variable name A is *permissible* for S_i if either:

(i) A is set by S_i, or
(ii) the scope of S_i is not null and A is not active within the scope of S_i, or
(iii) the scope of S_i is null and A is not active at S_{i+1} (or U if $i = m$).

Example 2.3. Let π be as in Example 2.1. Then the scope of the statement $T \leftarrow *XX$ is $G \leftarrow +FT$; $T \leftarrow *TT$, and the scope of $G \leftarrow +GT$ is U. T is active over the statements $G \leftarrow +FT$; $T \leftarrow *TT$; $G \leftarrow +GT$. F is active over the last 4 statements and U. No other name is active over the statement $G \leftarrow +FT$. Thus, the set of permissible names for this statement is $\Sigma - \{F, T\}$. Roughly speaking, a name is permissible for S_i if it can be substituted for the variable actually set by S_i everywhere within the scope of S_i, without altering the value of the program.

2.4. Transformations on programs. Formally, a *transformation* T on Θ-Σ programs is a mapping from Θ-Σ programs to subsets of Θ-Σ programs.

We would like to have a set S of transformations such that for any pair of equivalent programs π_1 and π_2 there is a finite sequence of transformations from S which maps π_1 into π_2. Strictly speaking there is one trivial transformation of this nature, namely, $T(\pi) = \{\pi' | \pi \equiv \pi'\}$. However, we would like to develop a theory built on transformations that have more intuitive appeal and are easier to apply from a computational point of view. The following four transformations are "natural" ones to consider.

T1. Removal of useless statements and variables.

T2. Identification of two computations producing the same value.

T3. Renaming of variables.

T4. Flipping of adjacent statements.

These transformations are defined as follows.

Transformation T1. Let $\pi = (P, I, U)$, where $P = S_1 ; \cdots ; S_m$, and let S_i have null scope. Then $\pi' = (P', I, U)$ is in T1(π), where $P' = S_1 ; \cdots ; S_{i-1} ; S_{i+1} ; \cdots ; S_m$. Suppose A is in I and this A has null scope. Then $\pi'' = (P, I - \{A\}, U)$ is in T1(π).

Thus, transformation T1 removes from a program a statement which sets a variable that is never used or an input variable that is never used.

Transformation T1 is particularly simple to implement. Given a program $\pi = (P, I, U)$, we can scan the sequence of statements $P = S_1 ; S_2 ; \cdots ; S_m$ backwards to determine U_k, the set of active variables at statement S_k. Initially, $U_m = U$. Suppose U_k has been determined. If statement S_{k-1} is $A \leftarrow \theta B_1 \cdots B_n$ then:

(i) $U_{k-1} = U_k$ if A is not in U_k. In this case S_{k-1} is a useless statement.

(ii) $U_{k-1} = (U_k - \{A\}) \cup \{B_1, \cdots, B_n\}$ if A is in U_k. In this case S_{k-1} is not useless.

(iii) If A is in I but not in U_0, then A is a useless input variable.

Transformation T2. Let $\pi = (P, I, U)$, where $P = S_1 ; \cdots ; S_m$. Suppose that for some $i < j$, S_i is $B \leftarrow \theta A_1 \cdots A_r$, S_j is $C \leftarrow \theta A_1 \cdots A_r$ and none of $A_1, \cdots, A_r$ are set by $S_i, \cdots, S_{j-1}$. Then the program $\pi' = (P', I, U')$ is in T2(π) where $P' = S_1 ; \cdots ; S_{i-1} ; D \leftarrow \theta A_1 \cdots A_r ; S'_{i+1} ; \cdots ; S'_{j-1} ; S'_{j+1} ; \cdots ; S'_m$.

Here, S'_k, $i < k \leqq m$, is S_k with the following changes:

(i) If S_k is in the scope of S_i, then replace references to B by D.

(ii) If S_k is in the scope of S_j, then replace references to C by D.

Also, U' is U with B replaced by D if U is in the scope of S_i and C replaced by D if U is in the scope of S_j.

D may be any variable name such that $v(\pi) = v(\pi')$.

Clearly, any variable not used in π is suitable as the variable D. We can also use for D some variables already present in π. However, a precise set of conditions on the allowable names for D is complicated (although possible), and the actual set of conditions is not needed for our development here.

In terms of the DAG for a program, transformation T2 merges two nodes having the same label and the same descendants (in order). Thus, it is easy to implement transformation T2 when the DAG for a program is available.

Transformation T3. Let $\pi = (P, I, U)$ with $P = S_1 ; \cdots ; S_m$. Let C be a permissible name for $S_i = A \leftarrow \theta B_1 \cdots B_r$. Then $\pi' = (P', I, U')$ is in T3(π) if:

(i) P' is P with S_i replaced by $C \leftarrow \theta B_1 \ldots B_r$ and all references to A within the scope of S_i replaced by C,

(ii) U' is either $U - \{A\} \cup \{C\}$ or U, as U is or is not in the scope of S_i.

Transformation T4. Let $\pi = (P, I, U)$ and $P = S_1; \cdots; S_m$. Let $S_i = A \leftarrow \theta B_1 \cdots B_r$ and $S_{i+1} = C \leftarrow \psi D_1 \cdots D_s$. If $A \neq C$, $A \neq D_j$ for $1 \leqq j \leqq s$, and $C \neq B_j$ for $1 \leqq j \leqq r$, then π' is in T4(π), where $\pi' = (P', I, U)$ and $P' = S_1; \cdots; S_{i-1}; S_{i+1}; S_i; S_{i+2}; \cdots; S_m$.

Note that transformations T3 and T4 have no effect on the DAG representing a program.

Example 2.4. Let $\pi = (P, \{A, B\}, \{D, E\})$, where P is

$$C \leftarrow +AB;$$
$$D \leftarrow *CC;$$
$$E \leftarrow +AB;$$
$$F \leftarrow +ED.$$

The scope of statement $F \leftarrow +ED$ is null, so we can apply T1 to π to obtain $\pi_1 = (P_1, \{A, B\}, \{D, E\})$, where P_1 is

$$C \leftarrow +AB;$$
$$D \leftarrow *CC;$$
$$E \leftarrow +AB.$$

Applying T4 to the last two statements of π_1 we can obtain $\pi_2 = (P_2, \{A, B\}, \{D, E\})$, where P_2 is

$$C \leftarrow +AB;$$
$$E \leftarrow +AB;$$
$$D \leftarrow *CC.$$

We can then apply T2 to π_2 to obtain $\pi_3 = (P_3, \{A, B\}, \{D, X\})$, where P_3 is

$$X \leftarrow +AB;$$
$$D \leftarrow *XX.$$

We could also have applied T2 directly to π_1 to obtain π_3.

If π' is in T$i(\pi)$, for $1 \leqq i \leqq 4$, we say $\pi \underset{i}{\Rightarrow} \pi'$. If S is a subset of $\{1, 2, 3, 4\}$ we define a relation $\overset{*}{\underset{S}{\Leftrightarrow}}$ on programs by: $\pi \overset{*}{\underset{S}{\Leftrightarrow}} \pi'$ if there exists a sequence of programs $\pi_1, \cdots, \pi_k$, $k \geqq 1$, such that $\pi_1 = \pi$, $\pi_k = \pi'$ and for all i, $1 \leqq i \leqq k$, either $\pi_i \underset{j}{\Rightarrow} \pi_{i+1}$ or $\pi_{i+1} \underset{j}{\Rightarrow} \pi_i$ for some j in S. Thus, $\overset{*}{\underset{S}{\Leftrightarrow}}$ is the least equivalence relation containing $\underset{i}{\Rightarrow}$ for each i in S. Note that the transformation implied by $\overset{*}{\underset{S}{\Leftrightarrow}}$ allows application of T1–T4 in either the direction stated or in the inverse direction.

We say a set S of transformations is *complete* if $\pi_1 \equiv \pi_2$ if and only if $\pi_1 \overset{*}{\underset{S}{\Leftrightarrow}} \pi_2$. S is *minimal complete* if S is complete, but no proper subset of S is complete.

3. Completeness and independence theorems. We shall show some basic results about the four transformations we have defined. These can be stated briefly as follows.

1. $\{1, 2\}$ is the only minimal complete subset of $\{1, 2, 3, 4\}$.
2. $\pi_1 \overset{*}{\underset{3,4}{\Leftrightarrow}} \pi_2$ if and only if π_1 and π_2 have the same DAG.

3.1. Preservation of equivalence by the transformations. In this section we shall show that the four transformations preserve program equivalence, and that the effect of transformations T3 and T4 can be obtained using only T1 and T2.

LEMMA 3.1. *If* $\pi \underset{1}{\Rightarrow} \pi'$, *then* $\pi \equiv \pi'$.

Proof. The proof is a straightforward consequence of the definitions and is omitted.

T2 preserves equivalence of programs by definition, but it is important to know that it is not vacuous. We state this as the next lemma.

LEMMA 3.2. *Let* $\pi = (P, I, U)$ *and let* D *not appear in* P *or be a member of* I. *Then* $\pi \equiv \pi'$, *where* π' *is constructed as in the definition of* T2 *with this value of* D.

Proof. Again, the proof is straightforward and is omitted.

LEMMA 3.3. *If* $\pi \underset{3}{\Rightarrow} \pi'$, *then* $\pi \overset{*}{\underset{1,2}{\Leftrightarrow}} \pi'$.

Proof. Let $\pi = (P, I, U)$ where $P = S_1; \cdots ; S_m$, with $S_i = A \leftarrow \theta B_1 \ldots B_r$. Let $\pi' = (P', I, U')$, with $P' = S_1; \cdots ; S_{i-1}; \ C \leftarrow \theta B_1 \ldots B_r; \ S'_{i+1}; \cdots ; S'_m$, where S'_j is S_j with references of A replaced by C if S_j is in the scope of S_i. U' is U with A replaced by C if U is in the scope of S_i.

Let X be a variable name not appearing in π. Define π_1 to be (P_1, I, U), where $P_1 = S_1; \cdots ; S_{i-1}; X \leftarrow \theta B_1 \cdots B_r; A \leftarrow \theta B_1 \cdots B_r; S_{i+1}; \cdots ; S_m$. That is, insert an extra copy of S_i before S_i. Since X does not appear elsewhere, X is a useless variable in π_1, and we can state $\pi_1 \underset{1}{\Rightarrow} \pi$.

To show $\pi_1 \underset{2}{\Rightarrow} \pi'$, it should be clear that C can play the role of D in the definition of T2, if S_i is taken to be $X \leftarrow \theta B_1 \cdots B_r$ and S_j to be $A \leftarrow \theta B_1 \cdots B_r$ in that definition.

We have shown $\pi_1 \underset{1}{\Rightarrow} \pi$ and $\pi_1 \underset{2}{\Rightarrow} \pi'$. Thus, we may conclude $\pi \overset{*}{\underset{1,2}{\Leftrightarrow}} \pi'$. Note that a use of T1 "backwards" is essential to this argument.

LEMMA 3.4. *If* $\pi \underset{4}{\Rightarrow} \pi'$, *then* $\pi \overset{*}{\underset{1,2}{\Leftrightarrow}} \pi'$.

Proof. Let $\pi = (P, I, U)$ where $P = S_1; \cdots ; S_m$ with $S_i = A \leftarrow \theta B_1 \cdots B_r$, $S_{i+1} = C \leftarrow \psi D_1 \cdots D_s$, $A \neq C$, $A \neq D_k$, $1 \leqq k \leqq s$, and $C \neq B_k$, $1 \leqq k \leqq r$. Let $\pi' = (P', I, U)$, where $P' = S_1; \cdots ; S_{i-1}; \ S_{i+1}; \ S_i; \ S_{i+2}; \cdots ; S_m$. That is, P' is P with S_i and S_{i+1} interchanged.

Consider $\pi_1 = (P_1, I, U)$, where P_1 is $S_1; \cdots ; \ S_{i-1}; \ X \leftarrow \psi D_1 \cdots D_s$; $A \leftarrow \theta B_1 \cdots B_r$; $C \leftarrow \psi D_1 \cdots D_s; \cdots ; S_m$, and X is a new variable name. In P_1, $X \leftarrow \psi D_1 \cdots D_s$ is useless, so $\pi_1 \underset{1}{\Rightarrow} \pi$. We can then show that $\pi_1 \underset{2}{\Rightarrow} \pi'$ by showing that C can play the role of D in the definition of T2, with $S_i = X \leftarrow \psi D_1 \cdots D_s$ and $S_j = C \leftarrow \psi D_1 \cdots D_s$.

Putting the previous two lemmas together, we have the following results.

THEOREM 3.1. *If* $\pi \overset{*}{\underset{1,2,3,4}{\Leftrightarrow}} \pi'$, *then* $\pi \overset{*}{\underset{1,2}{\Leftrightarrow}} \pi'$.

Proof. The proof is immediate from Lemmas 3.3 and 3.4.

THEOREM 3.2. *If* $\pi \overset{*}{\underset{1,2,3,4}{\Leftrightarrow}} \pi'$, *then* $\pi \equiv \pi'$.

Proof. The proof is immediate from Lemmas 3.1, 3.3 and 3.4.

3.2. Characterization of DAGs. We shall now show that $\pi \overset{*}{\underset{3,4}{\Leftrightarrow}} \pi'$ if and only if $D(\pi) = D(\pi')$. Thus, the DAGs are the natural representatives for the equivalence classes under $\overset{*}{\underset{3,4}{\Leftrightarrow}}$.

LEMMA 3.5. *If* (a) $\pi \underset{3}{\Rightarrow} \pi'$ *or* (b) $\pi \underset{4}{\Rightarrow} \pi'$, *then* $D(\pi) = D(\pi')$.

Proof. Again these results are direct applications of definitions, and we shall prove only part (b). Let $\pi = (P, I, U)$ and $\pi' = (P', I, U)$, where $P = S_1; \cdots ; S_m$

and $P' = S_1; \cdots; S_{i-1}; S_{i+1}; S_i; S_{i+2}; \cdots; S_m$. Let $S_i = A \leftarrow \theta B_1 \cdots B_r$ and $S_{i+1} = C \leftarrow \psi D_1 \cdots D_s$. Certainly the leaves of $D(\pi)$ and $D(\pi')$ are the same, and the nodes corresponding to $S_1, \cdots, S_{i-1}$ in each have the same labels and the same descendants. We claim that the nodes corresponding to S_i in $D(\pi)$ and $D(\pi')$ are the same. They certainly have the same labels. Let the descendants of this node in $D(\pi)$ be $n_1, \cdots, n_r$, where n_j corresponds to the statement most recently setting B_j. Since C is not among $B_1, \cdots, B_r$, the node in $D(\pi')$ also has descendants $n_1, \cdots, n_r$.

Similarly, the nodes corresponding to S_{i+1} in $D(\pi)$ and $D(\pi')$ have the same descendants. Since $A \neq C$, after $i + 1$ nodes of the two DAGs have been constructed, the nodes corresponding to the last definitions of each variable are the same.

It is straightforward that the distinguished nodes of the two DAGs are the same. Thus, $D(\pi) = D(\pi')$.

THEOREM 3.3. *$D(\pi) = D(\pi')$ if and only if* $\pi \overset{*}{\underset{3,4}{\Leftrightarrow}} \pi'$.

Proof. Lemma 3.5 is the "if" portion, so we need prove only the converse. Suppose $D(\pi) = D(\pi') = D$, where $\pi = (P, I, U)$ and $\pi' = (P', I, U')$. Since the DAGs are the same, we note that π and π' have the same input sets, and that P and P' have the same length. Let $P = S_1; S_2; \cdots; S_n$, and $P' = T_1; T_2; \cdots; T_n$. We shall construct a sequence of programs $\pi_0, \pi_1, \cdots, \pi_n$ such that $\pi_0 = \pi$, $\pi_n = \pi'$, $\pi_i \overset{*}{\underset{\{3,4\}}{\Leftrightarrow}} \pi_{i+1}$ for all $0 \leqq i < n$, and if $\pi_i = (P_i, I, U_i)$, then the first i statements of P_i are $T_1; T_2; \cdots; T_i$. Thus, $\pi \overset{*}{\underset{\{3,4\}}{\Leftrightarrow}} \pi'$ will follow directly.

Suppose we have π_i for some $i \geqq 0$. By Lemma 3.5, $D(\pi_i) = D$. Let $P_i = R_1; \cdots; R_n$, where $R_j = T_j$, $1 \leqq j \leqq i$. Let η be the node in D corresponding to T_{i+1} in P' and R_k the statement in P_i which corresponds to η. Certainly $k > i$, else the node η corresponds to two statements of P'. Suppose $R_k = A \leftarrow \theta B_1 \cdots B_m$.

Since η corresponds to both R_k and T_{i+1}, each of the B's must either be in I or set by one of $T_1, \cdots, T_i$ $(= R_1, \cdots, R_i)$. That is, the descendants of η are each nodes corresponding to $T_1, \cdots, T_i$ and the input variables. We can say more about the B's. They must not be set by any of $R_{i+1}, \cdots, R_{k-1}$, else the node corresponding to R_k in $D(\pi_i)$ would have a descendant which could not be a descendant of η. Finally, we conclude that T_{i+1} is $C \leftarrow \theta B_1 \cdots B_m$ for some C, since the descendants of η correspond to statements in both P_i and P' which set these variables.

We now construct π_{i+1} as follows.

1. Let X be a new variable not appearing in π_i. By T3, replace R_k by $X \leftarrow \theta B_1 \cdots B_m$. (Modification of $R_{k+1}, \cdots, R_n$ and U may take place.)

2. Repeatedly using T4, bring the statement $X \leftarrow \theta B_1 \cdots B_m$ to the position immediately following R_j. This is possible since none of $R_{i+1}, \cdots, R_{k-1}$ set any of $X, B_1, \cdots, B_m$ or reference X.

3. Then, using T3, cause all statements setting C in the statements now following $X \leftarrow \theta B_1 \cdots B_m$ to set some new variable Y.

4. Finally using T3, replace $X \leftarrow \theta B_1 \cdots B_m$ by $C \leftarrow \theta B_1 \cdots B_m$. This is possible because the scope of any statement among $T_1, \cdots, T_i$ which sets C does not extend past T_{i+1} in P'.

The four steps above convert π_i into some program π_{i+1} whose first $i + 1$ statements are $T_1, \cdots, T_{i+1}$. The proof is now complete.

3.3 Reduced programs and DAGs. A program π is said to be *reduced* if there is no program π' such that $\pi \underset{1,2}{\Rightarrow} \pi'$. Intuitively, a reduced program is one which has no redundancy and no useless variables. A DAG D is *reduced* if $D = D(\pi)$ for some reduced program π.

LEMMA 3.6. *If E is an expression over $\Theta \cup \Sigma$ in prefix Polish notation, then either E is a single variable or E can be uniquely expressed as $\theta E_1 \cdots E_m$, where θ is an m-ary operator and each $E_1, \cdots, E_m$ forms a prefix expression.*

Proof. This lemma is well known. Since we know how many arguments an operator takes, a method based on counting the number of operators and arguments when scanning the expression E can be used to obtain a unique "parse" of E. The details will be omitted.

LEMMA 3.7. *Let $\pi = (P, I, U)$ be a reduced program with $P = S_1; \cdots; S_n$. Suppose statement S_k sets A_k, $1 \leqq k \leqq n$. Then for all i and j, $1 \leqq i < j \leqq n$, $v_i(A_i) \neq v_j(A_j)$.*

Proof. We shall show that if $v_i(A_i) = v_j(A_j)$ for some i and j, $i \neq j$, then π cannot be reduced.

Suppose two statements in P define the same value. Then choose that pair of integers i and j such that if (i', j') is any other pair of integers violating the stated condition, then $i < i'$ or $i = i'$ and $j < j'$. Suppose S_i is $A_i \leftarrow \theta C_1 \cdots C_m$ and S_j is $A_j \leftarrow \theta D_1 \cdots D_m$. (Clearly, the operators must be the same if the expressions are the same.)

We must have $C_k = D_k$, $1 \leqq k \leqq m$. Otherwise, suppose for some k, $C_k \neq D_k$. Then by Lemma 3.6, $v_{i-1}(C_k) = v_{j-1}(D_k)$. Let i' and j' be the largest integers less than i and j, respectively, such that $A_{i'} = C_k$ and $A_{j'} = D_k$. (The case in which one or both of C_k and D_k were not previously set can be easily ruled out.) Then $i' \neq j'$, $v_{i'}(C_k) = v_{j'}(D_k)$, and we contradict our assumption on (i, j). Thus, $C_k = D_k$ for $1 \leqq k \leqq m$.

It is easy to show that none of $C_1, \cdots, C_m$ is set by $S_i, \cdots, S_{j-1}$. For if the first such instance is C_k set by S_l, $i \leqq l < j$, then $v_{l-1}(C_k) = v_l(C_k)$, else $v_i(A_i) \neq v_j(A_j)$. Thus, the same value was defined at S_l and the previous time C_k was set. (Again, there must be such a time.) This previous time was before S_i, so we again have a contradiction to our choice of i.

We may thus apply T2 to π, replacing S_i and S_j by $X \leftarrow \theta C_1 \cdots C_m$, where X is a new variable. Hence, π is not reduced.

THEOREM 3.4. *Let π_1 and π_2 be reduced programs. Then $\pi_1 \equiv \pi_2$ if and only if $D(\pi_1) = D(\pi_2)$.*

Proof. The "if" portion is trivial. Let $\pi_1 = (P_1, I_1, U_1)$ and $\pi_2 = (P_2, I_2, U_2)$, with $\pi_1 \equiv \pi_2$. We shall show that $D(\pi_1) = D(\pi_2)$. It is elementary to argue that $I_1 = I_2$, else, since π_1 and π_2 are reduced, the value of one program involves a variable name not appearing in the value of the other program.

Let $P_1 = S_1; S_2; \cdots; S_k$ and $P_2 = T_1; T_2; \cdots; T_l$. Let f_i, $1 \leqq i \leqq k$, be the value at time i of the variable defined by S_i, and let g_i, $1 \leqq i \leqq l$, be the value at time i of the variable defined by T_i. Then we claim that for all i, $1 \leqq i \leqq k$, there exists j such that $g_j = f_i$, and for all i, $1 \leqq i \leqq l$, there exists j such that $f_j = g_i$. We can prove the first contention by choosing i to be as large as possible such that $f_i \neq g_j$, $1 \leqq j \leqq l$. Certainly, f_i is not in $v(\pi_1)$. Thus, since π_1 is reduced, there exists f_m, $m > i$, such that f_i is a subexpression of f_m. By choice of

i, $f_m = g_n$ for some n. By Lemma 3.6, there exists g_p, $p < n$, such that $g_p = f_i$, contrary to hypothesis.

By the above paragraph and Lemma 3.6, the set $\{f_1, \cdots, f_k\}$ is equal to $\{g_1, \cdots, g_l\}$, $f_i \neq f_j$ for $i \neq j$, $g_i \neq g_j$ for $i \neq j$, and $k = l$. Thus, by Lemma 3.7, we can uniquely pair the interior nodes of $D(\pi_1)$ and $D(\pi_2)$ in such a way that the node of $D(\pi_1)$ corresponding to some statement S_i is paired with the node of $D(\pi_2)$ corresponding to T_j with $f_i = g_j$. By Lemma 3.6, the descendants of paired nodes are also paired, in the correct order. Hence, this pairing is in fact a graph isomorphism showing that $D(\pi_1)$ and $D(\pi_2)$ are the same.

COROLLARY 3.1. *All reduced programs equivalent to a given program have the same DAG.*

COROLLARY 3.2. *If a DAG D is reduced, then every program π such that $D = D(\pi)$ is reduced.*

Proof. Since D is reduced, $D = D(\pi_1)$ for some reduced π_1. If π were not reduced, let π' be a reduced program equivalent to π. By Theorems 3.2 and 3.3, $\pi \equiv \pi_1$, so $\pi' \equiv \pi_1$. By Theorem 3.4, $D(\pi_1) = D(\pi') = D$.

However, we can assume that π' has fewer statements than π, since a reduced program can be obtained from π by applying T1 and T2 in their forward directions as often as possible. By hypothesis, this is possible at least once. Thus, $D(\pi')$ has fewer nodes than $D(\pi) = D$ and $D(\pi') \neq D$. But we have already shown $D(\pi') = D$, and so must conclude that π was reduced.

3.4. Characterization of equivalent programs. We can now prove the central result, that two programs are equivalent if and only if they can be transformed into each other using only T1 and T2.

THEOREM 3.5. *Let π_1 and π_2 be programs. Then $\pi_1 \equiv \pi_2$ if and only if $\pi_1 \overset{*}{\underset{1,2}{\Leftrightarrow}} \pi_2$.*

Proof. The "if" portion is Lemma 3.1. For the "only if" portion, let π'_1 and π'_2 be reduced programs such that $\pi_1 \overset{*}{\underset{1,2}{\Leftrightarrow}} \pi'_1$ and $\pi_2 \overset{*}{\underset{1,2}{\Leftrightarrow}} \pi'_2$. Such programs exist, as we can apply T1 and T2 to any program only a finite number of times since the length of the program decreases with each application. By Lemma 3.1, $\pi'_1 \equiv \pi_1$ and $\pi'_2 \equiv \pi_2$, so $\pi'_1 \equiv \pi'_2$. By Theorem 3.4, $D(\pi'_1) = D(\pi'_2)$. By Theorem 3.3, $\pi' \overset{*}{\underset{3,4}{\Leftrightarrow}} \pi'_2$. Thus, by Theorem 3.1, $\pi_1 \overset{*}{\underset{1,2}{\Leftrightarrow}} \pi_2$.

We have thus provided a method of generating, in a systematic way, all programs equivalent to a given one. We are actually interested not in all equivalent programs, but in an optimal equivalent one, so we shall restrict the sequence of application of transformations further in § 4. There we shall show that under reasonable conditions the sequence of applications of transformations to go from an arbitrary program to an equivalent optimal one is of bounded length and can be made to have a special form.

3.5. Incomplete subsets of the transformations. As an essentially incidental result, we show that no subset of $\{T1, T2, T3, T4\}$ which does not contain T1 and T2 is complete, and, consequently, that T1 and T2 form a minimal complete set of transformations. The following two theorems together embody these statements.

THEOREM 3.6. *There exist equivalent programs π_1 and π_2 such that $\pi_1 \overset{*}{\underset{2,3,4}{\Leftrightarrow}} \pi_2$ is false.*

Proof. Let $\pi_1 = (P_1, \{A, B\}, \{C\})$ and $\pi_2 = (P_2, \{A, B\}, \{C\})$, where $P_1 = C \leftarrow +AB$ and $P_2 = C \leftarrow *AB; C \leftarrow +AB$.

Then, certainly, $v(\pi_1) = v(\pi_2) = \{+AB\}$. Transformations T3 and T4 preserve the number of statements that have a particular operator, *, in this case. Transformation T2 applied in the forward direction does not increase the number of statements with operator *, and in the backward direction increases that number only if the number is not zero to begin with. A formal proof that no sequence of T2, T3 and T4 can transform π_1 into π_2 should now be evident.

THEOREM 3.7. *There exist equivalent programs π_1 and π_2 such that $\pi_1 \overset{*}{\underset{1,3,4}{\Leftrightarrow}} \pi_2$ is false.*

Proof. Let $\pi_1 = (P_1, \{A, B\}, \{C, D\})$ and $\pi_2 = (P_2, \{A, B\}, \{C, D\})$, where P_1 and P_2 are given by:

P_1:	P_2:
$T \leftarrow +AB;$	$C \leftarrow +AB;$
$D \leftarrow *TT;$	$D \leftarrow *CC.$
$C \leftarrow +AB;$	

We shall outline an intuitive argument as to why π_1 cannot be transformed into π_2 by any sequence of T1, T3 and T4.

Consider any program π obtained from π_1 by a sequence of T1, T3 and T4. We must be able to find in the statement of π a sequence of statements of the form $X \leftarrow +AB; \cdots; W \leftarrow *XX$, such that:

(i) W is an output variable, and X is not set between $X \leftarrow +AB$ and $W \leftarrow *XX$.

(ii) There must be a statement $Y \leftarrow +AB$, where Y is an output variable, in this sequence.

(iii) If the statement $Y \leftarrow +AB$ appears between $X \leftarrow +AB$ and $W \leftarrow *XX$, inclusive, then $X \neq Y$.

A formal proof of these assertions is a straightforward induction on the number of transformations applied to go from π_1 to π. Since every π has at least three statements, π_2 cannot be π.

COROLLARY 3.3. *{T1, T2} is minimal complete.*

4. Optimization algorithms. We shall now provide a method for finding an optimal program equivalent to a given one, where "optimal" is defined with respect to any of a wide variety of cost criteria. This method isolates the easy and hard parts of straight line code optimization. Since optimality is defined in terms of a broad class of cost functions, we can only provide universal exhaustive algorithms for the hard parts. For the specific cost function at hand the user should find more efficient algorithms or heuristics for these aspects of the optimization.

DEFINITION. A *cost criterion* C is a function which maps programs to real numbers (the *cost*). We say cost criterion C is *reasonable* if $C(\pi') \leqq C(\pi)$, whenever $\pi \underset{1,2}{\Rightarrow} \pi'$.[1] That is, a cost criterion is reasonable if an application of T1 and T2 in their forward directions does not increase the cost of a program. A program π is *optimal under* cost C if $C(\pi) \leqq C(\pi')$ for any $\pi' \equiv \pi$.

Under the definition of reasonable cost criterion every program which has an equivalent optimal program[2] has an optimal program that is reduced. Thus, given a program π, we can confine our search for an optimal program for π to the class of reduced programs equivalent to π.

DEFINITION. A program is *open* if no two statements set the same variable, and no input variable is ever set.

LEMMA 4.1. *Let* π_1 *be an open program and suppose* $\pi_1 \overset{*}{\underset{3}{\Leftrightarrow}} \pi_2 \underset{4}{\Rightarrow} \pi_3$. *Then* $\pi_1 \underset{4}{\Rightarrow} \pi'_2 \overset{*}{\underset{3}{\Leftrightarrow}} \pi_3$ *for some open program* π'_2.

Proof. Let the statements of π_2 be $P_2 = S_1; \cdots; S_i; S_{i+1}; \cdots; S_n$ and those of π_3 be P_3, where P_3 is P_2 with statements S_i and S_{i+1} interchanged. Let the statements of π_1 be $P_1 = S'_1; \cdots; S'_n$, where P_2 is obtained from P_1 by applications of T3. Let S_i be $A \leftarrow \theta B_1 \cdots B_m$ and S_{i+1} be $C \leftarrow \psi D_1 \cdots D_r$. Then $A \neq C$, $A \neq D_j$ and $C \neq B_j$. Let S'_i be $A' \leftarrow \theta B'_1 \cdots B'_m$ and S'_{i+1} be $C' \leftarrow \psi D'_1 \cdots D'_s$. Define $P'_2 = S'_1; \cdots; S'_{i-1}; S'_{i+1}; S'_i; S'_{i+2}; \cdots; S'_n$, and let π'_2 be π_1 with P_1 replaced by P'_2.

We claim that $\pi_1 \underset{4}{\Rightarrow} \pi'_2$. Clearly, π'_2 is an open program, $A' \neq C'$ and none of $B'_1, \cdots, B'_m$ can be C'. If A' is one of $D'_1, \cdots, D'_r$, then A will be one of $D_1, \cdots, D_r$, no matter where the renaming in the sequence $\pi_1 \overset{*}{\underset{3}{\Leftrightarrow}} \pi_2$ occurred. Thus, the conditions for an application of T4 are fulfilled.

It is elementary, given these inequalities, to show that $\pi'_2 \overset{*}{\underset{3}{\Leftrightarrow}} \pi_3$ by renaming the same variables in the same way as was done in the transformation $\pi_1 \overset{*}{\underset{3}{\Leftrightarrow}} \pi_2$. The details are omitted.

LEMMA 4.2. *Let* π_1 *be an open program and suppose* $\pi_1 \overset{*}{\underset{3,4}{\Leftrightarrow}} \pi_3$. *Then there is an open program* π_2 *such that* $\pi_1 \overset{*}{\underset{4}{\Leftrightarrow}} \pi_2 \overset{*}{\underset{3}{\Leftrightarrow}} \pi_3$.

Proof. We observe that T3 and T4 are invertible, that is, $\pi \underset{3,4}{\Rightarrow} \pi'$ if and only if $\pi' \underset{3,4}{\Rightarrow} \pi$. Thus we can find a sequence $\pi_1 = \pi^{(0)} \underset{3,4}{\Rightarrow} \pi^{(1)} \underset{3,4}{\Rightarrow} \cdots \underset{3,4}{\Rightarrow} \pi^{(r)} = \pi_3$. Let k of these r steps be by T4. We show by induction on k that we can find π_2 such that $\pi_1 \overset{*}{\underset{4}{\Leftrightarrow}} \pi_2 \overset{*}{\underset{3}{\Leftrightarrow}} \pi_3$. The basis, $k = 0$ is trivial.

Assume the hypothesis is true for $k - 1$, where $k > 0$. Without loss of generality, we can find $j \geqq 0$ such that $\pi^{(0)} \overset{*}{\underset{3}{\Leftrightarrow}} \pi^{(j)} \underset{4}{\Rightarrow} \pi^{(j+1)} \overset{*}{\underset{3,4}{\Leftrightarrow}} \pi^{(r)}$. By Lemma 4.1, there exists an open π_4 such that $\pi^{(0)} \underset{4}{\Rightarrow} \pi_4 \overset{*}{\underset{3}{\Leftrightarrow}} \pi^{(j+1)}$. By the inductive hypothesis, there is some π_2 such that $\pi_4 \overset{*}{\underset{4}{\Leftrightarrow}} \pi_2 \overset{*}{\underset{3}{\Leftrightarrow}} \pi_3$. Thus, $\pi_1 \overset{*}{\underset{4}{\Leftrightarrow}} \pi_2 \overset{*}{\underset{3}{\Leftrightarrow}} \pi_3$.

We shall now give a characterization of optimization procedures under reasonable cost criteria.

---

[1] It would seem that "reasonable" in the formal sense would include all cost criteria which were "reasonable" in the intuitive sense. However, M. D. McIlroy has pointed out a situation where this is not so. Suppose we are generating code for a computer with three very fast accumulators, and our program requires constant use of the values A, B and $A + B$. It might be advantageous to use two accumulators to store A and B, and compute $A + B$ whenever it was needed. Eliminating redundant computations of $A + B$ could actually increase the cost. Nevertheless, we feel "reasonable" is still a pretty reasonable assumption.

[2] In bizarre cases a program need not have an equivalent optimal program. For example, consider the "reasonable" cost criterion where the cost of a program is $1/n$ where n is the length of the longest variable name appearing in a useful, irredundant statement of the program. Given a program π, no optimal program for π exists.

THEOREM 4.1. *Let π_1 be any program. Then there exists a program π_2 equivalent to π_1 such that for any reasonable cost criterion C such that π_1 has an equivalent optimal program under C, there exist programs π_3 and π_4 where:*

(i) $\pi_2 \overset{*}{\underset{4}{\Leftrightarrow}} \pi_3$,

(ii) $\pi_3 \overset{*}{\underset{3}{\Leftrightarrow}} \pi_4$, *and*

(iii) π_4 *is optimal under C.*

Proof. Let π'_1 be a reduced program equivalent to π_1. Using T3, we can transform π'_1 into π_2, an open program equivalent to π_1 and π'_1. By Theorem 3.3, $D(\pi_2) = D(\pi'_1)$. By Corollary 3.2, π_2 is reduced.

If an optimal program for π_1 exists, we know we can find a reduced optimal program π_4 equivalent to π_1. Thus, by Theorem 3.4, $D(\pi_4) = D(\pi_2)$. By Theorem 3.3, $\pi_2 \overset{*}{\underset{3,4}{\Leftrightarrow}} \pi_4$. By Lemma 4.2, there exists π_3 such that $\pi_2 \overset{*}{\underset{4}{\Leftrightarrow}} \pi_3 \overset{*}{\underset{3}{\Leftrightarrow}} \pi_4$.

Intuitively, what is implied by Theorem 4.1 is that if an optimal program exists under any reasonable cost criterion, we can find one by first removing useless and redundant statements, then applying some techniques to reorder the statements and, if necessary, rename variables. The first task, removing useless and redundant statements can be quickly done in a straightforward manner.[3]

Often, the names of the variables are unimportant, so only statement order counts. An example occurs when cost is the speed of the resulting program. At other times, only the names are important, for example if cost is the number of different locations needed for temporary storage. In these cases, part of the difficulty of optimizing disappears; that is, we can go from π_2 to an optimal program using only T3 or only T4.

In any case, the problem of considering all programs accessible from another using only T3 or T4 is a finite but large combinatorial problem. It is the force of Theorem 4.1 that one can restrict oneself to finding efficient algorithms (or heuristics) to search for good choices for π_3 or π_4. That this is the case is probably intuitively obvious. However, it is comforting to know that our intuition can be backed up by formal analysis in this case.

Example 4.1. We shall postulate a computer with a single accumulator. The operations which the machine can do are the following:

1. LOAD a memory address into the accumulator.
2. STORE the accumulator into a memory address.
3. If θ is an n-ary operator, it may apply operation θ with the first argument found in the accumulator and the remaining $n - 1$ arguments found in designated memory addresses.

The statement $A \leftarrow \theta B_1 \cdots B_n$ can be executed on such a machine by the following sequence of instructions:

(a) If the value of B_1 is not already in the accumulator, LOAD B_1. If already there, do nothing.

[3] Elgot has pointed out that the process can be likened exactly to the process of reducing a finite automaton. The DAG can be thought of as a transition diagram, with each node a state, transitions moving down the DAG, and the distinguished nodes as initial states. Removal of useless statements thus becomes removal of a state which is inaccessible from any initial state; removal of redundancy is merger of equivalent states.

(b) Apply operation θ to the accumulator and the locations of $B_2, \cdots, B_n$.

(c) If necessary, STORE the accumulator.

The cost of executing the statement $A \leftarrow \theta B_1 \cdots B_n$ can thus be 1, 2 or 3. It is 3 if B_1 is not found in the accumulator, and there is a subsequent reference to this computation of A that is not the first argument of the next statement (i.e., A has to be stored). It is 1 if B_1 is found in the accumulator and there is no reference to this computation of A other than as the first argument of the next statement. The cost is 2 if one, but not both of the above conditions hold.

Let us consider the following program, which might be obtained from the FORTRAN statements

$$F = (A + B) * (A - B),$$

$$G = (A - B) * (A - C) * (B - C).$$

The resulting program is $\pi_1 = (P, \{A, B, C\}, \{F, G\})$, where P is

$$T \leftarrow +AB;$$
$$S \leftarrow -AB;$$
$$F \leftarrow *TS;$$
$$T \leftarrow -AB;$$
$$S \leftarrow -AC;$$
$$R \leftarrow -BC;$$
$$T \leftarrow *TS;$$
$$G \leftarrow *TR.$$

We note one instance of redundancy, between the second and fourth statements. We can eliminate this redundancy and then make the program open. The output variables become X_3 and X_7. Call the resulting program π_2.

$$X_1 \leftarrow +AB;$$
$$X_2 \leftarrow -AB;$$
$$X_3 \leftarrow *X_1X_2;$$
$$X_4 \leftarrow -AC;$$
$$X_5 \leftarrow -BC;$$
$$X_6 \leftarrow *X_2X_4;$$
$$X_7 \leftarrow *X_6X_5.$$

The DAG for the above program is shown in Fig. 4.1. Node n_i is created from the statement which sets X_i.

While the problem of this example is simple enough to enumerate all possible orderings of π_2, we cannot afford the time to do this for an arbitrary program.

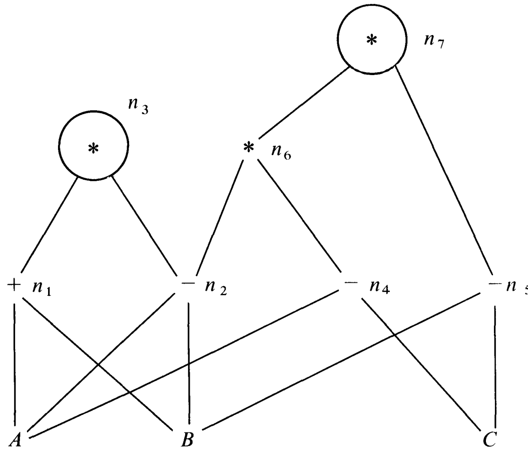


FIG. 4.1. *DAG for example program*

Some heuristic that will produce good, although not necessarily optimal, orderings quickly is needed. We propose one here. This algorithm produces an ordering of the nodes of a DAG. The desired program has statements corresponding to these nodes in reverse order. We express the algorithm as follows.

1. We construct a list L. Initially L is empty.

2. Choose a node n of the DAG such that n is not on L, and all n's ancestors are on L. Add n to L. If no such n exists, halt.

3. If n_1 is the last node added to L, the left-most descendant of n_1 is an interior node n not on L, and all n's ancestors are on L, add n to L and repeat step 3. Otherwise go to step 2.

For example, using the DAG of Fig. 4.1, we might begin with $L = n_3$. By step 3, we would add n_1 to L. Then we could choose n_7, add it to $L$and follow it by n_6 and n_2. Two uses of rule 2 would add n_4 and n_5, so L is $n_3, n_1, n_7, n_6, n_2, n_4, n_5$. Recalling that the statement defining X_i creates node n_i, and that the list L corresponds to the statements in reverse, we obtain the following program: $\pi_3 = (P', \{A, B, C\}, \{X_3, X_7\})$, where P' is

$$X_5 \leftarrow -BC;$$

$$X_4 \leftarrow -AC;$$

$$X_2 \leftarrow -AB;$$

$$X_6 \leftarrow *X_2X_4;$$

$$X_7 \leftarrow *X_6X_5;$$

$$X_1 \leftarrow +AB;$$

$$X_3 \leftarrow *X_1X_2.$$

It is easy to check that $\pi_2 \overset{*}{\underset{4}{\Leftrightarrow}} \pi_3$. No application of T3 changes the cost.

The assembly language programs obtained from π_2 and π_3 are shown in Fig. 4.2.

(a) From π_2	(b) From π_3
LOAD A	LOAD B
ADD B	SUBTR C
STORE X_1	STORE X_5
LOAD A	LOAD A
SUBTR B	SUBTR C
STORE X_2	STORE X_4
LOAD X_1	LOAD A
MULT X_2	SUBTR B
STORE X_3	STORE X_2
LOAD A	MULT X_4
SUBTR C	MULT X_5
STORE X_4	STORE X_7
LOAD B	LOAD A
SUBTR C	ADD B
STORE X_5	MULT X_2
LOAD X_2	STORE X_3
MULT X_4	
MULT X_5	
STORE X_7	

FIG. 4.2. *Assembly language programs*

It is easy, incidentally, to show that π_3 is optimal under our cost criterion. The following reasoning suffices.

1. Any program whose DAG in Fig. 4.1 has at least four LOADs, because each of n_1, n_2, n_4 and n_5 requires its left operand be in the accumulator immediately before the value of the node is computed.

2. Every interior node which is either distinguished or other than the leftmost descendant of some node must be stored. Thus, in Fig. 4.1, n_3, n_7, n_2, n_4 and n_5, a total of five nodes must be STOREd.

3. There are seven interior nodes in Fig. 4.1; hence at least seven machine arithmetic instructions occur.

4. The assembly program of Fig. 4.2(b) meets each of these bounds exactly.

Of course, these simple bounds may not be attainable for an arbitrary DAG, and optimization under this cost criterion is a large combinatorial problem.

5. Summary. We have given a set of transformations capable of transforming a straight line program into any other equivalent one. These are:

T1. Removal (or insertion) of useless statements,

T2. Removal (or insertion) of redundant statements,

T3. Renaming of variables,

T4. Flipping of independent statements.

T1 and T2 alone are sufficient to characterize equivalent programs, although no set not containing these is sufficient.

We showed that programs can be represented graphically by directed acyclic graphs (DAGs), and that two programs have the same DAG if and only if they can be transformed into one another by T3 and T4.

The concept of a reasonable cost criterion, one that does not increase when T1 and T2 are applied to remove statements, was introduced. An optimization procedure under any reasonable criterion can be expressed in three steps, the

first of which is independent of the cost criterion. Of the second and third, one or the other will often be unnecessary.

1. Remove redundant and useless statements by T1 and T2; then rename variables by T3 until no variable is set more than once or is set and is also an input variable.

2. Reorder statements by T4.

3. Rename variables by T3.

For a given program, all equivalent programs optimal under any reasonable cost criterion have the same DAG.

6. Concluding remarks. There are several directions for further work. One area is to focus on specific cost criteria in order to provide efficient algorithms for the reordering of computations and renaming of variables. One example of this occurs in [17], where an algorithm for generating optimal machine code for certain arithmetic expressions is presented. This algorithm provides an efficient method for choosing the order of the computations in an arithmetic expression minimizing both the length of the output code and the number of temporary stores required.

We can also consider optimization knowing that certain algebraic identities such as the associative or commutative laws hold among certain operators and operands. These algebraic identities may often expose new common subexpressions which would not arise using the four transformations considered here.

Unfortunately, it is well known that under some sets of algebraic identities it is recursively undecidable to determine whether two programs are equivalent, even for the rudimentary model of programs we are considering here. However, it is possible to extend Theorem 4.1 to apply to optimization problems in the presence of certain types of operator and operand preserving algebraic identities [16].

Finally, the basic model should be extended to encompass larger classes of programs. Along these lines the authors have considered the equivalence of programs when structured variables such as array variables are added [18].

Acknowledgment. The authors would like to express their appreciation to Calvin C. Elgot, Douglas McIlroy and Donald M. Kaplan for their helpful comments. We also would like to thank the referees for their perceptive comments on the original manuscript.

REFERENCES

[1] M. S. PATERSON, *Program schemata*, Machine Intelligence, 3 (1968), pp. 19–32.

[2] D. C. LUCKHAM, D. M. R. PARK AND M. S. PATERSON, *On formalized computer programs*, J. Comput. System Sci., 4 (1970), pp. 220–249.

[3] F. E. ALLEN, *Program Optimization*, Annual Review in Automatic Programming, vol. 5, Pergamon Press, New York, 1969.

[4] E. LOWRY AND C. W. MEDLOCK, *Object code optimization*, Comm. ACM, 12 (1969), pp. 13–22.

[5] V. A. BUSAM AND D. E. ENGLAND, *Optimization of expressions in Fortran*, Ibid., 12 (1969), pp. 666–674.

[6] J. NIEVERGELT, *On the automatic simplification of computer programs*, Ibid., 8 (1965), pp. 366–370.

[7] M. D. MCILROY, Unpublished manuscript.

[8] J. COCKE AND J. T. SCHWARTZ, *Programming Languages and Their Compilers*, Courant Institute, New York University, New York, 1970.
[9] I. I. IANOV, *On the logical schemata of algorithms*, Problems of Cybernetics, 1 (1958), pp. 75–127.
[10] J. D. RUTLEDGE, *On Ianov's program schemata*, J. Assoc. Comput. Mach., 11 (1964), pp. 1–9.
[11] S. K. BASU, *Transformations of program schemes to standard forms*, Conference Record Ninth Annual Symposium on Switching and Automata Theory 1968, pp. 99–105.
[12] T. ITÔ, *Some formal properties of a class of nondeterministic program schemata*, Ibid., pp. 85–98.
[13] D. M. KAPLAN, *Regular expressions and the equivalence of programs*, J. Comput. System Sci., 3 (1968), pp. 361–386.
[14] S. IGARISHI, *An axiomatic approach to the equivalence problems of algorithms with applications*, Report of the Computer Center, Univ. of Tokyo, 1 (1968), pp. 1–101.
[15] J. W. DE BAKKER, *Axiomatics of simple assignment statements*, Symposium on Sematics of Algorithmic Languages, E. Engeler, ed., Springer-Verlag, Berlin, 1970, pp. 1–22.
[16] A. V. AHO AND J. D. ULLMAN, *Transformations on straight line programs*, Conference Record Second Annual ACM Symposium on Theory of Computing, 1970, pp. 136–148.
[17] R. SETHI AND J. D. ULLMAN, *The generation of optimal code for arithmetic expressions*, J. Assoc. Comput. Mach., 17 (1970), pp. 715–728.
[18] A. V. AHO AND J. D. ULLMAN, *Equivalence of programs with structured variables*, Conference Record Eleventh Annual Symposium on Switching and Automata Theory, 1970, pp. 25–31.

SIAM J. COMPUT.
Vol. 1, No. 1, March 1972

TERMINAL CONTEXT IN CONTEXT-SENSITIVE GRAMMARS*

RONALD V. BOOK†

Abstract. If every non-context-free rewriting rule of a context-sensitive (with erasing) grammar has as left context a string of terminal symbols and the left context is at least as long as the right context, then the language generated is context-free. If every non-context-free rewriting rule of a context-sensitive (with erasing) grammar has as left and right context strings of terminal symbols, then the language generated is context-free.

Key words. context-sensitive grammar, context-free language, terminal context, "messages," "barriers"

Introduction. It is well known that there exist context-sensitive grammars which generate languages which are not context-free and that it is undecidable whether a context-sensitive grammar generates a context-free language. However, the mechanism by which the use of context allows a non-context-free language to be generated is not well understood (in fact, the question itself is vague: what does context do for you?). In this paper it is shown that if certain nontrivial constraints are placed on the form of the rules of a context-sensitive (with erasing) grammar, then only a context-free language will be generated. These constraints involve the use of terminal strings as part of context. The first restriction (Theorem 1) is that for every non-context-free rule, the left context is a string of terminal symbols which is at least as long as the (arbitrary) right context. (It is shown that the length restriction cannot be dropped.) The second restriction (Theorem 2) is that both left and right context be strings of terminal symbols.

If one is constructing a context-sensitive grammar to generate some non-context-free language, then one often proceeds as if context can be used to "store and transmit" information. Thus one builds rules so that "messages" or "pulses" are transmitted along a string in the course of the derivation. Sometimes this effect is achieved by building a grammar which imitates the action of a Turing machine; hence, the action of the read–write head must be imitated as it travels back and forth across the tape. Notions of "connectivity" in derivations and in grammars reflect this action, and restrictions on context sometimes alter the structure of derivations (see [2], [7], [8]).

The "ability to send messages" has not been formalized in such a way as to explain "what context does for you." However, this notion does provide an intuitive "handle" for studying some questions and for gaining perspective on some results concerning context-sensitive (with or without erasing) grammars and

* Received by the editors August 31, 1971, and in revised form November 8, 1971.

† Center for Research in Computing Technology, Division of Engineering and Applied Physics, Harvard University, Cambridge, Massachusetts 02138. This research was supported in part by the National Aeronautics and Space Administration under Grant NGR-22-007-176 and by the National Science Foundation under Grant GJ-803. Part of this research was performed while the author was visiting the Department of System Science, University of California, Los Angeles.

languages.[1] Let us consider two examples:

(i) Ginsburg and Greibach [5] have shown that if $G = (V, \Sigma, R, X)$ is a grammar such that every rule in R is of the form $\rho \to \theta$, where $\rho \in (V - \Sigma)^*$ and $\theta \in V^*\Sigma V^*$, then $L(G)$ is context-free. Thus the left-hand side of a rule cannot contain any terminal symbol and at each step of a derivation at least one new terminal symbol is generated. In this way a "barrier" is erected at each step and no message can cross the barrier since the left-hand side of a rule contains only nonterminal symbols. Thus messages can be transmitted only a bounded distance.

(ii) Hibbard [9] has shown that if $G = (V, \Sigma, R, X)$ is a grammar and $<$ is a partial order on V with the property that for every rule $Z_1 \cdots Z_p \to Y_1 \cdots Y_q$ in R, there exists $Y \in \{Y_1, \cdots, Y_q\}$ such that for every $Z \in \{Z_1, \cdots, Z_p\}$, $Z < Y$, then $L(G)$ is context-free. Thus no message can cross a maximal symbol so that a maximal symbol serves as a barrier when generated. The condition on the rules and the finiteness of V restrict the number of applications of rules in a given region since a maximal symbol must be generated after a bounded number of steps in a derivation. Thus messages can be transmitted only a bounded distance and only a bounded number of messages can be transmitted past any given symbol.

The results established in this paper may be interpreted as constraining the "message-sending" capacity by means of strings of terminal symbols which act as barriers when used as context. In the case of Theorem 1, this allows one to show that all terminal strings generated can be obtained from derivations which are left-to-right with bounded "lookahead." In the case of Theorem 2, this allows a series of reductions which eliminate all context. Thus both results are obtained from first principles. Also, neither result appears to imply or be implied by the two results cited above.

1. It is assumed that the reader is familiar with the basic facts of context-free grammars and languages, regular sets, and gsm mappings (see [4]). However, there are certain conventions which need to be emphasized here.

A *grammar* is a quadruple $G = (V, \Sigma, R, X)$, where V is a finite set of symbols, $\Sigma \subset V$ is the set of *terminal* symbols, $X \in V - \Sigma$, and R is a finite set of *rewriting rules* (*productions*) of the form $\alpha_1 y_1 \cdots \alpha_n y_n \alpha_{n+1} \to \alpha_1 w_1 \cdots \alpha_n w_n \alpha_{n+1}$ with each $\alpha_i \in \Sigma^*$, $y_i \in (V - \Sigma)^*(V - \Sigma)$, $w_i \in V^*$, and for some i, $w_i \neq y_1$.[2] If $\rho \to \theta \in R$, then for any $\alpha, \beta \in V^*$, write $\alpha\rho\beta \Rightarrow \alpha\theta\beta$ and say that the rule $\rho \to \theta$ is *applicable* to the string $\alpha\rho\beta$ and that $\rho \to \theta$ *transforms* $\alpha\rho\beta$. A *derivation* in G is a sequence $\Gamma_0, \Gamma_1, \cdots, \Gamma_n \in V^*$ such that for $i = 1, \cdots, n$, $\Gamma_{i-1} \Rightarrow \Gamma_i$. The transitive reflexive closure of $\Rightarrow$ is $\overset{*}{\Rightarrow}$. The *language generated by* G is $L(G) = \{w \in \Sigma^* | X \overset{*}{\Rightarrow} w\}$.

If $G = (V, \Sigma, R, X)$ is a grammar and $\Gamma_0 \Rightarrow \Gamma_1 \Rightarrow \cdots \Rightarrow \Gamma_n$ is a derivation in G, then a *production sequence* [7] for this derivation is a sequence of n ordered pairs, $\{\langle (B_i, P_i, C_i), (B_i, Q_i, C_i)\rangle\}_{i=1}^n$, where for each $i = 1, \cdots, n$, $B_i P_i C_i = \Gamma_{i-1}$, $B_i Q_i C_i = \Gamma_i$, and $P_i \to Q_i \in R$.

[1] There are two other approaches to these problems. In [3], [10]–[12] it is shown that for an arbitrary grammar certain types of derivations yield only context-free languages. Also, one can consider "regulating" the application of rewriting rules, such as in matrix grammars, programmed grammars, etc. See [1], [6], [13], [14], [15].

[2] For any set T of symbols, T^* is the free semigroup with identity e generated by T.

A grammar $G = (V, \Sigma, R, X)$ is *type* O (or *context-sensitive with erasing*) if each rule in R is of the form $\alpha Z\beta \to \alpha\gamma\beta$, $\alpha, \beta, \gamma \in V^*$, $Z \in V - \Sigma$, where α is the left context and β is the right context. When the rule $\alpha Z\beta \to \alpha\gamma\beta$ is applied to a string $y_1\alpha Z\beta y_2$ to yield $y_1\alpha\gamma\beta y_2$, Z is the *transformed symbol* of $y_1\alpha Z\beta y_2 \Rightarrow y_1\alpha\gamma\beta y_2$. It is well known that a set L is recursively enumerable if and only if there is a type O grammar G such that $L(G) = L$.

A grammar $G = (V, \Sigma, R, X)$ is *context-free* if each rule in R is of the form $Z \to \gamma$, $Z \in V - \Sigma$, $\gamma \in V^*$. (In any grammar a rule of this form is called a context-free rule.) A language L is context-free if and only if there is a context-free grammar G such that $L(G) = L$.

The length of a string w is denoted by $|w|$.

2. In this section we establish Theorem 1 below and show that the hypothesis cannot be weakened.

THEOREM 1. *Let $G = (V, \Sigma, R, X)$ be a type* O *grammar. If each non-context-free rule in R is of the form $\alpha Z\beta \to \alpha\gamma\beta$ where $\alpha \in \Sigma^*$, $Z \in V - \Sigma$, $\beta, \gamma \in V^*$, and $|\alpha| \geqq |\beta|$, then $L(G)$ is context-free.*

A non-context-free rule in R has left context which is a terminal string and right context which is no longer than the left (no other restriction is placed on the right context). We shall show that these two restrictions imply that a "message" cannot be "transmitted" to the left over a string of symbols longer than $1 + m(m + 1)/2$, where $m = \max\{|\alpha| \,|\alpha Z\beta \to \alpha\gamma\beta \in R\}$. Thus any $w \in L(G)$ can be generated by a derivation such that at each step the transformed symbol is no farther than $m(m + 1)/2$ from the leftmost nonterminal symbol in the string being transformed—hence, $L(G)$ is context-free. In order to prove this, we first state some definitions and review some facts about grammars and languages.

Let $G = (V, \Sigma, R, X)$ be a type O grammar. For $\Gamma, \Psi \in V^*$, $t \geqq 1$, if $\Gamma \Rightarrow \Psi$, where $\Gamma = \alpha Z \cdots Y\beta$, Z is the left-most nonterminal symbol in Γ (i.e., $\alpha \in \Sigma^*$ and $Z \in V - \Sigma$), Y is the transformed symbol in $\Gamma \Rightarrow \Psi$, and $|Z \cdots Y| \leqq t$, then $\Gamma \Rightarrow \Psi$ is *t bounded*. If $\Gamma \Rightarrow \Psi$ is t bounded, then it is r bounded for every $r \geqq t$. A derivation is a *t bounded derivation* if each step is t bounded. For any $t \geqq 1$, $\mathrm{LEFT}(t, G) = \{w \in \Sigma^* |$ there is a t-bounded derivation $X \Rightarrow \cdots \Rightarrow w$ in $G\}$. Clearly, for every $t \geqq 1$, $\mathrm{LEFT}(t, G) \subseteq L(G)$. In [3], [10] it is shown that for any grammar G and any $t \geqq 1$, $\mathrm{LEFT}(t, G)$ is a context-free language.

Notation. For any $t \geqq 1$, let $M(t) = t(t + 1)/2$. For a type O grammar $G = (V, \Sigma, R, X)$, let $m = \max\{|\alpha| \,|\alpha Z\beta \to \alpha\gamma\beta \in R\}$ and let $M_G = 1 + M(m)$.

If G satisfies the hypothesis of Theorem 1, then for any rule $\alpha Z\beta \to \alpha\gamma\beta \in R$, $|\beta| \leqq M(|\beta|) < M_G$ since $|\alpha| \geqq |\beta|$. To prove Theorem 1 we shall show that $\mathrm{LEFT}(M_G, G) = L(G)$ so that $L(G)$ is context-free because $\mathrm{LEFT}(M_G, G)$ is context-free.

The formal argument proving Theorem 1 is based on the following observations:

(i) Consider $\mu\alpha Z x_1 \cdots x_n$, where $\mu, \alpha \in \Sigma^*$. Suppose one wishes to apply the rule $\alpha Z\beta \to \alpha\gamma\beta$ in order to transform Z. Consider $x_1 \cdots x_n$. If β is a prefix of $x_1 \cdots x_n$, then this rule can be applied. If not, then other rules must be applied to $x_1 \cdots x_n$ in order to obtain a string with β as prefix.

(ii) Suppose that $\beta = x_1 \cdots x_q\delta$, for some $\delta \neq e$, where $q < n$, and that some rule must be applied to $x_1 \cdots x_n$ in order to transform x_{q+1} so that eventually β

is obtained as a prefix. Further, suppose that the rule to be applied is a context-sensitive rule, so that its left context is a suffix of $x_1 \cdots x_q$ (since $Z \in V - \Sigma$, Z cannot occur as part of terminal left context). Then the rule to be applied must have left context which is shorter than $q + 1 \leqq |\beta|$. But $|\alpha| \geqq |\beta|$ so that the left context of this rule is shorter than α.

(iii) By induction on $|\beta|$, it is seen that if it is possible to apply rules to $x_1 \cdots x_n$ in order to obtain a string with β as prefix, then it is possible to do this by transforming symbols that are no farther than $1 + 2 + \cdots + |\alpha|$ from Z, so that each step in the resulting derivation is $M(|\alpha|) + 1$ bounded. Since $M_G \geqq 1 + M(|\alpha|)$, the resulting derivation is M_G bounded.[3]

The result of these observations is argued formally through two lemmas (the proof of Lemma 1 containing the main argument). The theorem then follows easily.

LEMMA 1. *Let $G = (V, \Sigma, R, X)$ be a type O grammar which satisfies the hypothesis of Theorem 1. For any $\beta \in V^*$ and any $n \geqq 1$, if $\Gamma_0 \Rightarrow \cdots \Rightarrow \Gamma_n$ is a derivation of length n in G such that β is not a prefix of Γ_0 but is a prefix of Γ_n, then there exist $\Pi_0, \cdots, \Pi_n \in V^*$ such that $\Pi_0 = \Gamma_0$, $\Pi_n = \Gamma_n$, $\Pi_0 \Rightarrow \cdots \Rightarrow \Pi_n$ is a derivation of length n in G, and the step $\Pi_0 \Rightarrow \Pi_1$ is $M(|\beta|)$ bounded.*

Proof. Since β is not a prefix of Γ_0, $\beta \neq e$. The proof proceeds by induction on $|\beta|$.

(i) For any $n \geqq 1$, let $\Gamma_0 \Rightarrow \cdots \Rightarrow \Gamma_n$ be a derivation of length n in G, let $\{\langle (B_i, P_i, C_i), (B_i, Q_i, C_i) \rangle\}_{i=1}^n$ be a production sequence for this derivation, and let $\Gamma_0 = Y_1 \cdots Y_t$, $t \geqq 1$, each $Y_i \in V$. If $|\beta| = 1$, then $\beta \in V$. Since β is not a prefix of $\Gamma_0 = Y_1 \cdots Y_t$, $\beta \neq Y_1$. But β is a prefix of Γ_n so that there is some step in $\Gamma_0 \Rightarrow \cdots \Rightarrow \Gamma_n$ which transforms Y_1. Hence, $Y_1 \in V - \Sigma$. Thus Y_1 does not occur as part of the left context of any rule in R (since such context is in Σ^*). Let $\Gamma_{k-1} \Rightarrow \Gamma_k$ be the first step which transforms Y_1. Since Y_1 is the left-most symbol in Γ_0 and since $\Gamma_{k-1} \Rightarrow \Gamma_k$ is the first step which transforms Y_1, for each $i = 1, \cdots, k$, Y_1 is the left-most symbol of $\Gamma_{i-1} = B_i P_i C_i$. Thus Y_1 cannot be used as part of the right context for any of the rules $P_i \to Q_i$, $1 \leqq i \leqq k - 1$. Hence for each $i = 1, \cdots, k - 1$, $B_i = Y_1 D_i$ for some $D_i \in V^*$, $B_k = e$, and $P_k \to Q_k$ is a context-free rule with $P_k = Y_1$. Thus the derivation $\Pi_0 \Rightarrow \cdots \Rightarrow \Pi_n$ can be constructed by first applying $P_k \to Q_k$ to Y_1 in Γ_0, then imitating the derivations $\Gamma_0 \Rightarrow \cdots \Rightarrow \Gamma_{k-1}$ and $\Gamma_k \Rightarrow \cdots \Rightarrow \Gamma_n$. Since Y_1 is transformed by $\Pi_0 \Rightarrow \Pi_1$, this step is $M(|\beta|) = 1$ bounded.

(ii) Assume the result for all $\beta \in V^*$ such that $|\beta| < r$ for some r > 1, all $n \geqq 1$, and all derivations of length n in G.

(iii) Consider $\beta \in V^*$ such that $|\beta| = r$ and consider a derivation $\Gamma_0 \Rightarrow \cdots \Rightarrow \Gamma_n$ of length n in G such that β is not a prefix of Γ_0 and β is a prefix of Γ_n. Let $\{\langle (B_i, P_i, C_i), (B_i, Q_i, C_i) \rangle\}_{i=1}^n$ be a production sequence for this derivation. Let $\Gamma_0 = Y_1 \cdots Y_t$, $t \geqq 1$, for each $Y_i \in V$. If some step of $\Gamma_0 \Rightarrow \cdots \Rightarrow \Gamma_n$ transforms Y_1, then the argument is just as in (i). Suppose no step of $\Gamma_0 \Rightarrow \cdots \Rightarrow \Gamma_n$ transforms any part of $Y_1 \cdots Y_{q-1}$ for some $q \leqq t$ but some step transforms Y_q. Since β is not a prefix of $\Gamma_0 = Y_1 \cdots Y_t$, this implies $q \leqq |\beta| = r$. Since Y_q is transformed at

[3] One of the referees has pointed out that in the proof of Lemma 1 it is sufficient for M to be any function such that for $t \geqq 1$, $t + M(t - 1) \leqq M(t)$. The function we have used is the least function with this property although a function such as $M(t) = t^2$ will yield the result.

some step, $Y_q \in V - \Sigma$. Let $\Gamma_{k-1} \Rightarrow \Gamma_k$ be the first step such that Y_q is transformed. Since no part of $Y_1 \cdots Y_{q-1}$ is transformed in $\Gamma_0 \Rightarrow \cdots \Rightarrow \Gamma_n$ and $Y_q \in V - \Sigma$, no part of $Y_1 \cdots Y_q$ can serve as part of the left context in any step of $\Gamma_0 \Rightarrow \cdots \Rightarrow \Gamma_{k-1}$.

If the rule applied in $\Gamma_{k-1} \Rightarrow \Gamma_k$ is a context-free rule, then the argument is just as in (i). If the rule used in $\Gamma_{k-1} \Rightarrow \Gamma_k$ is a non-context-free rule, say $\alpha_1 Y_q \alpha_2 \rightarrow \alpha_1 \gamma \alpha_2$, then by choice of q, α_1 must be a suffix of $Y_1 \cdots Y_{q-1}$, and so $|\alpha_1| \leqq q - 1 < q \leqq |\beta| = r$. Thus $\Gamma_{k-1} = B_k P_k C_k$, where $P_k = \alpha_1 Y_q \alpha_2$ and $B_k \alpha_1 = Y_1 \cdots Y_{q-1}$. If $B_k \alpha_1 Y_q \alpha_2$ is a prefix of Γ_0, then the argument is just as in (i), applying $\alpha_1 Y_q \alpha_2 \rightarrow \alpha_1 \gamma \alpha_2$ first and noticing that this step is $q \leqq |\beta| < M(|\beta|)$ bounded.

If $B_k \alpha_1 Y_q \alpha_2$ is not a prefix of Γ_0, then by choice of q the application of rules in $\Gamma_0 \Rightarrow \cdots \Rightarrow \Gamma_{k-1}$ transforms no symbols in $\Gamma_0 = Y_1 \cdots Y_t$ to the left of Y_{q+1}, and these steps use no part of $B_k \alpha_1 Y_q = Y_1 \cdots Y_q$ as context. Thus for each $i = 1, \cdots, k - 1$, $B_k \alpha_1 Y_q = Y_1 \cdots Y_q$ is a prefix of B_i, say $B_i = B_k \alpha_1 Y_q D_i$ for some $D_i \in V^*$. For each $i = 1, \cdots, k - 2$, let $\Delta_i \in V^*$ be defined by $\Gamma_{i-1} = B_k \alpha_1 Y_q \Delta_i$, so that $\Delta_i = D_i P_i C_i$, and let $\Delta_{k-1} = D_{k-1} Q_{k-1} C_{k-1}$. Then $\Delta_0 \Rightarrow \cdots \Rightarrow \Delta_{k-1}$ is a derivation of length $k - 1$ in G (with production sequence $\{\langle (D_i, P_i, C_i), (D_i, Q_i, C_i) \rangle\}_{i=1}^{k-1}$) such that α_2 is not a prefix of Δ_0 but α_2 is a prefix of Δ_{k-1}. Since $\alpha_1 Y_q \alpha_2 \rightarrow \alpha_1 \gamma \alpha_2$ is a rule in R, $|\alpha_1| \geqq |\alpha_2|$. Also, $|\alpha_1| < r$. Hence, $|\alpha_2| < r$ and $k - 1 < n$ so that the induction hypothesis applies to the derivation $\Delta_0 \Rightarrow \cdots \Rightarrow \Delta_{k-1}$ and the string α_2, that is, there is a derivation $\nabla_0 \Rightarrow \cdots \Rightarrow \nabla_{k-1}$ of length $k - 1$ in G such that $\nabla_0 = \Delta_0$, $\nabla_{k-1} = \Delta_{k-1}$, and the step $\nabla_0 \Rightarrow \nabla_1$ is $M(|\alpha_2|)$ bounded.

Construct the derivation $\Pi_0 \Rightarrow \cdots \Rightarrow \Pi_n$ as follows. For each $i = 1, \cdots, k - 1$, let $\Pi_i = B_k \alpha_1 Y_q \nabla_i$, so that $\Pi_0 \Rightarrow \cdots \Rightarrow \Pi_{k-1}$. Now

$$\Pi_0 = B_k \alpha_1 Y_q \nabla_0 = B_k \alpha_1 Y_q \Delta_0 = B_k \alpha_1 Y_q D_1 P_1 C_1 = B_1 P_1 C_1 = \Gamma_0$$

and

$$\begin{aligned} \Pi_{k-1} &= B_k \alpha_1 Y_q \nabla_{k-1} = B_k \alpha_1 Y_q \Delta_{k-1} = B_k \alpha_1 Y_q D_{k-1} Q_{k-1} C_{k-1} \\ &= B_{k-1} Q_{k-1} C_{k-1} = \Gamma_{k-1}. \end{aligned}$$

For $i = k, \cdots, n$, let $\Pi_i = \Gamma_i$. Thus, $\Gamma_0 = \Pi_0 \Rightarrow \cdots \Rightarrow \Pi_{k-1} = \Gamma_{k-1}$ and $\Gamma_k = \Pi_k \Rightarrow \cdots \Rightarrow \Pi_n = \Gamma_n$, so that $\Pi_0 \Rightarrow \cdots \Rightarrow \Pi_n$. Further, the step $\Pi_0 \Rightarrow \Pi_1$ is $|B_k \alpha_1 Y_q| + M(|\alpha_2|)$ bounded since $\nabla_0 \Rightarrow \nabla_1$ is $M(|\alpha_2|)$ bounded. But $|B_k \alpha_1 Y_q| = q \leqq |\beta|$ and $|\alpha_2| \leqq |\alpha_1| < |\beta|$ so that even in the worst case (where $|\alpha_2| = |\beta| - 1$ and $q = |\beta|$),

$$|B_k \alpha_1 Y_q| + M(|\alpha_2|) \leqq |\beta| + M(|\beta| - 1) = M(|\beta|).$$

Thus $\Pi_0 \Rightarrow \Pi_1$ is $M(|\beta|)$ bounded.

LEMMA 2. *Let $G = (V, \Sigma, R, X)$ be a type 0 grammar which satisfies the hypothesis of Theorem 1. For any $w \in L(G)$, if $X \Rightarrow \Gamma_1 \Rightarrow \cdots \Rightarrow \Gamma_n = w$ is a derivation in G such that there is a least t where the step $\Gamma_t \Rightarrow \Gamma_{t+1}$ is not M_G bounded, then there exist $\Pi_1, \cdots, \Pi_n \in V^*$ such that $X \Rightarrow \Pi_1 \Rightarrow \cdots \Rightarrow \Pi_n$ is a derivation in G, $\Pi_n = w$, and the derivation $X \Rightarrow \Pi_1 \Rightarrow \cdots \Rightarrow \Pi_{t+1}$ is M_G bounded.*

Proof. Let $\Gamma_t = a_1 \cdots a_p Z_1 \cdots Z_q$, where each $a_i \in \Sigma$, each $Z_j \in V$, and $Z_1 \in V - \Sigma$. Since the step $\Gamma_t \Rightarrow \Gamma_{t+1}$ is not M_G bounded, $Z_2 \cdots Z_q \neq e$. Since $X \overset{*}{\Rightarrow} \Gamma_t$

$\overset{*}{\Rightarrow} \Gamma_n$ and $\Gamma_n = w \in \Sigma^*$, there is some first step of $\Gamma_t \Rightarrow \cdots \Rightarrow \Gamma_n$ which transforms Z_1. Let $\Gamma_j \Rightarrow \Gamma_{j+1}$ be that step. If $j = t$, then Z_1 is transformed in $\Gamma_t \Rightarrow \Gamma_{t+1}$, so that the step $\Gamma_t \Rightarrow \Gamma_{t+1}$ is $1 \leqq M_G$ bounded, contrary to the choice of t. Hence, $t \nleqq j$. Since $Z_1 \in V - \Sigma$, no part of $a_1 \cdots a_p Z_1$ serves as part of the left context of any rule applied in $\Gamma_t \Rightarrow \cdots \Rightarrow \Gamma_j$. Since Z_1 is the left-most nonterminal symbol in each of $\Gamma_t, \Gamma_{t+1}, \cdots, \Gamma_j$, no part of $a_1 \cdots a_p Z_1$ serves as part of the right context of any rule applied in $\Gamma_t \Rightarrow \cdots \Rightarrow \Gamma_j$. Thus, as in the proof of Lemma 1, the derivation $X \Rightarrow \Pi_1 \Rightarrow \cdots \Rightarrow \Pi_n$ can be constructed by rearranging the order of application of rules in the derivation $\Gamma_t \Rightarrow \cdots \Rightarrow \Gamma_j$.

In particular, if the rule applied in $\Gamma_j \Rightarrow \Gamma_{j+1}$ is a context-free rule $Z_1 \to \gamma$ or is a context-sensitive rule $\alpha Z_1 \beta \to \alpha\gamma\beta$ and

$$\Gamma_t = a_1 \cdots a_p Z_1 \cdots Z_q = a_1 \cdots a_{p-|\alpha|} \alpha Z_1 \beta Z_{|\beta|+2} \cdots Z_q,$$

then this rule can be applied to Π_t to yield

$$\Pi_{t+1} = a_1 \cdots a_p \gamma Z_2 \cdots Z_q,$$

where $\Pi_i = \Gamma_i$ for $i = 1, \cdots, t, j+1, \cdots, n$, and $\Pi_{t+2}, \cdots, \Pi_{j+1}$ are obtained from $\Gamma_t \Rightarrow \cdots \Rightarrow \Gamma_j$ just as in the proof of Lemma 1. In this case the step $\Pi_t \Rightarrow \Pi_{t+1}$ is 1 bounded.

If the rule applied in $\Gamma_j \Rightarrow \Gamma_{j+1}$ is a context-sensitive rule $\alpha Z_1 \beta \to \alpha\gamma\beta$ and $\alpha Z_1 \beta$ is not a prefix of Γ_t, then β is not a prefix of $Z_2 \cdots Z_q$ but is a prefix of δ, where $\Gamma_j = \alpha Z_1 \delta$. Since the rules applied in $\Gamma_t \Rightarrow \cdots \Rightarrow \Gamma_j$ use no part of $a_1 \cdots a_p Z_1$ as either left or right context, this sequence of rules can be applied to $Z_2 \cdots Z_q$ to obtain δ in a derivation of length $j - t$. By Lemma 1, this derivation can be converted to another derivation $Z_2 \cdots Z_q \Rightarrow \cdots \Rightarrow \delta$ of length $j - t$ such that the first step is $M(\beta)$ bounded. From the latter derivation (just as in the proof of Lemma 1), $\Pi_t \Rightarrow \cdots \Rightarrow \Pi_j$ is obtained such that $\Pi_t = \Gamma_t, \Pi_j = \Gamma_j$, and the step $\Pi_t \Rightarrow \Pi_{t+1}$ is $M(\beta) + 1 \leqq M_G$ bounded. Letting $\Pi_i = \Gamma_i$ for $i = 1, \cdots, t-1, j+1, \cdots, n$, one obtains a derivation $X \Rightarrow \Pi_1 \Rightarrow \cdots \Rightarrow \Pi_n = \Gamma_n = w$, where the derivation $X \Rightarrow \Pi_1 \Rightarrow \cdots \Rightarrow \Pi_{t+1}$ is M_G bounded.

Proof of Theorem 1. For any $w \in L(G)$, consider any derivation $X \Rightarrow \Gamma_1 \Rightarrow \cdots \Rightarrow \Gamma_n = w$ in G. Either this derivation is M_G bounded, or applying Lemma 2 at most $n - 2$ times yields a derivation $X \Rightarrow \Pi_1 \Rightarrow \cdots \Rightarrow \Pi_n = w$ in G which is M_G bounded. Hence $w \in \text{LEFT}(M_G, G)$. Thus

$$L(G) \subseteq \text{LEFT}(M_G, G) \subseteq L(G).$$

Since $\text{LEFT}(M_G, G)$ is context-free, $L(G)$ is context-free. This completes the proof.

Let $G = (V, \Sigma, R, X)$ be a type O grammar which satisfies the hypothesis of Theorem 1, so that if $\alpha Z \beta \to \alpha\gamma\beta \in R$, then $|\alpha| \geqq |\beta|$. If this restriction on length is weakened, then $L(G)$ is not necessarily context-free. To see that this is true, recall that there exist type O grammars $G = (V, \Sigma, R, X)$ such that each non-context-free rule is of the form $ZY \to Z'Y$, where $Z \in V - \Sigma$, and $Z', Y \in V$, and such that $L(G)$ is not context-free.[4] In such a grammar the non-context-free rules are of the form $\alpha Z \beta \to \alpha\gamma\beta$, where $\alpha = e$ so $\alpha \in \Sigma^*$, $\beta \in V^*$, and $0 = |\alpha| < |\beta| = 1$. Thus the length

[4] This fact has been observed by L. H. Haines (private communication).

restriction in the hypothesis of Theorem 1 cannot be dropped (or even weakened to, say, $|\alpha| + 1 \geqq |\beta|$). Hence, with respect to the comparison of lengths of context, Theorem 1 is as strong a result as is possible.

The family of context-free languages is closed under the operation of reversal: for $a \in \Sigma \cup \{e\}$, $a^R = a$; for $a_1 \cdots a_n \in \Sigma^*$, $n \geqq 1$, $(a_1 \cdots a_n)^R = a_n \cdots a_1$; $L^R = \{w^R | w \in L\}$. This fact is easy to prove by means of context-free grammars. Hence, if the restriction on the form of the rules in Theorem 1 is altered to $\alpha Z \beta \to \alpha \gamma \beta$ where $\beta \in \Sigma^*$, $Z \in V - \Sigma$, $\alpha, \gamma \in V^*$, and $|\alpha| \leqq |\beta|$, then again $L(G)$ is context-free.

3. This section is devoted to showing that if the rules of a grammar have only terminal strings as context, then the language generated is context-free. Formally, this is stated in the following theorem.

THEOREM 2. *If $G = (V, \Sigma, R, X)$ is a type O grammar such that every non-context-free rule is of the form $\alpha Z \beta \to \alpha \gamma \beta$ where $\alpha\beta \in \Sigma^*$, $Z \in V - \Sigma$, and $\gamma \in V^*$, then $L(G)$ is context-free*

Intuitively one sees that "messages" cannot be transmitted over sufficiently long terminal strings and here it is only terminal strings which are allowed as context. However, the formal proof of the theorem is based on a lemma which allows one to reduce the length of terminal context, so that by repeated use a context-free grammar is generated.

Notation. For any type O grammar $G = (V, \Sigma, R, X)$, let $L_G = \max\{|\alpha| \,|\, \alpha Z \beta \to \alpha \gamma \beta \in R\}$ and $R_G = \max\{|\beta| \,|\, \alpha Z \beta \to \alpha \gamma \beta \in R\}$.

LEMMA 3. *Let $G = (V, \Sigma, R, X)$ be a type O grammar satisfying the hypothesis of Theorem 2. If $R_G \geqq L_G$ and $R_G \geqq 1$, then one can construct a type O grammar G_1, a regular set T, and a* gsm *f such that*:

(i) *G_1 satisfies the hypothesis of Theorem 2;*

(ii) *$R_{G_1} = R_G - 1$ and $L_{G_1} = L_G$; and*

(iii) *$f(L(G_1) \cap T) = L(G)$.*[5]

Proof of Theorem 2. As pointed out at the end of § 2, the family of context-free languages is closed under reversal. Thus Lemma 3 still holds if R_G and L_G (and R_{G_1} and L_{G_1}) are interchanged throughout—refer to the result as Lemma 3′. Given G as in the hypothesis, applying Lemmas 3 and 3′ $m = R_G + L_G$ times yields a sequence $G_1, \cdots, G_m$ of grammars, a sequence $T_1, \cdots, T_m$ of regular sets, and a sequence $f_1, \cdots, f_m$ of gsm's such that for $i = 1, \cdots, m$, $f_i(L(G_i) \cap T_i) = L(G_{i-1})$, where $G_0 = G$ and G_m is context-free (since $L_{G_m} = R_{G_m} = 0$ implies G_m is context-free). Since the family of context-free languages is closed under intersection with regular sets and under gsm mappings, this implies that $L(G) = L(G_0)$ is context-free. This completes the proof.

The proof of Lemma 3 rests on the following observation. If $X \Rightarrow \Gamma_1 \Rightarrow \cdots \Rightarrow \Gamma_n$ is any derivation in G and some Γ_i has a substring $\beta \in \Sigma^*$ of length R_G or greater, then every step of $\Gamma_i \Rightarrow \cdots \Rightarrow \Gamma_n$ either transforms a symbol to the right of β independent of what is to the left of β or transforms a symbol to the left of β independent of what is to the right of β.

[5] See [4] for definitions and facts about regular sets and gsm's.

Proof of Lemma 3. Without loss of generality, assume that every non-context-free rule in R either has left context but no right context, or has right context but no left context. Partition the set R as follows: let

$$S_1 = \{Z \to \gamma \in R | Z \in V - \Sigma\},$$
$$S_2 = \{\alpha Z \to \alpha\gamma \in R | \alpha \in \Sigma^*, Z \in V - \Sigma, \text{ and } |\alpha| < R_G\},$$
$$S_3 = \{\alpha Z \to \alpha\gamma \in R | \alpha \in \Sigma^*, Z \in V - \Sigma, \text{ and } |\alpha| = R_G\},$$
$$S_4 = \{Z\beta \to \gamma\beta \in R | Z \in V - \Sigma, \beta \in \Sigma^*, \text{ and } |\beta| < R_G\},$$
$$S_5 = \{Z\beta \to \gamma\beta \in R | Z \in V - \Sigma, \beta \in \Sigma^*, \text{ and } |\beta| = R_G\}.$$

For each $\beta \in \Sigma^*$ such that $|\beta| = R_G$ and there exist $Z \in V - \Sigma$, $\gamma \in V^*$ with $Z\beta \to \gamma\beta \in R$, let $[\beta]$ be a new symbol, and let Σ_1 be the set of such new symbols. For each such β, let β_{-1} be the prefix of β of length $|\beta| - 1$, i.e., if $\beta = \delta a$ where $a \in \Sigma$, then $\beta_{-1} = \delta$; since $|\beta| = R_G, |\beta_{-1}| = R_G - 1$.

Construct the grammar $G_1 = (V_1, \Sigma \cup \Sigma_1, R_1, X)$ as follows. Let $V_1 = V \cup \Sigma_1$. Let

$$U_5 = \{Z\beta_{-1} \to \gamma\beta_{-1}[\beta]\beta_{-1} | Z \in V - \Sigma \text{ and } Z\beta \to \gamma\beta \in S_5\}.$$

Let $R_1 = (R - S_5) \cup U_5$. From the construction it is clear that G_1 is a type O grammar such that $L_{G_1} = L_G$ and $R_{G_1} = R_G - 1$.

Let

$$T = (\Sigma \cup (\bigcup_{[\beta]\in\Sigma_1} \beta_{-1}([\beta]\beta_{-1})^*[\beta]\beta))^*$$

so that T is a regular set. Let f be a gsm which yields the identity mapping on Σ until a symbol $[\beta] \in \Sigma_1$ is scanned. Strings of the form $[\beta]\beta_{-1}$ are erased and f returns to its initial mode of operation. (Recall that $|[\beta]\beta_{-1}| = R_G$, so that f need only remember finitely many strings.) Any other operation outputs "garbage." Hence for any $[\beta] \in \Sigma_1$ and any $n \geqq 0$,

$$f(\beta_{-1}([\beta]\beta_{-1})^n[\beta]\beta) = \beta_{-1}f(([\beta]\beta_{-1})^{n+1}a) = \beta_{-1}a = \beta,$$

where $a \in \Sigma$ and $\beta = \beta_{-1}a$. Also, $f(w) = w$ if and only if $w \in \Sigma^*$.

Before proving that $f(L(G_1) \cap T) = L(G)$, let us informally explain the construction of G_1 and the role of T and f. Since $R - S_5 = R_1 - U_5$ and U_5 is a "copy" of S_5, it is enough to explain the use of rules in U_5. When a rule

$$Z\beta_{-1} \to \gamma\beta_{-1}[\beta]\beta_{-1}$$

is applied to a string $\mu Z\beta_{-1}([\beta]\beta_{-1})^t\sigma$ to obtain $\mu\gamma\beta_{-1}([\beta]\beta_{-1})^{t+1}\sigma$, one "guesses" that β is a prefix of $\beta_{-1}\sigma$ so that an application of $Z\beta \to \gamma\beta$ in G is being imitated. The new occurrence of the symbol $[\beta]$ serves as a "marker" to indicate this guess and also a "barrier" so that further steps take place either to the right of $(\beta_{-1}[\beta])^{t+1}$ or to the left of $([\beta]\beta_{-1})^{t+1}$. The "new" copy of β_{-1} in $\gamma\beta_{-1}[\beta]\beta_{-1}$ is available for use as right context in the future application of some rule. The "old" β_{-1} can still serve as part of the left context, since if $\beta = \beta_{-1}a$, one still wishes to be able to apply a rule such as $\delta aY \to \delta a\psi$ if such a rule is in R. By hypothesis, $L_G \leqq R_G$, so that $|\delta a| \leqq |\beta| = R_G$. Hence if one were able to apply this rule in a derivation in

G and if the guess that β is present is correct, then one can still apply this rule in an "imitating" derivation in G_1—the symbol $[\beta]$ does not cause a conflict since

$$|[\beta]\beta_{-1}a| = 1 + |\beta_{-1}a| = 1 + |\beta| = 1 + R_G > L_G.$$

The regular set T serves as a "filter" to restrict attention to terminal strings which do have substrings in $(\beta_{-1}[\beta])^{t+1}\beta$—i.e., to check that the guesses were correct. The symbol $[\beta]$ also serves as a marker to tell the gsm to erase the substring $[\beta]\beta_{-1}$.

The equality $f(L(G_1) \cap T) = L(G)$ is established by showing that a derivation in G resulting in a string in $L(G)$ can be imitated by a derivation in G_1 resulting in a string in $L(G_1) \cap T$, with the role of f being obvious. The inclusion $f(L(G_1) \cap T) \subseteq L(G)$ is somewhat more complicated only because a derivation in G_1 resulting in a string in $L(G_1) \cap T$ may need to be "rearranged" in order to be imitated in G—the "guess" may have been made too soon. The proofs of these inclusions are only sketched since the detailed induction arguments do not yield any additional insight.

Claim 1. $L(G) \subseteq f(L(G_1) \cap T)$.

Sketch of the proof. It is sufficient to show that derivations in G can be imitated in G_1 such that any resulting terminal string is also in T. Derivations in G which do not use rules in S_5 can be considered to be derivations in G_1 and $L(G) \subseteq \Sigma^* \subset T$. Thus one need be concerned only with those derivations in G which do use rules in S_5.

The set R_1 of rules of G_1 contains all the rules in $R - S_5$ as well as the set U_5, which is a "copy" of S_5. A rule in U_5 is of the form

$$Z\beta_{-1} \to \gamma\beta_{-1}[\beta]\beta_{-1},$$

where $Z \in V - \Sigma$, $\beta_{-1} \in \Sigma^*$, $|\beta_{-1}| = R_G - 1$, and $Z\beta \to \gamma\beta$ is in S_5. Hence the symbol $[\beta]$ is generated only as part of a string $\beta_{-1}[\beta]\beta_{-1}$. If one "imitates" a derivation of G in G_1, then to imitate an application of $Z\beta \to \gamma\beta$, the rule $Z\beta_{-1} \to \gamma\beta_{-1}[\beta]\beta_{-1}$ in U_5 is applied to a string $\delta_1 Z(\beta_{-1}[\beta])^k\beta\delta_2$, $k \geqq 0$, to yield $\delta_1\gamma(\beta_{-1}[\beta])^{k+1}\beta\delta_2$ so that $[\beta]$ is generated as part of $(\beta_{-1}[\beta])^{k+1}\beta$. Thus it is easy to see that if $X \Rightarrow \Gamma_1 \Rightarrow \cdots \Rightarrow \Gamma_n$ is a derivation in G with $\Gamma_n \in \Sigma^*$, then one can construct a derivation $X \Rightarrow \Pi_1 \Rightarrow \cdots \Pi_n$ in G_1 with $\Pi_n \in (\Sigma \cup \Sigma_1)^*$ such that $\Pi_n \in T$ and $f(\Pi_n) = \Gamma_n$. Hence, $L(G) \subseteq f(L(G_1) \cap T)$.

Claim 2. $f(L(G_1) \cap T) \subseteq L(G)$.

Sketch of the proof. It is sufficient to show that derivations in G_1 which generate strings in $L(G_1) \cap T$ can be imitated in G. Such derivations in G_1 generate strings in $L(G_1) \cap \Sigma^*$ if and only if they use no rules from U_5. But clearly such derivations are already derivations in G, so $L(G_1) \cap \Sigma^* \subseteq L(G)$; also, $f(L(G_1) \cap \Sigma^*) = L(G_1) \cap \Sigma^*$. Hence one need consider only those derivations in G_1 which use at least one application of a rule in U_5.

Suppose $X \Rightarrow \Gamma_1 \Rightarrow \cdots \Rightarrow \Gamma_n$ is a derivation in G_1 such that $\Gamma_n \in L(G_1) \cap T$ and such that for some $k \leqq n$ the rule applied at the step $\Gamma_{k-1} \Rightarrow \Gamma_k$ is a rule in U_5. Let $\{\langle (B_i, P_i, C_i), (B_i, Q_i, C_i)\rangle\}_{i=1}^n$ be a production sequence for $X \Rightarrow \Gamma_1 \Rightarrow \cdots \Rightarrow \Gamma_n$, and let $P_k \to Q_k$ be $Z\beta_{-1} \to \gamma\beta_{-1}[\beta]\beta_{-1}$, where $Z \in V - \Sigma$ and $\beta_{-1} \in \Sigma^*$.

If $P_kC_k = Z(\beta_{-1}[\beta])^t\beta D$ for some $t \geqq 0$ and $D \in V_1^*$, then in the "imitating" derivation $X \Rightarrow \Pi_1 \Rightarrow \cdots \Rightarrow \Pi_n$ in G, the rule $Z\beta \to \gamma\beta$ can be applied at the step $\Pi_{k-1} \Rightarrow \Pi_k$. Since

$$f((\beta_{-1}[\beta])^{t+1}\beta) = \beta,$$

the substring $(\beta_{-1}[\beta])^{t+1}\beta$ of Γ_k is mapped onto the appropriate substring β of Π_k.

Suppose for some $t \geqq 0$, $Z(\beta_{-1}[\beta])^t\beta_{-1}$ is a prefix of P_kC_k but for every $j \geqq 0$, $Z(\beta_{-1}[\beta])^j\beta$ is not a prefix of P_kC_k. Let $\Gamma_{q-1} \Rightarrow \Gamma_q$ be the first step which generates an occurrence of $[\beta]$. Since no rule of R_1 has any symbol of Σ_1 on its left-hand side and since $|\beta_{-1}| = R_{G_1}$, one loses no generality by assuming that for some m, $q < m \leqq n$, every step of $\Gamma_q \Rightarrow \cdots \Rightarrow \Gamma_m$ transforms a symbol to the left of $[\beta]$ and every step of $\Gamma_m \Rightarrow \cdots \Rightarrow \Gamma_n$ transforms a symbol to the right of $[\beta]$. Since $\Gamma_n \in T$, the derivation $\Gamma_m \Rightarrow \cdots \Rightarrow \Gamma_n$ produces a substring $[\beta]\beta$. Hence in the imitating derivation $X \Rightarrow \Pi_1 \Rightarrow \cdots \Rightarrow \Pi_n$ in G, the portion $\Pi_{q-1} \Rightarrow \cdots \Rightarrow \Pi_{q-1+n-m}$ imitates $\Gamma_m \Rightarrow \cdots \Rightarrow \Gamma_n$ so that when imitating $\Gamma_{q-1} \Rightarrow \Gamma_q$, the string β is available to use as right context. This step is imitated by $\Pi_{q-1+n-m} \Rightarrow \Pi_{q+n-m}$ and then $\Gamma_q \Rightarrow \cdots \Rightarrow \Gamma_m$ is imitated by $\Pi_{q+n-m} \Rightarrow \cdots \Rightarrow \Pi_n$.

(A formal proof of Claim 2 can be carried out by induction based on the number of applications of rules from U_5 in a derivation $X \Rightarrow \Gamma_1 \Rightarrow \cdots \Rightarrow \Gamma_n$ such that $\Gamma_n \in L(G_1) \cap T$ (equivalently, the induction can be based on the number of occurrences of symbols from Σ_1 in Γ_n). The induction step is based on the informal description given in the last paragraph.)

A construction similar to that in the proof of Lemma 3 can be used in conjunction with the result of Ginsburg and Greibach cited in the Introduction to establish the following generalization of Theorem 2.

COROLLARY. *If $G = (V, \Sigma, R, X)$ is a grammar such that every non-context-free rule is of one of the forms*

(i) $\alpha\rho \to \alpha\theta$, *where* $\alpha \in \Sigma^*$, $|\alpha| \geqq 1$, *and* $\rho \in (V - \Sigma)^*$;

(ii) $\rho\beta \to \theta\beta$, *where* $\beta \in \Sigma^*$, $|\beta| \geqq 1$, *and* $\rho \in (V - \Sigma)^*$;

then $L(G)$ is context-free.

There is one further restriction which generalizes the hypotheses of both Theorems 1 and 2. Let $G = (V, \Sigma, R, X)$ be a type O grammar such that every non-context-free rule is of one of the forms

(i) $\alpha Z\beta \to \alpha\gamma\beta$, where $\alpha \in \Sigma^*$, $Z \in V - \Sigma$, $\beta, \gamma \in V^*$, and $|\alpha| \geqq |\beta|$;

(ii) $\alpha Z\beta \to \alpha\gamma\beta$, where $\beta \in \Sigma^*$, $Z \in V - \Sigma$, $\alpha, \gamma \in V^*$, and $|\alpha| \leqq |\beta|$.

Thus it is required that either left or right context be a terminal string, and in either case, the terminal context has length at least as great as the other context. We conjecture that this restriction forces the language generated to be context-free.

REFERENCES

[1] S. ABRAHAM, *Some questions of phrase structure grammars*, Comp. Linguistics, 4 (1965), pp. 61–70.

[2] R. BOOK, *Time-bounded grammars and their languages*, J. Comput. System Sci., 5 (1971), pp. 397–429.

[3] R. EVEY, *The theory and application of pushdown store machines*, Doctoral thesis, Harvard Univ., Cambridge, Mass., 1963. Also appears as *Math. linguistics and automatic translation*, NSF-10, Computation Lab., Harvard Univ., Cambridge, Mass., 1963.

[4] S. GINSBURG, *The Mathematical Theory of Context-free Languages*, McGraw-Hill, New York, 1966.

[5] S. GINSBURG AND S. A. GREIBACH, *Mappings which preserve context-sensitive languages*, Information and Control, 9 (1966), pp. 563–582.

[6] S. GINSBURG AND E. SPANIER, *Control sets on grammars*, Math. Systems Theory, 2 (1968), pp. 159–177.

[7] A. GLADKII, *On the complexity of derivations in phrase structure grammars*, Algebri i Logika Sem., 3 (1964), pp. 29–44.

[8] T. GRIFFITHS, *Some remarks on derivations in general rewriting systems*, Information and Control, 12 (1968), pp. 27–54.

[9] T. HIBBARD, *Scan limited automata and context limited grammars*, Doctoral thesis, Univ. of California, Los Angeles, 1966.

[10] G. H. MATTHEWS, *Discontinuity and asymmetry in phrase structure grammars*, Information and Control, 6 (1963), pp. 137–146.

[11] ———, *A note on asymmetry in phrase structure grammars*, Ibid., 7 (1964), pp. 360–365.

[12] ———, *Two-way languages*, Ibid., 10 (1967), pp. 111–119.

[13] D. ROSENKRANTZ, *Programmed grammars and classes of formal languages*, J. Assoc. Comput. Mach., 16 (1969), pp. 107–131.

[14] A. SALOMAA, *On grammars with restricted use of productions*, Ann. Acad. Sci. Fenn., Ser. A, 1969, no. 454.

[15] ———, *On some families of formal languages obtained by regulated derivations*, Ibid., 1970, no. 479.

SIAM J. COMPUT.
Vol. 1, No. 1, March 1972

A SIMPLE ALGORITHM FOR MERGING TWO DISJOINT LINEARLY ORDERED SETS*

F. K. HWANG AND S. LIN†

Abstract. In this paper we present a new algorithm for merging two linearly ordered sets which requires substantially fewer comparisons than the commonly used tape merge or binary insertion algorithms. Bounds on the difference between the number of comparisons required by this algorithm and the information theory lower bounds are derived. Results from a computer implementation of the new algorithm are given and compared with a similar implementation of the tape merge algorithm.

Key words. algorithms, merging

1. Introduction. Suppose we are given two disjoint linearly ordered subsets A and B of a linearly ordered set S, say

$$A = \{a_1 < a_2 < \cdots < a_m\},$$
$$B = \{b_1 < b_2 < \cdots < b_n\}.$$

The problem is to determine the linear ordering of their union (i.e., to merge A and B) by means of a sequence of pairwise comparisons between an element of A and an element of B. Given any algorithm s to solve this problem, we are interested in the maximum number of comparisons, $K_s(m, n)$, required under all possible orderings of $A \cup B$. An algorithm s is said to be M-optimal if $K_s(m, n) = K(m, n)$, where $K(m, n) = \min_x K_x(m, n)$. In this paper, we give a simple algorithm for solving this problem, called the generalized binary algorithm g, and derive some bounds for $K_g(m, n) - K(m, n)$ which are substantially better than two other known algorithms.

2. Some preliminary discussions and results. Let the cardinality of A and B be m and n respectively. We assume $m \leqq n$. Let $\mathscr{D}_0$ be the set of all possible orderings of $A \cup B$ and $\mathscr{D}_k$ be the subset of $\mathscr{D}_0$ consistent with the results of the first k comparisons we have made thus far. It is clear that, after making the ith comparison, $i = 1, 2, \cdots, k$, one of the two possible outcomes must have $|\mathscr{D}_i| \geqq \frac{1}{2}|\mathscr{D}_{i-1}|$ and that merging is achieved if and only if $\mathscr{D}_k$ contains exactly one element. Since $\mathscr{D}_0$ has $\binom{m+n}{n}$ elements, or as we say, data points, we must have, for any algorithm s,[1]

$$K_s(m, n) \geqq \left\lceil \log_2 \binom{m+n}{m} \right\rceil \equiv I(m, n).$$

$I(m, n)$ is usually called the information theory bound.

For $m = 1$, the binary insertion algorithm is optimal and hence

$$K(1, n) = I(1, n) = \lceil \log_2 (n + 1) \rceil. \tag{1}$$

* Received by the editors August 30, 1971.

† Bell Laboratories, Murray Hill, New Jersey 07974.

[1] As usual, we let $\lceil x \rceil$ denote the smallest integer $\geqq x$ and $\lfloor x \rfloor$ the largest integer $\leqq x$.

In a recent paper [1], the authors constructed an M-optimal algorithm for $m = 2$ and thereby determined the values of $K(2, n)$. It can also be shown that [2]

$$K(m, n) = m + n - 1 \quad \text{for } 3 < m \leqq n \leqq m + 3 \tag{2}$$

and

$$K(m, 2m) \leqq 3m - 2 \quad \text{for } m \geqq 3. \tag{3}$$

The determination of $K(m, n)$ for $m \geqq 3$ appears to be a very difficult problem.

3. Two existing algorithms. For the purpose of comparing with the generalized binary algorithm to be presented in the next section, we mention two existing algorithms.

I. *The "tape merge" algorithm t.* The "tape merge" algorithm is the commonly used procedure to merge two tapes or lists of sorted items. It can be described by the following steps (details of storing and stop conditions are omitted):

TM1. Compare a_m with b_n.

TM2. If $a_m < b_n$, set $n = n - 1$ and go to TM1.

TM3. If $a_m > b_n$, set $m = m - 1$ and go to TM1.

It can be easily shown that

$$K_t(m, n) = m + n - 1$$

and hence the "tape merge" algorithm is M-optimal for $n \leqq m + 3$ [2].

II. *The "simple binary" algorithm s.* The "simple binary" algorithm can be described by the following steps:

SB1. Merge a_m into B by the binary search procedure.

SB2. Pull out a_m and elements of $B > a_m$. (These are already in order and larger than the rest of the elements of $A \cup B$.) Set $m = m - 1$ and redefine m and n. (The new $n \geqq$ new m.) Go back to SB1.

It is clear that under the worst possible outcome, a_m is always larger than b_n and hence no element of B is discarded. Therefore,

$$K_s(m, n) = m\lceil \log_2 (n + 1) \rceil.$$

For $m = 1$, we have

$$K_s(m, n) = K(m, n).$$

However, we shall show in the next section that

$$K_s(m, n) > K(m, n) \quad \text{for } m > 2.$$

The distinctive feature of these two algorithms is their simplicity, although in general, they are quite inefficient in the sense that both $K_t(m, n) - K(m, n)$ or $K_s(m, n) - K(m, n)$ can be very large. In the next section, we shall present an algorithm which matches the two abovementioned algorithms in simplicity and yet improves a great deal on their efficiency.

4. The generalized binary algorithm g**.** For the sake of simplicity, we shall assume that whenever we are required to merge two disjoint linearly ordered

sets with cardinalities x and y respectively, n will always refer to $\max(x, y)$ and m, to $\min(x, y)$, so that $n \geqq m$.

The generalized binary algorithm may now be described as follows (again, details of storage and stop criteria are omitted):

GB1. Let $\alpha = \lfloor \log_2 (n/m) \rfloor$ and $x = n - 2^\alpha + 1$.

GB2. Compare a_m with b_x. If $a_m < b_x$, pull out the set of all elements in $B \geqq b_x$, say C. We are then left with the problem of merging two disjoint sets A and $B - C$. Redefine m and n and go back to GB1. (Note that $B - C$ has $n - 2^\alpha$ elements and we need to interchange the role of m and n if and only if $n = m$.)

GB3. If $a_m > b_x$, merge a_m into the set $C - b_x$ by the simple binary algorithm. Note that $C - b_x$ has exactly $2^\alpha - 1$ elements and a_m can be merged into the set in exactly α more comparisons. Pull out a_m and the set D of all elements in $B > a_m$. We are then left with the problem of merging the set $A - a_m$ with the set $B - D$. Redefine m and n and go back to GB1.

For this algorithm g, $K_g(m, n)$ is given by the following theorem.

THEOREM 1. *Let* $\alpha = \lfloor \log_2 (n/m) \rfloor$. *Write* $n = 2^\alpha m + 2^\alpha p + \theta$, *where* p *and* θ *are uniquely determined nonnegative integers satisfying* $0 \leqq p < m$, $0 \leqq \theta < 2^\alpha$. *Then* $K_g(m, n) = (2 + \alpha)m + p - 1$.

Proof. If $\alpha = 0$, $n = m + p$, and it is clear that the worst possible data forces the algorithm g to be identical with the algorithm t discussed in the previous section.

Hence $K_g(m, n) = K_t(m, n) = m + n - 1 = 2m + p - 1$.

If $m = 1$, p must be zero and $n = 2^\alpha + \theta$. It is clear that $a_m > b_x$ is the worst outcome and hence $K_g(1, n) = K(1, n) = 1 + \alpha$.

We now prove Theorem 1 by induction on $m + n$. Assume the theorem true for all m', n' such that $m' + n' < m + n$, and for all m, n with $\alpha = 0$, or $m = 1$. We prove the theorem true for m, n with $\alpha > 0$ and $m > 1$. The theorem is trivially true for $m + n = 2$.

After making the first comparison of a_m with b_x, we have two possibilities:

(i) $a_m < b_x$, and we are left with the problem of merging two sets with m and $n - 2^\alpha$ elements.

(ii) $a_m > b_x$. After merging a_m into the set $C - b_x$ in α more comparisons, we are left, in the worst case, with the problem of merging two sets with $m - 1$ and n elements. Hence

$$K_g(m, n) = \max\,[1 + K_g(m, n - 2^\alpha), 1 + \alpha + K_g(m - 1, n)].$$

Now,

$$n - 2^\alpha = \begin{cases} 2^\alpha m + 2^\alpha(p - 1) + \theta & \text{if } p \neq 0, \\ 2^{\alpha-1}m + 2^{\alpha-1}(m - 1) + \theta - 2^{\alpha-1} & \text{if } p = 0 \text{ and } \theta \geqq 2^{\alpha-1}, \\ 2^{\alpha-1}m + 2^{\alpha-1}(m - 2) + \theta & \text{if } p = 0 \text{ and } \theta < 2^{\alpha-1}. \end{cases}$$

Hence by induction,

$$K_g(m, n - 2^\alpha) = \begin{cases} (2 + \alpha)m + (p - 1) - 1 & \text{if } p \neq 0, \\ (1 + \alpha)m + (m - 1) - 1 & \text{if } p = 0 \text{ and } \theta \geqq 2^{\alpha-1}, \\ (1 + \alpha)m + (m - 2) - 1 & \text{if } p = 0 \text{ and } \theta < 2^{\alpha-1}. \end{cases}$$

Similarly,

$$n = \begin{cases} 2^\alpha(m-1) + 2^\alpha(p+1) + \theta & \text{if } p < m-2, \\ 2^{1+\alpha}(m-1) + \theta & \text{if } p = m-2, \\ 2^{1+\alpha}(m-1) + 2\alpha + \theta & \text{if } p = m-1. \end{cases}$$

Hence by induction,

$$K_g(m-1,n) = \begin{cases} (2+\alpha)(m-1) + (p+1) - 1 & \text{if } p < m-2, \\ (3+\alpha)(m-1) - 1 & \text{otherwise}. \end{cases}$$

Therefore,

$$1 + K_g(m, n - 2^\alpha) = \begin{cases} (2+\alpha)m + p - 2 & \text{if } p = 0 \text{ and } \theta < 2^{\alpha-1}, \\ (2+\alpha)m + p - 1 & \text{otherwise}, \end{cases}$$

and

$$1 + \alpha + K_g(m-1,n) = \begin{cases} (2+\alpha)m + p - 2 & \text{if } p = m-1, \\ (2+\alpha)m + p - 1 & \text{otherwise}. \end{cases}$$

Since the conditions $p = 0$ and $p = m - 1$ are mutually exclusive for $m > 1$, we have

$$\begin{aligned} K_g(m,n) &= \max\,[1 + K_g(m, n - 2^\alpha), 1 + \alpha + K_g(m-1,n)] \\ &= (2+\alpha)m + p - 1, \end{aligned}$$

and hence the theorem is proved.

Comparing the general binary algorithm g with the tape merge algorithm t and the simple binary algorithm s, we have

$$\begin{aligned} K_t(m,n) - K_g(m,n) &= (m + n - 1) - [(\alpha+2)m + p - 1] \\ &= m + 2^\alpha m + 2^\alpha p + \theta - 1 - (\alpha+2)m - p + 1 \\ &= (2^\alpha - \alpha - 1)m + (2^\alpha - 1)p + \theta. \end{aligned}$$

Hence $K_t(m, n) = K_g(m, n)$ only if $\alpha = 0$, or $\alpha = 1$ and $p = \theta = 0$. Otherwise, $K_t(m, n) - K_g(m, n) = n - (\alpha + 1)m - p > 0$. Similarly,

$$\begin{aligned} K_s(m,n) - K_g(m,n) &= m\lceil \log_2(n+1)\rceil - [(\alpha+2)m + p - 1] \\ &\geqq m(\alpha + \lfloor \log_2(m+p)\rfloor + 1) - [(\alpha+2)m + p - 1] \\ &= m(\lfloor \log_2(m+p) - 1\rfloor - p + 1. \end{aligned}$$

Hence $K_s(m, n) = K_g(m, n)$ only if $m = 1$, or $m = 2$, $p = 1$. Otherwise $K_s(m, n) - K_g(m, n) \geqq m(\lfloor \log_2(m + p)\rfloor - 1) - p + 1 > 0$.

It is often convenient to refer to a set of numbers $n_l(m, k)$ as the largest n such that $K_l(m, n) \leqq k$. Table 1 gives some of these numbers for the algorithms t, s and g. Also we have for $k = (2 + \alpha)m + p - 1$,

$$n_g(m,k) = 2^\alpha(m+p+1) - 1,$$
$$n_t(m,k) = (1+\alpha)m + p,$$
$$n_s(m,k) = 2^{\alpha+2} - 1 \quad \text{provided } 2^{\alpha+2} - 1 \geqq m.$$

TABLE 1

(m, k)	(2, 4)	(2, 24)	(4,14)	(4,90)	$(10^3, 10^4)$
$n_g(m, k)$	3	4095	15	$2^{23} - 1$	256,511
$n_t(m, k)$	3	23	11	87	9,001
$n_s(m, k)$	3	2047	7	$2^{22} - 1$	1,023

5. Bounds on $K_g(m, n) - I(m, n)$**.** Let $n = 2^\alpha m + 2^\alpha p + \theta$ with $0 \leqq p < m$, $0 \leqq \theta < 2^\alpha$, $\alpha \geqq 0$; and $k = K_g(m, n) = (2+\alpha)m + p - 1$.

THEOREM 2. $K_g(m,n) - I(m,n) \leqq m - 1$.

Proof. We have

$$I(m,n) = \left\lceil \log_2 \binom{m+n}{m} \right\rceil,$$

and

$$\binom{m+n}{m} \geqq \frac{(n+1)^m}{m!} = \frac{(2^\alpha m + 2^\alpha p + \theta + 1)^m}{m!}$$
$$> \frac{[2^\alpha m(1+p/m)]^m}{m!} = \frac{2^{\alpha m} m^m (1+p/m)^m}{m!} \geqq 2^{\alpha m + m - 1 + p}$$

since

$$m! \leqq m^m 2^{1-m} \quad \text{and} \quad (1+p/m)^m \geqq 2^p.$$

Hence

$$\log_2 \binom{m+n}{m} > (\alpha+1)m + p - 1,$$

and

$$\left\lceil \log_2 \binom{m+n}{m} \right\rceil \geqq (\alpha+1)m + p.$$

Therefore,

$$K_g(m,n) - I(m,n) \leqq m - 1.$$

COROLLARY 1. *For $m > 1$ and $\theta = 2^\alpha - 1$,*

$$K_g(m,n) - I(m,n) \leqq m - 2.$$

Proof. For $m > 1$ and $\theta = 2^\alpha - 1$, we have

$$\binom{m+n}{m} > \frac{(n+1)^m}{m!} \geqq \frac{[2^\alpha m(1 + (p+1)/m)]^m}{m!}$$

and the proof parallels the proof of Theorem 1.

For larger m, a much sharper bound for $K_g(m,n) - I(m,n)$ can be derived by means of Stirling's formula. First we prove a lemma.

LEMMA 1. *Let $\hat{e} = (1 + m/n)^{n/m}$ and x_m be defined by*

$$(n + x_m)^m = (n+m)(n+m-1)\cdots(n+1).$$

Then

$$x_m \geqq \max\left[1, \frac{\hat{e}}{e}(n+m) - n\right].$$

Proof. It is clear that $x_m \geqq 1$. From Stirling's formula, we have

$$\begin{aligned}(n + x_m)^m &= (n+m)(n+m-1)\cdots(n+1) = \frac{(n+m)!}{n!}\\ &= \frac{\sqrt{2\pi}(n+m)^{n+m+1/2}\, e^{-(n+m)+\theta_1/(12(n+m))}}{\sqrt{2\pi}n^{n+1/2}\, e^{-n+\theta_2/(12n)}} \qquad \left(0 < \begin{matrix}\theta_1\\ \theta_2\end{matrix} < 1\right)\\ &> \sqrt{\frac{n+m}{n}}\left(1 + \frac{m}{n}\right)^n (n+m)^m\, e^{-m-1/(12n)}\\ &> \hat{e}^m (n+m)^m\, e^{-m}\end{aligned}$$

since

$$\sqrt{\frac{m+n}{n}} = \left(1 + \frac{m}{n}\right)^{n/m \cdot m/(2n)} \geqq 2^{1/(2n)} = 4^{1/(4n)} > e^{1/(12n)}.$$

Therefore,

$$x_m > \frac{\hat{e}}{e}(n+m) - n = m\left[\frac{\hat{e}}{e}\left(1 + \frac{n}{m}\right) - \frac{n}{m}\right].$$

Some typical values of x_m/m are given below in Table 2.

TABLE 2

n/m	1	2	10	100	1000
$\hat{e}$	2	2.25	2.594	2.705	2.717
x_m/m	>0.4715	>0.4831	>0.4907	>0.5065	>0.5279

THEOREM 3. *Let* $\varepsilon = (\theta + x_m)/2^\alpha$ *and* $t = \min(p + \varepsilon, m)$. *Then*

$$I(m, n) = \left\lceil \log_2 \binom{m+n}{m} \right\rceil \geqq (1 + \alpha)m + \lceil t + q_m \rceil,$$

where

$$q_m = (\log_2 e - 1)m - \frac{\log_2 e}{12m} - \tfrac{1}{2}\log_2(2\pi m)$$

$$\sim 0.442695m - \frac{0.12}{m} - \tfrac{1}{2}\log_2(2\pi m).$$

Proof.

$$\binom{n+m}{m} = \frac{(n + x_m)^m}{m!}$$

$$> \frac{(n + x_m)^m}{\sqrt{2\pi m}\, m^m e^{-m+1/(12m)}}$$

$$= \frac{(2^\alpha m + 2^\alpha p + \theta + x_m)^m e^{m-1/(12m)}}{\sqrt{2\pi m}\, m^m}$$

$$= \frac{(2^\alpha m)^m (1 + (p + \varepsilon)/m)^m e^{m-1/(12m)}}{\sqrt{2\pi m}\, m^m}$$

$$> 2^{\alpha m + t + (\log_2 e)(m - 1/(12m)) - \log_2 \sqrt{2\pi m}}.$$

Therefore

$$I(m, n) = \left\lceil \log_2 \binom{m+n}{m} \right\rceil \geqq (1 + \alpha)m + \lceil t + q_m \rceil,$$

which is to be proved.

Since $K(m, n) \geqq I(m, n)$, we have the following corollary.

COROLLARY 2.

$$K_g(m, n) - K(m, n) \leqq K_g(m, n) - I(m, n) \leqq m + p - 1 - \lceil t + q_m \rceil.$$

Table 3 gives some values for

$$q_m = (\log_2 e - 1)m - \frac{\log_2 e}{12m} - \tfrac{1}{2}\log_2(2\pi m).$$

TABLE 3

m	1	2	3	4	5	6	16	1024
q_m	−2.08	−1.0005	−0.832	−0.585	−0.299	0.179	3.65	446.9

Note that $t > p$ and $q_m > -1$ for $m \geqq 3$ so that Corollary 2 implies Theorem 2 for $m \geqq 3$. For $m \geqq 6$, $q_m > 0$ and hence Corollary 2 also implies the conclusion of Corollary 1 regardless of the value of θ. For large m, say $m > 100$, we have $K_g(m, n) - I(m, n) < 0.6m - 1$, and the "best" bound occurs when $\alpha = 0$, $p \approx 0.52m$, $x_m \approx 0.48m$, $q_m \approx 0.44m$ and this gives $K_g(m, n) - I(m, n) \approx 0.08m$.

6. Computational results. In this section, we discuss the storage requirements when the generalized binary algorithm g is implemented by a computer program and compare its running time with a similar program implementing the commonly used tape merge algorithm t. We assume that the sorted lists A_m and B_n to be merged are stored on tapes (or other external devices) if they are too large to be accommodated in core. These can then be read in sequentially in sorted order as needed and the elements of the merged list C_{m+n} written in similar sorted order onto output devices as soon as they are sequentially determined. As can be seen from the description of the algorithm g, for efficient comparison we need the elements a_m and those in B_n from b_x to b_n in core. This requires a storage space of 2^α elements ($\alpha = \lfloor \log_2 (n/m) \rfloor$) which is approximately equal to n/m. In general, this will not be excessive. For example, if $n = 10^7$ and $m = 10^4$, an average of 10^3 elements of B are required to be in core and this ratio will be approximately maintained if the data in B_n and A_m are uniformly distributed in some interval. If n/m becomes too large, a slight modification of the algorithm can be made, say, to compare a_m with b_x, where $x = n - 2^\beta + 1$ for some smaller β, without substantially affecting its efficiency.

Assuming the data in A_m and B_n are uniformly distributed in some interval, the expected number of comparisons $E_t(m, n)$ required by the tape merge algorithm t can be seen to satisfy the following recurrence relation:

$$\text{(R)} \qquad E_t(m, n) = 1 + \frac{m}{m + n} E_t(m - 1, n) + \frac{n}{m + n} E_t(m, n - 1); \qquad E_t(1, 1) = 1.$$

Solving (R), we have

$$E_t(m, n) = mn\left(\frac{1}{m + 1} + \frac{1}{n + 1}\right)$$

$$= m + n - \left(\frac{m}{n + 1} + \frac{n}{m + 1}\right),$$

which is only slightly less than $K_t(m, n) = m + n - 1$.

When n/m is large, as in the case of updating telephone directories or library materials, we see that $E_t(m, n) \approx m + n - n/m$ can be considerably larger than $K_g(m, n) \approx (3 + \lfloor \log_2 (n/m) \rfloor)m$, the maximal number of comparisons required using the algorithm g. Even when the logic involved in making one comparison using the proposed algorithm g is more involved than making one comparison using the tape merge algorithm t, substantial savings in computer time can be achieved. A computer program (FORTRAN, GE-635), implementing both the tape merge algorithm t and the generalized binary algorithm g (hopefully with equal degrees of efficiency), was written to test our assertions on some problems with

randomly generated data. The results are presented in Table 4. As can be seen, the saving in time is great when n/m is large.

TABLE 4

m	n	n/m	C_t	C_g	T_t	T_g	T_g/T_t
150	1500	10	1649	784	75.6	40.2	$\approx 1/2$
100	2000	20	2099	620	94.8	32.6	$\approx 1/3$
100	10000	100	10069	765	462.0	58.0	$\approx 1/8$
20	3000	150	2873	180	131.5	13.5	$\approx 1/10$

C_t = number of comparisons made by the tape merge algorithm t;
C_g = number of comparisons made by the generalized binary algorithm g;
T_t = time (in milliseconds) spent in making the comparisons using t;
T_g = time (in milliseconds) spent in making the comparisons using g.

REFERENCES

[1] F. K. HWANG AND S. LIN, *Optimal merging of* 2 *elements with n elements*, Acta Informatica, 1 (1971), pp. 145–158.
[2] ———, *Some optimal results for merging two disjoint linearly-ordered sets*, internal memorandum.

SIAM J. COMPUT.
Vol. 1, No. 1, March 1972

SYMMETRIES IN DATA GRAPHS*

ARNOLD L. ROSENBERG†

Abstract. Data graphs were introduced as a vehicle for studying uniformities in the structure of graphs underlying data structures. This paper is devoted to investigating symmetries in data graphs. Such symmetries are of no little significance in every phase of the computational process. They can often be exploited to formulate more efficient algorithms, to simplify the specification of these algorithms, and to facilitate analysis of the resulting programs. The main thrust of this paper is to investigate the influence of various structural features of data graphs on the types of symmetries the data graphs enjoy: what features ensure the presence of symmetries of various types, and what features limit the possible types of symmetries. These questions are studied first on the class of all data graphs, and then on the class of addressable (= realizable by relative addressing) data graphs.

Key words. Data graph, symmetry, automorphism, addressable data graph, addressing scheme

1. Introduction. A *data graph* can be viewed as a strongly-connected labeled directed graph or, alternatively, as a finitely generated transitive partial operand [1, pp. 250 ff.]. From the former vantage point, one can employ data graphs to study uniformities in the structure of data structures. The latter vantage point permits one to enlist the theory of semigroups in these studies. Numerous other graph-like models for data structures have been formulated; an extensive bibliography on this subject appears in [4].

In [2], [3] we studied certain uniformities in data graphs, which relate to the task of realizing these data graphs in a random access memory. The main results of those papers were characterizations of the class of data graphs which are realizable by "relative addressing" [1] and of the proper subclass whose members admit "relocatable" realizations. In each case we presented a characterization which favored the graph-oriented point of view and one which was more algebraic in nature. We further obtained a number of results concerning the structure of data graphs in these classes, which suggest that the classes are of interest independent of the motivating problem of implementation.

The present paper is devoted to studying another class of uniformities of data graphs, which are important in a computational environment. From a graph-oriented point of departure, the uniformities studied are *symmetries* of data graphs; from an algebraic viewpoint, they can be thought of as data graph *automorphisms*. The main thrust of our investigation is to ascertain what types of symmetries a data graph can possess, and how the set of symmetries of a data graph is constrained by various features in the structure of the graph. Of special interest is the relationship between the symmetries of a data graph and various other types of uniformity it may enjoy.

---

* Received by the editors August 4, 1971, and in revised form December 22, 1971.

† 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.

[1] For background, we have included definitions of "realization" and "relative addressing" in an Appendix which presupposes the notions in § 2.

Motivating our study is the observation that symmetries in data structures can be of no little significance in a computational environment. At every stage of the computing process, symmetries can be exploited. They are often helpful in formulating efficient algorithms, in simplifying the specification of algorithms, and in analyzing the resulting programs. If parallel computing facilities are available, one may be able to exploit symmetries by using several (virtually identical) copies of a single program to operate on symmetric portions of a data graph. Moreover, one can often enlist symmetries to facilitate the detection of other uniformities of interest, for instance, those which guarantee relative addressability. The development in the sequel is strongly influenced by this computational motivation.

The remainder of the paper is organized in three sections. Section 2 introduces the basis of our study, data graphs and symmetries thereof. An elementary result which is useful in the sequel is presented, and some pertinent related literature is mentioned. Section 3 is devoted to investigating the set of symmetries of a data graph. Particular attention is paid to fixed points of symmetries, to symmetries induced by links in the graph, and to conditions which guarantee the presence of symmetries. In § 4, we coalesce the material of § 3 with that of our earlier papers [2], [3] by considering symmetries of addressable (= realizable by relative addressing) data graphs. The presence of the additional uniformities allows us to sharpen materially a number of results from § 3. We discover a strong relationship between the symmetries of an addressable data graph and its "addressing schemes." This relationship lends insight into the nature of both types of structural uniformity; it further facilitates verifying the presence of one type of uniformity in the known presence of the other. Throughout the paper, we present sample applications of the results derived.

We have attempted, insofar as is feasible, to make this paper self-contained. Perforce we could not reproduce from [2] our discussion of the considerations which led to the form of our model.

Preliminary to our technical development, a word about notation will be useful. Since we are studying uniformities in data graphs, it is not surprising that many of our results are most succinctly presented as asserting the commutativity of diagrams. To facilitate subsequent exposition, we establish the following conventions. Let A, B, C, D be sets; let $\alpha : A \to B$, $\beta : B \to D$, $\gamma : A \to C$ and $\delta : C \to D$ be *partial* functions. The symbology

$$\begin{array}{ccc} A & \xrightarrow{\alpha} & B \\ {\scriptstyle\gamma}\downarrow & = & \downarrow{\scriptstyle\beta} \\ C & \xrightarrow[\delta]{} & D \end{array}$$

is intended to mean that $\alpha\beta = \gamma\delta$ as partial functions from A into D; i.e., the two functions are simultaneously defined and are equal on their joint domain.

2. Data graphs and symmetries. In this section we define the basic notions to be studied and mention briefly some related work. While we attempt to motivate our formal notion of symmetry, we make no analogous arguments about the data graph model. The considerations which shaped our formulation

of data graphs are discussed at length in §§ 1 and 3 of [2], to which the interested reader is referred.

2.1. Data graphs. A *data graph* is specified as an ordered pair $\Gamma = (C, \Lambda)$, where

(i) C is a countable set (of *data cells*);

(ii) Λ is a finite set of partial transformations of C (the *link-transformations*); subject to the condition

(iii) for each pair of cells $c, d \in C$, there is a transformation $\xi \in \Lambda^\tau$ such that $c\xi = d$.[2]

One can view the system (C, Λ) as a labeled directed graph in a straightforward manner: The set C is the set of vertices of the graph; there is a directed edge $\langle c, c\lambda \rangle$, labeled by λ, from cell c to cell $c\lambda$ precisely when $c \in \Delta(\lambda)$ (the *domain* of λ). Under this interpretation, condition (iii) above asserts the strong-connectivity of the graph.

In order to illustrate our model and to motivate the sequel, we present two data graphs which display two distinct types of symmetry.

Example 2.1. Our first example, depicted in Fig. 2.1, is a tree[3] with two successors and a single predecessor. More formally,

$$\Gamma_{2.1} = (C, \Lambda),$$

where

(a) $C = N = \{1, 2, 3, \cdots\}$;

(b) $\Lambda = \{\sigma_r, \sigma_l, \pi\}$;

for each $n \in C$,

$$\begin{aligned} n\sigma_r &= 2n, \text{ the } \textit{right-successor}, \\ n\sigma_l &= 2n+1, \text{ the } \textit{left-successor}, \\ n\pi &= \lfloor n/2 \rfloor, \text{ the } \textit{predecessor}. \end{aligned}$$

Here $\lfloor q \rfloor$ denotes the integer part of the real number q. Note that both σ_r and σ_l are total, while π is not since $1\pi = 0 \notin N$.

Example 2.2. Our second example, depicted in Fig. 2.2, is a one-ended ladder. More formally,

$$\Gamma_{2.2} = (C, \Lambda),$$

where

(a) $C = N \times \{0, 1\}$;

(b) $\Lambda = \{\sigma, \pi, \varphi\}$;

---

[2] Throughout the paper, Λ^τ denotes the monoid generated by Λ under functional composition, with identity 1_C (the identity map on C).

[3] Such anthropomorphic names for our data graphs are presented only to aid the reader's intuition; no formal connotations should be assumed.

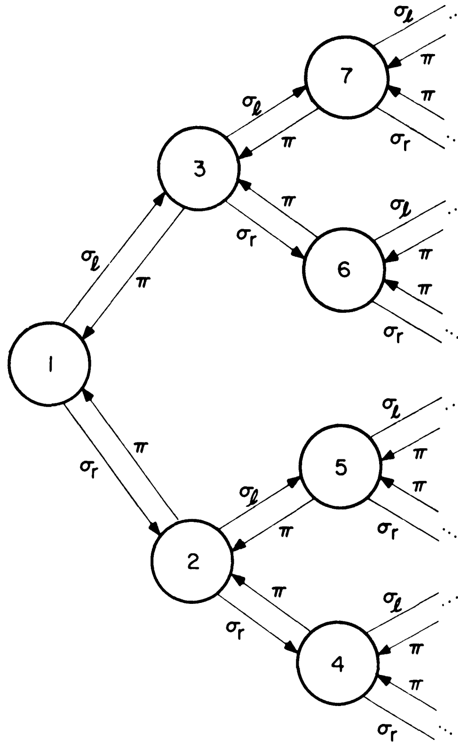


FIG. 2.1. $\Gamma_{2.1}$: *A binary tree*

for each $\langle n, a \rangle \in C$,

$$\langle n, a\rangle\sigma = \langle n + 1, a\rangle, \text{ the } \textit{successor},$$
$$\langle n, a\rangle\pi = \langle n - 1, a\rangle, \text{ the } \textit{predecessor},$$
$$\langle n, a\rangle\varphi = \langle n, a + 1 (\text{mod } 2)\rangle, \text{ the } \textit{flip}.$$

Note that, in this example, the functions σ and φ are total, but π is not since $\langle 1, a\rangle\pi = \langle 0, a\rangle \notin C$.

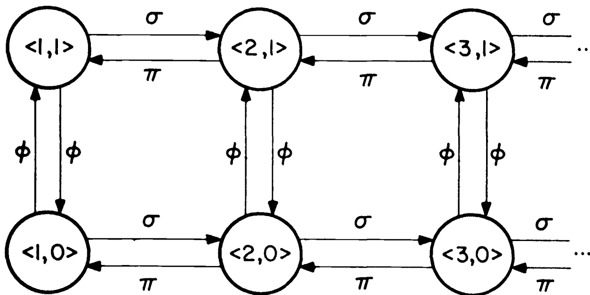


FIG. 2.2. $\Gamma_{2.2}$: *A one-ended ladder*

2.2. Symmetries. A glance at Figs. 2.1 and 2.2 renders obvious the assertion that the two data graphs presented enjoy nontrivial symmetries. Our formal notion of symmetry is shaped by considering these data graphs, and certain special types of symmetry emerge from this consideration.

Looking first at $\Gamma_{2.2}$, we observe that the shape of this "ladder" is unchanged by interchanging the "uprights." More formally, let us define the bijection $\beta: N \times \{0, 1\} \to N \times \{0, 1\}$ by

$$\langle n, 0\rangle\beta = \langle n, 1\rangle, \qquad \langle n, 1\rangle\beta = \langle n, 0\rangle,$$

for each $n \in N$. The system $(C\beta, \Lambda)$ is then isomorphic[4] to $\Gamma_{2.2} = (C, \Lambda)$. Thus, the bijection β exposes (or induces) a nontrivial symmetry of $\Gamma_{2.2}$. Two properties of this symmetry are worthy of note: (i) The symmetry leaves unchanged the set Λ of link-transformations. (ii) The symmetry is induced by an element of Λ^τ; in fact, $\beta = \varphi$.

Turning now to $\Gamma_{2.1}$, we remark that this tree enjoys a more complicated type of symmetry, one which has neither of the previously noted properties. Intuitively, the symmetry alluded to permits one to "fold" the tree about its root (cell 1) so that its two maximal subtrees coincide. This symmetry requires interchanging not only cells, but also links. Formally, let us define a cell bijection $\beta_c: C \to C$ and a link bijection $\beta_l: \Lambda \to \Lambda$ as follows.

(a) For each $k \in N$ and $i \in \{0, \cdots, 2^k - 1\}$,

$$(2^k + i)\beta_c = 2^{k+1} - (i + 1);$$

(b) $$\sigma_r\beta_l = \sigma_l, \quad \sigma_l\beta_l = \sigma_r, \quad \pi\beta_l = \pi.$$

One easily verifies that the system $(C\beta_c, \Lambda\beta_l)$ is isomorphic to the data graph $\Gamma_{2.1}$. Further, this symmetry does not leave elements of Λ unchanged; nor is it induced by a link transformation—i.e., $\beta_c \notin \Lambda^\tau$.

Our definition of a symmetry is a direct generalization of the latter example. The former example, however, suggests special types of symmetries which are studied further in the sequel.

DEFINITION. A *symmetry* of a data graph $\Gamma = (C, \Lambda)$ is specified by a pair of bijections $\Sigma = (\beta_c, \beta_l)$, where

(i) $\beta_c: C \to C$ is the *cell bijection*;

(ii) $\beta_l: \Lambda^\tau \to \Lambda^\tau$, the *link bijection*, is a monoid isomorphism which maps Λ onto Λ;

subject to the condition,

(iii) for all $\lambda \in \Lambda$,

$$\begin{array}{ccc} C & \xrightarrow{\lambda} & C \\ {\scriptstyle\beta_c}\downarrow & = & \downarrow{\scriptstyle\beta_c} \\ C & \xrightarrow[\lambda\beta_l]{} & C \end{array}$$

As we noted earlier, a symmetry can be viewed in algebraic terms as a data graph automorphism. In [1], the notion of an operand automorphism is discussed

[4] The intended notion of isomorphism should be obvious.

briefly. In the present context, this latter notion corresponds to a *fixed-link* symmetry, i.e., one with $\beta_l = 1_{\Lambda^\tau}$.

In order to emphasize the restriction that β_l map Λ *onto* Λ, we present one final example of a symmetry. This example indicates that the gross topology of a data graph does not suffice to characterize its symmetries.

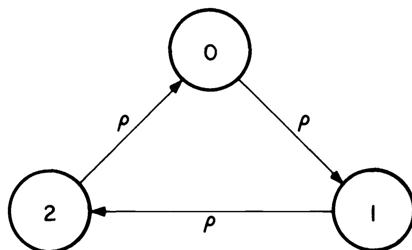


FIG. 2.3. $\Gamma_{2.3}$: *A triangle with only rotational symmetries*

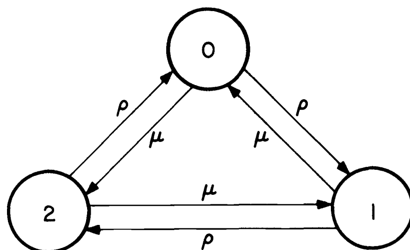


FIG. 2.4. $\Gamma_{2.4}$: *A triangle with a "complete" set of symmetries*

Example 2.3. Consider the following two "triangles," depicted in Figs. 2.3 and 2.4, respectively.

(A) $$\Gamma_{2.3} = (C, \Lambda_1),$$

where

(a) $C = \{0, 1, 2\}$;
(b) $\Lambda = \{\rho\}$;

for each $n \in C$, $n\rho = n + 1 \pmod 3$.

(B) $$\Gamma_{2.4} = (C, \Lambda_2),$$

where

(c) $\Lambda_2 = \{\rho, \mu\} = \Lambda_1 \cup \{\mu\}$;

for each $n \in C$, $n\mu = n + 2 \pmod 3) = \rho^2$.[5]

[5] If λ is a transformation of a set C, then $\lambda^0 = 1_C$, $\lambda^1 = \lambda$ and $\lambda^{k+1} = \lambda\lambda^k$ for all $k \in N$; thus, the multiplication is in Λ^τ.

Now the group of symmetries of a triangle comprises six symmetries, corresponding to the following permutations of the cells:

(i) $\theta_1 = (0)(1)(2)$, the identity,
(ii) $\theta_2 = (012)$, rotation 1,
(iii) $\theta_3 = (021)$, rotation 2,
(iv) $\theta_4 = (0)(12)$, flip 1,
(v) $\theta_5 = (01)(2)$, flip 2,
(vi) $\theta_6 = (02)(1)$, flip 3.

Both $\Gamma_{2.3}$ and $\Gamma_{2.4}$ have symmetries (i)–(iii), as fixed link symmetries. In contrast, $\Gamma_{2.4}$ enjoys symmetries (iv)–(vi), while $\Gamma_{2.3}$ does not. To verify this, let us restrict attention, with no loss of generality, to (iv).

For $\Gamma_{2.3}$: $1\rho = 2$, $1\theta_4 = 2$, $2\rho = 0$, $2\theta_4 = 1$. Thus,

$$1\rho\theta_4 = 1 \neq 0 = 1\theta_4\rho.$$

(Since Λ_1 is a singleton, we have no choice but to fix links.)

For $\Gamma_{2.4}$: Define β_l by $\rho\beta_l = \mu$, $\mu\beta_l = \rho$. Then we have

$$0\rho\theta_4 = 2 = 0\theta_4\mu, \qquad 0\mu\theta_4 = 1 = 0\theta_4\rho,$$

$$1\rho\theta_4 = 1 = 1\theta_4\mu, \qquad 1\mu\theta_4 = 0 = 1\theta_4\rho,$$

$$2\rho\theta_4 = 0 = 2\theta_4\mu, \qquad 2\mu\theta_4 = 2 = 2\theta_4\rho.$$

Thus, $\Gamma_{2.4}$ enjoys all the symmetries of a triangle, while $\Gamma_{2.3}$ has only rotations. Note that $\mu = \rho^2$ so that, if we dropped the requirement that β_l map Λ onto Λ, the distinction just noted would disappear. However, from a practical point of view it appears to be essential, given the complexity of data structures, to operate insofar as is possible with the data graph which is presented rather than with its transitive closure. This is all the more important in view of our strong-connectivity postulate.

We close this preliminary section with an elementary result that is extremely useful in the sequel. This result asserts that symmetries of even infinite data graphs can be specified in a very simple finite manner.

THEOREM 2.1. *Let $\Gamma = (C, \Lambda)$ be a data graph. Given an arbitrary bijection $\beta : \Lambda \to \Lambda$ and an arbitrary pair of cells $\langle c, d\rangle \in C \times C$, there is at most one symmetry $\Sigma = (\beta_c, \beta_l)$ of Γ which satisfies*: (i) $\beta \subseteq \beta_l$, *and* (ii) $\{\langle c, d\rangle\} \subseteq \beta_c$.[6]

Proof. The result follows from the strong connectivity of data graphs. To wit, let $\Sigma = (\beta_c, \beta_l)$ be a symmetry of $\Gamma = (C, \Lambda)$. Let $c_0 \in C$ be designated.

Pick an arbitrary cell $e \in C$. By strong-connectivity, there is a transformation $\xi = \lambda_1 \cdots \lambda_n \in \Lambda^\tau$ such that $c_0\xi = e$. By a straightforward induction on the definition of symmetry, one finds that

$$e\beta_c = (c_0\xi)\beta_c = (c_0\beta_c)(\xi\beta_l).$$

Moreover, we know that $\xi\beta_l = (\lambda_1\beta_l) \cdots (\lambda_n\beta_l)$ since β_l is a monoid isomorphism.

Thus, if we are given the cell $d = c_0\beta_c$ and the restriction[7] β of β_l to Λ, we know that $e\beta_c = d(\lambda_1\beta) \cdots (\lambda_n\beta)$. In other words, the entire symmetry Σ is induced by the pair $\langle c_0, d\rangle$ and the finite function β, as was claimed.

---

[6] Whenever convenient, we identify a function with its graph; i.e., we view a function as a relation; hence, the notation "$\subseteq$".

[7] Since β_l maps Λ onto Λ, β must be a bijection of Λ onto Λ.

2.3. Related work. The model of a data graph is so general as to admit a multitude of disparate interpretations. One can, for example, view a data graph as an outputless automaton which is incompletely specified (since elements of Λ need not be total); C is the set of states and Λ the set of inputs to the "automaton." From this viewpoint our subject of study is input-renaming (since β_l can be nontrivial) automorphisms of automata. There is a rich literature on (input-fixing) automorphisms of (completely specified) automata; selections [5], [6] from this literature have particular pertinence to the present paper. Several of the results in the sequel can be viewed as generalizations of results in [5], [6], the generalizations arising from the partialness of elements of Λ and the nontriviality of β_l. Although the proof techniques of the cited papers often find close analogues in our proofs of these results, the two generalized assumptions demand full proofs rather than citations. In §§ 3.3 and 3.4 particularly, very close analogues of certain results in [5], [6] appear; in fact there is some duplication in one or two instances. The reader may find it of interest to compare the results and development in the cited papers with that of the present paper, given the materially different points of departure.

3. General properties of symmetries. In this section we investigate properties of symmetries, which do not presuppose additional structure in the data graph. We begin with a number of elementary results which expose certain relationships between cells/links and their images under symmetries. We then investigate symmetries with fixed points, either fixed cells or fixed links. Next, we consider the conditions under which a symmetry of $\Gamma = (C, \Lambda)$ is induced by a transformation $\xi \in \Lambda^\tau$; the reader will recall that the data graph $\Gamma_{2.2}$ has such a symmetry. Finally, we note certain conditions which guarantee the presence of certain types of symmetries.

3.1. Elementary properties of symmetries. Our identification of symmetries as automorphisms of data graphs presages our first result.

THEOREM 3.1. *The set of symmetries of a data graph is a group.*

Proof. If $\Sigma_1 = (\beta_c^{(1)}, \beta_l^{(1)})$ and $\Sigma_2 = (\beta_c^{(2)}, \beta_l^{(2)})$ are symmetries of $\Gamma = (C, \Lambda)$, then their product symmetry is defined to be

$$\Sigma_1\Sigma_2 = (\beta_c^{(1)}\beta_c^{(2)}, \beta_l^{(1)}\beta_l^{(2)}).$$

We show that the set of symmetries of Γ is a group under this (obviously well-defined) multiplication.

1. The associativity of the defined multiplication follows immediately from the associativity of functional composition.

2. The *identity symmetry* $\Sigma_I = (1_C, 1_{\Lambda^\tau})$ is an identity under this multiplication.

3. Given a symmetry $\Sigma = (\beta_c, \beta_l)$, we claim that the pair $\Sigma^{-1} = (\beta_c^{-1}, \beta_l^{-1})$ is a symmetry which is inverse to Σ under the defined multiplication. First note that $\Sigma\Sigma^{-1} = \Sigma^{-1}\Sigma = \Sigma_I$. Thus it suffices to show that Σ^{-1} is a symmetry. Clearly β_c^{-1} and β_l^{-1} are both bijections of the appropriate types. Further, for all $c \in C$ and all $\lambda \in \Lambda$:

(i) If $(c\beta_c^{-1}) \in \Delta(\lambda\beta_l^{-1})$, then

$$((c\beta_c^{-1})(\lambda\beta_l^{-1}))\beta_c = (c\beta_c^{-1}\beta_c)(\lambda\beta_l^{-1}\beta_l) = c\lambda;$$

hence, $c \in \Delta(\lambda)$ and

$$(c\lambda)\beta_c^{-1} = (c\beta_c^{-1})(\lambda\beta_l^{-1}).$$

(ii) If $c \in \Delta(\lambda)$, then immediately,

$$c\lambda = (c\beta_c^{-1}\beta_c)(\lambda\beta_l^{-1}\beta_l);$$

but, since Σ is a symmetry,[8] this implies $c\beta_c^{-1} \in \Delta(\lambda\beta_l^{-1})$.

This establishes part 3, and the theorem is proved.

Our next result sets off the argument in part 3 of the preceding proof. While the result is obvious, it has numerous applications, and so merits explicit mention.

THEOREM 3.2. *Let $\Sigma = (\beta_c, \beta_l)$ be a symmetry of the data graph $\Gamma = (C, \Lambda)$.*

(a) *For each $\xi \in \Lambda^\tau$, $\Delta(\xi\beta_l) = (\Delta(\xi))\beta_c$.*

(b) *For all $c \in C$ and all $\xi \in \Lambda^\tau$, $c \in \Delta(\xi)$ if and only if $c\beta_c \in \Delta(\xi\beta_l)$.*

Proof. A straightforward induction on the diagram defining symmetries yields, for each $\xi \in \Lambda^\tau$, the diagram

$$\begin{array}{ccc} C & \xrightarrow{\xi} & C \\ \beta_c \downarrow & = & \downarrow \beta_c \\ C & \xrightarrow[\xi\beta_l]{} & C \end{array}$$

COROLLARY 3.1. *The image of a total (respectively, nontotal) link-transformation under a symmetry is again a total (respectively, a nontotal) link-transformation.*

COROLLARY 3.2 *If only total link-transformations are defined at a given cell, the same must be true of the image of that cell under any symmetry.*

As we come upon general principles which have direct applications, we shall present samples of these applications. Corollaries 3.1 and 3.2 yield such applicable principles for delimiting the set of symmetries of data graphs.

PROPOSITION 3.1. *The data graph $\Gamma_{2.1}$ enjoys precisely two symmetries, namely, the identity symmetry and the symmetry presented at the beginning of* § 2.2. (This latter must, by Theorem 3.1, be self-inverse.)

Proof. Our assertion is verified via the following observations.

Observation 1. The cell denoted 1 is fixed by all symmetries.[9]

One easily verifies that cell 1 is the unique cell of $\Gamma_{2.1}$ which is in the domain of only total link transformations. (This is verified rigorously in [3].) By Corollary 3.2, therefore, cell 1 must be fixed by all symmetries.

Observation 2. The transformation π is fixed by all symmetries.

Since σ_r and σ_l are both total while π is not, this assertion is immediate by Corollary 3.1 and the fact that β_l maps Λ onto Λ.

Thus, the only symmetries (β_c, β_l) of $\Gamma_{2.1}$ are those for which $\beta_l = 1_{\Lambda^\tau}$, and those for which $\sigma_r\beta_l = \sigma_l$ and $\sigma_l\beta_l = \sigma_r$. By Observations 1 and 2 in conjunction with Theorem 2.1, $\Gamma_{2.1}$ can have at most one of each type (since either case would specify the pair $\langle 1, 1\rangle \in C \times C$ and the restriction of β_l to Λ). However, we have exhibited one of each type, so the proposition is established.

[8] Recall that the defining diagram asserted that, for all $d \in C$ and $\mu \in \lambda$, $d \in \Delta(\mu)$ if and only if $d\beta_c \in \Delta(\mu\beta_l)$. Apply this with $d = c\beta_c^{-1}$ and $\mu = \lambda\beta_l^{-1}$.

[9] That is, $1\beta_c = 1$ for any symmetry (β_c, β_l) of $\Gamma_{2.1}$.

To lead to a second application, we present the following data graph.

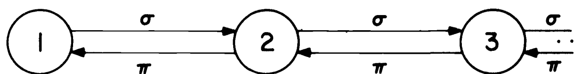


FIG. 3.1. $\Gamma_{3.1}$: *A one-ended linear list*

Example 3.1. We present a linear list, depicted in Fig. 3.1.

$$\Gamma_{3.1} = (C, \Lambda),$$

where

(i) $C = N$;
(ii) $\Lambda = \{\sigma, \pi\}$;

for each $n \in C$,

$n\sigma = n + 1$, the *successor* link,
$n\pi = n - 1$, the *predecessor* link.

PROPOSITION 3.2. *The data graph $\Gamma_{3.1}$ has no symmetry save the identity symmetry.*

Proof. Again, a series of observations yields the result.

Observation 1. Cell 1 is fixed by every symmetry.

As in the preceding proposition, cell 1 is unique in being in the domain of only total transformations. We can, therefore, invoke Corollary 3.2.

Observation 2. Both σ and π are fixed by all symmetries.

This is immediate by Corollary 3.1 since σ is the unique total, and π the unique nontotal member of Λ.

As before, an invocation of Theorem 2.1 completes the proof.

We are not yet in a position to characterize the symmetries of the other data graphs we have introduced. We turn now to a study of fixed points of symmetries; new principles, which give us a handle on these other data graphs, will emerge directly. One final general result will facilitate the development.

THEOREM 3.3. *Let $\Sigma = (\beta_c, \beta_l)$ be a symmetry of $\Gamma = (C, \Lambda)$. For all $\lambda \in \Lambda$, both λ and $\lambda\beta_l$ have the same order.*[10]

Proof. The proof is obvious, since β_l is a monoid isomorphism.

3.2. Fixed points of symmetries. Our study of fixed points of symmetries is dually motivated. First, one would expect "partially degenerate" symmetries—those for which one of the bijections is the identity map—to obey stricter laws than do general symmetries. Such strengthened properties may prove computationally significant. Second, in studying parallel computation on data graphs, as we suggested in the Introduction, cells which are fixed by symmetries represent potential memory conflicts. More information about how such fixed points of

[10] The *order* of λ in Λ is the smallest *positive* integer k such that $\lambda^k = 1_C$; it is 0 if no such k exists.

β_c occur may help to detect and avoid such problems. We begin with some general results on partially degenerate symmetries.

We call $\Sigma = (\beta_c, \beta_l)$ a *fixed-cell* symmetry if $\beta_c = 1_C$, and we call it a *fixed-link* symmetry if $\beta_l = 1_{\Lambda^\tau}$; in the former case we write $\Sigma = (1, \beta_l)$ and in the latter $\Sigma = (\beta_c, 1)$.

THEOREM 3.4 (a) *If Σ is a fixed-cell symmetry, then $\Sigma = \Sigma_I$. That is, the identity symmetry is the unique symmetry which fixes all cells.*

(b) *If $\Sigma = (\beta_c, 1)$ is a fixed-link symmetry, and if β_c has a fixed point, then $\Sigma = \Sigma_I$. That is, nonidentity fixed-link symmetries can fix no cells.*

Proof. (a) For arbitrary $\lambda \in \Lambda$, if $\beta_c = 1_C$, then for all $c \in C$,

$$c\lambda = (c\lambda)\beta_c = (c\beta_c)(\lambda\beta_l) = c(\lambda\beta_l).$$

In other words, $\lambda = \lambda\beta_l$. Since λ was arbitrary, we conclude $\Sigma = \Sigma_I$.

(b) If $\beta_l = 1_{\Lambda^\tau}$, and if $c_0\beta_c = c_0$ for some cell c_0, then $\Sigma = \Sigma_I$ by Theorem 2.1.

The reader will recall that the nonidentity symmetry of $\Gamma_{2.2}$, described at the beginning of § 2.2, fixes no cells. Theorem 3.4 assures us that this property is not mere coincidence, since that symmetry fixes all links. The nonidentity symmetry which we found for $\Gamma_{2.1}$ demonstrates that a symmetry can fix *some* cells and *some* links without degenerating.

Theorem 3.4 has a direct application to $\Gamma_{2.3}$.

PROPOSITION 3.3. *The data graph $\Gamma_{2.3}$ has no nonidentity symmetry which fixes a cell. In particular, it does not enjoy any of the "flip" symmetries.*

Proof. Since Λ is a singleton, any symmetry of $\Gamma_{2.3}$ must fix links. Theorem 3.4(b) thus yields the result.

In [2] we briefly considered fixed-link symmetries under the name *transpositions.* A result from [3], which deals with a weaker version of transpositions, yields the following theorem which supersedes Theorem 3.4(b). (See Lemma 4.3 below.)

THEOREM 3.5. *Any two fixed-link symmetries are either equal or disjoint. That is, if $c\beta_c^{(1)} = c\beta_c^{(2)}$ for some cell c, then $\beta_c^{(1)} = \beta_c^{(2)}$.*

Clearly Theorem 3.4(b) is an immediate corollary since any β_c with a fixed point intersects the identity 1_C, hence must coincide with it.

When Σ is a fixed-link symmetry, Theorem 3.2 can be sharpened. We state this result as a corollary of that theorem.

COROLLARY 3.3. *Let $\Sigma = (\beta_c, 1)$ be a symmetry of $\Gamma = (C, \Lambda)$. For all $c \in C$ and all $\xi \in \Lambda^\tau$, $c \in \Delta(\xi)$ if and only if $c\beta_c \in \Delta(\xi)$.*

We are now in a position to characterize the symmetries of $\Gamma_{2.2}$.

PROPOSITION 3.4. *The data graph $\Gamma_{2.2}$ enjoys precisely two symmetries, namely, the identity symmetry and the symmetry presented at the beginning of* § 2.2. (By Theorem 3.1, the latter must be self-inverse.)

Proof. The assertion follows via the following observations.

Observation 1. Every symmetry of $\Gamma_{2.2}$ fixes all links.

By Corollary 3.1, the transformation π is fixed by all symmetries, since it is the only nontotal element of Λ. By Theorem 3.3, neither of σ or φ can be the image of the other under any symmetry since $\varphi^2 = 1_C \neq \sigma^2$. We must, therefore, conclude that all links are fixed by all symmetries.

Observation 2. Any symmetry of $\Gamma_{2.2}$ either fixes cells $\langle 1, 0\rangle$ and $\langle 1, 1\rangle$, or interchanges them.

This is obvious by Corollary 3.3 since cells $\langle 1, 0\rangle$ and $\langle 1, 1\rangle$ are the sole elements of $C - \Delta(\pi)$; moreover, for all $\xi \in \Lambda^\tau$, $\langle 1, 0\rangle \in \Delta(\xi)$ if and only if $\langle 1, 1\rangle \in \Delta(\xi)$.

Observations 1 and 2 combine with Theorem 2.1 to assure that $\Gamma_{2.2}$ has at most two symmetries, namely, one that fixes cells $\langle 1, 0\rangle$ and $\langle 1, 1\rangle$, and one that interchanges them. However, we have exhibited one of each type. The proposition follows.

We now turn from considering symmetries which fix all links to consider briefly symmetries which fix some links. Although problems concerning such fixed points are of great significance, we have regrettably few results to report on this topic. In § 4, we have considerably more to say about fixed points of symmetries of addressable data graphs.

THEOREM 3.6. *Let $\Sigma = (\beta_c, \beta_l)$ be a symmetry of $\Gamma = (C, \Lambda)$. For each $\xi \in \Lambda^\tau$, we have $\xi\beta_l = \xi$ precisely when*

$$\begin{array}{ccc} C & \xrightarrow{\xi} & C \\ {\scriptstyle \beta_c}\downarrow & = & \downarrow{\scriptstyle \beta_c} \\ C & \xrightarrow[\xi]{} & C \end{array}$$

Proof. If $\xi\beta_l = \xi$, then, for all $c \in \Delta(\xi)$, $(c\xi)\beta_c = (c\beta_c)(\xi\beta_l) = (c\beta_c)\xi$.

Conversely, if the indicated diagram holds, then, for all $c \in \Delta(\xi)$, $(c\xi)\beta_c = (c\beta_c)\xi = (c\beta_c)(\xi\beta_l)$. Hence, restricted to the set $(\Delta(\xi))\beta_c$, $\xi = \xi\beta_l$. However, by Theorem 3.2(a), $(\Delta(\xi))\beta_c = \Delta(\xi\beta_l)$; by the diagram, $(\Delta(\xi))\beta_c = \Delta(\xi)$. It follows, therefore, that $\xi = \xi\beta_l$, as was claimed.

One easily verifies that the set of all links which are fixed by a symmetry is not devoid of structure.

THEOREM 3.7. *Let $\Sigma = (\beta_c, \beta_l)$ be a symmetry of $\Gamma = (C, \Lambda)$. The set $\Phi = \{\xi \in \Lambda^\tau | \xi\beta_l = \xi\}$ is a submonoid of Λ^τ, qua monoid of transformations. In contrast, the set $\Lambda^\tau - \Phi$ need not be a semigroup.*

Proof. The structure of Φ is easily discernible from the fact that β_l is a monoid isomorphism. The lack of structure of $\Lambda^\tau - \Phi$ can be observed in any of the "flip" symmetries of $\Gamma_{2.4}$. We showed in Example 2.3 that each of these symmetries exchanges the transformations ρ and μ; thus, $\{\rho, \mu\} \subseteq \Lambda^\tau - \Phi$ but $\rho\mu = \mu\rho = 1_C \in \Phi$.

It is not difficult to glean from our previous results conditions which ensure the nonemptiness of $\Phi \cap \Lambda$.

COROLLARY 3.4. *Each of the following conditions on Λ assures that any symmetry of $\Gamma = (C, \Lambda)$ fixes some link in Λ:*

Λ has an odd number of:

(a) *elements,*

(b) *total transformations,*

(c) *nontotal transformations,*

(d) *elements of a given order.*

Since Λ is a finite set, the conditions of this corollary are usually easy to check. Unfortunately though, these conditions may yield no substantive information

about the set Φ. In particular, there exist symmetries of data graphs for which the set Φ is infinite but is disjoint from Λ.

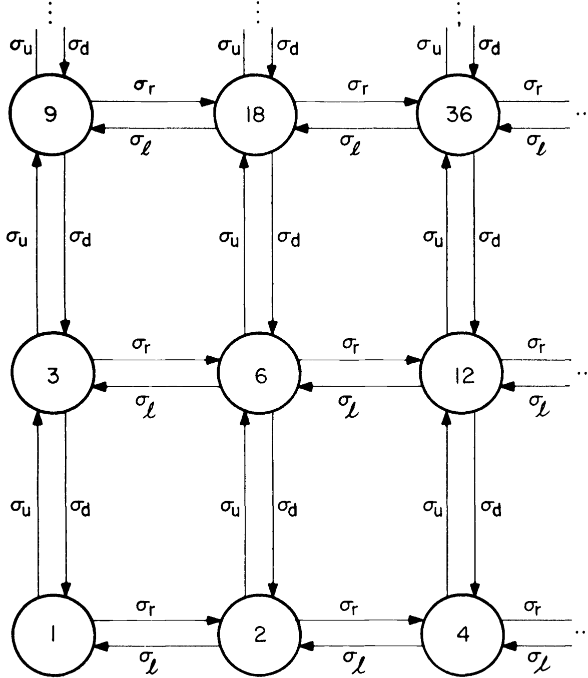


FIG. 3.2. $\Gamma_{3.2}$: *A two-dimensional quadrant*

Example 3.2. We present a two-dimensional quadrant graph, depicted in Fig. 3.2.

$$\Gamma_{3.2} = (C, \Lambda),$$

where

(i) $C = \{2^x 3^y | x, y \in N \cup \{0\}\}$;
(ii) $\Lambda = \{\sigma_r, \sigma_l, \sigma_u, \sigma_d\}$;

for each $n \in C$,

$n\sigma_r = 2n$, the *right*-move,
$n\sigma_l = n/2$, the *left*-move,
$n\sigma_u = 3n$, the *up*-move,
$n\sigma_d = n/3$, the *down*-move.

PROPOSITION 3.5. *The data graph $\Gamma_{3.2}$ enjoys precisely two symmetries, namely, Σ_I and the symmetry $\Sigma = (\beta_c, \beta_l)$, where*[11]

$$2^x 3^y \beta_c = 2^y 3^x \quad \textit{for all } x, y \in N \cup \{0\},$$

[11] Intuitively, Σ flips the quadrant about its diagonal.

and $\sigma_r\beta_l = \sigma_u$, $\sigma_u\beta_l = \sigma_r$, $\sigma_l\beta_l = \sigma_d$, $\sigma_d\beta_l = \sigma_l$. *Moreover, the latter symmetry fixes infinitely many links.*

Proof. Once again we proceed via a chain of observations.

Observation 1. Cell 1 is fixed by all symmetries (Corollary 3.2).

Since cell 1 must be fixed, no nonidentity symmetry of $\Gamma_{3.2}$ can fix links. Thus, by Corollary 3.1, either σ_r and σ_u are interchanged by a nonidentity symmetry, or σ_l and σ_d are. However, the relations $\sigma_r\sigma_l = \sigma_u\sigma_d = 1_C$ force one to conclude that a symmetry must effect both of these interchanges if it effects either. Since only two behaviors are possible for β_l, Theorem 2.1 and Observation 1 indicate that $\Gamma_{3.2}$ can have at most two symmetries. But we have exhibited two, so these must be the only ones.

Finally, since $\sigma_u\sigma_r = \sigma_r\sigma_u$, one notes that $(\sigma_r^n\sigma_u^n)\beta_l = \sigma_u^n\sigma_r^n = \sigma_r^n\sigma_u^n = (\sigma_u^n\sigma_r^n)\beta_l$ for all $n \in N \cup \{0\}$.

3.3. Symmetries induced by links. We say a symmetry $\Sigma = (\beta_c, \beta_l)$ of $\Gamma = (C, \Lambda)$ is *link-induced* if $\beta_c \in \Lambda^\tau$. We noted in § 2.2 and Proposition 3.4 that both symmetries of $\Gamma_{2.2}$ are link-induced—the identity by $\beta_c = 1_C$, and the other by $\beta_c = \varphi$. This section is devoted to studying such link-induced symmetries.

THEOREM 3.8. *Let* $\Gamma = (C, \Lambda)$ *be a data graph; let* $\xi \in \Lambda^\tau$ *be a bijection, and let* $\beta : \Lambda^\tau \to \Lambda^\tau$ *be a monoid isomorphism mapping* Λ *onto* Λ. Γ *enjoys the symmetry* $\Sigma = (\xi, \beta)$ *if and only if* $\lambda\xi = \xi(\lambda\beta)$ *for all* $\lambda \in \Lambda$.

Proof. The proof is obvious by definition.

THEOREM 3.9. *If* $\xi \in \Lambda^\tau$ *induces a symmetry* $\Sigma = (\xi, \beta)$, *then* ξ *is a fixed point of* β; *i.e.*, $\xi = \xi\beta$.

Proof. For all $c \in C$,[12] $c\xi^2 = (c\xi)\xi = (c\xi)(\xi\beta)$ since ξ induces the symmetry (ξ, β). Thus $\xi = \xi\beta$ on the set $C\xi$. However, ξ is surjective, so $C\xi = C$, and the result follows.

Thus, the fact that the link φ of $\Gamma_{2.2}$ is fixed by all symmetries was not due just to Theorem 3.3.

Let us now turn our attention to link-induced fixed-link symmetries. We obtain an interesting strengthening of Theorem 3.8.

DEFINITION. (i) Given any monoid M we define, as usual,

$$\text{center}(M) = \{a \in M | (\forall b \in M) ab = ba\}.$$

(ii) Given any monoid of transformations Λ^τ, we define

$$\text{unicenter}(\Lambda^\tau) = \{\xi \in \Lambda^\tau | \xi \text{ is injective}\} \cap \text{center}(\Lambda^\tau).$$

Thus, each $\xi \in \text{unicenter}(\Lambda^\tau)$ is one-to-one and commutes with all elements of Λ^τ.

LEMMA 3.1. *Let* $\Gamma = (C, \Lambda)$ *be a data graph. Every nonempty* $\xi \in \text{center}(\Lambda^\tau)$ *is total and surjective.*

Proof. (a) Let $c \in C$ be an arbitrary element of $\Delta(\xi)$. For arbitrary $d \in C$, there is, by strong-connectivity of Γ, a transformation $\eta \in \Lambda^\tau$ with $d\eta = c$. Since $\xi \in \text{center}(\Lambda^\tau)$, we have $c\xi = d\eta\xi = d\xi\eta$; thus $d \in \Delta(\xi)$ also. Since c, d were arbitrary, we conclude that ξ is total.

[12] Since ξ induces a symmetry, it must be total.

(b) Again, let $c \in C$ be arbitrary; by part (a), $c \in \Delta(\xi)$. By strong-connectivity, there is an $\eta \in \Lambda^\tau$ such that $(c\xi)\eta = c$. Since $\xi \in \text{center}(\Lambda^\tau)$ $c\xi\eta = c\eta\xi = c$, whence $c \in C\xi$. Since c was arbitrary, we conclude that ξ maps C onto C.

THEOREM 3.10. *Let* $\Gamma = (C, \Lambda)$ *be a data graph. The transformation* $\xi \in \Lambda^\tau$ *induces a fixed-link symmetry of* Γ *if and only if* $\xi \in \text{unicenter}(\Lambda^\tau)$.

Proof. Assume first that the system $(\xi, 1_{\Lambda^\tau})$ is a symmetry of Γ. By definition, ξ is one-to-one; moreover, by Theorem 3.8, $\xi \in \text{center}(\Lambda^\tau)$. Hence $\xi \in \text{unicenter}(\Lambda^\tau)$.

Conversely, if $\xi \in \text{unicenter}(\Lambda^\tau)$, then ξ is one-to-one by definition and is total and onto by Lemma 3.1. Since ξ is also in $\text{center}(\Lambda^\tau)$, Theorem 3.8 implies that ξ induces a fixed-link symmetry of Γ.

Theorem 3.10 cannot be sharpened materially since, in general, $\text{unicenter}(\Lambda^\tau)$ is a proper subset of $\text{center}(\Lambda^\tau)$. The following example, due to H. R. Strong, verifies this assertion.

Example 3.3. Consider the following data graph, depicted in Fig. 3.3:

$$\Gamma_{3.3} = (C, \Lambda),$$

where

(i) $C = N \cup (N \times \{0, 1\})$;
(ii) $\Lambda = \{\sigma, \pi, \rho\}$;

for each $n \in N$,

$$n\sigma = n + 1,$$

$$n\pi = \begin{cases} n - 1 & \text{if } n > 1, \\ \langle 1, 0 \rangle & \text{if } n = 1, \end{cases}$$

$$n\rho = n;$$

for each $\langle n, k \rangle \in N \times \{0, 1\}$,

$$\langle n, k \rangle\sigma = \begin{cases} \langle n - 1, k \rangle & \text{if } n > 1, \\ 1 & \text{if } n = 1, \end{cases}$$

$$\langle n, k \rangle\pi = \langle n + 1, 0 \rangle,$$

$$\langle n, k \rangle\rho = \langle n, k + 1 \ (\text{mod } 2) \rangle.$$

PROPOSITION 3.6. *The transformation* σ *of* $\Gamma_{3.3} = (C, \Lambda)$ *is in* $\text{center}(\Lambda^\tau)$ *but not in* $\text{unicenter}(\Lambda^\tau)$.

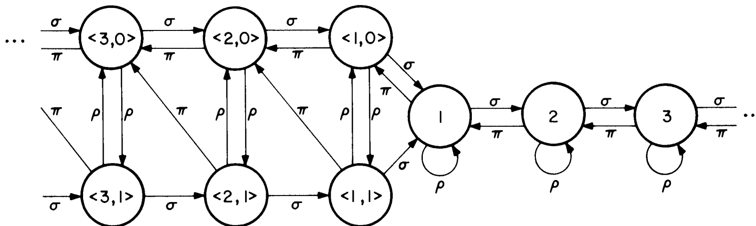


FIG. 3.3. $\Gamma_{3.3}$: *A data graph with* $\sigma \in \text{center}(\Lambda^\tau) - \text{unicenter}(\Lambda^\tau)$

Proof. Since $\langle 1, 0\rangle\sigma = \langle 1, 1\rangle\sigma = 1$, it is clear that $\sigma \notin \text{unicenter}(\Lambda^\tau)$. However, the following calculations show that $\sigma \in \text{center}(\Lambda^\tau)$.

(a) For $n \in N$,

(i) $n\sigma\rho = (n + 1)\rho = n + 1 = (n\rho) + 1 = n\rho\sigma$;

(ii) $n\sigma\pi = (n + 1)\pi = n$, while

$$n\pi\sigma = \begin{cases} (n-1)\sigma = n & \text{if } n > 1, \\ \langle 1, 0\rangle\sigma = 1 = n & \text{if } n = 1. \end{cases}$$

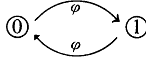
(b) For $\langle n, k\rangle \in N \times \{0, 1\}$,

$$\text{(i) } \langle n,k\rangle\sigma\rho = \begin{cases} \langle n-1,k\rangle\rho = \langle n-1, k+1 \,(\text{mod } 2)\rangle & \\ \quad = \langle n, k+1\,(\text{mod } 2)\rangle\sigma & \\ \quad = \langle n,k\rangle\rho\sigma & \text{if } n > 1, \\ 1\rho = 1 = \langle n, k+1\,(\text{mod } 2)\rangle\sigma = \langle n,k\rangle\rho\sigma & \text{if } n = 1; \end{cases}$$

$$\text{(ii) } \langle n,k\rangle\sigma\pi = \begin{cases} \langle n-1,k\rangle\pi = \langle n,0\rangle = \langle n+1,0\rangle\sigma & \\ \quad = \langle n,k\rangle\pi\sigma & \text{if } n > 1, \\ 1\pi = \langle 1,0\rangle = \langle 2,0\rangle\sigma = \langle n,k\rangle\pi\sigma & \text{if } n = 1. \end{cases}$$

Thus $\sigma\pi = \pi\sigma$, and $\sigma\rho = \rho\sigma$, as was claimed.

The transformation φ of $\Gamma_{2.2}$ satisfies the conditions of Theorem 3.10 because $\Gamma_{3.2}$ is a "direct product" of $\Gamma_{3.1}$ and the data graph $\Gamma_{3.4}$, defined by the figure



Thus, in this case the existence of a link-induced, fixed-link symmetry was guaranteed by construction. We turn now to a study of conditions which guarantee the presence of symmetries in data graphs.

3.4. Conditions which induce symmetries. In the preceding subsections, we have derived a number of principles which delimit the set of symmetries of a data graph. Most of these principles could be paraphrased, "If Γ has a symmetry of a given type, then the following condition must be satisfied." Thus, these results are more useful in eliminating proposed symmetries than in detecting symmetries. In contrast, this subsection is devoted to finding conditions which ensure the presence of symmetries. Each of the ensuing results will, therefore, be of the form, "If the following condition is satisfied, then Γ has a symmetry of a given type."

We consider two classes of conditions, those concerning the structure of Λ^τ and/or Λ, and those concerning the way Γ is constructed.

A. The structure of Λ^τ. Throughout this section, let us focus on a fixed data graph $\Gamma = (C, \Lambda)$.

DEFINITION. We say that Γ is *loop-free* if no nonidentity $\xi \in \Lambda^\tau$ has a fixed point;[13] i.e., $c\xi = c$ for some $c \in C$ implies $\xi = 1_C$.

[13] Thus, Γ is really free of nontrivial loops.

The following result, stated here without proof, is an excerpt from Theorem 6.11 of [2]. It yields our first set of conditions which ensure the existence of symmetries, in this case, fixed-link symmetries.

THEOREM 3.11 (see [2]). *The following conditions are equivalent; moreover, each implies that* Λ^τ *is a group*:

(a) Γ *is loop-free.*

(b) *For all* $\xi, \eta \in \Lambda^\tau$, *for all* $c \in \Delta(\xi) \cap \Delta(\eta)$, $\xi = \eta$ *whenever* $c\xi = c\eta$.

(c) *Any two* $\xi, \eta \in \Lambda^\tau$ *are either equal or disjoint.*

(d) *For all* $c, d \in C$, Γ *has a fixed-link symmetry* $\Sigma = (\beta_c, 1)$ *with* $c\beta_c = d$.[14]

Other conditions which either imply or are equivalent to these four can be found in [2]. An interesting condition not presented there is stated in the following which can also be found in [5].

THEOREM 3.12. *If* Λ^τ *is abelian—that is,* $\xi\eta = \eta\xi$ *for all* $\xi, \eta \in \Lambda^\tau$*—then* Γ *is loop-free. In this case, the symmetries of Theorem* 3.11(d) *are link-induced.*

Proof. Say $c\xi = c$ for some $c \in C$ and $\xi \in \Lambda^\tau$. Given arbitrary $\eta \in \Lambda^\tau$ with $c \in \Delta(\eta)$, one finds $c\eta = (c\xi)\eta = (c\eta)\xi$. Thus ξ fixes $c\eta$ also. Since η was arbitrary, ξ must fix all cells in $c\Lambda^\tau = C$. Hence $\xi = 1_C$, so Γ is loop-free.

Given arbitrary $c, d \in C$, let ξ_{cd} be the (unique by Theorem 3.11(b)) transformation with $c\xi_{cd} = d$. Since Λ^τ is a group, $\xi_{cd} \in \Lambda^\tau$ must be a bijection. By Lemma 3.1, each $\lambda \in \Lambda$ is total; hence, $(e\lambda)\xi_{cd} = (e\xi_{cd})\lambda$ for all $e \in C$, $\lambda \in \Lambda$ since Λ^τ is abelian. Thus, each system $\Sigma_{cd} = (\xi_{cd}, 1)$ is a fixed-link symmetry of Γ with $c\xi_{cd} = d$. By Theorem 2.1, Σ_{cd} must be the symmetry referred to in Theorem 3.11(d). Since c, d were arbitrary, the result follows.

Theorem 3.12 accounts for the rotational symmetries of $\Gamma_{2.3}$ and $\Gamma_{2.4}$. With one additional assumption, we can ensure the existence in abelian data graphs of symmetries which need not fix all links.

DEFINITION. We say Λ^τ is an *atomic group* if (i) it is a group of transformations, and (ii) for each $\lambda \in \Lambda$, $\lambda^{-1} \in \Lambda$.

For example, Λ^τ is an atomic group for $\Gamma_{2.4}$ but not for $\Gamma_{2.3}$ (although both are groups by Theorems 3.11 and 3.12).

THEOREM 3.13. *Let* Λ^τ *be abelian, hence a group; further, assume that* Λ^τ *is atomic. For each cell* $d \in C$, *there is a unique symmetry* $\Sigma^{(d)} = (\beta_c^{(d)}, \beta_l)$ *of* Γ *with* $d\beta_c^{(d)} = d$ *and with* $\lambda\beta_l = \lambda^{-1}$ *for all* $\lambda \in \Lambda$.

Proof. By Theorem 2.1, at most one such symmetry can exist for each $d \in C$. We need, therefore, establish only the existence of $\Sigma^{(d)}$. Let $d \in C$ be arbitrary.

For each cell $e \in C$, define

$$e\beta_c^{(d)} = d\xi_e^{-1},$$

where $\xi_e \in \Lambda^\tau$ is such that $d\xi_e = e$. We first verify that $\beta_c^{(d)}$ is a bijection.

1. $\beta_c^{(d)}$ is a total function.

This is obvious, since Λ^τ is a group of transformations.

2. $\beta_c^{(d)}$ is one-to-one.

$e\beta_c^{(d)} = f\beta_c^{(d)}$ implies $d\xi_e^{-1} = d\xi_f^{-1}$ so that $\xi_e = \xi_f$, whence $e = f$.

3. $\beta_c^{(d)}$ is onto.

[14] In [2] condition (b) is termed *universal-rootedness* (cf. § 4); condition (c) is called *relational homogeneity*; condition (d) is designated *uniform transposability*.

Since Λ^τ is a group, the strong-connectivity of Γ guarantees that

$$C = \{d\xi^{-1} | \xi \in \Lambda^\tau\}.$$

Finally:

4. The system $\Sigma^{(d)}$ is a symmetry of Γ.

By Lemma 3.1, each $\lambda \in \Lambda$ is total; therefore, for all $e \in C$, $\lambda \in \Lambda$,

$$(e\lambda)\beta_c^{(d)} = d(\xi_e\lambda)^{-1} = d\lambda^{-1}\xi_e^{-1} = (d\xi_e^{-1})\lambda^{-1} = (e\beta_c^{(d)})(\lambda\beta_l).$$

The theorem follows from parts 1–4. (Note the uses of commutativity.)

The symmetries of Theorem 3.13 fix all links only when each $\lambda \in \Lambda$ is self-inverse. In this case, each $\Sigma^{(d)} = \Sigma_I$.

Theorem 3.13 accounts for the "flip" symmetries of $\Gamma_{2.4}$.

B. Construction of Γ. It is probably safe to guess that a rather small percentage of the data graphs which arise "naturally" in applications are abelian or even loop-free. However, this percentage increases appreciably if one broadens his view to include data graphs, such as $\Gamma_{2.2}$, which are "constructed" using loop-free graphs. We now discuss symmetries which arise from the construction of data graphs. Although we focus on only one method of construction, other methods will readily occur to the reader, and results analogous to those presented here will be obtainable. (See, for example, [3, § 5].)

DEFINITION. Let $\Gamma_1 = (C_1, \Lambda_1)$ and $\Gamma_2 = (C_2, \Lambda_2)$ be data graphs. The *direct product* of Γ_1 and Γ_2 is the data graph

$$\Gamma_1 \times \Gamma_2 = (C, \Lambda),$$

where

(i) $C = C_1 \times C_2$;

(ii) $\Lambda = \{\lambda' | \lambda \in \Lambda_1\} \cup \{\lambda'' | \lambda \in \Lambda_2\}$;

for $\langle c_1, c_2 \rangle \in C$,

$$\langle c_1, c_2 \rangle \lambda' = \langle c_1\lambda, c_2 \rangle \text{ for all } \lambda' \in \Lambda,$$
$$\langle c_1, c_2 \rangle \lambda'' = \langle c_1, c_2\lambda \rangle \text{ for all } \lambda'' \in \Lambda.$$

The reader will easily verify that $\Gamma_{3.2}$ is isomorphic[15] to $\Gamma_{3.1} \times \Gamma_{3.1}$, and that $\Gamma_{2.2}$ is isomorphic to $\Gamma_{3.1} \times \Gamma_{3.4}$.

We present two results about symmetries of direct products. The first indicates how symmetries can arise from a construction; the second indicates how symmetries can be preserved by a construction.

THEOREM 3.14. *Let $\Gamma = (C, \Lambda)$ be a data graph. The data graph*

$$\Gamma \times \Gamma = (C^\#, \Lambda^\#),$$

where $C^\# = C \times C$ and $\Lambda^\# = \{\lambda', \lambda'' | \lambda \in \Lambda\}$, enjoys the symmetry $\Sigma = (\beta_c, \beta_l)$ defined by

$$\langle c, d \rangle \beta_c = \langle d, c \rangle \quad \textit{for all } \langle c, d \rangle \in C^\#;$$

$$\lambda'\beta_l = \lambda'', \qquad \lambda''\beta_l = \lambda' \quad \textit{for all } \lambda \in \Lambda.$$

[15] Cell $\langle m, n \rangle$ is encoded as $2^{m-1}3^{n-1}$; $\sigma' = \sigma_r$, $\pi' = \sigma_l$, $\sigma'' = \sigma_u$, $\pi'' = \sigma_d$.

The proof is obvious, and is omitted. Theorem 3.14 accounts for the nonidentity symmetry of $\Gamma_{3.2}$.

THEOREM 3.15. *Let* $\Gamma_1 = (C_1, \Lambda_1)$ *and* $\Gamma_2 = (C_2, \Lambda_2)$ *be data graphs. If* Γ_i *enjoys the symmetry* $\Sigma^{(i)} = (\beta_c^{(i)}, \beta_l^{(i)})$, *then* $\Gamma_1 \times \Gamma_2$ *enjoys the symmetry* $\Sigma = (\beta_c, \beta_l)$ *defined by*

$$\langle c, d\rangle\beta_c = \langle c\beta_c^{(1)}, d\beta_c^{(2)}\rangle \quad \text{for all } \langle c, d\rangle \in C_1 \times C_2,$$

$$\lambda'\beta_l = (\lambda\beta_l^{(1)})' \quad \text{for all } \lambda \in \Lambda_1,$$

$$\lambda''\beta_l = (\lambda\beta_l^{(2)})'' \quad \text{for all } \lambda \in \Lambda_2.$$

Again the straightforward proof is omitted. Theorem 3.15, in conjunction with Theorem 3.12, accounts for the nonidentity symmetry of $\Gamma_{2.2}$.

4. Symmetries in addressable data graphs. In this section we study the symmetries of data graphs which enjoy a strong type of uniformity, termed addressability. The presence of this uniformity affords us strengthened versions of a number of the results from § 3. In an analogous vein, knowledge about the symmetries enjoyed by an addressable data graph yields further information about the addressing structure of the graph. Thus, the results in this section supplement the study in [2], [3] as well as that in the preceding section.

4.1. Addressable data graphs. We review a number of the basic notions and results from [2], [3] which are pertinent to the sequel. Fix on a data graph $\Gamma = (C, \Lambda)$.

DEFINITION. An *addressing scheme* for the data graph Γ is a total function

$$\alpha : C \to \Lambda^\tau$$

such that

(i) there is a designated *base cell* $c_0 \in C$ for which $c_0\alpha = 1_C$;

(ii) for all $\lambda \in \Lambda$, for all $c \in \Delta(\lambda)$,

$$(c\lambda)\alpha = (c\alpha)\lambda.$$ [16]

We say Γ is *addressable* if it admits an addressing scheme.

LEMMA 4.1. (a) *If* α *exists, it is one-to-one.*

(b) *If* $c\alpha = (d\alpha)\lambda$ *for some* $c, d \in C$ *and* $\lambda \in \Lambda$, *then* $c = d\lambda$.

(c) $C\alpha$ *is the submonoid of* Λ^τ *comprising all and only total functions.*

(d) *For each* $\xi \in C\alpha$ *there is an* $\eta \in \Lambda^\tau$ *with* $\xi\eta = 1_C$.

DEFINITION. A cell $c_0 \in C$ is a *root* of Γ if, for all $\xi, \eta \in \Lambda^\tau$ which are defined at c_0, $\xi = \eta$ whenever $c_0\xi = c_0\eta$.

LEMMA 4.2. (a) *Every transformation defined at a root is one-to-one and total.*

(b) *Let* c_0 *be a root of* Γ. *Let* Ω *be the maximal submonoid of* Λ^τ *which is a group. Then, the set of all roots of* Γ *is dually characterized by the sets* $c_0\Omega = \{c_0\omega | \omega \in \Omega\}$ *and* $\{c_0\xi^{-1} | \xi \in \Lambda^\tau \text{ is total}\}$.

DEFINITION. (i) A *self-insertion* of Γ is a total function $\theta : C \to C$ such that, for all $\lambda \in \Lambda$, $\lambda\theta \subseteq \theta\lambda$.

(ii) Γ is *uniformly self-insertable* if there is a cell $c_0 \in C$ such that, for every cell $c \in C$, there is a self-insertion θ_c of Γ with $c_0\theta_c = c$. We say c_0 is *relocatable.*

[16] Thus the *address* of $c\lambda$ is obtained as a product in the monoid Λ^τ.

LEMMA 4.3. *Any two self-insertions of* Γ *are either equal or disjoint.*[17]

The following theorem relates the three notions just defined.

THEOREM 4.1. *The following assertions about a data graph* Γ *are equivalent*:

(a) Γ *is addressable.*

(b) Γ *is rooted.*

(c) Γ *is uniformly self-insertable.*

Moreover, the base cell of every addressing scheme is a root of Γ; *every root of* Γ *is relocatable*; *every relocatable cell is the base cell of an addressing scheme.*

It is not hard to verify (cf. [2], [3]) that $\Gamma_{2.1}$, $\Gamma_{3.1}$ and $\Gamma_{3.2}$ each has a unique root; $\Gamma_{2.2}$ and $\Gamma_{3.4}$ each has two roots; $\Gamma_{2.3}$ and $\Gamma_{2.4}$ have three roots apiece; and $\Gamma_{3.3}$ has none. A major result in [2] is the proof of the coextension of addressability and realizability by relative addressing.

4.2. Symmetries and roots. We establish some basic connections among the uniformities we have introduced. The reader should apply these results to the rooted data graphs just mentioned.

Our first result strengthens Corollary 3.2.

THEOREM 4.2. *The image of a root under any symmetry is again a root.*

Proof. Let c_0 be a root, and $\Sigma = (\beta_c, \beta_l)$ a symmetry of $\Gamma = (C, \Lambda)$.

Since c_0 is a root, it is in the domain of only total transformations (Lemma 4.2(a)); the same must, therefore, be true of cell $c_0\beta_c$ (Corollary 3.2). The result is now immediate by the strong-connectivity of Γ and by the second part of Lemma 4.2(b).

COROLLARY 4.1. *Every symmetry of a singly-rooted data graph fixes the root cell.*

These results yield immediately that cells 1 of $\Gamma_{2.1}$, $\Gamma_{3.1}$ and $\Gamma_{3.2}$ are fixed by all symmetries. They further yield that cells $\langle 1, 0\rangle$ and $\langle 1, 1\rangle$ of $\Gamma_{2.2}$ are either fixed or interchanged by all symmetries. (The roots of these four data graphs are readily identifiable using techniques from [2], [3].)

COROLLARY 4.2. *If an addressable data graph* Γ *enjoys a nonidentity link-induced symmetry, then* Γ *is multiply-rooted.*

Proof. Say $\Gamma = (C, \Lambda)$ has root c_0 and nonidentity link-induced symmetry $\Sigma = (\beta_c, \beta_l)$. By Theorem 4.2, $c_0\beta_c$ must also be a root of Γ. Moreover, since $\beta_c \in \Lambda^\tau$, $c_0\beta_c = c_0$ only when $\beta_c = 1_C$; however, by Theorem 3.4, only the identity symmetry fixes all cells. Thus our assumption that $\Sigma \neq \Sigma_I$ forces us to conclude that $c_0\beta_c$ is a root distinct from c_0.

We can establish a strong converse to Theorem 4.2. Not only are any two roots of a data graph "connected" by a symmetry, they are connected by a fixed-link symmetry.

THEOREM 4.3. *Given any pair of roots of* $\Gamma = (C, \Lambda)$, *say* c_1 *and* c_2, *there is a (unique) fixed-link symmetry* $\Sigma = (\beta_c, 1)$ *of* Γ *satisfying* $c_1\beta_c = c_2$.

Proof. The proof technique mirrors that of Theorem 4.1 in [3]. Let c_1 be the base cell of the addressing scheme α. Define the map $\beta: C \to C$ by:

$$d\beta = c_2(d\alpha) \quad \text{for all } d \in C.$$

[17] Clearly any fixed-link symmetry of Γ is a self-insertion. Hence, Theorem 3.5 is actually a corollary of Lemma 4.3.

We claim that $(\beta, 1_{\Lambda^\tau})$ is the desired symmetry. This claim is established by the following chain of observations.

Observation 1. β is a total function.

Functionality follows from the functionality of each $\xi \in \Lambda^\tau$. Totality is immediate by Lemma 4.1(c).

Observation 2. $c_1\beta = c_2$.

Since α has c_1 as base cell, $c_1\beta = c_2(c_1\alpha) = c_2 1_C = c_2$.

Observation 3. β is one-to-one.

$d\beta = e\beta$ implies $c_2(d\alpha) = c_2(e\alpha)$, by definition; hence $d\alpha = e\alpha$, since c_2 is a root of Γ; hence $d = e$, since α is one-to-one (Lemma 4.1(a)).

Observation 4. β is onto.

Let $d \in C$ be arbitrary. By strong-connectivity of Γ, there is a $\xi \in \Lambda^\tau$ with $c_2\xi = d$. By Lemma 4.2(a) ξ is total; hence $\xi \in C\alpha$, by Lemma 4.1(c). Thus $\xi = e\alpha$ for some $e \in C$, and for this e, $e\beta = c_2(e\alpha) = d$.

Observation 5. $(\beta, 1_{\Lambda^\tau})$ is a symmetry.

Let $\lambda \in \Lambda$ be arbitrary. If $d \in \Delta(\lambda)$, then

$$(d\lambda)\beta = c_2((d\lambda)\alpha) = (c_2(d\alpha))\lambda = (d\beta)\lambda.$$

Conversely, if $(d\beta)\lambda \in C\beta = C$, then for some $e \in C$,

$$(d\beta)\lambda = c_2(d\alpha)\lambda = c_2(e\alpha) = e\beta.$$

Hence, $(d\alpha)\lambda = e\alpha$ since c_2 is a root of Γ. By Lemma 4.1(b), $d\lambda = e$, so $d \in \Delta(\lambda)$.

The theorem follows from these observations and Theorem 2.1.

The preceding results yield an unexpected symmetry-based test for multiple roots. This exemplifies how knowledge of one type of uniformity can yield information about another.

COROLLARY 4.3. *An addressable data graph is multiply-rooted if and only if it enjoys a nonidentity fixed-link symmetry.*

Proof. If Γ is multiply-rooted, Theorem 4.3 ensures the existence of such a symmetry.

Conversely, if Γ is singly-rooted, Corollary 4.1 asserts that this root must be fixed by all symmetries. By Theorem 2.1, therefore, the identity symmetry is the only fixed-link symmetry of Γ.

Thus the multiple-rootedness of, say, $\Gamma_{2.2}$ can be inferred from knowledge of its rootedness and of its symmetries.

It is quite easy to show that the symmetries of Theorem 4.3 are not the unique ones which exchange a pair of root cells. For example, consider the data graph $\Gamma_{2.1} \times \Gamma_{3.4}$, obtained as a direct product of our tree and our two-cell data graph. This data graph clearly possesses two distinct symmetries which interchange its two roots $\langle 1, 0\rangle$ and $\langle 1, 1\rangle$. On the one hand, it enjoys the fixed-link symmetry of Theorem 4.3. On the other hand it enjoys a symmetry which interchanges σ_r and σ_l, and which uniformly has cell $\langle 2^k + i, l\rangle$ as the image of cell $\langle 2^{k+1} - (i+1), l+1 \pmod 2\rangle$. The latter symmetry "combines" the nonidentity symmetries of $\Gamma_{2.1}$ and $\Gamma_{3.4}$. (Cf. Theorem 3.15.)

4.3. Symmetries, addressing schemes and self-insertions. Our first results relating symmetries and addressing schemes are actually corollaries of previously obtained results.

COROLLARY 4.4. *The sets* $C\alpha$ *and* $\Lambda^\tau - C\alpha$ *are invariant under all symmetries. That is,* $(C\alpha)\beta_l = C\alpha$, $(\Lambda^\tau - C\alpha)\beta_l = \Lambda^\tau - C\alpha$ *for any symmetry* $\Sigma = (\beta_c, \beta_l)$.

Proof. The proof is by Corollary 3.2 and Lemma 4.1(c).

COROLLARY 4.5. *Every fixed-link symmetry of a rooted data graph is induced by an addressing scheme. That is, given* $\Gamma = (C, \Lambda)$ *with addressing scheme* α *and symmetry* $\Sigma = (\beta_c, 1)$, *there is a cell* $e \in C$ *such that* $d\beta_c = e(d\alpha)$ *for all* $d \in C$.

Proof. If c_0 is the base cell of α, then let $e = c_0\beta_c$. By Theorem 4.2 and the proof of Theorem 4.3, the system $(\beta, 1_{\Lambda^\tau})$ is a symmetry of Γ, where $d\beta = e(d\alpha)$ for all $d \in C$. By Theorem 2.1, therefore, this symmetry must coincide with Σ.

We now establish a result which demonstrates a very intimate relationship between the set of symmetries of a data graph and the set of addressing schemes for the graph.

DEFINITION. Let $\Gamma = (C, \Lambda)$ have addressing schemes α_1 and α_2 (not necessarily distinct). An $\langle\alpha_1, \alpha_2\rangle$-*system* is a pair of bijections (β_1, β_2) such that

(i) $\beta_1 : C \to C$;

(ii) $\beta_2 : \Lambda^\tau \to \Lambda^\tau$ is a monoid isomorphism which maps Λ onto Λ;

(iii) $\alpha_1\beta_2 = \beta_1\alpha_2$; that is,

$$\begin{array}{ccc} C & \xrightarrow{\beta_1} & C \\ \alpha_1 \downarrow & = & \downarrow \alpha_2 \\ \Lambda^\tau & \xrightarrow[\beta_2]{} & \Lambda^\tau \end{array}$$

THEOREM 4.4. *Let* $\Gamma = (C, \Lambda)$ *be a data graph with (not necessarily distinct) addressing schemes* α_1 *and* α_2. *Say* α_1 *has base cell* c_1, *and* α_2 *has base cell* c_2. *The following assertions obtain*:

(a) *If* $\Sigma = (\beta_c, \beta_l)$ *is a symmetry of* Γ, *and if* $c_1\beta_c = c_2$, *then* Σ *is an* $\langle\alpha_1, \alpha_2\rangle$-*system.*

(b) *Any* $\langle\alpha_1, \alpha_2\rangle$-*system* (β_1, β_2) *is a symmetry of* Γ *for which* $c_1\beta_1 = c_2$.

Proof. (a) Let $c \in C$ be arbitrary. On the one hand, we find

$$c\beta_c = c_2((c\beta_c)\alpha_2)$$

since c_2 is the base cell of α_2. On the other hand,

$$c\beta_c = (c_1(c\alpha_1))\beta_c = (c_1\beta_c)((c\alpha_1)\beta_l) = c_2((c\alpha_1)\beta_l)$$

since c_1 is the base cell of α_1, and $c_1\beta_c = c_2$. By Theorem 4.1, c_2 is a root of Γ; hence $(c\beta_c)\alpha_2 = (c\alpha_1)\beta_l$. Since c was arbitrary, we conclude that Σ is an $\langle\alpha_1, \alpha_2\rangle$-system.

(b) To establish the converse, we need merely demonstrate that $\lambda\beta_1 = \beta_1(\lambda\beta_2)$ for each $\lambda \in \Lambda$, and that $c_1\beta_1 = c_2$.

To these ends, let $\lambda \in \Lambda$ and $c \in \Delta(\lambda)$ be arbitrary. We find the following chain of equations:

$$\begin{aligned} (c\lambda)\beta_1 &= c_2(((c\lambda)\beta_1)\alpha_2) && \text{(since } c_2 \text{ is the base cell of } \alpha_2) \\ &= c_2(((c\lambda)\alpha_1)\beta_2) && \text{(since } \alpha_1\beta_2 = \beta_1\alpha_2) \\ &= c_2(((c\alpha_1)\lambda)\beta_2) && \text{(since } \alpha_1 \text{ is an addressing scheme)} \\ &= c_2((c\alpha_1)\beta_2)(\lambda\beta_2) && \text{(since } \beta_2 \text{ is a monoid isomorphism)} \end{aligned}$$

$$= c_2((c\beta_1)\alpha_2)(\lambda\beta_2) \quad \text{(since } \alpha_1\beta_2 = \beta_1\alpha_2\text{)}$$
$$= (c\beta_1)(\lambda\beta_2) \quad \text{(since } c_2 \text{ is the base cell of } \alpha_2\text{)}.$$

Moreover, reversing these steps shows that $c \in \Delta(\lambda)$ whenever $c\beta_1 \in \Delta(\lambda\beta_2)$. Since λ and c were arbitrary, we conclude that the system (β_1, β_2) is a symmetry of Γ. (Actually, reversing the steps requires Lemma 4.1(b).)

Finally, we note that

$$\begin{aligned} c_1\beta_1 &= c_2((c_1\beta_1)\alpha_2) && \text{(since } c_2 \text{ is the base cell of } \alpha_2\text{)} \\ &= c_2((c_1\alpha_1)\beta_2) && \text{(since } \alpha_1\beta_2 = \beta_1\alpha_2\text{)} \\ &= c_2((1_C)\beta_2) && \text{(since } c_1 \text{ is the base cell of } \alpha_1\text{)} \\ &= c_2 && \text{(since } \beta_2 \text{ is a monoid isomorphism).} \end{aligned}$$

This completes our proof.

COROLLARY 4.6. *If the symmetry $\Sigma = (\beta_c, \beta_l)$ of $\Gamma = (C, \Lambda)$ fixes the root c_0, then $\beta_c\alpha = \alpha\beta_l$, where c_0 is the base cell of addressing scheme α.*

Corollary 4.6 is often useful in computing one of the functions β_c or β_l, when the other is known. In particular, when a symmetry is presented as in Theorem 2.1, this corollary is a very useful computing aid.

COROLLARY 4.7. *If $\Gamma = (C, \Lambda)$ is singly-rooted and, hence, has a unique addressing scheme α, then any symmetry $\Sigma = (\beta_c, \beta_l)$ of Γ satisfies the equation $\beta_c\alpha = \alpha\beta_l$.*

We remarked that Theorem 4.4 and its corollaries can be useful in computing symmetries of data graphs which are known to be addressable. Not surprisingly, known symmetries can often be helpful in detecting addressability by facilitating the "computation" of self-insertions.

Let c be a root of $\Gamma = (C, \Lambda)$. By Theorem 4.1, for each cell $d \in C$, there is a self-insertion of Γ which maps c to d. Let us uniformly denote this self-insertion by θ_d^c.

THEOREM 4.5. *Let $\Gamma = (C, \Lambda)$ have root c and symmetry $\Sigma = (\beta_c, \beta_l)$. For all $d \in C$,*

$$\theta_d^c\beta_c = \beta_c\theta_{d\beta_c}^{c\beta_c}.$$

Proof. Say c is the base cell of addressing scheme α_1. By Theorem 4.2, cell $c\beta_c$ is also a root of Γ; say that it is the base cell of α_2.

Let $d \in C$ be arbitrary. For each cell $e \in C$, we have,[18] using Theorem 4.4,

$$(e\theta_d^c)\beta_c = (d(e\alpha_1))\beta_c = (d\beta_c)((e\alpha_1)\beta_l) = (d\beta_c)((e\beta_c)\alpha_2) = (e\beta_c)\theta_{d\beta_c}^{c\beta_c}.$$

Since d, e were arbitrary, the theorem follows.

Since β_c is a bijection, the equation of Theorem 4.5 can be rewritten as

$$\theta_{d\beta_c}^{c\beta_c} = \beta_c^{-1}\theta_d^c\beta_c$$

which may be a more useful form. If, moreover, Σ fixes c, this last equation simplifies to $\theta_{d\beta_c}^c = \beta_c^{-1}\theta_d^c\beta_c$. Other related equations are readily derivable by the reader.

[18] These equations draw on the proof of Theorem 4.1 from [3]; cf. Corollary 4.5.

4.4. Fixed points of symmetries. We were able to say very little in § 3 about cells which are fixed by symmetries. When we restrict attention to addressable data graphs, we can say a bit more.

THEOREM 4.6. *Let* $\Gamma = (C, \Lambda)$ *have root* c_0 *and associated addressing scheme* α. *Let* $\Sigma = (\beta_c, \beta_l)$ *be a symmetry of* Γ. Σ *fixes the cell* $c \in C$ *if and only if* $(c\alpha')\beta_l = c\alpha$, *where* α' *is the addressing scheme of* Γ *with base cell* $c_0\beta_c$.

Proof. The proof is immediate by Theorem 4.4.

COROLLARY 4.8. *Let* $\Sigma = (\beta_c, \beta_l)$ *fix the root* c_0 *of* $\Gamma = (C, \Lambda)$. *Then, for all* $n \in N \cup \{0\}$, Σ *must fix the cells* $c_0(c\alpha)^n$ *whenever it fixes the cell* c.

An immediate consequence of Corollary 4.8 is that, since cells 1 and 6 of the quadrant $\Gamma_{3,2}$ are fixed by all symmetries, so also must be all cells 2^n3^n on the diagonal.

Thus, for singly-rooted data graphs, the fixing of cells by symmetries is a concomitant of the fixing of total link-transformations. Moreover, the fixing of a single nonroot cell induces the fixing of an entire family of cells.

4.5. Symmetries induced by links. The final topic we consider is how the results of § 3.3 can be sharpened in the presence of addressability.

LEMMA 4.4. *Let* $\Gamma = (C, \Lambda)$ *be an addressable data graph. Then* center(Λ^τ) = unicenter(Λ^τ). *Moreover,* center(Λ^τ) *is a subgroup of the group* Ω *of Lemma* 4.2.[19]

Proof. By Lemma 3.1, every $\xi \in$ center(Λ^τ) is total; hence, center$(\Lambda^\tau) \subseteq C\alpha$. By Lemma 4.2 and Theorem 4.1, every $\xi \in C\alpha$ is one-to-one. We conclude that center(Λ^τ) = unicenter(Λ^τ). To complete the proof, recall that Lemma 4.1(d) asserts that, for each $\xi \in C\alpha$ there is an $\eta \in \Lambda^\tau$ with $\xi\eta = 1_C$. In particular, if $\xi \in$ center$(\Lambda^\tau) \subseteq C\alpha$, $\xi\eta = \eta\xi = 1_C$; it clearly follows that $\eta = \xi^{-1} \in C\alpha$ also, so $\xi, \eta \in \Omega$. We need, therefore, show only that $\eta \in$ center(Λ^τ). To this end, let $\zeta \in \Lambda^\tau$ be arbitrary. We know that $\xi\zeta = \zeta\xi$, so that $\zeta = \eta\zeta\xi$, and finally $\zeta\eta = \eta\zeta$. Since ζ was arbitrary, we conclude that $\eta \in$ center(Λ^τ), and the lemma follows.

The following result is now immediate from Lemma 4.4 and Theorem 3.10.

THEOREM 4.7. *Let* $\Gamma = (C, \Lambda)$ *be an addressable data graph. The transformation* $\xi \in \Lambda^\tau$ *induces a fixed-link symmetry of* Γ *if and only if* $\xi \in$ center(Λ^τ).

Appendix. Realizations by relative addressing. In order to indicate some of the motivation for the notion of addressability, we present, excerpted from [2], the notions of a realization of a data graph and of a realization by relative addressing.

A realization of a data graph in a random access memory can be viewed in the following simple manner. Let A denote the *address space* of the memory, i.e., the set of addresses. A realization of a data graph (C, Λ) in A comprises (a) an assignment of a unique address (or memory location) to each cell $c \in C$, and (b) a mechanism (function) for determining, from the address $a \in A$ of a cell $c \in C$ and a transformation ξ defined on c, the address of cell $c\xi$. Our informal notion of realization is further simplified if the mechanism for obtaining the address of $c\xi$ is independent of c (hence, of the address of c). Since every data graph admits such a realization (Proposition A), this type of realization is the starting point of our development.

[19] Recall that Ω is the maximal submonoid of Λ^τ which is a group.

DEFINITION. Let $\Gamma = (C, \Lambda)$ be a data graph, and let A be a set with $\# C \leqq \# A$. A *realization of* Γ *in* A is a pair of mappings $\langle r, \rho \rangle$ where:

(i) r maps C one-to-one into A;

(ii) ρ, which maps Λ^τ one-to-one into the set of partial transformations of A, is a monoid homomorphism;

(iii) for all $\lambda \in \Lambda$, $\lambda r = r(\lambda\rho)$ as functions on C.

Note. If $\langle r, \rho \rangle$ realizes $\Gamma = (C, \Lambda)$, then $(Cr, \Lambda\rho)$ is isomorphic to Γ. This assures that structural properties of a realization depend only on structural properties of the data graph realized.

PROPOSITION A. *A data graph* $\Gamma = (C, \Lambda)$ *is realizable in any set* A *with* $\# A \geqq \# C$.

Perhaps the most familiar technique for implementing graphs such as arrays and trees is the method of *relative addressing*. Informally, this technique amounts to specifying a *base address* and representing the addresses of the various cells in the graph as *displacements* from this base address. We present an example of this technique using the binary tree $\Gamma_{2.1}$ depicted in Fig. 2.1. The "specification" of this data graph presented in § 2, was, strictly speaking, a realization of the graph in the set N of natural numbers. Employing Church's well-known lambda notation for functions, the mapping ρ was specified to be:

$$\sigma_r\rho = \lambda n[2n], \quad \sigma_l\rho = \lambda n[2n + 1], \quad \pi\rho = \lambda n[\lfloor n/2 \rfloor].$$

We now show that this realization (r is implicit from the figure) can be viewed as a realization of the graph by relative addressing.

Base address. The base address is 1.

Displacements. For any cell $c \in C$, let $\zeta_c = \omega_1 \cdots \omega_n$ be the (unique) sequence of σ_r's and σ_l's which describes a path from the root of the tree to c. Note that ζ_c is null if c is the root. For example, if $cr = 5$, then $\zeta_c = \sigma_r\sigma_l$. Now, for each such ζ_c, define $|\zeta_c|$ as follows:

(i) If ζ_c is null, then $|\zeta_c| = 0$.

(ii) For any $\zeta \in \{\sigma_r, \sigma_l\}^*$,

$$|\zeta\sigma_r| = 2|\zeta| + 1, \qquad |\zeta\sigma_l| = 2|\zeta| + 2.$$

The displacement of $c \in C$ is, then, $|\zeta_c|$. One easily verifies that, for each $c \in C$,

$$cr = 1 + |\zeta_c|.$$

Thus, the address of a cell can be uniquely expressed as the pair $\langle 1, \zeta_c \rangle$ or, more generally, as a pair $\langle$*base address, displacement*$\rangle$.

We proceed to the general definition.

DEFINITION. Let $\langle r, \rho \rangle$ be a realization of a data graph $\Gamma = (C, \Lambda)$ in a set A. We say $\langle r, \rho \rangle$ is a *realization by relative addressing* if the following condition obtains: There is an $a_0 \in Cr$ and a bijection (= one-to-one onto map) $\delta : C \to \Pi = \{\pi \in \Lambda^\tau\rho | a_0\pi \in Cr\}$ such that, for all $c \in C$, $cr = a_0(c\delta)$. We call a_0 the *base address* and δ the *displacement function* of $\langle r, \rho \rangle$.

Theorem 4.1 exhibits three conditions on a data graph, which are equivalent to the realizability of the graph by relative addressing.

Acknowledgment. It is a pleasure to acknowledge several helpful and stimulating conversations with Dr. R. E. Miller and Dr. H. R. Strong, Jr. The indications by the referees of related work are sincerely appreciated.

REFERENCES

[1] A. H. CLIFFORD AND G. B. PRESTON, *The Algebraic Theory of Semigroups, II.*, Mathematical Surveys No. 7, American Mathematical Society, Providence, R.I., 1967.

[2] A. L. ROSENBERG, *Data graphs and addressing schemes*, J. Comput. System. Sci., 5 (1971), pp. 193–238.

[3] ———, *Addressable data graphs*, J. Assoc. Comput. Mach., 19 (1972).

RELATED WORK

[4] J. EARLEY, *Toward an understanding of data structures*, Comm. ACM, 14 (1971), pp. 617–627.

[5] A. C. FLECK, *Isomorphism groups of automata*, J. Assoc. Comput. Mach., 9 (1962), pp. 469–476.

[6] C. A. TRAUTH, *Group-type automata*, Ibid., 13 (1966), pp. 170–175.

SIAM J. COMPUT.
Vol. 1, No. 1, March 1972

ON CLASSES OF PROGRAM SCHEMATA*

ROBERT L. CONSTABLE AND DAVID GRIES†

Abstract. We define the following classes of program schemata:

P = class of schemes using a finite number of simple variables;
P_A = class of schemes using simple and subscripted variables (arrays);
P_{Ae} = class of schemes in P_A, with the addition of an equality test on subscript values;
P_R = class of schemes allowing recursive functions;
P_L = class of schemes allowing labels as values;
P_M = class of schemes allowing a finite number of special markers as values;
P_{pds} = class of schemes using pushdown stores.

With these, we can also discuss, for example, P_{AM}, the class of schemes allowing arrays, and special markers as values; and P_{AL}, the class of schemes allowing arrays, and labels as values.

We argue that P_A, P_R, and P_L faithfully represent mechanisms of subscripting, recursion, and labels as values, that are present in many "real" programming languages.

We show that

$$P < P_R < P_A \equiv P_{AM} \equiv P_{pdsM} \equiv P_{Ae} \equiv EF,$$

where EF is Strong's class of effective functionals, assuming total functions and predicates. The inclusions $P < P_R < P_A$ and equivalences $P_{AL} \equiv P_{AM} \equiv P_{pdsM} \equiv P_{Ae}$ are effective. For example, given a program scheme in P_{AM} we can construct an equivalent one in P_{AL}. However, we show that for any scheme in P_{AM} an equivalent P_A scheme exists, but also prove it cannot (in general) be constructed!

We conjecture that P_A, P_{AL}, and equivalent classes are indeed "universal."

The above results assume that the uninterpreted functions and predicates are total. We discuss the problems which arise when they are partial. We define the class of multischemes and outline the relationship between the class P_{Ae}, multischemes, and Strong's nondeterministic and deterministic effective functionals.

Key words. Schemata, Programming Languages, functionals, effective functionals

TABLE OF CONTENTS

1. Introduction . . . 67
2. Basic notions of schemes, the concept of equivalence . . . 68
3. Recursive program schemes and macro expansion . . . 71
4. Other classes of schemes . . . 76
5. Locators and P_i-simulators . . . 80
6. The relation between P_A and P_R . . . 83
7. The relation between P_{pds} and P_L . . . 87
8. The equivalence of P_{AL}, P_{AM}, P_{pdsM} and P_{Ae} . . . 88
9. The noneffective equivalence of P_A and P_{AM} . . . 92
10. Effective functionals on total interpretations . . . 100
11. Multischemes and nondeterministic effective functionals . . . 103
Appendix . . . 110
References . . . 118

* Received by the editors September 9, 1971, and in revised form December 27, 1971.

† Computer Science Department, Cornell University, Ithaca, New York 14850. This research was supported by the National Science Foundation under Grants GJ-579 and GJ-28176.

1. Introduction. A fundamental result in logic and computing theory is that certain seemingly different computing systems (e.g., Turing machines, programs and recursive functions) are equivalent in their ability to compute functions. They are all universal because they allow computation of exactly the computable functions.

Paterson and Hewitt [5] and Strong [7] discovered that structural differences between these systems emerge at the level of schemata, i.e., on the task of computing functionals.

In [5] the authors exhibit a hierarchy of schemata: program schemata (P) contained in recursion schemata (P_R) contained in parallel schemata (P_p). These results indicate that the mechanism of function evaluation in recursion is "more powerful than" the mechanism in machines with a finite number of registers. Paterson and Hewitt also raise the question of whether there is a well-defined notion of *universal computing scheme.*

We are interested in discovering more exactly how different classes of schemata relate to each other, and in proposing a *natural* model of a computing scheme which is perhaps universal.

The discovery of universal computing schemata, if they exist, must proceed analogously to the discovery of universal computing systems. That is, a number of diverse candidates are proposed and are found to be equivalent. To start the study we need candidates. We propose our array schemata with special markers as one candidate. In [4] Paterson proposes program schemata with pushdown stores and special markers, and Strong [7] proposes his "effective functionals." We relate a version of our array program schemata to these other classes.

Our "proofs" of equivalence or inclusion of one class of schemata in another generally consist of showing, given a scheme S in one class, how to construct another equivalent scheme S′ in the second class. The constructions are usually fairly simple and obvious. We do not give formal proofs of equivalence.

The paper is arranged as follows. In § 2 we discuss the class P of program schemata and define what we mean by "equivalence" of two schemes. In § 3 we define the class P_R of program schemata, and relate them to the usual recursive functions used in mathematics. We describe macro expansion, which is the replacement of a function call by the corresponding function body, to yield an equivalent scheme.

Section 4 defines several other classes of schemes: P_A, P_{Ae}, P_M, P_L, $P_{\underset{\sim}{\mathbb{N}}}$ and P_{pds}. Section 5 is devoted to a discussion of two important concepts on which many of the results depend: locators and simulators. In § 6 we show that $P_R < P_A$, while in § 7 we include some results concerning P_{pds} and P_{pdsM}.

In §§ 8 and 9 we show that P_{AM}, P_{AL}, P_{pdsM} and P_{Ae} are all equivalent, and finally that $P_A \equiv P_{AM}$. The equivalence of Strong's [7] effective functionals (assuming total functions and predicates) with P_{Ae} is outlined in § 10.

Thus far, all the results concern schemes with *total* functions and predicates. In § 11 we discuss schemes with partial functions and predicates and arrive at a suitable class of schemes, called *multischemes*, to handle them. We assert the equivalence of the class of multischemes and effective functionals and attempt to show the relation between P_{Ae}, multischemes, nondeterministic effective functionals, and deterministic effective functionals.

2. Basic notions of schemes, the concept of equivalence.

(2.1) DEFINITION. A program scheme in the set P of *program schemata* is a sentence of the grammar G[⟨program⟩] given below, where

(i) v, w denote *simple variables* from the class $V = \{V_1, V_2, \cdots\}$;

(ii) h, f denote operations, or basic (total) functions from the class $F = \{F_1, F_2, \cdots\}$. Each F_i has rank (number of arguments) RF_i;

(iii) p, q denote (total) *predicates* from the class $P = \{P_1, P_2, \cdots\}$. Each P_i has rank (number of arguments) RP_i;

(iv) l denotes a *label* from the class $L = \{L_1, L_2, \cdots\}$.

The grammar G[⟨program⟩] contains the rules[1]

⟨program⟩ :: = (v{, v}): ⟨body⟩

⟨body⟩ :: = ⟨S-list⟩; [l:] HALT(v)

⟨S-list⟩ :: = [l:] ⟨S⟩ {; [l:] ⟨S⟩}

⟨S⟩ :: = "empty";

|$v \leftarrow w$

|$v \leftarrow f(v_1, \cdots, v_{Rf})$

|IF $p(v_1, \cdots, v_{Rp})$ THEN [l:] ⟨S⟩ ELSE [l:] ⟨S⟩

|HALT(v)

|GOTO l

|BEGIN ⟨S-list⟩ END

The list of variables just before the ⟨body⟩ indicates the simple variables whose values will be the program input. They must be different. In addition, any label l used in a GOTO l statement must also label a statement, as [l:] ⟨S⟩. Each label l can be used only once to label a statement.

(2.2) *Example*. The following is a program scheme:

($V1$, $V2$): $V1 \leftarrow F1(V1, V2)$;

$L1$: IF $P1(V1)$

THEN BEGIN $V2 \leftarrow F3(V2)$; $V1 \leftarrow F2(V1)$; GOTO $L1$ END

ELSE HALT($V2$);

HALT($V2$)

We shall from time to time use programming constructions which are not strictly allowed, but which could obviously be transformed into correct constructions. We use the constructions in order to make "algorithms" clearer. Typical examples are

IF $P(X)$ *and* $Q(X)$ THEN $X \leftarrow F(X)$

[1] We use BNF notation, with the following additional notations: $\{x\}$ indicates 0, 1, 2 or more occurrences of x. $[x]$ means 0 or 1 occurrences of x.

which is equivalent to

IF $P(X)$

THEN IF $Q(X)$ THEN $X \leftarrow F(X)$

ELSE

ELSE

and

WHILE $P(X)$ DO $X \leftarrow F(X)$

which is equivalent to

L: IF $P(X)$ THEN

BEGIN $X \leftarrow F(X)$; GOTO L END

The rest of this section is concerned with giving our idea of what a program scheme is. We also describe several other restrictions we place on schemes and their meanings.

Paterson [4] introduces schemata as flow charts. We have decided to define them formally in terms of ALGOL-like program statements solely in order to reduce the number of diagrams necessary in the paper. We shall, however, resort to flow charts from time to time for purposes of clarity. A flow chart equivalent to the program scheme in (2.2) is shown in Fig. 1.

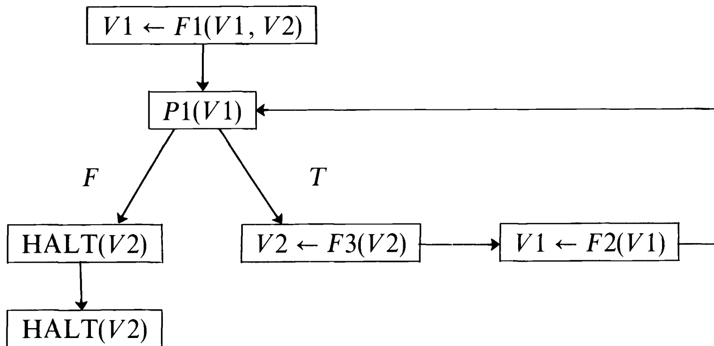


FIG. 1

Program schemes can express algorithms for computing over any domain D (e.g., $D = \mathbb{N} = \{0, 1, 2, \cdots\}$, or $D = \{\text{set of all binary trees}\}$). The operations over D are $f(\cdot): D^{Rf} \to D$, while the predicates are $p(\cdot): D^{Rp} \to \{T, F\}$. Therefore, parameters necessary in specifying a concrete programming language are the domain D, a subset of the operators $\mathscr{F}(D) = \{f(\cdot): D^{Rf} \to D\}$, and a subset of the predicates $\mathscr{P}(D) = \{p(\cdot): D^{Rp} \to \{T, F\}\}$.

Suppose we give an interpretation to the domain D, the predicates P_i and the functions F_i that can be used in a scheme. We then have a programming language, and any program scheme is a program in that language.

For instance, selecting $D = \mathbb{N}$, $f_1(x) = x + 1$, $f_2(x) = x \dot{-} 1$, and P_1 to be the predicate $\neq$, gives us the language G_3 which is universal for $\mathbb{N}$. That is, it expresses algorithms for all computable functions over $\mathbb{N}$.

To execute a program, the input values are put into the variables $v \{, v\}$ listed at the beginning of the ⟨program⟩, and the ⟨body⟩ of the program is executed. The way each of the statement types is executed should be obvious to any programmer, and we shall not discuss it further. We often use the phrase "execute a scheme" as an abbreviation for "execute a scheme under an interpretation of D, $\mathscr{F}$ and $\mathscr{P}$, with input values $\cdots$."

In summary, any particular program scheme can be interpreted as a map of n_1 operators, n_2 predicates, and n_3 input values into D:

$$\mathscr{F}(D)^{n_1} \times \mathscr{P}(D)^{n_2} \times D^{n_3} \to D.$$

Such maps are called *functionals* over D. The program scheme of (2.2) computes a functional

$$\mathscr{F}(D)^3 \times \mathscr{P}(D)^1 \times D^2 \to D.$$

(2.3) DEFINITION. $\mathbb{F}(\mathrm{P}, D)$ is the set of all functionals over D computable by program schemes in P.

Abstracting programs into program schemes may lead to a better understanding of conventional programming language features, because it allows us to separate completely the values being computed from the data structures and control mechanisms used in the program. For example, we can define schemes using arrays of subscripted variables, where the subscripting mechanism is completely divorced from the values computed. No subscript value can be used as an argument to a basic function and no basic value can be used as a subscript value.

This allows us to compare the class $\mathrm{P_A}$ of program schemes using arrays with the class $\mathrm{P_R}$ of programs allowing recursive procedures but no arrays, and we get the result that $\mathrm{P_A}$ is the more powerful class!

All our schemes yield only a single output value, the value of the variable v in the HALT(v) statement which halts the execution. This is not a real restriction; our only purpose in limiting the schemes this way is to arrive at a clear and concise discussion.

All basic predicates and functions are assumed to be *total*, that is, they yield a valid value in D for any set of input arguments whose values are in D. Other people working in the area have allowed partial functions and predicates. Some of our work carries over easily when using partial functions and predicates; other parts do not. Also no form of *parallelism* can be simulated without *parallelism* if basic functions or predicates are partial.

In any case, we feel that totality of functions and predicates is a more useful and desirable feature. In any "real" programming system, these basic functions should be total. They may yield an undefined value as a result, but this is better than a system function looping indefinitely because of an input error.

Furthermore, if one intends to characterize the behavior of schemata in which the function inputs are subroutines (hence possibly partial), this can be realistically done by allowing schemata as arguments to other schemata.

Our main purpose in this paper is to compare several different classes of program schemata and show they are equivalent, or not equivalent. Our equivalence concept should include schemes which execute "incorrectly" under some interpretation. For example, the scheme

$$(V): \quad \text{HALT}(W)$$

will always execute incorrectly, because the output variable W has never been assigned a value! Nevertheless, we must take such schemes into account.

Let us use the convention that each domain D includes an arbitrary value Ω, which we shall call the undefined value. In any scheme, before execution (under some interpretation), each variable is initialized to this value Ω. We can also assume the simple variable *OMEGA* is never assigned a value during execution, and thus always contains the value Ω.

We shall define program schemata using labels as values, special markers, and so on. In order to separate the different features as much as possible, we do not want to allow these labels or markers to be in the domain D. For example, if one class of schemata allows $v \leftarrow f(m, w)$ where m is a special marker and f is a basic function, then clearly we cannot have an equivalent scheme in a class which does not allow markers.

Our convention then is:

(2.4) If a value not in the domain D is used as an argument to a basic predicate, function, or HALT operation, then the value Ω is used in its place.

Again, this convention is not essential to the theory (the matter could be handled as it is in "real programming languages"), but it leads to cleaner results.

(2.5) DEFINITION. Two program schemes, S_1, S_2, with inputs $x_1, \cdots, x_{n_1}$, $f_1, \cdots, f_{n_2}, p_1, \cdots, p_{n_3}$ and output y are (input–output) *equivalent over D* if and only if they compute the same functional over D, i.e., $S_1[x_1, \cdots, x_{n_1}, f_1, \cdots, f_{n_2}, p_1, \cdots, p_{n_3}] = S_2[x_1, \cdots, x_{n_1}, f_1, \cdots, f_{n_2}, p_1, \cdots, p_{n_3}]$ for all $x_1, \cdots, x_n$ in D. Two schemes are *equivalent* if and only if they are equivalent over all D.

(2.6) DEFINITION. Let P_1 and P_2 be two classes of program schemes. We say that P_1 is *less powerful than* P_2, written $P_1 < P_2$, if

(1) for every scheme S in P_1 there exists an equivalent scheme in P_2;

(2) there exists a scheme S_2 in P_2 with no equivalent scheme in P_1.

Equivalently, $P_1 < P_2$ if $\mathbb{F}(P_1, D) \subset \mathbb{F}(P_2, D)$ for all D. P_1 has *the same power as* P_2, written $P_1 \equiv P_2$, if $\mathbb{F}(P_1, D) = \mathbb{F}(P_2, D)$ for all D. We write $P_1 \leqq P_2$ if either $P_1 < P_2$ or $P_1 \equiv P_2$.

3. Recursive program schemes and macro expansion. Recursive definition is one of the basic ways of specifying functions in mathematics. The underlying structure of recursive definition can be specified at the scheme level through the notion of a recursive scheme.

(3.1) DEFINITION. A (*recursive*) *definition of a function* f of rank Rf has the form $f(x_1, x_2, \cdots, x_{Rf}) = e_0$, where e_0 is an expression. An expression e (or e_i) has one of the three forms:

(i) x_i (the ith parameter; $1 \leqq i \leqq Rf$);

(ii) $h(e_1, e_2, \cdots, e_{Rh})$, where h is a function (it may be f);

(iii) IF $p(e_1, e_2, \cdots, e_{Rp})$ THEN e_i ELSE e_j, where p is a (total) predicate.

(3.2) DEFINITION. A *recursive scheme* consists of a finite set of definitions of functions $f_1, f_2, \cdots, f_n$, together with an initial call $f_i(v_1, \cdots, v_{Rf_i})$ of one of the functions f_i, using as arguments the values $v_1, \cdots, v_{Rf_i}$.

The class of recursive schemes is denoted by R. Note that a recursive scheme is a map from

$$\mathscr{F}(D)^{n_1} \times \mathscr{P}(D)^{n_2} \times D^{n_3} \to D$$

for some n_1, n_2, and n_3. Thus, both recursive and program schemes compute over the same set of domains. It is therefore reasonable to ask whether they compute the same class of functionals over all domains D. Precisely, is $\mathbb{F}(\mathrm{R}, D) = \mathbb{F}(\mathrm{P}, D)$ for all D? In [4] Paterson and Hewitt show the following result.

(3.3) THEOREM. *There exists a D such that* $\mathbb{F}(\mathrm{P}, D) \subset \mathbb{F}(\mathrm{R}, D)$.

In fact, they show the result for any infinite domain D by showing it for the free domain $D = \{F_i, P_i | i \in \mathbb{N}\}$. This result makes it clear that the evaluation mechanism for recursion is more "powerful" than that of programs using only simple variables. In other words, programming languages using a finite number of variables but allowing recursion are more powerful than those without recursion.

Since $\mathbb{F}(\mathrm{P}, D)$ is a subset of $\mathbb{F}(\mathrm{R}, D)$, it is convenient to introduce the notion of the class of recursive program schemes $\mathrm{P_R}$, in a way which yields $\mathbb{F}(\mathrm{R}, D) \equiv \mathbb{F}(\mathrm{P_R}, D)$.

(3.4) DEFINITION. A *recursive program scheme* in $\mathrm{P_R}$ is a sentence of the grammar G[⟨recursive program⟩] given below (it uses the rules for ⟨program⟩, etc. given in (2.1)).

$$\langle\text{recursive program}\rangle ::= \langle\text{program}\rangle \{\langle\text{function def}\rangle\}$$

$$\langle\text{function def}\rangle \quad ::= f(v_1, \cdots, v_{Rf}) : \langle\text{body}\rangle$$

Thus, a recursive program scheme consists of a program plus a sequence of function definitions. Those functions which are defined are called *recursive functions*, as opposed to the basic functions not defined. "Execution" is as before, except that when a recursive function f is called, the following happens:

(1) The values of the arguments of the call are assigned to the corresponding variables (formal parameters) $v_1, \cdots, v_{Rf}$, and the other variables used in the function are set to Ω, the undefined value.

(2) The function's ⟨body⟩ is executed until a statement HALT(v) is executed. The value of the variable v is then the value of the function, and execution resumes just after the point of call.

The variables and labels used in a function definition are assumed to be disjoint from those in the ⟨program⟩ and other function definitions. They are also assumed to be different for each invocation of the function. In programming terminology, they are *local* variables and labels. The value of v in the HALT(v) operation which ends execution of a function need not be in D. In other words (2.4) does not hold for HALTs executed within a function.

(3.5) *Example.*

(V, W): $V \leftarrow F1(V, W); W \leftarrow F2(V, W);$ HALT(V)

$F2(X, Y)$: L: IF $P(X)$ THEN

BEGIN $X \leftarrow F3(X)$; $Y \leftarrow F2(X, Y)$; GOTO L

END

ELSE HALT(Y);

HALT(Y)

The equivalence of P_R and R was first proved by McCarthy [3] (using different notation and different definitions). Let us show how to translate a recursive scheme into a recursive program scheme (with an intuitively equivalent meaning).

(3.6) *Transformation of* $S \in R$ *to* $S' \in P_R$. The initial call $f(v_1, \cdots, v_{Rf})$ is transformed into the ⟨program⟩

$$(V_1, \cdots, V_{Rf}): OUTPUT \leftarrow f(V_1, \cdots, V_{Rf}); \text{HALT } (OUTPUT)$$

Each function definition for a function f_i is translated into

$$f_i(V_1, V_2, \cdots, V_{Rf_i}); V \leftarrow e_0; \text{HALT}(V)$$

Within e_0, references to the arguments x_j are replaced by the variable names V_j. We now iterate the following two steps on each function definition until neither is applicable:

(a) if there is an assignment $v \leftarrow f(e_1, \cdots, e_{Rf})$ where at least one e_i is not a variable, then generate new variables $V_1, \cdots, V_{Rf}$ and replace the assignment by

BEGIN $V_1 \leftarrow e_1; \cdots; V_{Rf} \leftarrow e_{Rf}$;

$v \leftarrow f(V_1, \cdots, V_{Rf})$

END

(b) if there exists an assignment

$$v \leftarrow \text{IF } p(e_1, \cdots, e_{Rp}) \text{ THEN } e_i \text{ ELSE } e_j$$

then generate new variables $V_1, \cdots, V_{Rp}$ and replace it by

BEGIN $V_1 \leftarrow e_1; \cdots; V_{Rp} \leftarrow e_{Rp}$;

IF $p(V_1, \cdots, V_{Rp})$ THEN $v \leftarrow e_i$ ELSE $v \leftarrow e_j$

END

A particular function call in a "real" program can alternatively be thought of as a macro call. By this we mean that, before the program is ever executed, the function call is replaced by the ⟨body⟩ of the function definition, with suitable changes of variable names and suitable initialization of parameter and result variables, to yield a new equivalent program. During execution, the function evaluation is then performed "in-line." We can perform the same process with schemata.

(3.7) *Macro expansion.* Given a program scheme in P_R with a function call $v \leftarrow f(w_1, \cdots, w_{Rf})$, where f is defined by a ⟨function def⟩, we *expand* the call to yield a new, equivalent scheme in P_R, as follows:

(a) Make a copy of the ⟨body⟩ of the ⟨function def⟩ of f. Let the local variables and labels used be $V_1, \cdots, V_n, L_1, \cdots, L_m$. Create entirely new variables and

labels $V_1^*, \cdots, V_n^*, L_1^*, \cdots, L_m^*$, and change each reference to V_i (L_j) within the copy of the ⟨body⟩ to V_i^* (L_j^*).

(b) Create a new label L. Replace each statement HALT(V_i^*) within the copy of the ⟨body⟩ by

$$\text{BEGIN } v \leftarrow V_i^*; \text{ GOTO } L \text{ END}$$

(remember, the function call is $v \leftarrow f(w_1, \cdots, w_{Rf})$).

(c) Suppose the formal parameters of the function f were $V_1, V_2, \cdots, V_{Rf}$. Replace the call $v \leftarrow f(w_1, \cdots, w_{Rf})$ by the statement

$$\text{BEGIN } V_1^* \leftarrow w_1; \cdots; V_{Rf}^* \leftarrow w_{Rf};\ V_{Rf+1}^* \leftarrow OMEGA; \cdots; V_n^* \leftarrow OMEGA;$$

(copy of ⟨body⟩ resulting from b);

L:

END

(3.8) *Example.* Let S be

```
        (W):   V ← W;
               V ← F(V, W);
               HALT(V)
     F(X, Y):  X ← Y; HALT(X)
```

Expansion of $V \leftarrow F(V, W)$ yields

```
        (W):   V ← W;
               BEGIN X* ← V; Y* ← W;
                       X* ← Y*; BEGIN V ← X*; GOTO L END
          L:   END;
               HALT(V);
     F(X, Y):  X ← Y; HALT(X)
```

Later, when showing that $P_R < P_A$, we will have to be able to expand all possible "first-level" calls of recursive functions. A call $f(\cdots)$ is a *first-level call* if during execution of the scheme (under some interpretation) there is a chance the call will *not* occur during another execution of f. During execution, a call of f is *recursive* if it occurs during another execution of f.

(3.9) *Expansion of all possible first-level calls.* We construct a scheme S' (from a scheme S) which has all possible first-level calls expanded, as follows:

(a) Within the main ⟨program⟩, rewrite each assignment $v \leftarrow f(\cdots)$ where f is not basic, as

$$v \leftarrow f(\cdots)\{\ \}.$$

Similarly, within each definition of a function g, rewrite each assignment $v \leftarrow f(\cdots)$ where f is not basic, as

$$v \leftarrow f(\cdots)\{g\}.$$

{ } denotes the empty list, and indicates that the function call appears in no function. $\{g\}$ indicates the call definitely appears within an execution of g. In general, as the construction continues, inside the braces will be a list of functions in which the particular call appears.

(b) Iterate the following step until no longer possible, first on the main program scheme, and then on each of defined functions: Choose an assignment

$$v \leftarrow f(w_1, \cdots, w_{Rf})\{f_j, \cdots, f_k\}$$

such that f is not in the list $\{f_j, \cdots, f_k\}$. This indicates that this call does not appear within another execution of f, and is thus a possible first-level call. Expand the function call as described in (3.7). Add to the list attached to each nonbasic function call in the expanded function body the list $\{f, f_j, \cdots, f_k\}$ to indicate in which function executions the function call occurs.

(c) Delete all the lists $\{\cdots\}$ from the function calls.

The above construction, if it terminates, obviously results in an equivalent program scheme. Second, all possible first-level calls are expanded, since as long as one exists, step (b) will iterate and expand one.

Now note that each macro expansion deletes one call, and adds several new ones (the ones appearing in the expanded function body). However, the number of the elements in the list attached to each of the new ones is at least one more than the number in the list attached to the deleted one. This, together with the fact that the number of elements in the list attached to a first-level call must be less than the number of ⟨function def⟩s, can be used to show that step (b) terminates.

(3.10) COROLLARY. *Suppose a scheme in* $\mathrm{P_R}$ *has no recursive calls under any interpretation and any execution. Then there exists an equivalent scheme in* P.

Proof. Expand all possible first-level calls as described in (3.9). If any function calls still exist in the main ⟨program⟩, they can never be executed (they are not first-level calls, and are thus recursive calls). Replace each by a null statement. Since the functions defined by ⟨function def⟩s are no longer needed, they can be deleted yielding a scheme in P. Q.E.D.

The following interesting relationship between recursive calls and the predicates used in a scheme will be used later when comparing the class $\mathrm{P_R}$ with other classes of schemata.

(3.11) THEOREM. *Suppose during execution of a scheme in* $\mathrm{P_R}$ *that a function f is called recursively—that is, it is called a second, third,* $\cdots$, *n-th time before one execution is completed. Then, before the last (n-th) execution of f can return (can execute a* HALT(v)), *some predicate* P_i *must have been evaluated with two different lists of argument values* a_1 *and* a_2, *such that* $P_i(a_1) \neq P_i(a_2)$.

Proof. Label all the statements of the scheme with unique labels. During an execution of the scheme, record the sequence of labels of statements in the order in which the statements *begin* executing, beginning with the first statement executed in the first invocation of f, up until the nth call of f:

$$L_1, L_2, L_3, \cdots, L_m.$$

In a separate list, record the sequence of statements executed during the nth (deepest) execution of f, until a HALT(v) of f is executed:

$$L'_1, L'_2, L'_3, \cdots, L'_q.$$

Now, $L_1 = L'_1$, since both sequences begin with the label of the first statement of f. If $q \leqq m$, then $L'_q \neq L_q$, since L'_q labels a HALT(v) in f, while L_q cannot. If $q > m$, then $L_m \neq L'_m$ since L_m labels a call of f while L'_m cannot. In any case, the sequences start out the same, and at some point become different. There is a first integer i, $1 < i \leqq q, m$, such that $L_i \neq L'_i, L_{i-1} = L'_{i-1}$. This can only be if L_{i-1} labels a statement of the form

$$\text{IF } p(\cdots) \text{ THEN } S_1 \text{ ELSE } S_2$$

and if the evaluations of p yielded two different values.

(3.12) COROLLARY. *Given a scheme S in P_R, expand all possible first-level calls as macros, using* (3.9), *to yield scheme S'. Then, during execution under any interpretation of S', before any nonbasic function executes a* HALT(v) (*and returns*), *a predicate P_i will have been evaluated with two lists a_1 and a_2 of argument values, such that $P_i(a_1) \neq P_i(a_2)$.*

Proof. Suppose function f in S' executes a HALT(v) (and returns). In the equivalent scheme S, the same HALT(v) must be executed in an nth level of the recursive function f. By virtue of Theorem (3.11), therefore, a predicate P_i has been evaluated twice with the property described above.

4. Other classes of schemes. We now introduce other classes of program schemata, the class P_A of program schemata allowing arrays of subscripted variables, the class P_{Ae} of schemata allowing arrays of variables and the testing for equality of subscript values, the class P_L of schemata allowing the use of labels l as values which can be assigned to variables, the class P_M of schemata allowing a finite number of "markers" to be used as values, the class P_{pds} of schemata which can use pushdown stores, and the class $P_{\mathbb{N}}$ which allows integer arithmetic.

P_A and P_L are interesting because they contain features allowed in current high-level languages (e.g., PL/I). The pushdown store seems to be one of the favorite data structures of the theoreticians, and the purpose of P_{pds} and P_M is to compare the theoreticians' viewpoint on schemata with the programmers'.

(4.1) DEFINITION. P_A is defined as the class of program schemata P, extended by allowing v and w to denote either simple variables v or subscripted variables $a[w]$. Only the input variables must be simple variables.

A (one-dimensional) array B is a sequence $B_0, B_1, \cdots$ of simple variables. Any value in $\mathbb{N} = \{0, 1, 2, \cdots\}$ can be used as a subscript value. In order to be able to generate subscript values during execution, we allow the new statements

$$\langle S \rangle \quad ::= \quad v \leftarrow 0 \quad | v \leftarrow w + 1$$

Execution of $v \leftarrow 0$ assigns 0 to the variable v. $+1$ is the successor function; if v contains a value i in $\mathbb{N}$, then execution of $w \leftarrow v + 1$ puts into w its successor $i + 1$.

We assume that $\mathbb{N}$ is disjoint from the domain D under any interpretation (this will be discussed in a moment). Finally, if v contains a value not in $\mathbb{N}$, then $v + 1$ yields the value 1.

If v contains a value $i \in \mathbb{N}$, then $B[v]$ references the variable B_i. If v contains a value not in $\mathbb{N}$, then $B[v]$ references B_0.

Note that since $\mathbb{N}$ and D are disjoint, if $i \in \mathbb{N}$ is used as an argument to a predicate, function, or HALT operation, the value Ω is used instead. This restriction

is necessary to rule out meaningless comparisons between schemes with and without arrays. The restrictions are not arbitrary; they serve to separate the allocation task from the other computation mechanisms.

The reader may claim that we are not describing subscripting as performed in concrete programming languages, for three reasons:

(1) subscript values may not appear as input, since they are not in the domain D;
(2) functions like $+1$ are usually allowed as "basic" operations; and
(3) subscript values may not be used as input to, or be output from, basic functions and predicates.

Our reply is as follows. We are trying to isolate different programming language concepts and get as "clean" a description of them as possible. Second, we want to compare the computational power of different classes of program schemata. If one class is allowed to compute using the function $+1$ with restricted properties and the other class is not, then comparison is meaningless. However, if $+1$ is used *only* to help "allocate storage" and refer to different variables, and *cannot* be used in the actual computation, then a comparison may make sense.

Should a particular interpretation of a program scheme have a function F with the same properties as $+1$, then F and $+1$ can be identified with each other in that interpretation. Similarly, should the domain D contain $\tilde{\mathbb{N}}$ isomorphic to $\mathbb{N}$, then the two can be identified with each other in that interpretation.

Strictly speaking, statements like

$$(1)\ B[V+3] \leftarrow W, \quad (2)\ W \leftarrow 2, \quad \text{and} \quad (3)\ B[5] \leftarrow W$$

are not valid in P_A schemes. We allow them because they make schemes clearer, and they can be easily translated into P_A statements. For example, statements (1) and (2) above can be translated into

```
BEGIN X ← V + 1;                 BEGIN W ← 0;
      X ← X + 1;                       W ← W + 1;
      X ← X + 1;        and            W ← W + 1
      B[X] ← W                   END
END
```

Note, however, when we say "given some scheme S in P_A" (or P_{AM}, etc.) *we always mean a scheme which is strictly in that class.*

Similarly, we can prove that the use of the predecessor function $\dot{-}\,1$ defined by

$$w \dot{-} 1 = \begin{cases} i - 1 & \text{if } w \text{ contains } i \in \mathbb{N},\ i \neq 0, \\ 0 & \text{if } w \text{ contains } 0 \text{ or } w \notin \mathbb{N} \end{cases} \tag{4.2}$$

does not increase the power of the scheme.

(4.3) THEOREM. *Let S be a scheme in P_A, except for the use of the predecessor function. Then S can be translated into an equivalent scheme S' in P_A.*

Proof. S' uses a new array DOWN to simulate the predecessor function.

Thus DOWN[0] = 0, and DOWN[i] will contain $i \dot{-} 1$ for $i > 0$, whenever necessary. We translate S into S' as follows:

1. Add DOWN[0] $\leftarrow 0$; at the beginning of the scheme S.
2. Replace each statement $v \leftarrow w + 1$ in S by

$$\text{BEGIN DOWN}[w + 1] \leftarrow w; v \leftarrow w + 1 \text{ END}$$

3. Replace each statement $v \leftarrow w \dot{-} 1$ in S by

$$v \leftarrow \text{DOWN}[w]. \qquad \text{Q.E.D.}$$

A rather interesting (but easy) result is that any scheme needs really only *one* array. Note that the proof below depends only on the array feature, and therefore the same result holds for classes in P_{AL}, P_{AM}, etc., which will be defined later.

(4.4) THEOREM. *For any scheme S in P_A there exists an equivalent scheme S' in P_A which uses only one array.*

Proof. Let the arrays used in S be $A_0, \cdots, A_{n-1}$. The scheme in S' uses a single array A, where

$$\begin{array}{ll} A[0], A[n], A[2n], \cdots & \text{represents } A_0, \\ A[1], A[n+1], A[2n+1], \cdots & \text{represents } A_1, \\ \vdots & \vdots \\ A[n-1], A[2n-1], A[3n-1], \cdots & \text{represents } A_{n-1}. \end{array}$$

To do this, we need only change each assignment $v \leftarrow w + 1$ to $v \leftarrow w + n$, and then change each reference $A_i[v]$ to $A[v + i]$.

(4.5) DEFINITION. A scheme in the class P_{Ae} is a scheme in P_A with the following statement type allowed:

$$\langle S \rangle ::= \text{IF } v \ominus w \text{ THEN } [l:] \langle S \rangle \text{ ELSE } [l:] \langle S \rangle$$

The predicate $v \ominus w$ is defined as follows:

$$v \ominus w = \begin{cases} T & \text{if } v \text{ and } w \text{ are both not in } \mathbb{N} - \{0\}, \\ T & \text{if } v, w \in \mathbb{N} \text{ and are the same}, \\ F & \text{otherwise.} \end{cases}$$

Thus $v \ominus w$ if and only if $A[v]$ and $A[w]$ refer to the same simple variable.

(4.6) DEFINITION. A scheme in the class P_L of program schemata allowing labels as values is a scheme in P with the following two extra statements allowed:

$$\langle S \rangle ::= v \leftarrow l \mid \text{GOTO } v$$

If a statement $v \leftarrow l$ exists in a scheme, l must actually label some statement. Execution of $v \leftarrow l$ assigns the label l to the variable v. If v contains a label l, execution of GOTO v causes control to be transferred to the statement labeled l; if v contains any other value, GOTO v acts like a null statement. We assume that the set of labels L is disjoint from the domain of values D under any interpretation.

(4.7) DEFINITION. Let s denote a *pushdown store* from the class PD $= \{\mathrm{PD}_1, \mathrm{PD}_2, \cdots\}$ of pushdown stores. $\mathrm{P}_{\mathrm{pds}}$ is defined as the class of schemata P extended by the following two statements:

$$\langle S \rangle ::= \langle \text{pushdown} \rangle | \langle \text{popup} \rangle$$

$$\langle \text{popup} \rangle ::= v \leftarrow s$$

$$\langle \text{pushdown} \rangle ::= s \leftarrow v$$

Execution of a $\langle$pushdown$\rangle$ statement causes the contents of v to be placed on the store s, and they become the *top of the store*. Execution of a $\langle$popup$\rangle$ statement causes the value at the top of the store to be placed in v, after which the second element in the store becomes the new top. If $v \leftarrow s$ is executed and s is currently empty, no change is made to v or to s. Each pushdown store is initially empty.

Note carefully the effect of a $\langle$popup$\rangle$ if the store is empty. Sometimes the convention is used that such a pop causes the machine to jam, or loop infinitely. This interpretation could also have been used here.

(4.8) DEFINITION. Let m denote a member of the class of markers $\mathrm{M} = \{\mathrm{M}_1, \mathrm{M}_2, \cdots\}$. P_{M} is defined as the class of schemata P, extended by the statements:

$$\langle S \rangle ::= v \leftarrow m$$

$$|\ \text{IF } v = m \text{ THEN } [l:]\ \langle S \rangle \text{ ELSE } [l:]\ \langle S \rangle$$

We assume that M is disjoint from the domain D under any interpretation. Note that any program scheme in P_{M} can use only a finite number of markers.

(4.9) DEFINITION. Let $\underset{\sim}{\mathbb{N}} = \{\underset{\sim}{0}, \underset{\sim}{1}, \underset{\sim}{2}, \underset{\sim}{3}, \cdots\}$. $\mathrm{P}_{\underset{\sim}{\mathbb{N}}}$ is defined as the class of schemata P, extended by the statements:

$$\langle S \rangle ::= v \leftarrow \underset{\sim}{0}$$

$$|\ v \leftarrow w + \underset{\sim}{1} | v \leftarrow w \mathbin{\dot{-}} \underset{\sim}{1}$$

$$|\ \text{IF } v \boxed{=} w \text{ THEN } [l:]\ \langle S \rangle \text{ ELSE } [l:]\ \langle S \rangle$$

where $v \boxed{=} w$ is defined by

$$v \boxed{=} w = \begin{cases} T & \text{if } v \text{ and } w \text{ are both not in } \underset{\sim}{\mathbb{N}} - \{\underset{\sim}{0}\}, \\ T & \text{if } v, w \in \underset{\sim}{\mathbb{N}} \text{ and are the same}, \\ F & \text{otherwise}. \end{cases}$$

$v \leftarrow w + \underset{\sim}{1}$ $(v \leftarrow w \mathbin{\dot{-}} \underset{\sim}{1})$ adds (subtracts) one from w, as usual; any value $w \notin \underset{\sim}{\mathbb{N}}$ acts like $\underset{\sim}{0}$.

The reader will note that schemes in P_{Ae} already have the ability to do integer arithmetic as described above, using $\mathbb{N}$ instead of $\underset{\sim}{\mathbb{N}}$. We shall, however, find it interesting to describe schemes using integer arithmetic but not arrays.

Given these classes of schemata P, P_{R}, P_{A}, P_{Ae}, P_{L}, P_{M}, $\mathrm{P}_{\underset{\sim}{\mathbb{N}}}$, and $\mathrm{P}_{\mathrm{pds}}$, we can generate other classes in an obvious way. Thus $\mathrm{P}_{\mathrm{AL}} \equiv \mathrm{P}_{\mathrm{LA}}$ is the class of program schemata using arrays, and labels as values. We assume of course the obvious interpretation when the various mechanisms interact. Thus, if we look at $\mathrm{P}_{\mathrm{AeL}}$,

GOTO v has no effect if v contains a value in $\mathbb{N}$, while $v \ominus 0$ is true if v contains a label.

The classes we are mainly interested in are P_A, P_{Ae}, P_{AL}, P_{AM} and P_{pdsM}. In fact, we show they have the same computational power. However, the equivalence between P_A and the others is not effective. Thus, given a P_{AM} scheme, an equivalent P_A scheme exists, but one cannot always construct it.

5. Locators and P_i-simulators. One of the main results in this paper is that P_A has the same computational power as P_{AL} or P_{AM}. The proof of this result depends upon two things:

(1) simulating label values (or equivalently markers) by a predicate P_i and two values rt and rf where we *know* that $P_i(rt) = T$ and $P_i(rf) = F$. Thus, if the scheme uses k markers $m_1, m_2, \cdots, m_k$, we can use the vectors

$$\begin{gathered}(rt, rf, rf, \cdots, rf),\\(rf, rt, rf, \cdots, rf),\\\vdots\\(rf, \cdots, rf, rt)\end{gathered}$$

to represent them;

(2) being able to *locate* the predicate P_i and values rt and rf with the above properties.

Let us define these concepts formally.

(5.1) DEFINITION. Let a basic predicate P of rank n be used in a scheme $S_1 \in P_{AL}$. Let $RT_1, \cdots, RT_n$, $RF_1, \cdots, RF_n$ be variables not used in S_1. $S_2 \in P_A$ is called a *P-simulator of* S_1 if the following holds:

> if $P(RT_1, \cdots, RT_n) = T$ and $P(RF_1, \cdots, RF_n) = F$ before execution begins, then S_2 will execute equivalently to S_1.

The term P-simulator is used because the label variables of S_1 will be "simulated" using the predicate P. Given a scheme $S_1 \in P_{AM}$ (or P_{RM}), we similarly define a P-simulator S_2 of S_1 where S_2 is in P_A (or P_R). Here S_2 simulates the markers of S_1.

(5.2) DEFINITION. Given a scheme S (in any class of schemes) a *locator* for S is a scheme S' with the following properties:

- (i) S and S' have the same list of input variables and use the same basic functions and predicates.
- (ii) When executing, S' attempts to find a predicate P_i and two lists a_1 and a_2 of argument values such that $P_i(a_1) = T, P_i(a_2) = F$. If it finds them, S'
  - (a) puts the values in a_1 into variables $RT_1, \cdots, RT_{RP_i}$,
  - (b) puts the values in a_2 into variables $RF_1, \cdots, RF_{RP_i}$, and
  - (c) transfers control to a statement BEGINi: HALT($OMEGA$), where BEGINi is a new unique label.
- (iii) If S' does not find the predicate P_i and lists a_1 and a_2, then one of the following happens:
  - (a) S' executes infinitely long, and S does also,
  - (b) S' halts with value v, and S does too.

Before showing how to put locators and P_i-simulators together, let us first show how to construct a P-simulator in $\mathrm{P_A}$ for a scheme $\in \mathrm{P_{AL}}$.

(5.3) THEOREM. *For any scheme $S_1 \in \mathrm{P_{AL}}$ we can construct a P-simulator S_2 in $\mathrm{P_A}$, of S_1.*

Construction. We assume without loss of generality that P has rank 1. Let the simple variables RT and RF contain values rt and rf such that $P(RT) = T$ and $P(RF) = F$.

Suppose the labels which are used in statements $v \leftarrow l$ throughout the scheme are $L_1, L_2, \cdots, L_k$. Then, for each simple variable v in S_1, S_2 uses variables $v, v^0, v^1, \cdots, v^k$. For each array A used in S_1, S_2 uses arrays $A, A^0, A^1, \cdots, A^k$. During execution of S_2, if a variable v should currently contain a value in $D \cup \mathbb{N}$, then that value is in v, while $v^0 = rt$. If the value should be the equivalent of a label L_i in S_1, then $v^0 = \Omega$, $v^i = rt$, and $v^j = rf$ for $0 \leqq j \leqq k, j \neq i$.

It should be clear now how to transform S_1 into $S_2 \in \mathrm{P_A}$:

(a) change every assignment $v \leftarrow w$ to

$$\text{BEGIN } v \leftarrow w; v^0 \leftarrow w^0; \cdots; v^k \leftarrow w^k \text{ END}$$

(b) change every assignment $v \leftarrow L_i$ to

$$\text{BEGIN } v \leftarrow OMEGA; v^0 \leftarrow RF; \cdots; v^k \leftarrow RF; v^i \leftarrow RT \text{ END}$$

(c) change every assignment $v \leftarrow f(\cdots)$ to

$$\text{BEGIN } v \leftarrow f(\cdots); v^0 \leftarrow RT \text{ END}$$

(d) change every statement GOTO v to

```
BEGIN IF P(v^0) THEN
      ELSE IF P(v^1) THEN GOTO L_1
           ELSE IF P(v^2) THEN GOTO L_2
                ELSE ···
                     ELSE IF P(v^k) THEN GOTO L_k
END
```

Note that the simulation of GOTO v works correctly if v does not contain a label L_i; control passes to the next statement in the sequence. Note also that if v has never been assigned a value, then $v = v^0 = \cdots = v^k = \Omega$. The simulator works correctly whether $P(\Omega) = T$ or $P(\Omega) = F$.

(5.4) THEOREM. *For any scheme $S_1 \in \mathrm{P_{AM}}$ we can construct a P-simulator $S_2 \in \mathrm{P_A}$.*

Proof. (The proof is similar to the proof of (5.3); the markers are simulated in a manner similar to the simulation of label variables. The proof is left to the reader.)

The problem is, of course, to find some predicate P which is not identically true or false (if it exists) and to find the corresponding arguments. This is the purpose of the locator. Given S_1 in $\mathrm{P_{AL}}$ (or $\mathrm{P_{AM}}$) let us show how to put a locator together with P_i-simulators together to form an equivalent scheme.

(5.5) THEOREM. *Let S_1 in $\mathrm{P_{AL}}$ (or $\mathrm{P_{AM}}$) have a locator in $\mathrm{P_A}$. Then there exists a scheme in $\mathrm{P_A}$ equivalent to S_1.*

Proof. Suppose the locator in $\mathrm{P_A}$ is

$$(v_1, \cdots, v_m): \langle \text{body} \rangle$$

Suppose S_1 uses n predicates $P_1, \cdots, P_n$, and suppose for $i = 1, \cdots, n$ that the P_i-simulator in $\mathrm{P_A}$ for S_1 is

$$(v_1, \cdots, v_m): \langle \text{body} \rangle_i$$

Then the following scheme, where each statement BEGINi: HALT(*OMEGA*) in the locator's ⟨body⟩ is replaced by the null statement, is clearly in $\mathrm{P_A}$ and is clearly equivalent to S_1. $W_1, \cdots, W_m$ are new, unique variables. Remember, when the locator finds a two-valued predicate P_i, it initializes RT_j, RF_j, $j = 1, \cdots, RP_i$, and then jumps to BEGINi. Each simulator uses the global variables RT_j, RF_j:

$$
\begin{aligned}
&(v_1, \cdots, v_m): W_1 \leftarrow v_1; \cdots; W_m \leftarrow v_m; \langle \text{body} \rangle; \\
&\quad \text{BEGIN1}: v_1 \leftarrow W_1; \cdots; v_m \leftarrow W_m; \langle \text{body} \rangle_1; \\
&\quad \vdots \qquad\qquad\qquad\qquad \vdots \\
&\quad \text{BEGIN}n: v_1 \leftarrow W_1; \cdots; v_m \leftarrow W_m; \langle \text{body} \rangle_n
\end{aligned}
$$

Q.E.D.

The following result will help us later in proving that $\mathrm{P_R} < \mathrm{P_A}$.

(5.6) THEOREM. *Any scheme S in $\mathrm{P_R}$ has a locator S_1 in* P.

Construction. We assume without loss of generality that the predicates $P_1, \cdots, P_n$ used in S have rank 1. The construction of S_1 proceeds as follows:

(a) Expand all possible first-level calls of S (construction (3.9)).

(b) Replace each statement of the form

IF $P_i(v)$ THEN S_1 ELSE S_2

by

```
IF Pi(v)
THEN IF Pi(RT) THEN S1
                ELSE BEGIN RF ← RT; RT ← v;
                           GOTO BEGINi
                     END
ELSE IF Pi(RT) THEN BEGIN RF ← v; GOTO BEGINi
                     END
     ELSE S2
```

(c) Add statements BEGINi: HALT(*OMEGA*) at the end of the main ⟨body⟩ of the scheme. All jumps GOTO BEGINi are assumed to be jumps to these "global" labels.

(d) Note that, as soon as a predicate is determined with two argument values a_1, a_2 such that $P_i(a_1) \neq P_i(a_2)$, control transfers to BEGINi. By virtue of Theorem (3.12) and step (b), no recursive function of the scheme constructed so far ever executes a HALT (or return). Replace each HALT(v) within a function definition by the null statement.

(e) No function ever returns. Suppose a function B is called within a function A (see Fig. 2). Since B never returns, there is no need to save A's variables.

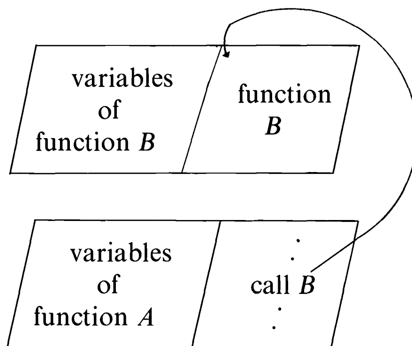


FIG. 2

We can therefore replace each call by a GOTO and change each function to a normal sequence of statements in the main scheme. To do this, perform the following for each function f. Assume that the variables $V_1, \cdots, V_{rf}, \cdots, V_n$ used in each f are different from those in the main scheme or other functions. Change the function definition

$$f(V_1, \cdots, V_{rf})\colon \langle \text{body} \rangle$$

to

$$f\colon \langle \text{body} \rangle$$

For each call $v \leftarrow f(w_1, \cdots, w_{rf})$, generate new variables $W_1, \cdots, W_{rf}$ and replace the call by

$$\begin{aligned}\text{BEGIN } & W_1 \leftarrow w_1; \cdots; W_{rf} \leftarrow w_{rf};\\ & V_1 \leftarrow W_1; \cdots; V_{rf} \leftarrow W_{rf};\\ & V_{rf+1} \leftarrow OMEGA; \cdots; V_n \leftarrow OMEGA;\\ & \text{GOTO } f\\ \text{END} & \end{aligned}$$

The result is a locator in P for S_1.

6. The relation between P_A and P_R. We begin by showing how to construct a scheme in P_{AL} equivalent to a given scheme in P_R. We then show how to get rid of the labels as values (using the results of the last section) to arrive at an equivalent

scheme in P_A. We end with a scheme which can be described in P_A but not in P_R, which yields $P_R < P_A$. Thus, the programming language feature of arrays of subscripted variables is more powerful than that of recursive procedures.

Given a P_R scheme, the equivalent P_{AL} scheme uses an array A of subscripted variables as a stack to hold the parameters, local variables, and return information of the functions currently being executed. This information for a particular execution of a function is collected in a "contiguous" set of subscripted variables in the array A, called a *data area*. In a typical implementation of recursive procedures in an ALGOL-like environment, when a function execution is complete, the corresponding data area locations on the stack are released for other use. Here we are not interested in efficiency of any sort, and these locations will not be freed.

A global variable TOP will always contain the value i of the index of the last subscripted variable allocated. A second variable AA (for Active Area) always contains the index j of the data area of the function currently being executed (the *active* function). If its data area is $k + 1$ locations long, it consists of the variables $A[\mathrm{AA}]$, $A[\mathrm{AA} + 1], \cdots, A[\mathrm{AA} + k]$.

Consider a function f. Assume the variables it uses are $V_1, V_2, \cdots, V_p$, where $V_1, V_2, \cdots, V_{Rf}$ are the formal parameter variables. Then the data area for an execution of f is $p + 2$ locations long and has the format shown in Fig. 3.

AA value for function that called this one
V_1
$\vdots$
V_p
return label

FIG. 3

The "return label" is the statement label to which control must return on completion of the function execution.

A call to f in the recursive scheme is replaced by a compound statement which (1) allocates $p + 2$ locations for the function's data area, (2) saves the current active area index AA in the first location of the new data area, (3) moves the return label and the values of the actual parameters into the new data area, (4) assigns the undefined value Ω to all the other variables used in the function, (5) puts into the global variable AA the index of the new area, and (6) jumps to the first statement of f to begin executing.

Within each function ⟨body⟩ a reference to V_i is translated into a reference to $A[\mathrm{AA} + i]$, since the variables for the function are in the data area defined

by AA. Each HALT(V_i) in the original function definition is correspondingly replaced by

$$\text{BEGIN } RV \leftarrow A[\text{AA} + i]; \text{GOTO } A[\text{AA} + (p + 1)] \text{ END}$$

where RV is a global variable. Thus, when executing, the value of the function is stored in RV and control transfers to the statement defined by the label in the last location of the data area, which will be the statement following the original jump to the function. At this return point, the following statements will reinitialize the calling function's data area and store the value in RV in the appropriate variable:

$$\text{AA} \leftarrow A[\text{AA}]; v \leftarrow RV$$

This outline should be enough to convince the reader; the detailed construction is given in the Appendix.

(6.1) LEMMA. *Given a scheme in* P_R, *one can construct an equivalent scheme in* P_{AL}.

(6.2) *Example.* Let $S \in P_R$ be

```
(X, Y): IF P(X) THEN Y = f(X);
        HALT(Y);

f(X):   IF Q(X) THEN V = X
        ELSE BEGIN V = g(x); V = f(V) END;
        HALT(V)
```

Then S' is

```
(X, Y): TOP ← 0; AA ← 0;

        IF P(X) THEN

          BEGIN TOP ← TOP + 1; A[TOP] ← AA;                ⎡Y = f(x)
                A[TOP + 3] ← RL1;                          ⎢
                A[TOP + 1] ← X; A[TOP + 2] ← OMEGA;        ⎢
                AA ← TOP; TOP ← TOP + 3;                   ⎢
                GOTO Lf;                                   ⎢
            RL1: AA ← A[AA]; Y ← RV                        ⎢
          END;                                             ⎣
        HALT(Y);
```

```
Lf: IF Q(A[AA + 1]) THEN A[AA + 2] ← A[AA + 1]            [V = X
    ELSE BEGIN A[AA + 2] ← g(A[AA + 1]);                  [V = g(X)
          BEGIN TOP ← TOP + 1; A[TOP] ← AA;               ┌V = f(V)
                A[TOP + 3] ← RL2;                         │
                A[TOP + 1] ← A[AA + 1];                   │
                A[TOP + 2] ← OMEGA;                       │
                AA ← TOP; TOP ← TOP + 3;                  │
                GOTO Lf;                                  │
           RL2: AA ← A[AA]; A[AA + 2] ← RV                │
          END                                             │
         END;                                             └
    BEGIN RV ← A[AA + 2]; GOTO A[AA + 3] END              [HALT(V)
```

(6.3) THEOREM. *Given* $S \in \mathrm{P_R}$, *an equivalent scheme* S_3 *exists in* $\mathrm{P_A}$.

Proof. Construct (by Lemma (6.1)) a scheme $S_1 \in \mathrm{P_{AL}}$ equivalent to S. Construct a locator $S_2 \in \mathrm{P}$ for S (by Theorem (5.6)). Since S_1 is equivalent to S, S_2 is *also* a locator for S_1. Since S_2 is a locator for S_1, by Theorem (5.5) there is a scheme $S_3 \in \mathrm{P_A}$ equivalent to S_1, and thus equivalent to S. Q.E.D.

The generation of a $\mathrm{P_A}$ scheme from a $\mathrm{P_R}$ scheme was an effective construction. We now show that there exists a $\mathrm{P_A}$ scheme for which no equivalent $\mathrm{P_R}$ scheme exists, which indicates that the array mechanism is more powerful than recursion!

In [5] Paterson introduced a form of parallelism using a new operator /OR/.

(6.4) DEFINITION. $P(x)$ /OR/ $Q(x)$ has the value

(a) true if and only if either $P(x)$ or $Q(x)$ or both are defined and true;

(b) false if and only if both $P(x)$ and $Q(x)$ are false;

(c) undefined otherwise.

(6.5) *Example.*

```
   (X): IF Q(X) THEN V ← F1(X) ELSE V ← X; HALT(V)
Q(X): IF P(X) THEN V1 ← true
                 ELSE V1 ← Q(L(X)) /OR/ Q(R(X));
                 HALT(V1)
```

In [5] it was proved that the scheme *leaftest* of Example (6.5) is not $\mathrm{P_R}$ computable.

(6.6) THEOREM. $\mathrm{P_R} < \mathrm{P_A}$.

Proof. We show that Example (6.5) can be computed in $\mathrm{P_A}$. Suppose we write $L(R(x))$, $L(L(x))$, $R(L(x))$ as $LR(x)$, $LL(x)$, etc. Then (6.5) halts if and only if there exists a string $\alpha \in \{L, R\}^*$ such that $P(\alpha(x)) =$ true. The idea is to use an array to describe a breadth-first search of the binary tree rooted at x (with $L(x)$, $R(x)$ as branches for any x).

The P_A scheme is simply

```
(X): TOP ← 0; CURRENT ← 0; A[TOP] ← X;
     WHILE ¬P(A[CURRENT]) DO
           BEGIN TOP ← TOP + 1; A[TOP] ← L(A[CURRENT]);
                 TOP ← TOP + 1; A[TOP] ← R(A[CURRENT]);
                 CURRENT ← CURRENT+1;
           END;
     V ← F1(X); HALT(V)
```

7. The relation between P_{pds} and P_L. Let us write $P_{(m,n)}$ to indicate a P_{pdsM} scheme which uses m pushdown stores and n special markers. In this section, we give constructions which show that

$$P_{(n,m)} \equiv P_{(2,n+m+1)} \equiv P_{(2,1)} \quad \text{for } n \geqq 2,\ m \geqq 1;$$

$$P_R \leqq P_{(1,0)}.$$

(7.1) LEMMA. $P_{(n,m)} \leqq P_{(2,n+m+1)}$.

Outline of proof. Given a scheme S in $P_{(n,m)}$ which uses pds's $PD_1, \cdots, PD_n$, and markers $M_1, \cdots, M_m$, reserve new markers $A_0, A_1, \cdots, A_n$.

When S references a pds PD_i, the corresponding scheme S' in $P_{(2,n+m+1)}$ references its main pds PD.

A ⟨pushdown⟩ statement $PD_i \leftarrow v$ in S is executed in S' by

$$\text{BEGIN } PD \leftarrow v; \quad PD \leftarrow A_i \text{ END}$$

Thus the marker A_i on the PD indicates that a value in PD_i is on top of the stack. A_0 is put on the bottom of PD to indicate when it is empty.

A ⟨popup⟩ statement $v \leftarrow PD_i$ is simulated in S' by a compound statement which searches down the main pds PD, looking for a marker A_i. The PD element just below it is then the value needed to store in v. This search down PD requires a second, auxiliary pds to save elements of PD. After executing the pop, the main pds PD is restored using the auxiliary pds.

(7.2) LEMMA. $P_{(2,m)} \leqq P_{(2,1)}$ *for* $m \geqq 0$.

Outline of proof. The m markers used in a $(2, m)$ scheme S can be encoded in binary and decoded using the finite control of the program scheme. For example, suppose $m \geqq 2$ and $M_1, \cdots, M_m$ are the markers being used. Then let the equivalent scheme $S' \in P_{(2,1)}$ use a marker M, and represent a marker M_i in S by

$$M \underbrace{\Omega \cdots \Omega}_{\substack{i-1\\ \text{times}}} \quad M \underbrace{\Omega \cdots \Omega}_{\substack{m-1\\ \text{times}}}$$

in the S' scheme. See the Appendix for details.

(7.3) THEOREM. $P_{(n,m)} \equiv P_{(2,1)}$ *for* $n \geqq 2$, $m \geqq 1$.

Proof. Apply Lemmas (7.1) and (7.2).

(7.4) LEMMA. *For any scheme* $S \in P_R$ *there exists an equivalent scheme in* P_{pdsL} *which uses only* one *pds.*

Proof. We leave the proof to the Appendix. The idea is the same as the simulation of recursion using an array.

(7.5) THEOREM. $P_R \leqq P_{(1,0)}$.

Proof. By Lemma (7.4) a scheme S in P_R has an equivalent scheme S_1 in P_{pdsL} which uses one pds. If S_1 uses predicates $P_1, \cdots, P_n$, we can certainly construct P_i-simulators for S_1, which will all be in $P_{(1,0)}$ and all use the same stack. The proof then hinges on finding a locator for S_1 and thus S, the locator being in $P_{(1,0)}$. Theorem (5.6) gives us the locator.

8. The equivalence of P_{AL}, P_{AM}, P_{pdsM}, and P_{Ae}. In this section we first prove that P_{AL} and P_{AM} are equivalent. Lemmas (8.2) and (8.3) then establish the equivalence of P_{AM} and P_{pdsM}. The equivalence of P_{pdsM} and P_{pdsL} could also be established; we shall not do that here.

Generally speaking, we regard P_{AM} as the superior class of schemes. Algorithms using pushdown stores often tend to be cumbersome, while arrays are more natural to work with and can be easily used to simulate pushdown stores. P is probably a better class of schemes to use than P_L. In general, programs using few "GOTO's" are clearer, easier to read, and easier to debug, and the feature of labels as variables just tends to increase the number of branches.

(8.1) THEOREM. $P_{AL} \equiv P_{AM}$.

Proof. We can show that given any scheme $S \in P_{AL}$ we can construct an equivalent scheme S' in P_{AM}, and vice versa.

Suppose the scheme S uses n labels $L_1, \cdots, L_n$. Then the scheme S' will use corresponding markers $M_1, \cdots, M_n$. Replace each statement $v \leftarrow L_i$ in S by $v \leftarrow M_i$. Replace each statement GOTO v by

BEGIN IF $v = M_1$ THEN GOTO L_1
ELSE IF $v = M_2$ THEN GOTO L_2
ELSE $\cdots$
IF $v = M_n$ THEN GOTO L_n
END

The result is clearly an equivalent scheme $S' \in P_{AM}$.

Now suppose we are given a scheme $S' \in P_{AM}$ which uses n markers $M_1, M_2, \cdots, M_n$. Consider these to be labels and append to the scheme the sequence of statements

M_1: GOTO V_1;
M_2: GOTO V_2;
$\vdots$
M_n: GOTO V_n;

where the V_i are new, unique simple variables. For each statement

$$\text{IF } v = M_i \text{ THEN } S_1 \text{ ELSE } S_2$$

generate three new labels LT, LF, and DONE, and replace the statement by

```
BEGIN V1 ← LF; V2 ← LF; ··· ; Vn ← LF; Vi ← LT;
    GOTO v;
    LF: S2; GOTO DONE;
    LT: S1;
  DONE:
END
```

Inspection of this sequence shows that, when executed, S_1 is executed if and only if $v = M_i$; otherwise S_2 is executed. This is the desired property.

(8.2) LEMMA. $\mathrm{P_{pdsM}} \leqq \mathrm{P_{AM}}$.

Proof. Given a scheme S in $\mathrm{P_{pdsM}}$ we construct an equivalent scheme S' in $\mathrm{P_{AM}}$. Each pushdown store PD in S is simulated by an array A in S'. A counter CA indicates where the top of A is. Each element of PD is described by two consecutive locations in A, with the format

actual contents of *PD* element
marker

The second location indicates whether the PD is empty or not. For this purpose, the $\mathrm{P_A}$ scheme uses two extra markers M_0 and M_1. An empty stack is represented as in Fig. 4a while the PD of Fig. 4b would be described by the array A of Fig. 4c.

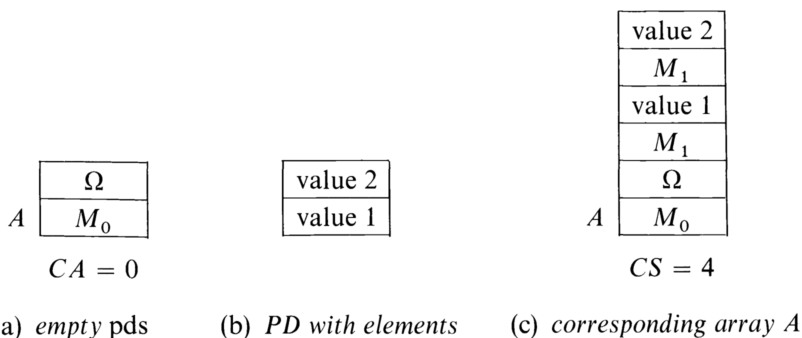


FIG. 4. *Simulating a pushdown store by an array*

One can easily see how to construct S' equivalent to S. For each PD used in S, perform the following three steps:

(1) Generate a unique array name A and a simple variable CA, and insert in the beginning of S the statements to initialize the array:

$$CA \leftarrow 0; \quad A[0] \leftarrow M_0; \quad A[1] \leftarrow OMEGA;$$

(2) Replace each statement $PD \leftarrow v$ by

BEGIN $CA \leftarrow CA + 2$; $A[CA] \leftarrow M_1$; $A[CA + 1] \leftarrow v$ END

(3) Replace each statement $v \leftarrow PD$ by

BEGIN IF $A[CA] = M_1$

THEN BEGIN $v \leftarrow A[CA + 1]$; $CA \leftarrow CA \dot{-} 2$ END

END

(By virtue of Theorem (4.3) we can use the function $\dot{-}1$ and thus also $\dot{-}2$.)

(8.3) LEMMA. $P_{AM} \leqq P_{pdsM}$.

Proof. Given a scheme S in P_{AM}, we construct an equivalent scheme S' in P_{pdsM}. In S', each value in D and each marker M used in S will be represented by itself. Let $*$ be a new, unique marker. Then each value $i \in \mathbb{N}$ computed in S will be represented in S' by a sequence of $i + 1$ $*$'s. Since any variable may contain a value i, *except* when that variable is used as an argument to a basic function or predicate, every variable in S will have to be represented in S' by a pushdown store! (If used as an argument, we can use either Ω or just a single asterisk in its place.)

Our task of transforming S into S' will be performed in four steps. First we create S_1 equivalent to S. In S_1, all the arguments to functions, predicates and HALT operations are newly introduced simple variables which, because of the way they are introduced, need not be treated as pds's. Second, we create S_2 equivalent to S_1 solely in order to simplify the transformation from a simple variable to a pds. For example, a statement $A[v] \leftarrow A[w]$ will be replaced by

$$\text{BEGIN } v_0 \leftarrow A[w]; \quad A[v] \leftarrow v_0 \text{ END}$$

where v_0 is a new simple variable.

Third, in step (3) we finally consider simple variables as pds's and show how each statement must be changed accordingly. For example, we must change $v \leftarrow w$ (v, w simple variables) into a compound statement which moves pds w into pds v.

Finally, we show how arrays are represented as pds's, and show how to transform the only two statement types which use arrays:

$$A[w] \leftarrow v \quad \text{and} \quad w \leftarrow A[v] \quad (\text{where } v, w \text{ are simple variables}).$$

In general, each array A is represented by a pds PA. In PA, the actual values of A are separated by a comma ",". Thus the array A in Fig. 5a is represented by the pds PA of Fig. 5b. To reference a value $A[i]$, in S' we first move all of PA onto an auxiliary stack $PDT1$. We then use the representation of i as $i + 1$ asterisks to pop i commas off $PDT1$, leaving the value $A[i]$ at the top of $PDT1$.

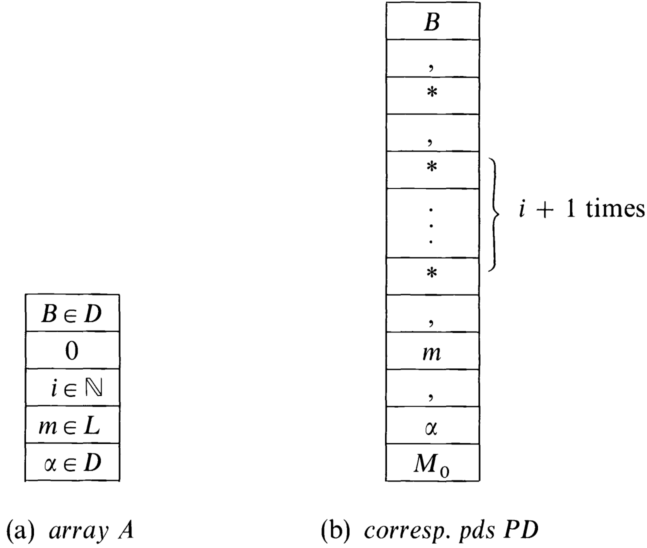
Details are left to the Appendix.

(8.4) THEOREM. $P_{AL} \equiv P_{pdsM}$.

Proof. Apply Lemmas (8.2) and (8.3).

(8.5) LEMMA. $P_{AM} \leqq P_{Ae}$.

Proof. In P_{Ae} one has the ability to test whether two subscript values are equal (see Definition (4.5)). Suppose we have a scheme S in P_A which uses markers $M_1, \cdots, M_n$. We translate S into an equivalent P_{Ae} scheme S' as follows.



(a) *array A* (b) *corresp. pds PD*

FIG. 5. *Simulation of array* A

We use $1, 2, \cdots, n \in \mathbb{N}$ in place of the n markers. However, we must be able to tell whether a value is actually a subscript or a marker. To this end, every simple v (except *OMEGA*) in S is represented by two variables v, v_0, while every array A is represented by arrays A and A_0. (Below, we use w_0 to represent a simple variable w_0 or subscripted variable $w_0[v]$.) If during the execution of S we have $w = M_i$, then $w ⊜ i \in \mathbb{N}$ while $w_0 = 1$. Otherwise, w contains a normal value or subscript while $w_0 = 0$ or w_0 contains a value in D.

Note that for both simple and subscripted variables w, initially w_0 and w both contain Ω, so initially everything is correct. Change S as follows:

1. Change every assignment $v \leftarrow w$ to

$$\text{BEGIN } v \leftarrow w;\, v_0 \leftarrow w_0 \text{ END}$$

2. Change every assignment $v \leftarrow f(\cdots)$ to

$$\text{BEGIN } v \leftarrow f(\cdots);\, v_0 \leftarrow 0 \text{ END}$$

3. Change every assignment $v \leftarrow 0$ to

$$\text{BEGIN } v \leftarrow 0;\, v_0 \leftarrow 0 \text{ END}$$

4. Change every assignment $v \leftarrow w + 1$ to

$$\text{IF } w_0 \ominus 0 \text{ THEN BEGIN } v \leftarrow w + 1; v_0 \leftarrow 0 \text{ END}$$
$$\text{ELSE } v \leftarrow 1$$

5. Change every assignment $v \leftarrow M_i$ to

$$\text{BEGIN } v \leftarrow 1; v_0 \leftarrow 1 \text{ END}$$

6. Change every test

$$\text{IF } v = M_i \text{ THEN } S_1 \text{ ELSE } S_2$$

to

```
BEGIN IF v0 ⊖ 1 and v ⊖ i
          THEN S1 ELSE S2
END
```

(8.6) LEMMA. $P_{Ae} \leqq P_{AM}$.

Proof. Suppose S is in P_{Ae}. We show how to construct an equivalent scheme S' in P_{AM}. All we need do is show how to translate a statement

(8.7) If $w \ominus v$ THEN S_1 ELSE S_2

where the result of the test is defined in (4.5). First of all, use a new array M and two markers M_0 and M_1. Put the statement $M[0] \leftarrow M_0$ in the beginning of the scheme S, and change each statement $v \leftarrow w + 1$ to

$$\text{BEGIN } v \leftarrow w + 1; M[v] \leftarrow M_1 \text{ END}$$

This assures us that during execution, if any value $i > 0$ is calculated, then $M[0] = M_0$ and $M[1] = M_1, \cdots, M[i] = M_1$. Let W, V be new variables, and change each statement (8.7) in S to

```
BEGIN V ← v; W ← w;
          WHILE M[V] = M1 and M[W] = M1 DO
                    BEGIN V ← V ∸ 1; W ← W ∸ 1 END;
          IF V = M0 and W = M0
          THEN S1 ELSE S2
END
```

(8.8) THEOREM. $P_{Ae} \equiv P_{AM}$.

Proof. Apply Lemmas (8.5) and (8.6).

9. The noneffective equivalence of P_A and P_{AM}. Given any scheme S (in P_A or P_{AM}) we can construct an equivalent "completely labeled" scheme, in which all statements (in the syntactic sense, all statements $\langle S \rangle$; see § 2.1) are labeled. For example, we do this in Fig. 6b for the scheme in Fig. 6a. We can then assume that every scheme is completely labeled.

```
(V, W): V ← F1(V, W);            (V, W): L2: V ← F1(V, W);
   L1: IF P1(V) THEN                     L1: IF P1(V) THEN
           GOTO L7                                L3: GOTO L7
       ELSE BEGIN                             ELSE L4: BEGIN
               V ← F2(V);                                  L5: V ← F2(V);
               GOTO L1                                     L6: GOTO L1
               END;                                        END;
   L7: HALT(V)                           L7: HALT(V)
           (a)                                    (b)
```

FIG. 6. *Completely "labeling" a scheme*

(9.1) DEFINITION. The *behavior of* (the execution of) *a scheme* (under some interpretation) is the sequence of labels of the statements executed, in the order they *begin* executing.

Thus, suppose scheme 6b is executed under some interpretation, and that $P(V)$ is false the first time it is evaluated and true the second time. Then the behavior is

$$L2, L1, L4, L5, L6, L1, L3, L7.$$

Of importance to us is the "autonomous" behavior of a scheme.

(9.2) DEFINITION. Let S be a (completely labeled) scheme which uses n predicates $P_1, \cdots, P_n$. Let $v = (v_1, \cdots, v_n)$ be a vector of n values from the set {true, false}. Then the *v-autonomous behavior* of S is the behavior of S assuming that

$$P_1 \equiv v_1, \quad P_2 \equiv v_2, \cdots, P_n \equiv v_n.$$

Note that we have defined the autonomous behavior independently of both the input values of an execution and the basic functions. This is of course not obvious and must be proved.

(9.3) LEMMA. *Let S be a scheme in* $\mathrm{P_{AM}}$ *which uses n predicates $P_1, \cdots, P_n$, and suppose that $P_1, \cdots, P_n$ are all constant. Then the behavior of S is independent of the domain D, the interpretation of the basic functions, and the input values.*

Proof. Suppose we begin executing S under two different interpretations, or under the same interpretation with different input values. Suppose we get two sequences of labels

$$L_1, L_2, \cdots, L_k, L_{k+1}, \cdots,$$
$$L_1, L_2, \cdots, L_k, L'_{k+1}, \cdots$$

such that they first differ at the $(k + 1)$st label. Clearly, L_k must label a conditional

$$\text{IF } P(\ ) \text{ THEN } S_1 \text{ ELSE } S_2$$

or

$$\text{IF } v = m \text{ THEN } S_1 \text{ ELSE } S_2$$

Since all predicates are constant, L_k must label the latter conditional. We can show a contradiction if we can show that,

(9.4) just before execution of any step, a simple or subscripted variable contains either

(1) the *same* marker m under *both* executions, or

(2) the *same* subscript value in $\mathbb{N}$ under *both* executions, or

(3) values in the domain D (possibly different).

We prove this by induction on the number of statements whose execution has begun. For $i = 1$, just before the statement labeled L_1 begins executing, all variables contain values in the domain D. Suppose (9.4) is true for $n = 1, 2, \cdots, i < k$, and suppose we begin executing the statement labeled L_i. The only statements which can change a value (and thus possibly contradict property (9.4)) have the forms

(1) $v \leftarrow w$ (v and w simple or subscripted),

(2) $v \leftarrow m$,

(3) $v \leftarrow 0$,

(4) $v \leftarrow F(\cdots)$,

(5) $v \leftarrow w + 1$.

Since under either execution, up to step k the *same* statements are executed, after execution of L_i, (9.4) will still hold regardless of which of (1)–(5) is performed.

We now can begin our discussion of the nonconstructability of $S' \in \mathrm{P_A}$ equivalent to S in $\mathrm{P_{AM}}$.

(9.5) THEOREM. *Let v be a vector of n values from the set* {true, false}. *Let S be a scheme in $\mathrm{P_A}$ using n predicates. Then it is decidable whether the v-autonomous behavior of S is finite or infinite.*

Proof. Since the v-autonomous behavior is independent of the interpretation, we can begin constructing the behavior without giving an interpretation. Let S have p labels (and thus p statements $\langle S \rangle$). Execute and record the behavior of S until either it halts or until $p + 1$ labels have been recorded, whichever comes first. If it halts before $p + 1$ labels have been recorded, the v-autonomous behavior is finite. If $p + 1$ labels are recorded, then one label is recorded more than once. Thus a statement is executed more than once, which means there is a loop. This looping process cannot stop, since all predicates are constant and the only way to change the sequence is to execute a test

$$\text{IF } P(\cdots) \text{ THEN } L1: S_1 \text{ ELSE } L2: S_2$$

where $P(\cdots)$ yields a different value.

(9.6) LEMMA. *It is undecidable whether the v-autonomous behavior of a scheme S in $\mathrm{P_{AM}}$ is finite or not.*

Proof. Given a Turing machine M using a single tape, infinite to the right, and a 0, 1 alphabet, we can construct a program scheme S_M in $\mathrm{P_A}$ which has finite autonomous behavior if and only if M halts on blank tape (all 0's). This reduces the T.M. halting problem to the finiteness problem for autonomous behavior, and since the halting problem is undecidable, so is the finiteness problem.

The T.M. tape is represented by an array T, with a counter I (initially 0) indicating the position of the read-write head. Two marks $\underset{\sim}{0}$ and $\underset{\sim}{1}$ are used. S_M executes $T[I] \leftarrow m$ when M writes the mark m in the ith tape cell. When M reads the cell, the scheme S_M asks "If $T[I] = m \cdots$." When M moves right, S_M executes

$I \leftarrow I + 1$, and when M moves left, by virtue of Theorem (4.3), S_M can execute $I \leftarrow I \dot{-} 1$.

We leave the details of the construction of S_M to the reader.

(9.7) THEOREM. *One cannot effectively construct a scheme S' in P_A equivalent to a scheme S in P_{AM}.*

Proof. If $S' \in P_A$ and $S \in P_{AM}$ are equivalent, then the v-autonomous behavior of S is finite if and only if that of S' is finite. If S' could be effectively constructed from S, then applying the decision procedure for finiteness of the autonomous behavior of S' would contradict Lemma (9.6). Q.E.D.

Our last step is to show the existence of (but not an effective construction for) a scheme S' in P_A equivalent to a scheme S in P_{AM}. Theorem (5.5) indicates we need only prove the existence of a locator in P_A for S (see Definition (5.2)). Assuming S uses n predicates $P_1, \cdots, P_n$, this locator has the form shown in Fig. 7, where the arguments of the predicates are all *OMEGA*. (Thus if P_1 has two arguments, the cell in the upper box is P_1(*OMEGA*,*OMEGA*).) We should point out that it is not necessary to use Ω as the argument, it just simplifies the construction. Any of the input values could have been used.

Let us suppose that an interpretation is given to the domain D, predicates P_i and functions F_i, and that the locator with the above form is executed with some input values. Then the locator tests *all* the predicates P_i and executes one of the statements S_j, depending on the outcome of the tests. There are 2^n different S_j, corresponding to the 2^n possible v-autonomous behaviors of the scheme S. For example, S_1 corresponds to $v \equiv$ (true, $\cdots$, true).

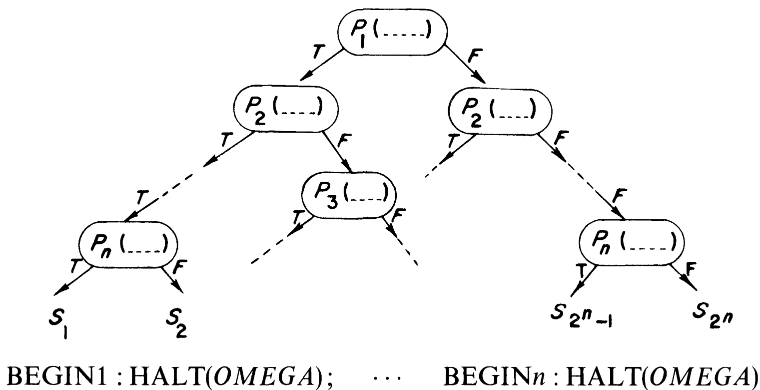
Let us assume that the execution mentioned finds $P_1(\cdots) = \cdots = P_n(\cdots) =$ true. Then $v =$ (true, $\cdots$, true). S_1 is executed, and must perform the following:

(9.8) S_1 must "simulate" the v-autonomous behavior of scheme S until either:

(i) it halts and outputs the same result that S would, or

(ii) a two-valued predicate P_i is evaluated. At this point, $RT_1, \cdots, RT_n$ are initialized to Ω, $RF_1, \cdots, RF_n$ are set to the argument list which yielded $P_i(\cdots) =$ false, and control is transferred to BEGINi.

Thus, since P_i(*OMEGA*) = true for $i = 1, \cdots, n$, S_1 assumes that all predicates are constant (and executes essentially as S would) until it finds out differently.



BEGIN1 : HALT(*OMEGA*); $\cdots$ BEGINn : HALT(*OMEGA*)

FIG. 7

We now show the construction of S_1 only for v = (true, $\cdots$, true). The construction of the other S_j is similar. There are two cases, depending on whether the v-autonomous behavior of S is finite or infinite.

(9.9) *Construction of S_1 assuming the v-autonomous behavior of S is finite*:

$$L_1, L_2, \cdots, L_k.$$

We assume without loss of generality that all predicates have rank 1. The above sequence indicates the order in which the statements begin executing. In effect, we have unraveled all loops. Let us write this sequence as

```
        BEGIN L1:  <S>1
              L2:  <S>2
(9.10)           :
              Lk:  <S>k
        END
```

where $\langle S\rangle_j$ is the statement labeled L_j.

For example, the v = (true) behavior of the scheme in Fig. 6b is $L2$, $L1$, $L3$, $L7$, so we get the sequence

```
BEGIN L2: V ← F1(V, W);
      L1: IF P1(V) THEN L3: GOTO L7 ELSE L4: BEGIN ··· END
      L3: GOTO L7
      L7: HALT(V)
END
```

We now show how to translate (9.10) into a P_A statement with the required properties. As an illustration, we shall translate the above into

```
BEGIN V ← F1(V, W);
      IF P1(V) THEN
               ELSE BEGIN RT ← OMEGA; RF ← V;
                          GOTO BEGIN1
                    END;
      HALT(V);
END
```

Note that the unnecessary GOTO $L7$ was deleted and that the conditional statement involving $P1(V)$ was changed as necessary.

To translate (9.10), we first categorize the $\langle S\rangle_j$ into the types

(1) $v \leftarrow w, v \leftarrow 0, v \leftarrow w + 1, v \leftarrow F(\cdots)$, HALT($v$), "empty"

(2) IF $P_i(v)$ THEN S ELSE S

(3) IF $v = m$ THEN S ELSE S, BEGIN $\langle S$-list$\rangle$ END, GOTO l

(4) $v \leftarrow m$

Next, for $j = 1, \cdots, k$ execute one of the following, depending on the type of $\langle S\rangle_j$:

Type

(1) replace L_j: $\langle S\rangle_j$ by "$\langle S\rangle_j$;" (the simple statement remains);

(2) replace L_j: $\langle S\rangle_j$ by

IF $P_i(v)$ THEN ELSE BEGIN $RT \leftarrow OMEGA$: $RF \leftarrow v$;
GOTO BEGINi
END;

(3) delete L_j: $\langle S\rangle_j$ (it is not needed, since the statement to execute next is on the next line);

(4) replace L_j: $\langle S\rangle_j$ by $v \leftarrow OMEGA$.

We assert that this process yields a statement satisfying (9.8), assuming that $P_i(OMEGA) = \text{true}$ for all i. Note that if $P_i(OMEGA) \equiv \text{true}$, executing S_1 executes the "simple" statements of the v-autonomous behavior of S, and yields the necessary value. If a predicate P is detected which is not identically true, we jump out as required.

(9.11) *Construction of S_1 in case of infinite v-autonomous behavior.*

The aim of S_1 is either to find a predicate which changes value, or to HALT with the same output as S. We cannot "unravel" the statements executed, as in the finite case. However, in the case of infinite *v-autonomous* behavior, we need not worry about halting, since S does not. We need only guarantee that S_1 find a binary-valued predicate if it exists (regardless of whether S finds one). This is accomplished by systematically generating and testing *all* possible values that *might* be generated by S.

More precisely, consider the set of all *function expressions* of S defined as follows. Let S have as inputs $x_1, \cdots, x_n$, rank p functions $f_1^p, \cdots, f_{k_p}^p$ for $p = 1, \cdots, m$ and rank q predicates $p_1^q, \cdots, p_{l_q}^q$, $q = 1, \cdots, r$. The set of function expressions is generated by the following productions:

$$E \rightarrow x_1 \mid x_2 \mid \cdots \mid x_n \mid \Omega$$
$$E \rightarrow f_1^1(E) \mid \cdots \mid f_{k_1}^1(E)$$
$$\vdots$$
$$E \rightarrow f_1^m(E, \cdots, E) \mid \cdots \mid f_{k_m}^m(E, \cdots, E)$$

This set of expressions, $\{E\}$, contains all possible values that can be tested in the predicates.

S_1 searches the entire set $\{E\}$. The form of S_1 depends on m, the maximum rank of the functions and predicates; and S_1 for rank $m + 1$ is constructed inductively from S_1 for rank m. To indicate the general nature of S_1, let us first present S_1 for the case of one predicate P_1 of rank 1, one function f of rank 2, and a single initial value v_0. We write a program to generate and test values in the following order:

$$v_1 = f(v_0, v_0),$$

$$v_2 = f(v_1, v_0), \quad v_3 = f(v_0, v_1),$$
$$v_4 = f(v_2, v_0), \quad v_5 = f(v_1, v_1), \quad v_6 = f(v_0, v_2),$$
$$v_7 = f(v_3, v_0), \quad v_8 = f(v_2, v_1), \cdots$$
$$\vdots$$

This is illustrated by Fig. 8, where the row (column) specifies the value used for the first (second) argument, and the numbers at the grid points indicate the order in which the new values are generated.

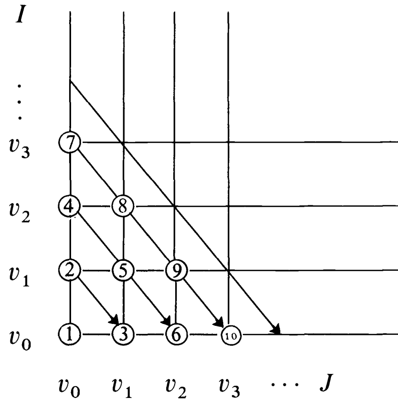


FIG. 8

An obvious statement for S_1 would be the following, where A contains the array of values, K indicates how many values are currently generated in A, and I and J are counters for the subscript of the first and second argument values. The only problem is the statement which tests I; it is not allowed in P_A.

```
BEGIN K ← 0; I ← 0; J ← 0; A[0] ← v0;
LOOP: IF P1(A[K]) THEN                              [Check new value in P1
      ELSE BEGIN RT ← OMEGA; RF ← A[K];             and jump out if
                 GOTO BEGIN1                        2-valued
           END;                                     predicate found.

      K ← K + 1; A[K] ← f(A[I], A[J]);              [Generate new value
                                                    using vI, vJ as arguments.

      J ← J + 1;                                    [J increases

      SAVE ← I; I ← I − 1;                          [Decrease I. If it does not
      IF I ⊜ SAVE THEN                              change, we are at end of
         BEGIN I ← J; J ← 0 END;                    diagonal (see Fig. 8)
                                                    and must begin new one.

      GOTO LOOP                                     [Test new value.
END
```

To get rid of this test on I, we use two extra arrays. $AD[I]$ is an array of "down" pointers; $AD[I] = I \dot{-} 1$ (except for $AD[0]$, which we explain in a moment). Thus, we replace the statement $I \leftarrow I \dot{-} 1$ by $I \leftarrow AD[I]$.

Secondly, we replace J by an array of subscripts AS. If $A[I]$ is used as the first argument to f, then $A[AS[I]]$ is used as the second argument. Note that the first time $A[I]$ is used as the first argument, the second argument is $A[0]$. Thus, when we generate a new value $A[K]$, we set $AS[K] = 0$. Moreover, the second (third, etc.) time $A[I]$ is used as the first argument, the second argument is $A[1]$ ($A[2]$, etc.). This means that after using $A[I]$ as the first argument, we should increase $AS[I]$ by 1. This leaves us with the following program which should be clear except for (i) the statement $AD[0] \leftarrow AS[0]$ and (ii) how we begin a new diagonal on the diagram in Fig. 8.

```
BEGIN K ← 0; I ← 0; A[0] ← v0;                      [Initialize
      AS[0] ← 0; AD[0] ← 0;

LOOP: IF P1 (A[K]) THEN                             [Check new value
      ELSE BEGIN RT ← OMEGA; RF ← A[K];
                 GOTO BEGIN1
           END;

      K ← K + 1; A[K] ← f(A[I], A[AS[I]]);          [Generate new value
      AS[K] ← 0; AD[K] ← K ∸ 1;
                                                    [Increase subscript
  L1: AS[I] ← AS[I] + 1;                            [for next time

  L2: AD[0] ← AS[0]; I ← AD[I];                     [Fix 0-down pointer
      GOTO LOOP

END
```

As long as we are decreasing I (going along a diagonal in Fig. 8), the program is easy to follow. Now, suppose $I = 0$ and we are evaluating $f(A[0], A[AS[0]])$. Then, according to Fig. 8, the next value to calculate is $f(A[AS[0] + 1], A[0])$, and I should be changed to $AS[0] + 1$. The statement labeled $L1$ increases $AS[0]$ as necessary, and line $L2$ puts the correct value in I! The statement $AD[0] \leftarrow AS[0]$ has essentially no effect unless $I = 0$, in which case it is used to help put the right value in I.

We leave the details of the construction of a general S_1, assuming infinite v-autonomous behavior and functions of rank greater than 2, to the Appendix. A more detailed explanation of the process can be found in [8].

To summarize: The locator in P_A for $S \in P_{AM}$ is given in Fig. 7. Each of the statements S_j depends on the corresponding v-autonomous behavior of the scheme S. If finite, (9.9) shows how to construct S_j. If infinite, the construction of S_j is outlined in (9.11) and the Appendix. If the behavior of S is infinite, the corresponding statement S_j just generates and tests in the predicates all possible values, until a predicate is found to be not constant. Remember this whole construction is noneffective, because we cannot decide whether the v-autonomous behavior is finite or infinite. This discussion, together with the fact that $P_A \leqq P_{AM}$, yields the following theorem.

(9.12) THEOREM. $P_A \equiv P_{AM}$.

10. Effective functionals on total interpretations. In [7] Strong proposed the class of effective functionals as a universal class. We outline here the equivalence of this class EF (assuming all predicates and functions are total) with P_{Ae}. While doing so, we also conclude that $P_{(1,0)\mathbb{N}}$ is universal (this latter result is actually due to Strong [private communication]), where $P_{(0,1)\mathbb{N}}$ is the class of schemes using one pushdown store and integer arithmetic. Hence we show that $P_{Ae} \leqq EF \leqq P_{(1,0)} \leqq P_{Ae}$.

Please remember, we do not give formal proofs, but very brief outlines of the constructions.

Let $X_1, \cdots, X_n$ be input variables, $F_1, \cdots, F_m$ be (total) basic function names, and $P_1, \cdots, P_l$ be (total) predicate names. An *expression* e (or e_i) has one of the forms

(1) X_i (the ith input variable),

(2) $F_j(e_1, \cdots, e_{RF_j})$.

A *proposition* has one of the forms

(1) $P_j(e_1, \cdots, e_{RP_j})$,

(2) $\neg P_j(e_1, \cdots, e_{RP_j})$ (complement of $P_j(\cdots)$).

(10.1) DEFINITION. A *computation* is a finite sequence of expressions and propositions, the last of which must be an expression.

(10.2) *Example.* $\langle P_1(X_1), \neg P_1(F_1(X_1)), F_1(X_1)\rangle$.

Given an interpretation of the predicates and functions and some input values, to *evaluate* a computation, we evaluate its expressions and propositions from left to right until either

(1) a proposition yields the value false, in which case the computation has *no* value; or

(2) the last expression is evaluated, in which case the value of the computation is the value of this last expression.

Computation (10.2) yields a value, the value of $F_1(X_1)$, if and only if both $P_1(X_1)$ and $\neg P_1(F_1(X_1))$ are true.

(10.3) DEFINITION. A (total) *effective functional* in the class EF is a recursively enumerable set of computations in which, given any interpretation of the predicates and functions and input values, if two or more computations yield a value, they yield the same value. This value is of course the result of the execution.

As an example, consider this completely labeled scheme in P:

$$
\begin{array}{ll}
 & (X)\colon L1\colon Y \leftarrow X; \\
(10.4) & \quad L2\colon \text{IF } P(Y) \text{ THEN} \\
 & \quad\quad L3\colon \text{BEGIN } L4\colon Y \leftarrow B(Y);\ L5\colon \text{GOTO } L2 \text{ END}; \\
 & \quad L6\colon \text{HALT}(Y)
\end{array}
$$

This scheme can be transformed into the effective functional

$$
(10.5)\quad \left\{
\begin{array}{ll}
\langle \neg P(X), X\rangle & \text{computation 1} \\
\langle P(X), \neg P(B(X)), B(X)\rangle & \text{computation 2} \\
\langle P(X), \ P(B(X)), \neg P(B(B(X))), B(B(X))\rangle & \text{computation 3} \\
\vdots & \vdots
\end{array}
\right\}
$$

A quick look at how we get (10.5) from (10.4) will give us the idea for transforming any scheme in P_{Ae} into an effective functional. Note that any finite behavior yields a corresponding output value, so that the effective functional should have one computation for each different finite behavior of (10.4). Let us begin recording the possible partial behaviors of (10.4) of length 1, 2, 3, etc.:

length 1: $L1$
length 2: $L1, L2$
length 3: $L1, L2, L3$ and $L1, L2, L6$ (HALTS)
length 4: $L1, L2, L3, L4$
length 5: $L1, L2, L3, L4, L5$
length 6: $L1, L2, L3, L4, L5, L2$
length 7: $L1, L2, L3, L4, L5, L2, L3$ and
$L1, L2, L3, L4, L5, L2, L6$ (HALTS)

Any partial behavior whose last label labels a HALT corresponds to a computation. Let us determine the computation for the second behavior which HALTS: $L1, L2, L3, L4, L5, L2, L6$.

We look at the statements executed by this finite behavior, in order, and keep track of the (symbolic) values assigned to each variable. Statement $L1$ indicates that the value of Y is obtained by evaluating the expression "X." $L2$ labels a conditional, and since the one following it is $L3$, $P(Y)$ must be true. So we build the first proposition of the computation by substituting for Y the symbolic expression which yields its current value. The first proposition is "$P(X)$."

Executing $L4$: $Y \leftarrow B(Y)$ indicates that Y now contains the value resulting from evaluating "$B(X)$," since Y previously contained "X." After executing $L5$ we again come to the conditional labeled $L2$. This time $L2$ is followed by $L6$, so $P(Y)$ must be false. Hence, we put the second proposition into the computation: "$\neg P(B(X))$." Now we come to $L6$: HALT(Y). This results in the last and only expression in the computation, "$B(X)$." Hence we end up with computation 2 of the effective functional (10.5).

Given a finite behavior, then, we generate the computation by sequencing through the statements, keeping for each variable the symbolic expression which yields its value, i.e., executing under the free interpretation. For each conditional statement a proposition is put into the computation. The final HALT(v) results in the final expression, the current symbolic expression for v.

(10.6) ASSERTION. *Let S be a scheme in* P_{Ae}. *Then there exists an equivalent effective functional on total interpretations.*

We must give a recursive procedure for enumerating the computations. At least for a scheme in P, the above discussion shows how to enumerate the finite behaviors of the scheme, and from a finite behavior how to construct the computation.

What about schemes in P_{Ae}? Can we enumerate the finite behaviors? Can we construct valid computations for them? The problem is of course with statements like:

(10.7) L_1: IF $v \ominus w$ THEN $L2$: $\langle S \rangle_2$ ELSE $L3$: $\langle S \rangle_3$

(If the reader is unsure as to the meaning of $\ominus$, please read Definition (4.5) carefully now.)

As we build the partial behaviors, we record with each a list of variable-value pairs—the list of all simple and subscripted variables whose values are in $\mathbb{N}$. Thus, if the partial behavior ends with labels for the statements $V \leftarrow 0$, $A[V] \leftarrow V + 1$, $W \leftarrow A[V] + 1$, $A[W + 1] \leftarrow V$, $V \leftarrow F(X)$, the list will contain the pairs $(A_0, 1)$, $(W, 2)$, $(A_3, 0)$. Now let us see what happens when we process a partial behavior $L_1, \cdots, L_n$, where L_n labels a statement (10.7). We are to build from it all partial behaviors of length $n + 1$. Since we have a list of all variables with values in $\mathbb{N}$, we can determine at this point whether $v \ominus w$ or not. If so, we get the single new partial behavior $L_1, \cdots, L_n$, $L2$. If $v \not\ominus w$, we get the single new partial behavior $L_1, \cdots, L_n$, $L3$. But we do *not* add both new partial behaviors to the list.

Hence we see that, by keeping a list of variable-value pairs where the values are in $\mathbb{N}$, we can determine which of the two paths the finite behavior will take when processing a statement (10.7), *irrespective of the actual interpretation of predicates, functions, and input values*, i.e., in the free interpretation.

When building the computations from a finite behavior, no extra processing need be done with statements of the form (10.7).

(10.8) ASSERTION. *For every effective functional* $F \in \mathrm{EF}$ *there exists a scheme* S *in* $\mathrm{P}_{(1,0)\mathbb{N}}$.

Outline of construction. F is given by a recursive function $f(i)$ which, given an integer i, returns the ith computation in some form. S is given in Fig. 9. By using Minsky's two counter machine [9, p. 255], f can be implemented using integer arithmetic $(+, \dot{-}, =)$ and two simple variables, which $\mathrm{P}_{(1,0)\mathbb{N}}$ allows. Let us assume that the computation produced by f is coded in Polish postfix, to be interpreted. For example, input variables $X_1, \cdots, X_n$ could be represented by integers $\underset{\sim}{1}, \cdots, \underset{\sim}{n}$, function names $F_1, \cdots, F_m$ by $\underset{\sim}{n} + \underset{\sim}{1}, \cdots, \underset{\sim}{n} + \underset{\sim}{m}$, and so forth. Second, assume the whole computation is coded as a single integer, in one variable. Thus, if the scheme uses X_1, F_1, F_2, and predicates P_1 and P_2, the expression $F_1(X_1, F_2(X_1))$ in Polish notation would be X_1, X_1, F_2, F_1. In coded form this

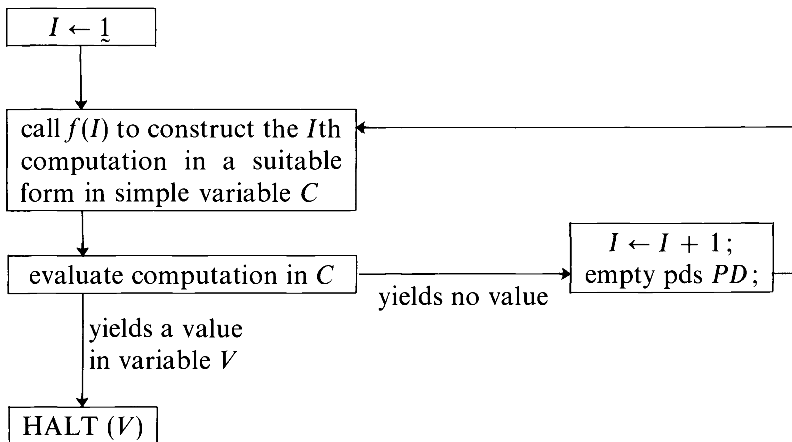


FIG. 9

would be the sequence of integers 1, 1, 3, 2, and to represent it as a single integer we write it as $2^1 * 3^1 * 5^3 * 7^2$.

To evaluate the computation C, we need only decode each name, in order, and execute the function calls. Here is where we use the pds PD; we need it to hold the temporary values generated. In general, we cannot put a bound on the number of temporaries needed in the evaluation of an expression of the effective functional; if we could, the scheme could be written in $P_{\underset{\sim}{\mathbb{N}}}$.

(10.9) COROLLARY. *Let S be any scheme in* P_{Ae}. *Then there exists an equivalent scheme* S_1 *in* $P_{\underset{\sim}{\mathbb{N}}}$ *if and only if there exist an equivalent effective functional f and an integer n such that all the expressions and propositions of f's computations can be evaluated using at most n variables to hold temporary results.*

Proof. Suppose S_1 exists. Construct an equivalent functional f, using a construction similar to that used in (10.6). The proposition and expressions of f correspond to expressions and predicates used to calculate values in S_1, and thus can be evaluated using only as many variables as S_1 uses.

Suppose an effective functional f exists with the required property. Then construct S_1 using the construction in (10.8), where we can use simple variables instead of a pds to hold temporary values.

(10.10) THEOREM. *For every scheme S in* $P_{(1,0)\underset{\sim}{\mathbb{N}}}$ *there exists an equivalent scheme in* P_{Ae}.

Proof. We can certainly simulate the single pds in P_{Ae} using an array. Now, S uses integer arithmetic using the natural numbers $\underset{\sim}{\mathbb{N}}$. However, P_{Ae} has the ability to do the same arithmetic using $\mathbb{N}$, and we need only identify $\mathbb{N}$ with $\underset{\sim}{\mathbb{N}}$. (The operation + and relation ⊜ are defined in P_{Ae}, and by Theorem (4.3) we can simulate $\dot{-}$.)

11. Multischemes and nondeterministic effective functionals. The theory of schemes with total inputs is adequate to model most situations arising in the theory of programming languages (as opposed to pure recursive function theory). In place of partial inputs one allows as inputs subroutines written in the schema language. But the basic inputs to all schemes are total.

Within this theory, we have discovered few "naturally" occurring classes of schemes: P, P_R, and P_{Ae}, with P_A being noneffectively equivalent to P_A. When beginning our research, we originally thought we would have $P_A < P_R$. The fact that P_A is universal and moreover is *noneffectively* equivalent to P_{Ae} is remarkable.

A subsequent paper will discuss multidimensional arrays, and will also describe a few other naturally occurring classes of schemes, mostly having to do with pushdown stores.

There are sound *mathematical* reasons for considering partial functions and predicates as inputs to a scheme. In particular, the theory of recursive functionals over $\mathbb{N}$, as developed by Kleene and others, allows such inputs, and one wants a comparable theory of schematic functionals (i.e., of schemes). Also, in any attempt to generalize various theories about recursive functions, such as abstract complexity theory, to the level of recursive functionals, one naturally considers partial inputs.

Moreover, there are situations in the theory of computing which can be nicely modeled using partial inputs. For instance, the relative computing times of different function inputs may be such that one program appears partial relative to the other. Or, we may wish to consider function inputs coming via communication with humans. The machine asks for the value of $f(\cdot)$ at x. If the human is unable to respond, the result is like a call to an undefined value of $f(\cdot)$.

Given the need for a theory of schemes with partial inputs, what is the appropriate model? The basic ideas for such a model can be found in the early work of recursive function theorists. More recently, Strong [7] presented his effective functionals as one machine independent model. We present a *procedure-oriented* model and prove the equivalence of the two models. The length of this paper precludes a full discussion of these very interesting schema formalisms, and we shall undertake a full presentation elsewhere.

We add to P_{Ae} the facility to process basic function and predicate calls in parallel. When a statement like $v \leftarrow f(\cdots)$ is executed, as soon as execution of f is *begun*, we go on to execute the next statement in the scheme. Thus, f and the scheme are executing simultaneously. We do not know how long f will take to store its result in v (or whether it ever will), so it is only fair to require that f indicate in some manner when it is finished.

This new type of scheme we call a *multischeme*.

(11.1) *Syntax of multischemes*. Programs are written as in P_{Ae}, with the following changes in statement types:

(1) IF $p(\cdots)$ THEN $[l:]$ $\langle S\rangle$ ELSE $[l:]$ $\langle S\rangle$ is *not* allowed.

(2) The following additional types are allowed (the first two replace the statement type deleted; the last one, the "wait" statement, is used to tell if a function evaluation is finished):

(i) $v \leftarrow p(e_1, \cdots, e_{R_p})$

(ii) IF v THEN $[l:]$ $\langle S\rangle$ ELSE $[l:]$ $\langle S\rangle$

(iii) IF $\bar{v}$ THEN $[l:]$ $\langle S\rangle$ ELSE $[l:]$ $\langle S\rangle$

(11.2) *Semantics of multischemes*. Multischemes may use a new value, *true*, which is not in the domain D and is not in $\mathbb{N}$. Secondly, each variable v consists of *two* parts:

(11.3)

conventional value or *true*	Ω or *true*
v	$\bar{v}$

The part labeled v holds the usual value assigned to a variable, while $\bar{v}$ is used only to indicate function evaluation completion. Of course, $\bar{v}$ is initially Ω.

Finally, we require that evaluation of a basic function or predicate $f(x)$ be performed in a unit of time $tf(x) \in \{0, 1, 2, 3, \cdots, \infty\}$. Thus, if f is partial when applied to x, $tf(x) = \infty$. We shall in a moment assign (rather arbitrary) units of time to the execution of statements of a multischeme. If we execute $v \leftarrow f(x)$ where $tf(x)$ is 5 (say), then the value of $f(x)$ is stored in v, and $\bar{v}$ is set to *true* after 5 more time units have lapsed.

Let us now describe the execution of statements in a multischeme.

(1) null statement. Takes 1 unit of time to execute.

(2) $v \leftarrow w$, $v \leftarrow 0$, $v \leftarrow w + 1$, GOTO l, HALT(w). Each takes 1 unit of time to execute. They are executed as in $\mathrm{P_{Ae}}$; only the "conventional value" part of v or w is changed or referenced.

(3) $v \leftarrow f(\cdots)$, $v \leftarrow p(\cdots)$. Each takes 1 unit of time to execute. v and $\bar{v}$ are both set to Ω, and execution of f or p is begun. Execution of this statement is then finished.

Suppose $tf(\cdots) = n$. Then just after n time units have lapsed, the resulting value of $f(\cdots)$ is stored in v and $\bar{v}$ is set to *true*; similarly for the execution of $v \leftarrow p(\cdots)$. Here p produces the value *true* or Ω (which stands for *false*).

(4) IF v ⊜ w THEN [l:] $\langle S \rangle_1$ ELSE [l:] $\langle S \rangle_2$
IF v THEN [l:] $\langle S \rangle_1$ ELSE [l:] $\langle S \rangle_2$
IF $\bar{v}$ THEN [l:] $\langle S \rangle_1$ ELSE [l:] $\langle S \rangle_2$

Each of these takes 1 time unit to evaluate the proposition and to decide whether $\langle S \rangle_1$ or $\langle S \rangle_2$ should be executed. The first one is executed exactly as in $\mathrm{P_{Ae}}$. With the second, $\langle S \rangle_1$ is executed next if and only if the "conventional value" part of v contains *true*; otherwise, $\langle S \rangle_2$ is executed. With the last, $\langle S \rangle_1$ is executed next if and only if the $\bar{v}$ part of variable v is *true*; otherwise $\langle S \rangle_2$ is executed. This latter statement can thus be used to wait for a value to be computed by a function or predicate, as illustrated in Example (11.4).

The following scheme, considered in $\mathrm{P_{Ae}}$ (with total functions and predicates), returns the value x if and only if $P(F^i(X))$ is true for some $i > 0$. Considered as a multischeme, however, the result may depend on the timing of the function evaluations, since the value $P(A)$ may not be stored in V in time for the conditional statement to execute properly.

(X): $A \leftarrow F(X)$;

LOOP: $V \leftarrow P(A)$;

IF V THEN HALT(X);

$A \leftarrow F(A)$; GO TO LOOP

(11.4) *Example.*

(X): $N \leftarrow 1$; $A[1] \leftarrow F(X)$;

LOOP: IF $\overline{A[N]}$ THEN

BEGIN $V[N] \leftarrow P(A[N])$;

$A[N + 1] \leftarrow F(A[N])$;

$N \leftarrow N + 1$

END;

[If possible, begin computation of $P(F^N(X))$ and $F^{N+1}(X)$.]

```
I ← 1;
WHILE I ≠ N DO                              [If P(F^i(X)) is com-
   BEGIN IF ‾V[I]‾ THEN                      puted and true for
                                             some i < N, then HALT.
           IF V[I] THEN HALT(X);
        I ← I + 1
   END;
GO TO LOOP
```

Scheme (11.4) uses an array A to store $F^1(X)$, $F^2(X)$, $\cdots$ and an array V to store $P(F^1(X))$, $P(F^2(X))$, $\cdots$. The statement LOOP begins computation of $P(F^N(X))$ and $F^{N+1}(X)$ *if and only if* $F^N(X)$ has been computed. The WHILE statement HALTs when some computed value $P(F^i(X))$ is true. Obviously, then, the scheme HALTs and outputs value X if and only if

$$P(F^i(X)) = \text{true} \quad \text{for some } i \geqq 1;$$

otherwise it does not stop. Note that the output is *finitely independent* of the times $tF(\cdot)$ and $tP(\cdot)$.[2]

(11.5) To execute a multischeme S, then, we are given not only input values $d_1, \cdots, d_l$, functions $\phi_1, \cdots, \phi_n$ and predicates $\Pi_1, \cdots, \Pi_m$, but also *implicitly* a *time set*

$$T = \{t\phi_1, \cdots, t\phi_n, t\Pi_1, \cdots, t\Pi_m\}$$

which satisfies the conditions $t\phi_i(X)\downarrow$ if and only if $\phi_i(X)\downarrow$ and $t\Pi_i(X)\downarrow$ if and only if $\Pi_i(X)\downarrow$.[3] (In the case that $\{\phi_i\}$ are partial recursive functions, we might want $\{t\phi_i\}$ to be Blum complexity measures.) We interpret $v \leftarrow \phi_i(X)$ to mean that the value gets stored in v just after $t\phi_i(X)$ more time units have lapsed.

The semantics we have proposed for multischemes tacitly assumes that function and predicate inputs are supplied as processes which can be "executed" in some manner. The details of this assumption are not necessary; all that we require is the associated *time set*.

It is interesting to compare this process interpretation with the usual set-theoretic interpretation of functions. A partial function ϕ is simply a set of ordered pairs, $S_\phi = \{\langle x, y\rangle | \phi(x) = y\}$. We might then be given ϕ as a set. To compute with a set we ask, "is $\langle x, y\rangle \in S_\phi$?", and we assume that an "oracle" can always answer this question. If the domain D is enumerable, say $D = \mathbb{N}$, then we can "compute" $\phi(x)$ from S_ϕ as follows: ask $\langle x, 0\rangle \in S_\phi$, $\langle x, 1\rangle \in S_\phi, \cdots, \langle x, n\rangle \in S_\phi, \cdots$. If $\phi(x) = y$, then for this process $t\phi(x) = y$; and if $\phi(x)$ is undefined, then so is $t\phi(x)$.

If D is not enumerable, this process fails, but the concept of a time set is still applicable.

---

[2] By *finitely independent* of the times $tF(\cdot)$, we mean it does not matter whether $tF(\cdot) = 1, 2, 3, \cdots, n$ for any finite n. It does of course depend on whether $tF(\cdot)$ is finite or infinite.

[3] $\phi(X)\downarrow$ means that ϕ halts on X.

(11.6) DEFINITION. A multischeme is *well-defined* if and only if the output value of the scheme is finitely independent of the time set T. The class of well-defined multischemes is denoted $\mathrm{MP_{Ae}}$.

We claim that $\mathrm{MP_{Ae}}$ is a universal scheme formalism over the domain of partial functions and predicates. A reasonable notation for the class of functionals computed is $\mathbb{F}(\mathrm{MP_{Ae}}, \mathscr{P}(D), \mathscr{P}_*(D))$. Previously, $\mathbb{F}(\mathrm{P_{Ae}}, D)$ was used for $\mathbb{F}(\mathrm{P_{Ae}}, \mathscr{T}(D), \mathscr{T}_*(D))$, where $\mathscr{T}(D)$ are the *total* functions over D and $\mathscr{T}_*(D)$ are the *total* predicates over D. To substantiate our claim of universality we want to show that

$$\mathbb{F}(\mathrm{EF}, \mathscr{P}(D), \mathscr{P}_*(D)) = \mathbb{F}(\mathrm{MP_{Ae}}, \mathscr{P}(D), \mathscr{P}_*(D))$$

for all D. We write this as $\mathrm{EF} \simeq \mathrm{MP_{Ae}}$ (on partial interpretations).

(11.7) ASSERTION. $\mathrm{EF} \leqq \mathrm{MP_{Ae}}$.

Let f be the effective procedure which enumerates the computations of an effective functional. The equivalent S in $\mathrm{MP_{Ae}}$ is constructed in the same fashion as that described in § 10 when we asserted that $\mathrm{EF} \leqq \mathrm{P_{Ae}}$; it is outlined in Fig. 10. S contains a loop which constructs the computations in Polish postfix form, stored as single integers in $A[1]$, $A[2]$, $\cdots$. Between the construction of computation $A[I]$ and $A[I + 1]$, we proceed to evaluate as much as possible the computations $A[1], \cdots, A[I]$.

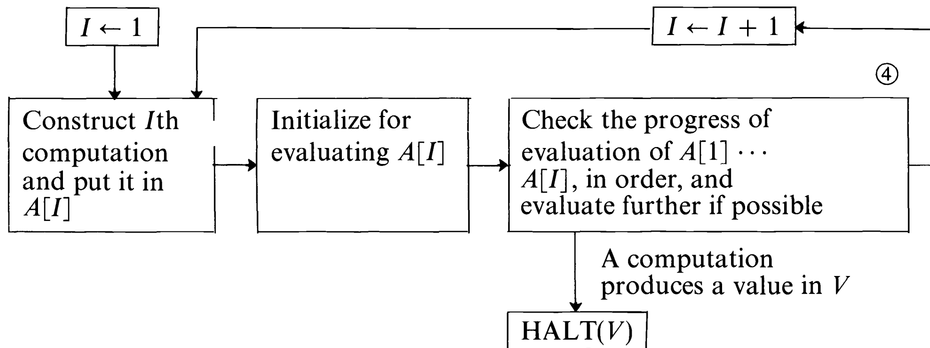


FIG 10

A more detailed description of box 4 of Fig. 10 is in order. We use a counter J to check the computations $A[1], A[2], \cdots, A[J], \cdots, A[I]$. We use a two-dimensional array[4] B, where row $B[J, 1], B[J, 2], \cdots$ is used as a stack to store temporary results of computation J. When we *first* "check the progress" of a computation, we begin evaluating it, and continue until a basic function or predicate must be evaluated. We then begin this evaluation and stop working on the computation.

The next time we have to "check the progress" of the computation, we see whether the basic function or predicate has returned its value. If not, we can do

[4] We cannot of course use multidimensional arrays in $\mathrm{MP_{Ae}}$. We leave it to the reader to show how they can be "simulated".

nothing more; if it has, we then continue evaluating the computation until another basic function or predicate must be evaluated.

All computations (of the effective functional) which produce a value, produce the same value. This means the multischeme we have described is well-defined because it only outputs the value of a computation.

(11.8) ASSERTION. $MP_{Ae} \leqq EF$ (*on partial interpretations*).

Let S be a well-defined multischeme. Then the result of execution does not depend on the time set T. Given a time set T we can clearly produce an effective functional for S and that time set (as we did in § 10), but the computations in it depend on the particular time set used.

To produce the effective functional *equivalent* to S, we systematically vary the time set T and record all possible finite behaviors of S, for *all* time sets T. For each of these finite behaviors we get a computation of the effective functional as in § 10. Clearly this effective functional is equivalent to S.

(11.9) *Example*. We give a scheme in MP_{Ae} which computes leaftest (Example (6.5)) assuming that P, R and L may be partial. The scheme uses an array A to hold values of the nodes and a corresponding array B to hold the values P (node). We continually process nodes generated so far, as follows:

```
IF the value of the node has been computed THEN
  IF we have not begun to compute P (node) THEN
     BEGIN begin computing P (node);
                begin computing L (node) to get a new node value;
                begin computing R (node) to get a new node value;
     END
  ELSE IF P (node) has yielded an answer THEN
             IF P (node) is true THEN HALT(X)
```

```
(X):     A[0] ← X; B[0] ← P(X);   (Start root node going)
         A[1] ← L(X); B[1] ← 1;   (Start computing first leafs)
         A[2] ← R(X); B[2] ← 1;
         T ← 2;
LOOP: IF ‾B[0]‾ THEN IF B[0] THEN HALT(X);   (Process root node)
         I ← 1;
         WHILE I ≦ T DO                        (Process 1 node at a time)
           BEGIN IF ‾A[I]‾ THEN
             IF B[I] ⊜ 1 THEN
               BEGIN B[I] ← P(A[I]);
                 T ← T + 1; A[T] ← L(A[I]); B[T] ← 1;
                 T ← T + 1; A[T] ← R(A[I]); B[T] ← 1;
               END
```

ELSE IF $\overline{B[I]}$ THEN IF $B[I]$ THEN HALT(X);

$I \leftarrow I + 1$

END;

GOTO LOOP

(11.10) To conclude we want to sketch very briefly how the example of (11.9) can be used to show that $P_{Ae} < MP_{Ae}$, i.e., P_{Ae} is not universal on partial interpretations, but MP_A is.

Strong [7] defines a subset of EF called the *deterministic effective functionals*, EF^d. Briefly, F is deterministic if and only if its computations can always be evaluated in the order enumerated; that is, if computation i is the first computation to yield a value for the given inputs, then in all previous computations $1, 2, \cdots, i - 1$ in the recursive enumeration defining F, all predicates and expressions up to the first false predicate are defined. Let EF^d denote these functionals.

Strong calls a functional *inherently nondeterministic* if and only if there is no deterministic functional equivalent to it. He then shows that leaftest in which P is partial but L and R are total (a special case of (11.9)) is inherently nondeterministic, and thus not computable in EF^d.

We can apply this result by relating EF^d to our classes. Note that there is a natural way to interpret P_{Ae} on partial inputs. First we modify P_{Ae} making the changes of (11.1) except for (2) (iii). Clearly these changes are inessential. Next, we say that whenever an assignment $V \leftarrow F(\cdots)$ or $V \leftarrow P(\cdots)$ is executed, control waits until a value is returned before executing the next statement. If the operator F or P is partial, then the scheme is said to *jam* because control never passes the assignment.

To be precise about this interpretation we define a canonical mapping of P_{Ae} into MP_{Ae}. Namely, for each assignment statement $V \leftarrow F(\cdots)$ (or $V \leftarrow P(\cdots)$) of scheme S in P_{Ae} generate a new label L and replace the statement by

BEGIN $V \leftarrow F(\cdots)$;

L: IF $\overline{V}$ THEN ELSE GOTO L END

We now identify P_{Ae} over partial domains as the image of P_{Ae} in MP_{Ae} under this mapping.

Relating P_{Ae} to EF^d is now easy, given the methods of § 10 and this section. The reader can check the following assertion.

(11.11) ASSERTION. $EF^d \leqq P_{Ae}$ *and* $P_{Ae} \leqq EF^d$.

Thus Strong's deterministic effective functionals correspond to the natural interpretation of P_{Ae} schemes on partial inputs.

Now applying Strong's result that leaftest is inherently nondeterministic, we get the following corollary.

(11.12) COROLLARY. $P_{Ae} < MP_{Ae}$.

On total interpretations all of the classes P_{Ae}, MP_{Ae}, EF^d and EF coincide.

On partial interpretations we get the following diagram (the notation $A \to B$ indicates an effective translation from A into B).

$$\begin{array}{ccc} \mathrm{EF} & \longleftrightarrow & \mathrm{MP_{Ae}} \\ \uparrow & & \uparrow \\ \mathrm{EF^d} & \longleftrightarrow & \mathrm{P_{Ae}} \end{array}$$

Appendix. The Appendix contains details of some of the constructions outlined in the paper. When reading them, refer back to the general outline for helpful comments.

(6.1) THEOREM. *Given a scheme S in $\mathrm{P_R}$, one can construct an equivalent scheme S' in $\mathrm{P_{AL}}$.*

Construction.

Step 1. (Generate initialization statements): Insert before the main ⟨body⟩ of S the statements to initialize the global counters TOP and AA: TOP ← 0; AA ← 0;

Step 2. (Change simple variables to subscripted variables): Change each simple variable V_i in each function definition ⟨body⟩ to $A[\mathrm{AA} + i]$.

Step 3. (Change each HALT): Generate a new variable RV and change each HALT(v) in each function definition ⟨body⟩ to

$$\text{BEGIN } RV \leftarrow v; \quad \text{GOTO } A[\mathrm{AA} + (p + 1)] \text{ END}$$

where the function definition uses p variables.

Step 4. Change each function heading $f(V_1, \cdots, V_{Rf})$: to 'Lf:' where Lf is a new unique label.

Step 5. (Change each function call): For each nonbasic function call

$$v \leftarrow f(v_1, \cdots, v_{Rf})$$

within the complete scheme, generate a new, unique label RL, and replace the statement with the following:

$$\begin{array}{rll}
\text{BEGIN TOP} & \leftarrow \text{TOP} + 1; & \text{(part of action 1)} \\
A[\text{TOP}] & \leftarrow \text{AA}; & \text{(action 2)} \\
A[\text{TOP} + (p + 1)] & \leftarrow RL; & \text{(action 3)} \\
A[\text{TOP} + 1] & \leftarrow v_1; & \\
\vdots & \quad \vdots & \\
A[\text{TOP} + Rf] & \leftarrow v_{Rf} & \\
A[\text{TOP} + (Rf + 1)] & \leftarrow OMEGA; & \text{(action 4)} \\
\vdots & \quad \vdots & \\
A[\text{TOP} + p] & \leftarrow OMEGA; & \\
\text{AA} & \leftarrow \text{TOP}; & \text{(action 5)} \\
\text{TOP} & \leftarrow \text{TOP} + (p + 1); & \text{(rest of action 1)}
\end{array}$$

GOTO Lf; (jump to function)

RL: AA $\leftarrow A[\text{AA}]$; $v \leftarrow RV$ Return here

END

where Lf is the label which replaced "$f(V_1, \cdots, V_{Rf})$:" in Step 4; and where the function definition of f used p simple variables.

(7.2) LEMMA. $P_{(2,m)} \leqq P_{(2,1)}$ *for* $m \geqq 0$.

Proof. Let S in $P_{(2,m)}$ use markers $M_1, \cdots, M_m$. The equivalent scheme S' in $P_{(2,2)}$ uses marker M. Each marker M_i in S is represented in S' by $m + 1$ values:

$$M_i \equiv M, \underbrace{\Omega, \cdots, \Omega}_{\substack{i-1 \\ \text{times}}}, M, \underbrace{\Omega, \cdots, \Omega}_{\substack{m-i \\ \text{times}}}.$$

Each simple variable v in S is represented by $m + 1$ variables $v, v_1, v_2, \cdots, v_m$ in S'. To translate S into S' perform the following steps:

1. Change each statement $v \leftarrow w$ into

   BEGIN $v \leftarrow w$; $v_1 \leftarrow w_1$; $\cdots$; $v_m \leftarrow w_m$ END

2. Change each statement $v \leftarrow M_i$ into

   BEGIN $v \leftarrow M$; $v_1 \leftarrow OMEGA$; $\cdots$; $v_m \leftarrow OMEGA$; $v_i \leftarrow M$ END

3. For each statement

   IF $v = M_i$ THEN S_1 ELSE S_2

   change the statement to

   IF $v = M$ *and* $v_i = M$

   THEN S_1 ELSE S_2

4. Change each statement (where PD is a pds) $PD \leftarrow v$

   to BEGIN $PD \leftarrow v_1$; $\cdots$; $PD \leftarrow v_m$; $PD \leftarrow v$; $PD \leftarrow M$ END

5. Let W be a new, unique variable. Change each statement

   $v \leftarrow PD$

   to BEGIN $W \leftarrow OMEGA$; [If $W \neq M$ after the pop, the pds PD was empty.]

   $W \leftarrow PD$;

   IF $W = M$ THEN

   BEGIN $v \leftarrow PD$; $v_m \leftarrow PD$; $\cdots$; $v_1 \leftarrow PD$ END

   END

(7.4) LEMMA. *For any scheme* $S \in P_R$ *there exists an equivalent scheme* S_1 *in* P_{pdsL} *which uses only one pds.*

Proof. Assume without loss of generality that each function ⟨body⟩ and the main scheme ⟨body⟩ of S all use p variables $V_1, \cdots, V_p$, where $V_1, \cdots, V_{Rf}$

are the normal parameters ($Rf = 0$ for the main program). S_1 uses one pds PD. We construct S_1 as follows, using new variables $A_1, \cdots, A_{Rf}$, RET, and RV.

Step 1. Change each nonbasic call $V_i \leftarrow f(v_1, \cdots, v_{Rf})$ to

BEGIN	$A_1 \leftarrow v_1; \cdots; A_{Rf} \leftarrow v_{Rf}$;	[Save arguments]
	$PD \leftarrow V_1; \cdots; PD \leftarrow V_p$;	[Put variables on stack]
	$PD \leftarrow RL$;	[Return label stacked]
	$V_1 \leftarrow A_1; \cdots; V_{Rf} \leftarrow A_{Rf}$;	[Initialize formal parameters]
	$V_{Rf+1} \leftarrow OMEGA; \cdots; V_p \leftarrow OMEGA$;	[Fix rest of local variables]
	GOTO Lf;	[Jump to function]
RL:	$V_i \leftarrow RV$;	[Get value of function]
END		

Step 2. Change each HALT(V_j) within a function definition to

BEGIN	$RV \leftarrow V_j$;	[Value returned]
	RET $\leftarrow PD$;	[Label to return to]
	$V_p \leftarrow PD; \cdots; V_1 \leftarrow PD$;	[Restore old values]
	GOTO RET	[Return]
END		

Step 3. Change each function heading "$f(V_1, \cdots, V_{RF})$" to "Lf:". This results in an equivalent scheme S_1 in P_{pdsL}, where one pds PD is used.

(8.3) LEMMA. $P_{AM} \leqq P_{pdsM}$.

Proof. Given S in P_{AM}, we give the steps for constructing S' in P_{pdsM} equivalent to S.

Step 1. Let scheme S be

$$(w_1, \cdots, w_n): \langle \text{body} \rangle$$

where $w_1, \cdots, w_n$ are simple variables. Change S into

$$(IN_1, \cdots, IN_n): w_1 \leftarrow IN_1; \cdots; w_n \leftarrow IN_n; \langle \text{body} \rangle$$

where $IN_1, \cdots, IN_n$ are new simple variables. Second, suppose the function or predicate with the largest rank has rank $m \geqq 1$. Generate m new simple variables $\text{ARG}_1, \cdots, \text{ARG}_m$, and a new simple variable V_0. Replace each statement

$$v \leftarrow f(v_1, \cdots, v_{Rf})$$

by

$$\begin{aligned}
&\text{BEGIN } V_0 \leftarrow v_1;\ \text{ARG}_1 \leftarrow V_0;\\
&\qquad\vdots\\
&\qquad V_0 \leftarrow v_{Rf};\ \text{ARG}_{Rf} \leftarrow V_0\\
&\qquad \text{ARG}_1 \leftarrow f(\text{ARG}_1, \cdots, \text{ARG}_{Rf});\ V_0 \leftarrow \text{ARG}_1;\ v \leftarrow V_0\\
&\text{END}
\end{aligned}$$

Replace each statement

$$\text{IF } p(v_1, \cdots, v_{Rp}) \text{ THEN } S_1 \text{ ELSE } S_2$$

by

$$\begin{aligned}
&\text{BEGIN } V_0 \leftarrow v_1;\ \text{ARG}_1 \leftarrow V_0;\\
&\qquad\vdots\\
&\qquad V_0 \leftarrow v_{Rp};\ \text{ARG}_{Rp} \leftarrow V_0;\\
&\qquad \text{IF } p(\text{ARG}_1, \cdots, \text{ARG}_{Rp}) \text{ THEN } S_1 \text{ ELSE } S_2\\
&\text{END}
\end{aligned}$$

Change each statement HALT(v) to

$$\text{BEGIN } V_0 \leftarrow v;\ \text{HALT}(V_0) \text{ END}$$

Obviously, these changes yield an equivalent scheme S_1. Again, the point of the transformations is to produce a scheme in which the input variables and arguments to functions, predicates, and HALTs need not be considered as pds's, ever. (They never contain a marker m or a value in $\mathbb{N}$. If they do, we use just a single asterisk * in P_{AM}.) Hence, the variables ARG_i, IN_i, do not have to be represented by push-down stores in S'.

Step 2. We leave it to the reader to show that the S_1 scheme can be further transformed to yield an equivalent P_A scheme S_2 with the following 12 types of simple statements, where T is one of the variables ARG_i or IN_j, v and w are *simple* variables which are not ARG_i or IN_j, and m is a marker:

(1) $T \leftarrow v$
(2) $v \leftarrow T$
(3) $v \leftarrow w$ (where v and w are not the same)
(4) $v \leftarrow w + 1$ (where v and w are not the same)
(5) $v \leftarrow m$
(6) $v \leftarrow A[w]$ (where v and w are not the same)
(7) $A[v] \leftarrow w$ (where v and w are not the same)
(8) $T \leftarrow f(\text{ARG}_1, \cdots, \text{ARG}_{Rf})$
(9) IF $p(\text{ARG}_1, \cdots, \text{ARG}_{Rp})$ THEN S_1 ELSE S_2
(10) IF $v = m$ THEN S_1 ELSE S_2
(11) GOTO l
(12) $v \leftarrow 0$

If we remember that each variable (except ARG_i, IN_j and V_0) will be represented by a pushdown store, we see that statements must be transformed accordingly.

Step 3. (Transform simple variables to stores): Let M_0 be a new marker. Consider each simple variable v (except the ARG_i, IN_j and V_0) to be a pushdown store. For each such v insert at the beginning of the scheme the statements

$$v \leftarrow M_0;\ v \leftarrow OMEGA$$

This will initialize the store to Ω. *Each store always will have M_0 as the first element, and will always contain a value.*

We assume we can write a (compound) statement to empty a store v completely, and another to copy a stack w to a completely empty store v, without changing w. Let us assume we denote these by

$$\text{EMPTY}(v) \quad \text{and} \quad \text{COPY}(w, v).$$

We now show the transformation of statement types (1)–(5), (10), and (12); TEMP is a new, unique simple variable:

(1) BEGIN $T \leftarrow v$; $v \leftarrow T$ END
(Remember T can only contain values in D, and this value must be at the top of the store v.)

(2) BEGIN EMPTY(v); $v \leftarrow M_0$; $v \leftarrow T$ END

(3) BEGIN EMPTY(v); COPY(w, v) END

(4) BEGIN EMPTY(v); COPY(w);

TEMP $\leftarrow v$; $v \leftarrow$ TEMP;

IF TEMP = * THEN $v \leftarrow$ TEMP

ELSE BEGIN EMPTY(v); $v \leftarrow M_0$; TEMP $\leftarrow$ *;

$v \leftarrow$ TEMP; $v \leftarrow$ TEMP

END

END

(Remember $v + 1 = 0 + 1$ if $v \notin \mathbb{N}$.)

(5) BEGIN EMPTY(v); TEMP $\leftarrow m$; $v \leftarrow M_0$; $v \leftarrow$ TEMP END

(10) BEGIN TEMP $\leftarrow v$; $v \leftarrow$ TEMP;

IF TEMP = m THEN S_1 ELSE S_2

END

(12) BEGIN EMPTY(v); $v \leftarrow M_0$; $v \leftarrow$ * END

Step 4. The only statements which have not yet been transformed into statements in P_{pdsM} are those of types (6) and (7) which reference arrays. We now show how arrays are represented.

Let ',' be a new, unique marker. Each array A in S is simulated in S' by a pds PD. Let A be as in Fig. 5a. Then PD is as in Fig. 5b. Thus, *array elements* of A are separated in PD by the marker ','.

Note that, during execution, if the "highest" array element assigned to thus far is $A[i]$, then the corresponding pds PD in S' will describe exactly the array elements $A[0], \cdots, A[i]$.

Now, for each array A used in S_3 (the result of *Step* 3), generate a pds PD and insert the statements

$$PD \leftarrow M_0; \; PD \leftarrow OMEGA;$$

at the beginning of S_3. Similarly, initialize three temporary pds's $PDT1$, $PDT2$, and $PDT3$.

The statement

$$v \leftarrow A[w] \qquad (v, w \text{ not the same})$$

is translated into a compound statement, which

(1) puts PD into store $PDT1$, *upside down* without changing PD;

(2) copies the store w into $PDT2$ (EMPTY($PDT2$); COPY(w, $PDT2$));

(3) empties the store v (EMPTY(v));

(4) deletes "array elements" from the store $PDT1$ until the one described by $PDT2$ is on top. Remember, array elements in the stack are separated by commas ',', and $PDT2$ contains a sequence of 0 or more asterisks '*' to indicate how many array elements to delete;

(5) moves the top array element from $PDT1$ into v.

If the number of asterisks in $PDT2$ is greater than the number of commas in $PDT1$, then we are executing $v \leftarrow A[w]$, where $A[w]$ is undefined. In this case we put Ω into v.

Similarly, a statement

$$A[w] \leftarrow v \qquad (v, w \text{ not the same})$$

is translated into a compound statement which

(1) puts PD into the store $PDT1$ upside down, and empties PD;

(2) copies w into $PDT2$, without changing w;

(3) copies as many "array elements" from $PDT1$ back into PD, upside down, as there are asterisks in $PDT2$;

(4) deletes the top array element from $PDT1$ (this is $A[w]$);

(5) copies v onto PD; adding a comma ',';

(6) moves the rest of $PDT1$ back to PD (upside down of course).

If, in step (3), there are more asterisks in $PDT2$ than commas in $PDT1$, extra array elements $= \Omega$ are put into PD.

We leave the programming details to the reader.

(9.1) *Construction of S_1 in case of infinite v-autonomous behavior.* We already gave a program to construct S_1 in case of one function of rank w. We now outline how to construct S_1 inductively on the maximum rank of the functions and predicates. A more general discussion of such "programming by induction" is given in [8].

Suppose there are functions $f_1, \cdots, f_n$ and predicates $P_1, \cdots, P_k$ of rank 1, and that the input variables to the original scheme S are $V_1, V_2, \cdots, V_l$. Then we

write the following statement:

(A.1)

$$
\begin{aligned}
&\text{BEGIN } I \leftarrow 0; \\
&\qquad K \leftarrow 0; A[K] \leftarrow OMEGA; \\
&\qquad K \leftarrow K + 1; A[K] \leftarrow V_1; \\
&\qquad \vdots \\
&\qquad K \leftarrow K + 1; A[K] \leftarrow V_l \qquad \text{(Initialize)} \\
&\text{LOOP: } I \leftarrow I + 1; \\
&\qquad [\text{test predicates of rank 1 on } A[I \dot{-} 1]]; \\
&\qquad K \leftarrow K + 1; A[K] \leftarrow f_1(A[I \dot{-} 1]); \\
&\qquad \vdots \\
&\qquad K \leftarrow K + 1; A[K] \leftarrow f_n(A[I \dot{-} 1]); \\
&\qquad \text{GOTO LOOP} \\
&\text{END}
\end{aligned}
$$

We leave the details of the tests of predicates of rank 1 to the reader. Initialization puts all initial values into array A. Variable K always indicates how many values are in A. This scheme could have been simplified by putting the incrementation of I just before the GOTO LOOP statement; it has been written this way in order to facilitate the induction process.

If there are no functions of higher rank, then clearly the above statement performs as desired. All possible values are put into the array A, while I is used to sequence through A testing these values.

Now suppose we have a statement S_1 which "takes care" of all functions and predicates of rank $m > 0$ or less. Its form is:

(A.2)

$$
\begin{aligned}
&\text{BEGIN } I \leftarrow 0; \\
&\qquad S_1; \cdots; S_n; \qquad \text{(Initialization)} \\
&\text{LOOP: } I \leftarrow I + 1; \\
&\qquad \vdots \\
&\qquad [\text{test predicates of rank } m \text{ on } v_1, v_2, \cdots, v_m] \\
&\qquad K \leftarrow K + 1; A[K] \leftarrow f^m_1(v_1, v_2, \cdots, v_m) \\
&\qquad \vdots \\
&\qquad K \leftarrow K + 1; A[K] \leftarrow f^m_{k_m}(v_1, v_2, \cdots, v_m); \\
&\qquad \text{GOTO LOOP} \\
&\text{END}
\end{aligned}
$$

The assumptions are that A will hold all possible values, K indicates how many values are in A, the $S_1, \cdots, S_n$ are initialization statements, I is changed only

where explicitly indicated, and that any given value is put in A at some time. All possible m-tuples $(v_1, \cdots, v_m)$ will be used as arguments to $f_1^m, \cdots, f_{k_m}^m$ at some point. Note that the statement programmed for rank 1 satisfies these requirements and has form (A.2).

Suppose, in addition, that the original scheme S used predicates and functions $f_p, \cdots, f_q$ of rank $m + 1$. Then we change (A.2) to use new, unique arrays AD, AS, $A1, \cdots, Am$, and simple variables $I1$ and $J1$. $A1, \cdots, Am$ will hold all possible m-tuples; $J1$ is their counter. AD, AS and $I1$ will be used as in the locator of rank 2 to reference, at each step, a value $A[I1]$ and an m-tuple $A1[AS[I1]], \cdots, Am[AS[I1]]$. $S1$ and $S2$ are new simple variables.

We change (A.2) to

BEGIN $I \leftarrow 0$; $S_1; \cdots; S_n$;	Old initialization
$A1[0] \leftarrow OMEGA; \cdots; Am[0] \leftarrow OMEGA$; $AD[0] \leftarrow 0; AS[0] \leftarrow 0$;	New initialization
$I1 \leftarrow 0; J1 \leftarrow 0$;	
LOOP: $I \leftarrow I + 1$; ⋮	
[test predicates of rank m on $v_1, v_2, \cdots, v_m$] $K \leftarrow K + 1; A[K] \leftarrow f_1^m(v_1, \cdots, v_m)$; ⋮ $K \leftarrow K + 1; A[K] \leftarrow f_{k_m}^m(v_1, \cdots, v_m)$;	As before
$J1 + J1 + 1; AD[J1] \leftarrow J1 \dot{-} 1; AS[J1] \leftarrow 0$; $A1[J1] \leftarrow v_1; \cdots; Am[J1] \leftarrow v_m$;	Put new m-tuple in $A1, \cdots, Am$.
$S1 \leftarrow I1; S2 \leftarrow AS[I1]$;	Get subscript for rank $m + 1$ work.
$AS[I1] \leftarrow AS[I1] + 1; AD[0] \leftarrow AS[0]$; $I1 \leftarrow AD[I1]$;	Fix 2nd subscript and go down diagonal.
[test predicates of rank $m + 1$ on $A[S1]$, $A1[S2], \cdots, Am[S]]$	Test rank $m + 1$ predicates.
$K \leftarrow K + 1; A[K] \leftarrow f_p(A[S1], A1[S2], \cdots, Am[S2])$; ⋮ $K \leftarrow K + 1; A[K] \leftarrow f_q(A[S1], A1[S2], \cdots, Am[S2])$;	Get new function values.
GOTO LOOP	
END	

Note first of all that the transformation does not change any of the original statements and, since none of the original variables are changed by new statements, that values on functions of rank m or less are calculated and stored as before. This new statement has form (A.2), where $v_1 = A[S1], v_2 = A1[S2], \cdots, v_{m+1} = Am[S2]$.

REFERENCES

[1] J. HOPCROFT AND J. ULLMAN, *Formal Languages and Their Relation to Automata*, Addison-Wesley, London, 1969.

[2] D. C. LUCKHAM, D. M. R. PARK AND M. S. PATERSON, *On formalized computer programs*, J. Comput. System Sci., 4 (1970), pp. 220–249.

[3] J. MCCARTHY, *Recursive functions of symbolic expressions and their computation by machine, Part I*, Comm. ACM, 3 (1960), pp. 184–195.

[4] M. S. PATERSON, *Equivalence problems in a model of computation*, Doctoral thesis, Cambridge Univ., Cambridge, England, 1967. Also Artificial Intelligence Tech. Memo, 1, M.I.T., Cambridge, Mass., 1970, 153 pp.

[5] M. S. PATERSON AND C. E. HEWITT, *Comparative schematology*, Conf. Record of Project MAC Conference on Concurrent Systems and Parallel Computation, Assoc. for Computing Machinery, New York, 1970, pp. 119–128.

[6] H. R. STRONG, *Translating recursion equations into flow charts*, J. Comput. System Sci., 5 (1971), pp. 254–285.

[7] ———, *High level languages of maximum power*, Proc. IEEE Conf. on Switching and Automata Theory, 1971, pp. 1–4.

[8] D. GRIES, *Programming by induction*, Information Processing Letters, 2 (1972).

[9] M. MINSKY, *Computation, Finite and Infinite Machines*, Prentice-Hall, Englewood Cliffs, N.J., 1967.

SIAM J. COMPUT.
Vol. 1, No. 1, March 1972

SOME UNDECIDABILITY RESULTS FOR PARALLEL PROGRAM SCHEMATA*

RAYMOND E. MILLER†

Abstract. Some theorems showing undecidability for computational commutativity, boundedness, termination and determinacy of parallel program schemata are proved. These results are then compared with contrasting decidability results in [1] showing that the deletion of the hypothesis of repetition-freeness from the decidability theorems produces undecidability.

1. Introduction. In attempts to better understand the structure of computer programs, and to circumvent the well-known undecidability results concerning, for example, the termination and equivalence of programs, people have turned to modeling certain restricted aspects of programs for which some more positive (decidability) results can be obtained. The determination and formulation of the important properties of program structure and behavior is of interest in itself. But in addition, results that establish boundaries between when such inherent properties are decidable or undecidable provide a fuller understanding of what modifications and simplifications of programs can be carried out in an algorithmic fashion.

One of these approaches, which uses the program schemata model, is aimed primarily at studying the control flow or sequencing aspects of programs rather than the particular functions that a program computes. Among the various schemata formulations, Ianov [2] (also see Rutledge [3]) showed that a certain type of equivalence between schemata was decidable. Luckham, Park and Paterson [4] showed that adding a little more structure to the memory, say by having two or more distinct memory locations, causes almost any reasonable type of equivalence to be undecidable.

In [1] Karp and Miller introduced the notion of parallelism into schemata and extensively studied the property of determinacy for such schemata. Intuitively, determinacy means that the values being computed are not dependent upon the relative sequencing between the operations being performed in parallel. It was shown that determinacy was decidable for a large class of parallel program schemata. In addition, it was shown that equivalence was undecidable for indeterminate parallel program schemata. In a recent paper by Itkin and Zwinogrodski [5] the construction of Karp and Miller to show undecidability for parallel program schemata equivalence was employed, together with the implementation of a "reset" operation, to show that equivalence was undecidable even for determinate parallel program schemata.[1] In addition Itkin and

* Received by the editors August 9, 1971, and in revised form January 3, 1972.

† 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.

[1] It is interesting to note that the general idea of a reset in schemata was first used by Luckham, Park and Paterson [4] to show undecidability of equivalence for their program schemata. Resets were previously introduced in a different context by Rosenberg [6].

Zwinogrodski showed that state accessibility and the existence of a finite computation were also undecidable.

In this paper we extend these undecidability results to detecting computational commutativity, boundedness, termination and determinacy. In particular, the unsolvability of determinacy holds for finite state schemata (Theorem 4). This answers negatively a problem posed in [1] concerning the existence of an effective test for determinacy for the class of finite state schemata. In § 5 we compare these results with the decidability results of [1], and show that the deletion of the single hypothesis of repetition-freeness provides a boundary between decidability and undecidability for these properties of parallel program schemata.

2. Preliminaries. To make this paper fairly self-contained we review some of the basic definitions of parallel program schemata [1].

A *parallel program schema* $\mathscr{S} = (M, A, \mathscr{T})$ consists of a set M of *memory locations*; a finite set A of *operations*: we associate with each $a \in A$ a positive integer $K(a)$ called the *number of outcomes* of a, and sets $D(a) \subseteq M$ and $R(a) \subseteq M$ called the *domain locations* and *range locations* of a respectively; and a transition system control $\mathscr{T} = (Q, q_0, \Sigma, \tau)$, where Q is a set of *states*, q_0 is a designated *initial state*, $\Sigma = \Sigma_i \cup \Sigma_t$ is the *alphabet*, where $\Sigma_i = \bigcup_{a \in A} \{\bar{a}\}$ is the set of *initiation symbols* and $\Sigma_t = \bigcup_{a \in A} \{a_1, \cdots, a_{K(a)}\}$ is the set of *termination symbols*, and τ is a partial transition function from $Q \times \Sigma$ to Q which is total on $Q \times \Sigma_t$.

An *interpretation* $\mathscr{I}$ of a schema $\mathscr{S}$ is given by: (a) a function C which associates a set of values $C(i)$ with each $i \in M$, (b) an initial memory contents $c_0 \in \bigtimes_{i \in M} C(i)$, and (c) for each $a \in A$, two functions $F_a : \bigtimes_{i \in D(a)} C(i) \to \bigtimes_{i \in R(a)} C(i)$ and $G_a : \bigtimes_{i \in D(a)} C(i) \to \{a_1, \cdots, a_{K(a)}\}$. For a performance of a, F_a determines the results to be stored in locations $R(a)$ and G_a determines the conditional branch to be taken.

A finite or infinite word z over Σ is called an $\mathscr{I}$*-computation* for $\mathscr{S}$ if for C, c_0, F_a and G_a defined by $\mathscr{I}$:

(i) every prefix $y\sigma$ of z with $\sigma \in \Sigma$ satisfies the constraints that $\tau(q_0, y\sigma)$ is defined,[2] and if σ is a termination symbol for $a \in A$, then the number of initiation symbols $\bar{a}$ in y is greater than the number of termination symbols in y for operation a;

(ii) if z is finite, then for all $\sigma \in \Sigma$, condition (i) is not satisfied for $z\sigma$;

(iii) if x is a prefix of z and $\sigma \in \Sigma$ with the property that for every y such that xy is a prefix of z it follows that $xy\sigma$ satisfies (i), then for some y', $xy'\sigma$ is a prefix of z;

(iv) if $x\sigma$ is a prefix of z and $\sigma \in \Sigma_t$, where σ is the ith termination symbol of operation a in $x\sigma$, then G_a evaluated after the ith $\bar{a}$ in x equals σ.

Condition (iii) is called the *finite delay property* and implies that the performance of an operation cannot be indefinitely delayed, or forced to be "infinitely slower" than the performance of other operations.

An $\mathscr{I}$-computation z thereby represents a sequence of initiations and terminations of operations which is consistent with the schema control $\mathscr{T}$ and the outcome function G_a. The memory locations are read and changed by the sequence of initiations and terminations. Upon the initiation of an operation a,

[2] The transition function τ is extended in the usual manner, namely $\tau(q, y\sigma) \equiv \tau(\tau(q, y), \sigma)$.

the values in locations $D(a)$ are used to compute new values in accord with functions F_a, and to determine the outcome of a defined by G_a for this performance of operation a. Upon termination of the operation a the values computed by F_a are stored in locations $R(a)$. In this way an $\mathscr{I}$-computation z defines a sequence of contents for each cell $i \in M$, and we denote this sequence by $\Omega_i(z)$. A more detailed definition of computations and the resulting sequences of memory values is given in [1], but this description should suffice for our current purposes.

We are interested in several properties of schemata which now can be defined. A schema is called *determinate* if, whenever x and y are $\mathscr{I}$-computations for the same interpretation $\mathscr{I}$, $\Omega_i(y) = \Omega_i(z)$ for all $i \in M$. Two schemata $\mathscr{S} = (M, A, \mathscr{T})$ and $\mathscr{S}' = (M, A, \mathscr{T}')$ are called *equivalent* if for each $i \in M$ and each interpretation $\mathscr{I}$,

$$\{\Omega_i(y) | y \text{ is an } \mathscr{I}\text{-computation for } \mathscr{S}\} = \{\Omega_i(z) | z \text{ is an } \mathscr{I}\text{-computation for } \mathscr{S}'\}.$$

A schema $\mathscr{S}$ is called *finite state* if Q is a finite set. A schema $\mathscr{S}$ is called *bounded* if there is a constant K such that for every interpretation $\mathscr{I}$ any prefix x of any $\mathscr{I}$-computation has a number of initiation symbols which is no more than K greater than the number of termination symbols in x. If K can be taken as 1, then the schema is said to be *serial.*

A schema $\mathscr{S}$ is *computationally commutative* if whenever for some given interpretation $\mathscr{I}$, $x\pi\sigma$ and $x\sigma\pi$ are prefixes of $\mathscr{I}$-computations, then $\tau(q_0, x\pi\sigma) = \tau(q_0, x\sigma\pi)$. $\mathscr{S}$ is *repetition-free* if whenever an $\mathscr{I}$-computation contains two initiation symbols of the same operation, as in $v\bar{a}w\bar{a}x$, then w contains a termination symbol of an operation c for which $R(c) \cap D(a) \neq \emptyset$. Finally, an operation $a \in A$ is said to be *terminating* if $\bar{a}$ occurs only a finite number of times in each computation of $\mathscr{S}$.

Other properties of schemata being persistent, commutative, permutable, and lossless are defined in [1], and these terms appear occasionally in this paper. The definitions are omitted, however, since they are not essential to understand the results in this paper.

3. The construction for undecidability. The elements of the constructions we use are taken from [1] along with the "reset" idea of [5]. In those papers, as well as here, undecidability is proved by using the Post correspondence problem. The form of the Post correspondence problem we use can be stated as follows: Given two n-tuples $X = x_1, x_2, \cdots, x_n$ and $Y = y_1, y_2, \cdots, y_n$ of words over the alphabet $\{b_1, b_2\}$, does there exist a sequence of indices $i_1, i_2, \cdots, i_p$ such that $x_{i_1}x_{i_2} \cdots x_{i_p} = y_{i_1}y_{i_2} \cdots y_{i_p}$? This problem is denoted as $P(X, Y)$. We show that the properties we wish to study for schemata are decidable only if this class of Post correspondence problems is decidable. Since undecidability holds for this class of problems, this is sufficient to prove undecidability for the schemata properties.

For any particular Post correspondence problem $P(X, Y)$ we construct an $\mathscr{S}(X)$ and $\mathscr{S}(Y)$ as in [1].[3] For $\mathscr{S}(X)$ and $\mathscr{S}(Y)$, $M = \{1, 2\}$, $A = \{a, b\}$, $D(a)$

[3] Actually, the $\mathscr{S}(X)$ and $\mathscr{S}(Y)$ constructions are slight variations from those shown in Fig. 4.1 of [1], correcting an error there, to insure that $\bar{a}a_3\bar{b}b_3$ is not a computation that reaches an end state. Also, the sink construction of Fig. 3 differs from [1] in order to provide serial operation.

$= R(a) = \{1\}$, $D(b) = R(b) = \{2\}$, $K(a) = K(b) = 3$. Since neither operation affects the domain location of the other, the sequence of outcomes of a and b depends only on the interpretation and not on how the performances of a and b are interspersed. $\mathscr{S}(X)$ and $\mathscr{S}(Y)$ are constructed in an identical manner so we describe the construction of $\mathscr{S}(X)$ only. We say that an interpretation $\mathscr{I}$ is *consistent* with $(X; i_1, \cdots, i_p)$ if and only if:

(i) if a could be executed repeatedly, beginning with the control in state q_0 and the initial assigned contents of memory location 1, the sequence of outcomes would have as a prefix

$$a_1^{i_1-1}a_2a_1^{i_2-1}a_2 \cdots a_1^{i_p-1}a_2a_3;$$

and

(ii) if b could be executed repeatedly, beginning with state q_0 and the initial assigned contents of memory location 2, the sequence of outcomes would have the prefix

$$x_{i_1}x_{i_2} \cdots x_{i_p}b_3.$$

Thus, $\mathscr{S}(X)$ is designed so that under a consistent interpretation the outcomes of a determine a sequence of indices and the outcomes of b determine the word generated from X by this sequence of indices. The actual computation for $\mathscr{S}(X)$ under a consistent interpretation would have performances of a and b interspersed so that the sequence of outcomes would be

$$a_1^{i_1-1}a_2x_{i_1}a_1^{i_2-1}a_2x_{i_2} \cdots a_1^{i_p-1}a_2x_{i_p}a_3b_3.$$

The control for $\mathscr{S}(X)$ to accomplish this is sketched in Fig. 1. If, for example, $x_1 = b_2b_1$, then the sequence of initiations and terminations of b that loop from

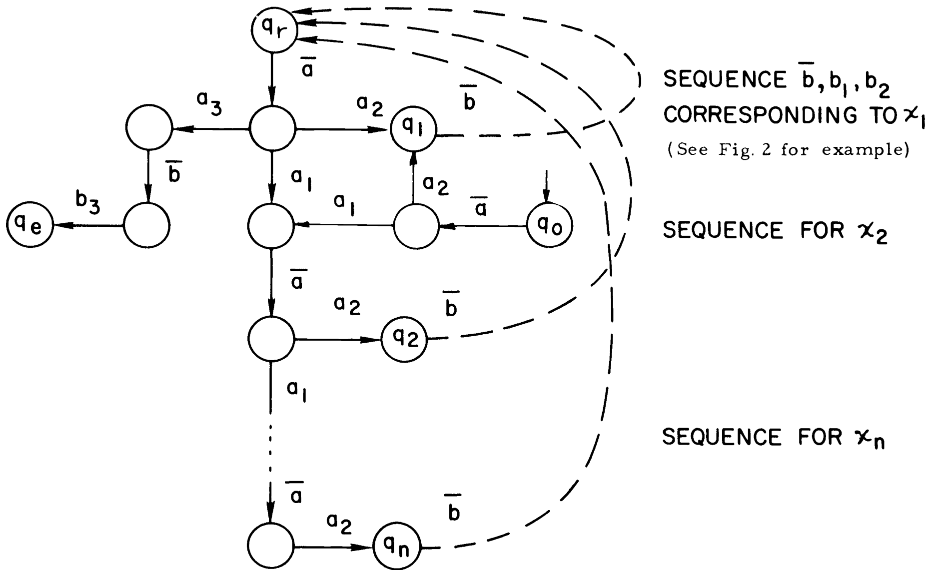


FIG. 1. *Sketch of* $\mathscr{S}(X)$ *control*

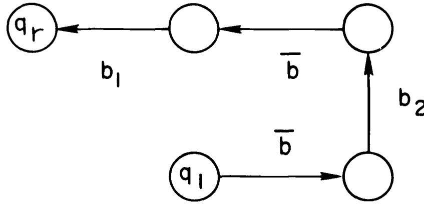


FIG. 2. *Example of x_1 representation*

q_1 to q_r are as shown in Fig. 2. For termination symbol transitions not shown in Figs. 1 and 2, the $\mathscr{S}(X)$ transitions are all assumed to go to the sink entry state of the "sink" construction of states shown in Fig. 3. In Fig. 3 the termination symbol transitions not shown are also assumed to enter the entry state of this sink construction.

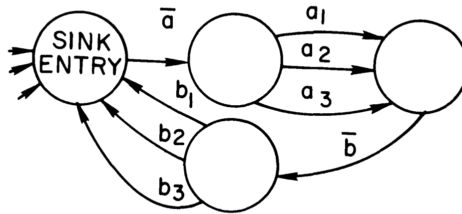


FIG. 3. *"Sink" construction*

From the construction of $\mathscr{S}(X)$ it is readily seen that for any pair $(X; i_1, i_2, \cdots, i_p)$ and interpretation $\mathscr{I}$, state q_e is reached in $\mathscr{S}(X)$ if and only if $\mathscr{I}$ is consistent with $(X; i_1, i_2, \cdots, i_p)$. If $\mathscr{I}$ is not consistent with $(X; i_1, i_2, \cdots, i_p)$, then each $\mathscr{I}$-computation reaches the "sink" and is infinite in length.

Now consider schema $\mathscr{S}(XY)$ depicted in Fig. 4. In schema $\mathscr{S}(XY)$ we let $M = \{0, 1, 2\}$, $D(r) = 0$, $R(r) = \{1, 2\}$ and operations a and b be defined as before for $\mathscr{S}(X)$ and $\mathscr{S}(Y)$. For convenience we denote state q_e of $\mathscr{S}(Y)$ in $\mathscr{S}(XY)$ as q_e^*. Note that the first performance of operation r, preceding $\mathscr{S}(X)$, initializes locations 1 and 2. The interpretation is consistent with $(X; i_1, i_2, \cdots, i_p)$ for some $(i_1, i_2, \cdots, i_p)$ if and only if the computation reaches state q_e in the $\mathscr{S}(X)$ part of $\mathscr{S}(XY)$. In this event operation r is performed a second time, and state q_0 of $\mathscr{S}(Y)$ is entered. The second performance of r resets locations 1 and 2 to the same values that they had upon entering $\mathscr{S}(X)$ (since F_r is single-valued, and location 0 is never changed). Thus, the sequence of outcomes for a in $\mathscr{S}(Y)$ must have the same prefix as the sequence that occurred in $\mathscr{S}(X)$; in particular the outcomes must have the prefix $a_1^{i_1-1}a_2a_1^{i_2-1}a_2 \cdots a_1^{i_p-1}a_2a_3$. Now, the interpretation is also consistent with Y if and only if state q_e^* of $\mathscr{S}(XY)$ is reached. Thus q_e^* is reached in some computation if and only if there is a solution to the Post correspondence problem $P(X, Y)$. From the $\mathscr{S}(XY)$ construction we see that the accessibility of states is undecidable for this kind of schemata. This is one of the main results of Itkin and Zwinogrodski [5], and the $\mathscr{S}(XY)$ construction is, in essence, the

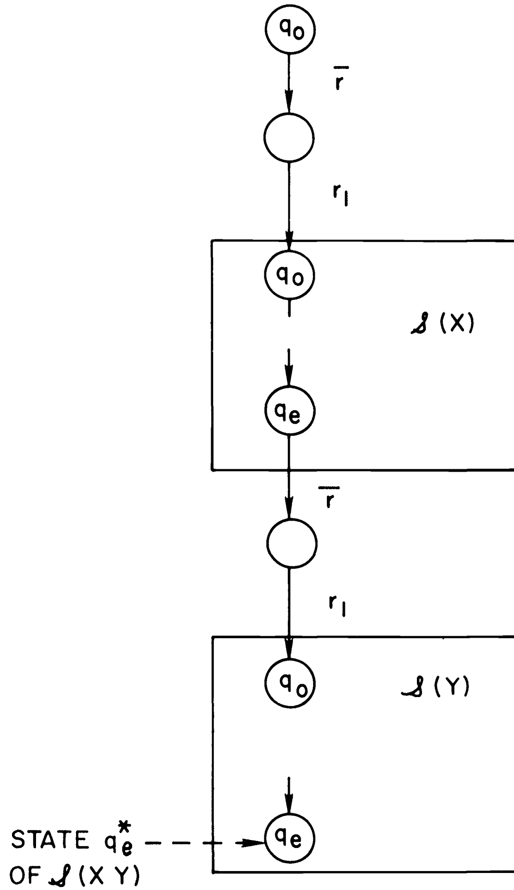


FIG. 4. *Schema $\mathcal{S}(XY)$ with reset operation r*

construction they use. They call a schema *one-valued* if for each interpretation $\mathcal{I}$ the schema has only one $\mathcal{I}$-computation. They note that $\mathcal{S}(XY)$ is clearly one-valued, so the state accessibility problem is undecidable for one-valued schemata. They also show that since a finite computation results only when q_e^* is reached, the equivalence of one-valued schemata is undecidable [5]. This follows easily by constructing a schema $\mathcal{S}^i(XY)$ which is identical to $\mathcal{S}(XY)$ except it loops upon entering state q_e^* in $\mathcal{S}^i(XY)$. Then $\mathcal{S}(XY)$ and $\mathcal{S}^i(XY)$ are equivalent only if this state is not reached. To maintain the one-valued nature of the schemata a looping construction as in Fig. 3 is required from state q_e^* of $\mathcal{S}^i(XY)$.

Before we prove some additional undecidable results in the next section, we show that any schema of the $\mathcal{S}(XY)$ variety also satisfies some of the other simple properties of schemata.

Since $\mathcal{S}(X)$ and $\mathcal{S}(Y)$ are formed from n-tuples of finite words over the alphabet (b_1, b_2) as described in Figs. 1, 2 and 3, the number of states in $\mathcal{S}(X)$ and $\mathcal{S}(Y)$ is finite. Thus, $\mathcal{S}(XY)$ is a finite state schema. Also, $\mathcal{S}(XY)$ is constructed

in such a way that from any state q with $\tau(q, \sigma)$ defined, where $\sigma \in \Sigma_i$, for all $\pi \in \Sigma_i$ and $\pi \neq \sigma$, $\tau(q, \pi)$ is undefined. Thus, at most one operation can be initiated at any state. Also, if $\tau(q, \sigma) = q'$ for $\sigma \in \Sigma_i$, then $\tau(q', \pi)$ is undefined for any $\pi \in \Sigma_i$. Thus, once an operation is initiated, it must be terminated before any other operation can be initiated. This leads not only to the one-valued property of $\mathscr{S}(XY)$ but also to the fact that $\mathscr{S}(XY)$ is serial, determinate, permutable, persistent and computationally commutative. Moreover, $\mathscr{S}(XY)$ is lossless since $R(a)$, $R(b)$ and $R(r)$ are not empty. Obviously $\mathscr{S}(XY)$ is not repetition-free since interpretations can be devised to make $\mathscr{S}(XY)$ enter q_e in $\mathscr{S}(X)$ forcing operation r to be performed twice. This gives a computation of the form $v\bar{r}w\bar{r}x$ with $R(c) \cap D(r) = \varnothing$ for each operation c of $\mathscr{S}(XY)$ since $D(r) = 0$ and 0 is not an element of $R(a)$, $R(b)$ or $R(r)$. It can also be shown that $\mathscr{S}(XY)$, although computationally commutative, is not commutative.

Finally, we wish to show that any $\mathscr{S}(XY)$ can also be represented by a counter schema. A *counter schema* [1] is a schema whose control $\mathscr{T}$ is specified by: a nonnegative integer k (the number of counters); a finite set S with a distinguished element s_0; an initial vector $\Pi \in N^k$, where N denotes the nonnegative integers; a function v from the alphabet Σ into N^k such that $\sigma \in \Sigma_t$ implies that $v(\sigma) \geqq 0$; and a partial function $\theta: S \times \Sigma \to S$ which is total on $S \times \Sigma_t$. Now for the control $\mathscr{T} = (Q, q_0, \Sigma, \tau)$ of the counter schema, the set of states $Q = S \times N^k$, $q_0 = (s_0, \Pi)$, and $\tau((s, x), \sigma)$ is defined if $\theta(s, \sigma)$ is defined and $x + v(\sigma) \geqq 0$; when defined, $\tau((s, x), \sigma) = (\theta(s, \sigma), x + v(\sigma))$. To show that $\mathscr{S}(XY)$ can be represented in the counter schema form we note the following:

(i) Since the constructions of Figs. 1, 2 and 3 give a finite state structure we can let this finite state structure be S in a counter schema. The transitions then become the partial function $\theta: S \times \Sigma \to S$ which is total on $S \times \Sigma_t$.

(ii) We can let $k = 0$, so there are no counters and the schema $\mathscr{S}(XY)$ satisfies, in a degenerate way, the requirements of a counter schema.

This demonstration that $\mathscr{S}(XY)$ can be represented by a counter schema is a simple result of the fact that any finite state schema can be represented by a counter schema with $k = 0$ counters. This is also used in the next section to obtain counter schemata for variants of $\mathscr{S}(XY)$.

To summarize we note that $\mathscr{S}(XY)$ is a finite state, one-valued, serial, determinate, permutable, persistent, computationally commutative, lossless, counter schema. In the proofs of the next section we are interested primarily in the fact that $\mathscr{S}(XY)$ is a counter schema.

4. The undecidability theorems. We shall use $\mathscr{S}(XY)$ and minor variations to prove several new undecidability results. These theorems use the fact that it is undecidable whether, under some interpretation, state q_e^* of $\mathscr{S}(XY)$ is reached or not. Through this result other properties are shown to be undecidable by adding suitable constructions onto $\mathscr{S}(XY)$ at q_e^*.

THEOREM 1. *It is undecidable whether a given counter schema is computationally commutative, one-valued or serial.*[4]

[4] Since $\mathscr{S}^{ii}(XY)$ in the proof is determinate, permutable, persistent, lossless, and finite state, Theorem 1 also holds for schemata with these properties.

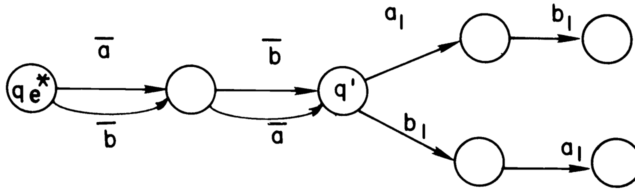


FIG. 5. *A simple structure which is not computationally commutative*

Proof. Construct for any $\mathscr{S}(XY)$ a schema $\mathscr{S}^{ii}(XY)$ which is identical to $\mathscr{S}(XY)$ except that a noncommutative behavior emanates from q_e^*. A simple structure of this type is shown in Fig. 5. Since this addition to $\mathscr{S}(XY)$ is finite state, $\mathscr{S}^{ii}(XY)$ is also finite state and therefore a counter schema. If x is a prefix of an $\mathscr{I}$-computation for $\mathscr{S}^{ii}(XY)$ that reaches state q_e^*, then both $x\bar{a}\bar{b}a_1b_1$ and $x\bar{a}\bar{b}b_1a_1$ are $\mathscr{I}$-computations, where $\tau(q_0, x\bar{a}\bar{b}) = q'$. Since $\tau(q', a_1b_1)$ and $\tau(q', b_1a_1)$ are both defined in $\mathscr{I}$-computations and unequal, $\mathscr{S}^{ii}(XY)$ is not computationally commutative if and only if q_e^* is reached. Similarly $\mathscr{S}^{ii}(XY)$ is not one-valued or serial if and only if q_e^* is reached. The theorem follows.

THEOREM 2. *It is undecidable whether a given counter schema is bounded.*[5]

Proof. Construct for any $\mathscr{S}(XY)$ a new schema $\mathscr{S}^{iii}(XY)$ which upon entering state q_e^* has unbounded behavior. This can be done by adding an operation c which loops on state q_e^*. It suffices to let $D(c) = 1$, $R(c) = 2$, $K(c) = 1$, where

$$\tau(q, \bar{c}) = \begin{cases} q_e^* & \text{for } q = q_e^*, \\ \text{undefined}, & \text{otherwise}, \end{cases}$$

and

$$\tau(q, c_1) = \begin{cases} q_e^* & \text{for } q = q_e^*, \\ \text{sink entry state}, & \text{otherwise}. \end{cases}$$

Now any schema of the form $\mathscr{S}^{iii}(XY)$ is unbounded if and only if q_e^* of $\mathscr{S}^{iii}(XY)$ is accessible. Also $\mathscr{S}^{iii}(XY)$ is finite state and thereby a counter schema.

Note that this construction also shows the undecidability of such schemata for the serial and one-valued properties.

THEOREM 3. *It is undecidable for a counter schema whether a given operation* $a \in A$ *is terminating.*[6]

Proof. Consider another variation $\mathscr{S}^{iv}(XY)$ of $\mathscr{S}(XY)$. In $\mathscr{S}^{iv}(XY)$ let all but the structure following q_e^* and the sink structure be the same as $\mathscr{S}(XY)$. But let the sink structure be a single state q_s, where $\tau(q_s, \bar{a})$, $\tau(q_s, \bar{r})$ and $\tau(q_s, \bar{b})$ are undefined but $\tau(q_s, \sigma) = q_s$ for each termination symbol. For the structure following q_e^* use the sink construction of Fig. 3 with the sink entry state being q_e^*. Then for $\mathscr{S}^{iv}(XY)$,

[5] Since $\mathscr{S}^{iii}(XY)$ in the proof is finite state, determinate, permutable, persistent, computationally commutative and lossless, Theorem 2 also holds for schemata with these properties.

[6] Since $\mathscr{S}^{iv}(XY)$ is finite state, one-valued, serial, determinate, permutable, persistent, computationally commutative, and lossless, Theorem 3 also holds for schemata with these properties.

operations a and b are terminating if and only if q_e^* is not entered. Note that $\mathscr{S}^{iv}(XY)$ is clearly a counter schema.

It is interesting to note that one sees directly from $\mathscr{S}(XY)$ that the question of the existence of a finite computation is undecidable; this is similar to the construction used in [5]. From $\mathscr{S}^{iv}(XY)$, we obtain a different result, namely, that the question of the existence of an infinite computation is undecidable.

THEOREM 4. *It is undecidable whether a given finite state counter schema is determinate.*[7]

Proof. Construct, for any $\mathscr{S}(XY)$, a new schema $\mathscr{S}^v(XY)$ which upon entering state q_e^* has indeterminate behavior. A simple indeterminate attachment is shown in Fig. 6, where m and n are two operations in $\mathscr{S}^v(XY)$ that are not in $\mathscr{S}(XY)$ and $R(m) \cap R(n) \neq \emptyset$. For example let $D(m) = D(n) = 0$ and $R(m) = R(n) = \{1, 2\}$. Then $\mathscr{S}^v(XY)$ is indeterminate if and only if q_e^* is reached in some computation, and this is undecidable. Clearly $\mathscr{S}^v(XY)$ is a finite state counter schema.

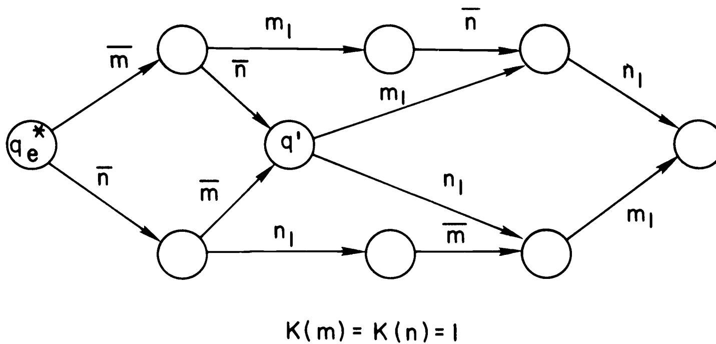


FIG. 6. *An indeterminate construction*

By a simple addition of an operation which is performed exactly once only when q_e^* of $\mathscr{S}(XY)$ is reached we can obtain the related result.

THEOREM 5. *It is undecidable for a given counter schema whether, for a given operation c, any computation exists containing $\bar{c}$.*[8]

5. Comparison of undecidability and decidability theorems. In [1] a number of results were given showing the decidability of commutativity, boundedness, termination, and determinacy for repetition-free counter schemata. The results

[7] Since $\mathscr{S}^v(XY)$ in the proof is permutable, persistent, computationally commutative and lossless, Theorem 4 also holds for schemata with these properties. It is also readily seen that the properties "persistent" and "permutable" can be removed and replaced by the term "serial." An appropriate construction is formed by eliminating q' and its incident edges from Fig. 6. Although this leads to a serial schema, it is not necessarily one-valued.

[8] As should be clear, Theorem 5 also holds for schemata that are finite state, one-valued, serial, determinate, permutable, persistent, computationally commutative and lossless.

of the previous section yield undecidability results for these properties. By comparing these results we show that deletion of repetition-freeness from the hypothesis of the decidability theorems changes each problem to an undecidable one. In particular it is decidable for any given repetition-free counter schema whether the schema is computationally commutative,[9] bounded, or a given operation $a \in A$ is terminating [1, Theorems 4.5–4.7]. Also, Theorem 4.9 of [1] states that it is decidable whether a repetition-free, lossless, persistent, commutative, counter schema is determinate. Theorems 1 through 4 show that without repetition-freeness these properties are undecidable. Thus, in a sense, we are "close" to the borderline between decidability and undecidability. Also, these results are as tight as can be expected since only the one hypothesis is removed.

It is interesting to note that the schema $\mathscr{S}(XY)$, which is the basis for the undecidability results, is itself very reliant on the repetitive character of the "reset" operation. In particular, operation r is the only repetitive operation of $\mathscr{S}(XY)$ and is performed at most twice in any computation. Thus $\mathscr{S}(XY)$ is in some sense minimally repetitive since only one operation can be repetitive and this operation can be repeated only once.

The properties noted in the footnotes of the previous section, or any combination of them, could be added back as hypotheses in the respective undecidability results. Of course, the properties could also be added into the hypotheses of the decidability theorems since adding further constraints to the hypotheses of the decidability theorems could only tend to simplify the problem further. This thereby gives a family of comparable pairs of theorems with whatever combination of constraining hypotheses, consistent with those noted, are desired.

Since it is decidable whether a given counter schema is repetition-free [1, Theorem 4.4], we can now see rather clearly the importance of repetition-freeness in schemata. A natural question for further study arises from these results as to whether similar results can be obtained by deletion of one or more of the other constraining hypotheses in the decidable theorems. For example, does undecidability result if either persistence or computational commutativity are deleted from the determinacy theorem? Since $\mathscr{S}(XY)$ is repetitive it is clear that a different basic construction would be required to answer these questions.

Acknowledgment. The author is grateful to Dr. Arnold L. Rosenberg for his careful reading and detailed comments on an earlier version of this paper.

REFERENCES

[1] RICHARD M. KARP AND RAYMOND E. MILLER, *Parallel program schemata*, J. Comput. System Sci., 3 (1969), pp. 147–195.

[2] I. I. IANOV, *The logical schemes of algorithms*, Problems of Cybernetics, 1 (1958), pp. 75–127.

[3] J. D. RUTLEDGE, *On Ianov's program schemata*, J. Assoc. Comput. Mach., 11 (1964), pp. 1–9.

[4] D. C. LUCKHAM, D. M. R. PARK AND M. S. PATERSON, *On formalized computer programs*, J. Comput. System Sci., 4 (1970), pp. 220–249.

[9] In [1] the decidability result was shown for commutativity, but by inspecting the proof of this theorem it becomes evident that the result also holds for this new and weaker notion of computational commutativity.

[5] V. E. ITKIN AND Z. ZWINOGRODSKI, *On the program schemata equivalence*, Rep., Computing Center of the Siberian Branch of the USSR Academy of Sciences, Novosibirsk 90, USSR, 1970.

[6] ARNOLD L. ROSENBERG, *Multitape finite automata with rewind instructions*, J. Comput. System Sci., 1 (1967), pp. 299–315.

SIAM J. COMPUT.
Vol. 1, No. 2, June 1972

THE TRANSITIVE REDUCTION OF A DIRECTED GRAPH*

A. V. AHO,† M. R. GAREY† AND J. D. ULLMAN‡

Abstract. We consider economical representations for the path information in a directed graph. A directed graph G^t is said to be a transitive reduction of the directed graph G provided that (i) G^t has a directed path from vertex u to vertex v if and only if G has a directed path from vertex u to vertex v, and (ii) there is no graph with fewer arcs than G^t satisfying condition (i). Though directed graphs with cycles may have more than one such representation, we select a natural canonical representative as the transitive reduction for such graphs. It is shown that the time complexity of the best algorithm for finding the transitive reduction of a graph is the same as the time to compute the transitive closure of a graph or to perform Boolean matrix multiplication.

Key words and phrases. Directed graph, binary relation, minimal representation, transitive reduction, algorithm, transitive closure, matrix multiplication, computational complexity.

1. Introduction. Given a directed graph G, one is often interested in knowing whether there is a path from one vertex to another in that graph. In many cases it is possible to represent this information by another directed graph that has fewer arcs than the given graph. Informally, we say that a graph G^t is a transitive reduction of the directed graph G whenever the following two conditions are satisfied:

(i) there is a directed path from vertex u to vertex v in G^t if and only if there is a directed path from u to v in G, and

(ii) there is no graph with fewer arcs than G^t satisfying condition (i).

Such minimal representations for graphs are of particular interest for efficiently executing certain computer algorithms, such as the precedence constrained sequencing algorithms of [1] and [2], whose operation is partially determined by an input-specified transitive relation. In particular, these minimal representations may require less computer memory for storage and, depending upon the precise nature of the algorithm, may also lead to a reduced execution time.

In this paper, we mathematically characterize the transitive reduction and provide an efficient algorithm for computing the transitive reduction of any given directed graph. Furthermore, we show that the computational complexity of computing a transitive reduction is equivalent to the computational complexity of computing a transitive closure or performing a Boolean matrix multiplication.

In [3], the *minimum equivalent* of a directed graph G is defined as a smallest subgraph G' of G such that there is a path from vertex u to vertex v in G' whenever there is a path from u to v in G. Our notion of transitive reduction is similar, but with the important exception that we do not require a transitive reduction to be a subgraph of the original graph. The two notions give rise to the same reduced representation when the original graph is acyclic. However, the transitive reduction of a graph G with cycles can be smaller and much easier to find than a minimal equivalent graph for G.

* Received by the editors August 9, 1971, and in revised form November 15, 1971.

† Bell Telephone Laboratories, Inc., Murray Hill, New Jersey 07974.

‡ Department of Electrical Engineering, Princeton University, Princeton, New Jersey 08540.
The work of this author was supported in part by the National Science Foundation under Grant GJ-1052.

2. Definitions and basic results. A *directed graph* G on the set of vertices $V = \{v_1, v_2, \cdots, v_n\}$ is a subset of $V \times V$, the members of G being called *arcs*. A *directed path* in G from vertex u to vertex v is a sequence of distinct arcs $\alpha_1, \alpha_2, \cdots, \alpha_p, p \geqq 1$, such that there exists a corresponding sequence of vertices $u = v_0, v_1, v_2, \cdots, v_p = v$ satisfying $\alpha_k = (v_k, v_{k+1}) \in G$, for $0 \leqq k \leqq p - 1$. A *cycle* is a directed path beginning and ending at the same vertex which passes through at least one other vertex. A *simple* cycle is a cycle which passes through no vertex more than once. A *loop* is an arc of the form (v, v). A graph will be called acyclic if and only if it contains no cycles. Notice that this differs slightly from conventional usage, since we do allow an acyclic graph to contain loops.

A graph G is said to be *transitive* if, for every pair of vertices u and v, not necessarily distinct, $(u, v) \in G$ whenever there is a directed path in G from u to v. The *transitive closure* G^T of G is the least subset of $V \times V$ which contains G and is transitive.

THEOREM 1. *For any finite acyclic directed graph G, there is a unique graph G^t with the property that $(G^t)^T = G^T$ and every proper subset H of G^t satisfies $H^T \neq G^T$. The graph G^t is given by*

$$G^t = \bigcap_{G_i \in S(G)} G_i,$$

where $S(G) = \{G_1 | G_1^T = G^T\}$.

Proof. The proof of Theorem 1 follows from the following two lemmas. (Note that Lemma 1 is actually a straightforword consequence of Theorem 1 in [3].)

LEMMA 1. *Let G_1 and G_2 be any two finite acyclic directed graphs (on the same vertex set) satisfying $G_1^T = G_2^T$. If there exists an arc $\alpha \in G_1$ such that $\alpha \notin G_2$, then $(G_1 - \{\alpha\})^T = G_1^T = G_2^T$.*

Proof. Let $\alpha = (u, v)$ be as described in the hypothesis of the lemma. Since $G_1^T = G_2^T$ and $\alpha \notin G_2, G_2$ must contain a path from u to v passing through some other vertex, say w. Then G_1 must contain a directed path from u to w and a directed path from w to v. If the path from u to w in G_1 includes arc α, then G_1 contains a path from v to w. But, since G_1 also contains a path from w to v, this contradicts G_1 being acyclic. If the path from w to v in G_1 includes arc α, then G_1 contains a path from w to u. But, since G_1 contains a path from u to w, this also contradicts G_1 being acyclic. Thus, G_1 contains a directed path from u to w and from w to v, which does not include arc α. Therefore, $G_1 - \{\alpha\}$ contains a path from u to v, so $(G_1 - \{\alpha\})^T = G_1^T = G_2^T$.

LEMMA 2. *Let G be any finite acyclic directed graph. Then the set $S(G) = \{G_1 | G_1^T = G^T\}$ is closed under union and intersection.*

Proof. Let G_1 and G_2 be any two members of $S(G)$. Since $G_1^T = G_2^T = G^T$, $G_1 \cup G_2 \subseteq G^T$. Because G^T is transitive, we then have $(G_1 \cup G_2)^T \subseteq G^T$. Furthermore, G_1 is contained in $G_1 \cup G_2$, so $G_1^T = G^T \subseteq (G_1 \cup G_2)^T$. Therefore, $(G_1 \cup G_2)^T = G^T$ and $(G_1 \cup G_2) \in S(G)$.

Now let $\{\alpha_1, \alpha_2, \cdots, \alpha_r\} = G_1 - (G_1 \cap G_2)$. By repeated application of Lemma 1, we have

$$(G_1 - \{\alpha_1\})^T = G_1^T,$$

$$(G_1 - \{\alpha_1\} - \{\alpha_2\})^T = G_1^T,$$

$$\vdots$$

$$(G_1 - \{\alpha_1\} - \{\alpha_2\} - \cdots - \{\alpha_r\})^T = G_1^T.$$

But the last equation merely says that $(G_1 \cap G_2)^T = G_1^T = G^T$, so $(G_1 \cap G_2) \in S(G)$.

Since $S(G)$ is a finite set, Theorem 1 is obtained as a straightforward application of Lemma 2.

Theorem 1 shows that the intuitive definition of transitive reduction actually yields a unique graph for any finite acyclic directed graph. Furthermore, the transitive reduction of any such graph G can be obtained by successively examining the arcs of G, in any order, and deleting those arcs which are "redundant," where an arc $\alpha = (u, v)$ is redundant if the graph contains a directed path from u to v which does not include α.

We now extend this analysis to graphs which contain cycles. Consider the graph $G = \{v_1, v_2), (v_2, v_3), (v_3, v_2), (v_2, v_1)\}$ of Fig. 1(a). If H is any proper subset of G, then $H^T \neq G^T$. Thus, G is its own minimum equivalent graph. However, the graph $G_1 = \{(v_1, v_2), (v_2, v_3), (v_3, v_1)\}$ of Fig. 1(b) contains only three arcs and has the same transitive closure as G, as does the graph $G_2 = \{(v_1, v_3),$

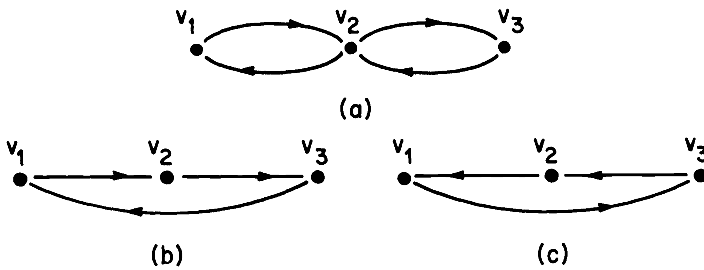


FIG. 1. *Graphs with cycles*

$(v_3, v_2), (v_2, v_1)\}$ of Fig. 1(c). No other graph with three or fewer arcs has the same transitive closure as G. Since $G_1 \neq G_2$, we see that for graphs with cycles there may not be a unique graph, with fewest arcs, having the same transitive closure as a given graph. Thus, Theorem 1 cannot be simply extended to encompass all finite directed graphs. This example also shows that the lemmas cannot be similarly extended. Furthermore, since neither G_1 nor G_2 is a subset of G, it may also be possible that no such minimal graph can be constructed by removing certain arcs from the given graph, as was the case for acyclic graphs. However, we shall show that all such minimal graphs, for any specific given graph, must have similar structure, and based on this result we choose a unique representative to be the unique transitive reduction.

Two vertices u and v of a directed graph G will be called *equivalent* if either $u = v$ or G contains a cycle which is incident with both u and v. Given any finite directed graph G, we say that G_1 is the *equivalent acyclic graph* for G when the vertices of G_1 are the vertex equivalence classes of G, denoted by E_k, and the arcs of G_1 satisfy: $(E_i, E_j) \in G_1$ if and only if there exists an arc $(u, v) \in G_1$ such that $u \in E_i$ and $v \in E_j$. If G_1 is the equivalent acyclic graph for G, and G_2 is a subset of G_1, then the graph G_3 is a *cyclic expansion* of G_2 if and only if:

(i) G_3 has the same vertices as G;

(ii) G_2 is the equivalent acyclic graph for G_3;

(iii) for each multimember vertex equivalence class E_i of G, G_3 contains a simple cycle incident with all vertices in E_i and G_3 contains no other arcs between members of E_i; and

(iv) for each arc $(E_i, E_j) \in G_2$, $E_i \neq E_j$, there is exactly one arc $(u, v) \in G_3$ satisfying $u \in E_i$ and $v \in E_j$.

A cyclic expansion simply replaces the loop and vertex corresponding to a multimember equivalence class by a simple cycle through the members of that equivalence class, with each arc between different equivalence classes transformed into a similarly directed single arc between some pair of vertices, one from each of the two equivalence classes.

THEOREM 2. *Given any finite directed graph G, let G_1 be its equivalent acyclic graph and G_1^t be the unique "transitive reduction" of G_1 given by Theorem 1. Then the directed graph H satisfies $H^T = G^T$ and has the fewest arcs of any such graph if and only if H is a cyclic expansion of G_1^t.*

Proof. The proof follows directly from the following lemmas, where G, G_1, and G_1^t are defined as above.

LEMMA 3. *If H_1 is the equivalent acyclic graph for a directed graph H satisfying $H^T = G^T$, then $H_1^T = G_1^T$.*

Proof. Since $H^T = G^T$, H contains a path from u to v if and only if G contains a path from u to v. Then H and G must have the same vertex equivalence classes, so H_1 and G_1 are on identical vertex sets. If H_1^T contains an arc (E_i, E_j) not contained in G_1^T, then H_1 contains a path from E_i to E_j and G_1 contains no such path. But then H contains a path from some $u \in E_i$ to some $v \in E_j$ and G contains no such path, a contradiction. Similarly, every arc of G_1^T is an arc of H_1^T, so $G_1^T = H_1^T$.

LEMMA 4. *If H has equivalent acyclic graph H_1 satisfying $H_1^T = G_1^T$, then H is either a cyclic expansion of G_1^T or H contains more arcs than any cyclic expansion of G_1^t.*

Proof. If $H_1^T = G_1^T$, we know from Theorem 1 that H_1 must contain G_1^t. Then H must contain at least one arc for each arc of H_1, i.e., if $(E_i, E_j) \in H_1$, there exist $u \in E_i$ and $v \in E_j$ such that $(u, v) \in H$. Furthermore, H must also contain enough arcs to ensure that every pair of vertices belonging to the same equivalence class lie on a cycle in H. But this requires at least as many arcs as there are members in the equivalence class, except for single member classes, and such a minimal number of arcs is used if and only if the only arcs between members of the equivalence class form a single cycle including exactly all members of the class. The definition of a cyclic expansion was chosen precisely to include all and only those graphs which use such a minimal number of arcs. Thus, H must either be a cyclic expansion of G_1^t or must have more arcs than any cyclic expansion of G_1^t.

LEMMA 5. *Every cyclic expansion of G_1^t has the same number of arcs.*

Proof. The number of arcs in any cyclic expansion of G_1^t is exactly equal to the number of arcs in G_1^t plus the number of vertices of G which belong to multimember vertex equivalence classes, minus the number of multimember vertex equivalence classes in G.

LEMMA 6. *If H is a cyclic expansion of G_1^t, then $H^T = G^T$.*

Proof. If $(u, v) \in G^T$, G contains a path from u to v. If u and v belong to the same vertex equivalence class, H must contain a path from u to v so $(u, v) \in H^T$.

If u and v belong to different equivalence classes, $u \in E_i$, $v \in E_j$, G_1 contains a path from E_i to E_j. But then G_1^t contains a path from E_i to E_j, so H contains a path from u to v and $(u, v) \in H^T$. Similarly, if $(u, v) \notin G^T$, G contains no path from u to v. Also, u and v belong to different vertex equivalence classes, $u \in E_i$, $v \in E_j$, and G_1 contains no path from E_i to E_j. But then G_1^t contains no path from E_i to E_j, and H cannot contain a path from u to v, so $(u, v) \notin H^T$.

This completes the proof of Theorem 2.

Theorem 2 tells us that if G_1 is the equivalent acyclic graph for G, then every cyclic expansion of the graph G_1^t given by Theorem 1 will satisfy our original intuitive definition for a transitive reduction of G. In fact, for most algorithms requiring such a transitively reduced graph, the most useful representation will simply be G_1^t along with the corresponding vertex equivalence classes of G. However, we also choose to select a unique representative from the various cyclic expansions of G_1^t to be defined as the transitive reduction of G.

Let the vertices of G be arbitrarily ordered by assigning them indices as $v_1, v_2, \cdots, v_n$. If G_1 is the equivalent acyclic graph for G and G_1^t is the "transitive reduction" of G_1 given by Theorem 1, then the *canonical cyclic expansion* of G_1^t is the unique cyclic expansion G_2 of G_1^t satisfying:

(i) If $(v_i, v_j) \in G_2$, $v_i \in E_k$, $v_j \in E_k$, and $v_i \neq v_j$, then either $j > i$ and none of $v_{i+1}, \cdots, v_{j-1}$ is in E_k or v_i has the largest index in E_k and v_j has the smallest index in E_k; and

(ii) For each arc $(E_i, E_j) \in G_1^t$, $E_i \neq E_j$, there is an arc in G_2 from the least index member of E_i to the least index member of E_j.

The canonical cyclic expansion merely expands each loop and vertex corresponding to a multimember equivalence class into an ordered simple cycle, with all arcs between equivalence classes transformed into arcs between the least members of the equivalence classes.

We then define the *transitive reduction* of a finite directed graph G to be the unique graph G^t which satisfies:

(i) $(G^t)^T = G^T$;

(ii) If $H^T = G^T$, then H contains at least as many arcs as G^t; and

(iii) If G is not acyclic, then G^t is the canonical cyclic expansion of the transitive reduction of the equivalent acyclic graph for G.

Existence and uniqueness of G^t follow from the previous results.

We do not attempt a definition of transitive reduction for graphs having infinite vertex sets. However, we point out that additional complications do arise for infinite graphs. Some of these difficulties are illustrated by attempting a reasonable definition of transitive reduction for (i) the countably infinite graph with arcs in both directions between every pair of vertices, and (ii) the infinite graph having a vertex for each real number with an arc from i to j if and only if $i < j$. In neither case does there exist a graph, having the same transitive closure as the given graph, such that no proper subset of that graph also has this property.

3. Computational complexity of the transitive reduction operation. We now turn to the question of how quickly the transitive reduction of a graph can be computed. In what follows, we assume that a graph G is represented by its *adjacency*

matrix, the matrix with a 1 in row i and column j if there is an arc from the ith vertex to the jth vertex and a 0 there otherwise. Our results clearly apply to any other graph representation that can be converted to an adjacency matrix and back in time $O(n^2)$, and in which transitive reduction takes $O(n^2)$ time.[1]

We proceed to demonstrate that under the above assumption, the number of steps of a random access computer (e.g., see [4]) needed to compute the transitive reduction of a graph with n vertices differs by at most a constant factor from the time needed to perform Boolean matrix multiplication or to compute the transitive closure of a graph. It should be noted that it was shown in [5] that multiplication of $n \times n$ Boolean matrices requires time which is at most a constant factor more than the time to compute the transitive closure of an n vertex graph, and the converse was shown in [6], [7].

THEOREM 3. *If there is an algorithm to compute the transitive closure of an n vertex graph in time $O(n^\alpha)$, then there is an algorithm to compute transitive reduction in time $O(n^\alpha)$.*

Proof. We can compute the transitive reduction of a graph G with n vertices as follows.

1. Find G_1, the equivalent acyclic graph of G.
2. Let G_2 be formed from G_1 by deleting loops.
3. Let M_1 be the incidence matrix of G_2, and let M_2 be the incidence matrix of G_2^T.
4. Compute $M_3 = M_1 M_2$, and let G_3 be the graph whose incidence matrix is M_3.
5. Then G_1^t is $G_1 - G_3$.
6. Let G^t be the canonical cyclic expansion of G_1^t.

It should be evident that steps 1, 2, 5 and 6 require $O(n^2)$ time. (See [8], e.g., for step 1.) Step 3 requires $O(n^\alpha)$ time to compute G_2^T. Step 4 requires $O(n^\alpha)$ time by [4]. Thus, the entire algorithm requires $O(n^\alpha)$ time, since $\alpha \geqq 2$ (see [4]).

It remains to show that $G_1^t = G_1 - G_3$. By Theorem 1, arc (u, v) is in G_1^t if and only if $(u, v) \in G_1$, and there is no path from u to v which does not include arc (u, v). Such a path exists if and only if there is some w not equal to u or v such that there is an arc (u, w) and a path from w to v in G_1. These are exactly the conditions under which there will be an arc (u, v) in G_3.

THEOREM 4. *If transitive reduction requires $O(n^\alpha)$ steps, $\alpha \geqq 2$, on a graph of n nodes, then transitive closure requires $O(n^\alpha)$ steps.*

Proof. Let G be a graph of n vertices. Construct a graph G' with nodes u, u' and u'' for each vertex u of G. The arcs of G' are the following.

1. If (u, v) is in G, it is in G'.
2. (u', u) and (u, u'') are in G' for each vertex u of G.
3. $(u', v'') \in G'$ for all vertices u and v of G. G' is shown in Fig. 2.

We observe that (u', v'') is in $(G')^t$ if and only if (u, v) is not in G^T. That is, since no arc enters u' or leaves v'', both u' and v'' are in vertex equivalence classes of their own. By Theorems 1 and 2, (u', v'') is in any transitive reduction of G' if and only if

[1] It may appear that we have defined away the problem. However, a little thought will suffice to conclude that in any reasonable representation, a transitive reduction algorithm will have to "look at" all the arcs and thus take at least time $O(n^2)$.

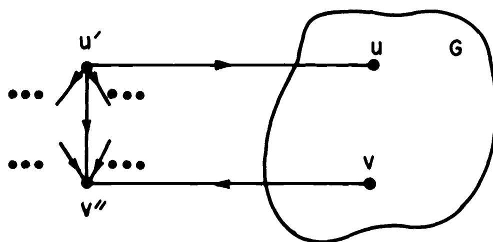


FIG. 2. *Construction of* G'

there is no path of length greater than one from u' to v''. But such a path is seen to exist if and only if there is a path from u to v in G.

Thus, we may compute G^T by the following algorithm.

1. Construct G'.
2. Compute $(G')^t$.
3. Say (u, v) is in G^T if and only if (u', v'') is not in $(G')^t$.

Steps 1 and 3 clearly take time $O(n^2)$ and step 2 requires time $O(n^\alpha)$. Thus, the entire algorithm requires $O(n^\alpha)$ steps.

Theorems 3 and 4 reduce the problem of finding a good algorithm for transitive reduction to that of finding a good algorithm for transitive closure. The method of [6], [7] is based on Strassen's matrix multiplication algorithm [9], and thus takes $O(n^{\log_2 7})$ steps. This method is the best known for large n. Under some conditions, transitive closure algorithms found in [10]–[12] may be preferred.

REFERENCES

[1] E. G. COFFMAN, JR. AND R. L. GRAHAM, *Optimal scheduling for two-processor systems*, to appear.

[2] M. R. GAREY, *Optimal task sequencing with precedence constraints*, to appear in Discrete Mathematics.

[3] D. M. MOYLES AND G. L. THOMPSON, *Finding a minimum equivalent graph of a digraph*, J. Assoc. Comput. Mach., 16 (1969), pp. 455–460.

[4] J. HARTMANIS, *Computational complexity of random access stored program machines*, Rep. TR 70-70, Dept. of Computer Science. Cornell Univ., Ithaca, N.Y., 1970.

[5] M. J. FISCHER AND A. R. MEYER, *Boolean matrix multiplication and transitive closure*, Conference Record, Twelfth Annual Symposium on Switching and Automata Theory, East Lansing, Mich., 1971, pp. 129–131.

[6] I. MUNRO, *Efficient determination of the strongly connected components and the transitive closure of a graph*, unpublished manuscript, Univ. of Toronto, Toronto, Canada, 1971.

[7] M. E. FURMAN, *Application of a method of fast multiplication of matrices in the problem of finding the transitive closure of a graph*, Dokl. Akad. Nauk SSSR, 11 (1970), no. 5, p. 1252.

[8] R. TARJAN, *Depth first search and linear graph algorithms*, Conference Record, Twelfth Annual Symposium on Switching and Automata Theory, East Lansing, Mich., 1971, pp. 114–121.

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

[10] S. WARSHALL, *A theorem on Boolean matrices*, J. Assoc. Comput. Mach., 9 (1962), pp. 11–12.

[11] V. L. ARLAZAROV, E. A. DINIC, M. A. KRONOD AND I. A. FARADZEV, *On economical construction of the transitive closure of an oriented graph*, Dokl. Akad. Nauk SSSR, 11 (1970), pp. 1209–1210.

[12] P. PURDOM, *A transitive closure algorithm*, BIT, 10 (1970), pp. 76–94.

SIAM J. COMPUT.
Vol. 1, No. 2, June 1972

ASYMPTOTIC SERVICE SYSTEM OUTPUT, WITH APPLICATION TO MULTIPROGRAMMING*

DONALD P. GAVER†

Abstract. In a multiprogramming computer system modeled as a cyclic queue the number of program segments completed at the CPU in a period of time is discussed. It is shown that this number is approximately normally distributed as the time period becomes long. Parameters of the normal distribution are determined. Numerical examples illustrate the results.

1. Introduction. It is often mathematically convenient and useful to represent the behavior of a computer system, or part thereof, as a single server queueing process. This is appropriate even when several servers are present, as in multiprogramming situations involving cyclic queues; see Gaver [2], and Lewis and Shedler [3]. Then the server singled out for particular attention usually possesses "general" (nonexponential) service times, while the others enjoy simple Markov-convenient properties. By assuming this structure, it is often possible to compute such system characteristics as waiting time properties and server idleness probability, where the latter depend upon server processing rates and the number of programs (customers) allowed to be present in the system simultaneously.

This paper is devoted to studying the distribution of the *output* of the server in such a process. By this we mean the following. Beginning at some moment t', the total number of service completions during $(t', t' + t)$ is observed. Denote this number by $Z(t', t' + t)$, or by $Z(t)$ if the process has stationary increments. We term Z the *output* of our server, and seek to characterize its behavior. It will be shown in the sequel that in interesting cases Z is approximately normally distributed as t becomes large. Furthermore, in cyclic models, e.g., for multiprogramming, a simple continuity argument shows that the outputs of both servers enjoy the same limiting normal distribution. The methods employed make possible a comparison of various multiprogramming situations. Some limited numerical illustrations are presented.

2. Inputs and outputs. In this section we record a simple observation upon which much of the later development rests.

(a) *The* M/G/1 *service system.* Here λ denotes the Poisson arrival rate, and S is a generic service time. Assume $E[S^2] < \infty$. Let $N(t)$ denote the number of customers in the system at time t, and let the *input*, $A(t)$, be the total number of arrivals to have occurred in $(0, t)$. Then $Z(t)$, the total *output* in $(0, t)$, is defined by the continuity relation

$$N(0) + A(t) = Z(t) + N(t). \tag{2.1}$$

Now suppose $\rho = \lambda E[S] < 1$. Then, for large t, $N(t)$ is finite while $A(t)$ becomes large, and hence Z is asymptotically similar to $A(t)$. Putting this formally, write

* Received by the editors August 31, 1971, and in revised form March 13, 1972.

† Department of Operations Research and Administrative Sciences, Naval Postgraduate School, Monterey, California 93940. This research was conducted under a grant from the IBM Corporation and was supported in part by the Office of Naval Research.

(2.1) as

$$\left[\frac{A(t)-\lambda t}{\sqrt{\lambda t}}\right]-\left[\frac{Z(t)-\lambda t}{\sqrt{\lambda t}}\right]=\frac{N(t)-N(0)}{\sqrt{\lambda t}}. \tag{2.2}$$

Now when $\rho < 1$ and $E[S^2] < \infty$, it is well known that

$$\lim_{t\to\infty} E[N(t)|N(0)] = E[N(\infty)] < \infty, \tag{2.3}$$

and by Chebyshev's inequality,

$$P\left\{\left|\frac{N(t)-N(0)}{\sqrt{\lambda t}}\right| > \varepsilon\right\} < \frac{E[N(t)]+N(0)}{\varepsilon\sqrt{\lambda t}} \to 0 \tag{2.4}$$

for any $\varepsilon > 0$ as $t \to \infty$. Thus it follows that the left-hand side of (2.2) approaches zero in probability, and hence the distribution of $(A(t) - \lambda t)/\sqrt{\lambda t}$ converges to the $N(0, 1)$ law, and so does that of $(Z(t) - \lambda t)/\lambda\sqrt{t}$. This result will hold true for many types of queueing systems, e.g., for the GI/G/1 as well as for various multiple server configurations.

(b) *The cyclic system.* Of particular interest in the multiprogramming computer system studies context is the cyclic arrangement depicted in Fig. 1. In the

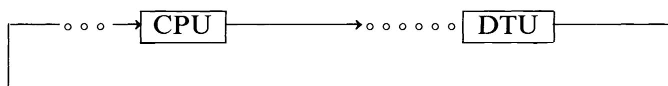


FIG. 1

most rudimentary model a fixed finite number, J, of tasks or programs is present in the system at any time. A program is processed at the *central processing unit* (CPU) until an interruption occurs ("page interrupt" in certain types of machines) for lack of information. At this moment the program enters the *data transfer unit* (DTU) stage, where it awaits and eventually receives the required information and is then returned to the CPU stage. In the mean time, the CPU may be busy processing another program, and is therefore kept busy. Programs that are completed at the CPU stage are assumed to leave the system and be instantaneously replaced. Models of this type have been considered by various authors (cf. Gaver [2], Lewis and Shedler [3], and Shedler [4]).

Although only a limited amount of actual data analysis has been carried out, it may be somewhat appropriate to assume that the service times of programs at the CPU are independently and exponentially distributed; see Smith [5]. Actually, recent data analysis has indicated that distributions of greater than exponential skewness (hyperexponentials) are more appropriate; our analysis can be extended, with some effort, to deal with such models. Service times at the DTU are apparently of nonexponential (more nearly constant) character. We shall, as a consequence, make such assumptions; it is noted that this represents a reversal of the assumptions made in Gaver [2].

In order to discuss the outputs of the servers, denote by $C(t)$ the number of programs that complete the CPU stage (number of CPU service completions) in

$(0, t)$, while $D(t)$ refers to the corresponding quantity for the DTU. Furthermore, $N_C(t)$ and $N_D(t)$ represent the number of programs at the CPU and DTU stages respectively. Then again the continuity relation (where C takes the place of Z) states that

$$N_C(0) + D(t) = C(t) + N_C(t). \tag{2.5}$$

Hence, if there exist norming constants μ and σ such that $[D(t) - \mu t](\sigma\sqrt{t})^{-1}$ has a limiting normal distribution, then, by the same argument as that outlined in connection with the M/G/1-system, $[C(t) - \mu t](\sigma\sqrt{t})^{-1}$ approaches the same normal distribution. That is, the output distributions of CPU and DTU are asymptotically identical. But for the present model it may be seen that $D(t)$ is actually a *cumulative process* in the sense of Smith (cf. Cox [1, Chap. 8, particularly § 8.5], or Smith [5]). Hence, asymptotic normality follows, e.g., from Smith's development [5, pp. 262–263].

3. The cyclic system, its busy periods, and a cumulative process. The process associated with the cyclic system of Fig. 1 may conveniently be viewed as a succession of busy and idle periods for the DTU. Let there be J programs circulating, and consider a moment t' such that $N_D(t' - 0) = 0$ but $N_D(t') = 1$. That is, the system is idle prior to t', becoming busy with the service of one customer at t'. A busy period, or first passage time, for the DTU is defined by

$$\tau_1(J) = \inf\{t | N_D(t + t') = 0\}. \tag{3.1}$$

Following each busy period is an idle period, during which all J programs are queued behind the CPU. By the Markov property of the CPU service times, a generic idle period of duration I is exponentially distributed with mean λ^{-1}. Successive idle periods and subsequent busy periods are independently and identically distributed random variables, since CPU and DTU service times are mutually independent. Put

$$X^{(n)} = I^{(n)} + \tau_1^{(n)}(J) \quad \text{for } n = 1, 2, \cdots; \tag{3.2}$$

$\{X^{(n)}\}$ represents the times between the successive regeneration points at which the DTU becomes idle. In terms of the $\{X^{(n)}, n = 1, 2, \cdots\}$ sequence, which is one of independently and identically distributed random variables, one can speak of a renewal counting process, $R(t)$, where

$$\begin{aligned} R(t) &= 0 \quad \text{iff} \quad X^{(1)} > t, \\ &\cdots \\ R(t) &= j \quad \text{iff} \quad \sum_{n=1}^{j} X^{(n)} \leqq t \quad \text{and} \quad \sum_{n=1}^{j+1} X^{(n)} > t \end{aligned} \tag{3.3}$$

for $j = 2, 3, \cdots$.

Let the output, or number of service completions, during a (the nth) busy period be

$$\beta_1^{(n)}(J) = D(\tau_1^{(n)}(J)) \quad \text{for } n = 1, 2, 3, \cdots. \tag{3.4}$$

Then the output process $D(t)$ is cumulative in the sense of Smith [5], with

$$D(t) \approx \sum_{n=1}^{R(t)} \beta_1^{(n)}(J) \quad \text{as } t \to \infty \tag{3.5}$$

(the approximation consists in neglecting outputs during part of an X-cycle; these are negligible for large t). A central limit theorem for such processes (cf. Cox [1]) enables one to show that $D(t)$, appropriately normalized, is approximately normally distributed for large t, and to find the parameters of the limiting distribution (asymptotic mean and variance) explicitly in terms of the CPU service rate, λ, and the distribution of service times at the DTU. To be specific, it may be shown that as $t \to \infty$,

$$E[D(t)] \sim t\frac{E[\beta_1(J)]}{E[X]} \equiv t\mu \tag{3.6}$$

and

$$\operatorname{var}[D(t)] \sim t\left\{\frac{\operatorname{var}[\beta_1(J)]}{E[X]} + \frac{\operatorname{var}[X](E[\beta_1(J)])^2}{(E[X])^3} - 2\frac{\operatorname{cov}[\beta_1(J), X]E[\beta_1(J)]}{(E[X])^2}\right\} \equiv \sigma^2 t \tag{3.7}$$

and that $(D(t) - t\mu)/\sigma\sqrt{t}$ tends to the $N(0, 1)$ law as t becomes large. Explicit evaluation of the parameters μ and σ^2 is discussed in the next section. Finally the development of § 2 then implies that $(C(t) - t\mu)/\sigma\sqrt{t}$ also has the limiting $N(0, 1)$ distribution.

4. Cumulative process parameters. Recursive evaluation of busy period properties as J increases was discussed in Gaver [2]. Let S be the DTU service time, with distribution $U(x)$ and Laplace–Stieltjes transform

$$u(s) = \int_0^\infty e^{-sx}U\{dx\}. \tag{4.1}$$

Then consider the following cases.

Representation.

(A) $J = 1$. Here clearly

$$\tau_1(1) = S, \quad \beta_1(1) = 1. \tag{4.2}$$

	Probability $e^{-\lambda S}$	$1 - e^{-\lambda S}$
$\tau_1(2)$	$S(=\tau_1(1))$	$S + \tau_1'(2)$
$\beta_1(2)$	1	$1 + \beta_1'(2)$

FIG. 2

(B) $J = 2$. *Condition on S to find the results in Fig.* 2. Use of the symbol $'$ means that, for example, $\tau_1'(2)$ has the $\tau_1(2)$ distribution but is independent of events leading up to the initial service completion. To explain further, consider the situation just following the initial service completion of the busy period. Either (i) no CPU output occurred during S, an event of probability $e^{-\lambda S}$, in which case the busy period is of duration S with one output, or (ii) exactly one CPU output occurred, an event of probability $1 - e^{-\lambda S}$, in which case the initial situation was reproduced, with one program at the CPU and one at the DTU, but with an initial component of busy period duration, S, and one initial output.

(C) *Arbitrary J. Again condition on S.* Define $\tau_i(J)$ and $\beta_i(J)$, $i = 1, 2, \cdots, J$, to be respectively the first passage time from i to $i - 1$ and the output therein, given that a service is just commencing at the moment $N_D = i$.

For the present setup, see Fig. 3. Now a little reflection shows that $\tau_i(J)$ has the same distribution as $\tau_1(J - i + 1)$, and similarly that $\beta_i(J)$ has the same distribution as $\beta_1(J - i + 1)$. This fact enables us to compute successively the various expectations required to evaluate (3.6) and (3.7). We now illustrate.

	Probability					
	$e^{-\lambda S}$	$\lambda S\, e^{-\lambda S}$	$\cdots$	$\frac{(\lambda S)^j e^{-S}}{j!}$	$\cdots$	$1 - e^{-\lambda S}\sum_{j=0}^{J-2}\frac{(\lambda S)^j}{j!}$
$\tau_1(J)$	S	$S + \tau_1'(J)$		$S + \sum_{i=1}^{j} \tau_i'(J)$		$S + \sum_{i=1}^{J-1} \tau_i'(J)$
$\beta_1(J)$	1	$1 + \beta_1'(J)$		$1 + \sum_{i=1}^{j} \beta_i'(J)$		$1 + \sum_{i=1}^{J-1} \beta_i'(J)$

FIG. 3

(A′) $J = 1$. Directly,

$$E[\tau_1(1)] = E[S], \quad \operatorname{var}[\tau_1(1)] = \operatorname{var}[S].$$

$$E[\beta_1(1)] = 1, \qquad \operatorname{var}[\beta_1(1)] = 0, \tag{4.3}$$

$$\operatorname{cov}[\tau_1(1), \beta_1(1)] = 0.$$

(B′) $J = 2$. The condition on S is

$$E[\tau_1(2)|S] = S + (1 - e^{-\lambda S})E[\tau_1(2)]. \tag{4.4}$$

Consequently after removal of the condition on S and use of (4.1),

$$E[\tau_1(2)] = \frac{E[S]}{E[e^{-\lambda S}]} = \frac{E[S]}{u[\lambda]}. \tag{4.5}$$

Likewise,

$$E[\beta_1(2)] = \frac{1}{E[e^{-\lambda S}]} = \frac{1}{u[\lambda]}. \tag{4.6}$$

Squaring column by column in Fig. 2 delivers second moments. For example,

$$E[\tau_1^2(2)|S] = S^2 e^{-\lambda S} + E[(S + \tau_1'(2))^2](1 - e^{-\lambda S}) \tag{4.7}$$
$$= S^2 + \{2SE[\tau_1(2)] + E[\tau_1^2(2)]\}(1 - e^{-\lambda S}).$$

So, upon removal of the condition on S,

$$E[\tau_1^2(2)] = \frac{E[S^2] + 2E[S(1 - e^{-\lambda S})]E[\tau_1(2)]}{E[e^{-\lambda S}]}; \tag{4.8}$$

the value of (4.5) is introduced to evaluate the latter expression. Next the variance is computed by subtracting off the square of (4.5). In analogous fashion,

$$E[\beta_1^2(2)] = \frac{1 + 2E[(1 - e^{-\lambda S})]E[\beta_1(2)]}{E[e^{-\lambda S}]}; \tag{4.9}$$

insertion of (4.6) and subtraction of its square yields the variance. The covariance is obtained from the expectation

$$E[\beta_1(2)\tau_1(2)] = \frac{E[S] + E[S(1 - e^{-\lambda S})]E[\beta_1(2)] + E[1 - e^{-\lambda S}]E[\tau_1(2)]}{E[e^{-\lambda S}]} \tag{4.10}$$

by subtraction of the product of (4.5) and (4.6). These expressions can be evaluated in terms of the transform (4.1) and its derivatives, and thus there is natural impetus to employ some explicitly transformable density, e.g., the gamma or hyperexponential, to represent DTU service times.

Examination of Fig. 3 makes it clear that the busy period moments for any J can be expressed in terms of the corresponding moments for smaller J-values. This step can perhaps be best carried out numerically, for neat closed-form exact expressions will not occur.

The busy period moments obtained by the procedure described may be employed to evaluate the output parameters (3.6) and (3.7). Some numerical illustrations are given in the following section.

5. Numerical examples. The effect of assuming various parameter values in our multi-programming model can be investigated numerically by putting the results of the previous section to work. Some rather limited examples appear in Table 1.

Notice that when $\lambda > (E[S])^{-1} = 1$, in which case the DTU stage acts as bottleneck, the move from $J = 1$ to $J = 2$ has dramatic effects. Although the output rate can never exceed unity, improvements of at least ten percent occur. The addition of further programs ($J > 2$) is justified if considerable overhead activity is present; this feature is not included in the present model. It is of interest to compare the numerical values of Table 1 to those obtained by Shedler [4]. Clearly λ[CPU utilization] $= \mu$, and a reference to the appropriate entries in Table 1 of [4] provides numerical confirmation.

Examination of the variance of output is of some interest. If λ is relatively small (CPU the bottleneck), it appears that

(a) for $J = 1$, var $[C(t)|S$ exponential$] >$ var $[C(t)|S$ constant$]$

TABLE 1

			CPU					
		λ:	0.20	0.5	1.0	1.5	2.0	5.0
DTU Constant, $E[S] = 1$	$J = 1$	μ	0.17	0.33	0.50	0.60	0.67	0.83
		σ^2	0.12	0.15	0.13	0.10	0.07	0.02
	$J = 2$	μ	0.20	0.45	0.73	0.87	0.94	0.98
		σ^2	0.18	0.31	0.23	0.12	0.05	0.01
Exponential, $E[S] = 1$	$J = 1$	μ	0.17	0.33	0.50	0.60	0.67	0.83
		σ^2	0.17	0.19	0.26	0.33	0.35	0.52
	$J = 2$	μ	0.19	0.43	0.67	0.79	0.86	0.92
		σ^2	0.17	0.29	0.37	0.47	0.56	0.81

but

(b) for $J = 2$, var $[C(t)|S$ exponential$]$ < var $[C(t)|S$ constant$]$.

By way of explanation, one sees that when $J = 2$ busy periods are more likely to involve more than one DTU service when S is constant than when S is exponential. Of course, if $\lambda > (E[S])^{-1}$, the DTU becomes the bottleneck. As anticipated in this situation the output behaves like a renewal process with inter-event times distributed according to S. Consequently when S is constant, $\sigma^2 = t^{-1}$ var $[C(t)]$ dwindles to zero as λ increases, reflecting the fact that outputs through the DTU bottleneck are regular. Of course, the regularity is even greater when $J = 2$ than when $J = 1$. If, on the other hand, S is exponential, the variance gradually approaches that of the DTU bottleneck, namely, unity.

It may be guessed from the numbers of Table 1 that when $\lambda > 1$ the assumption of exponential S provides an *underestimate* of output rate μ, and an *overestimate* of σ^2, provided S is more regular—of smaller variance—than the exponential. Another estimate of σ^2, useful when the DTU rate is smaller than that of the CPU, is obtained by simply assuming that the DTU is never idle, and thus

$$\sigma^2 \approx \frac{\text{var}\,[S]}{(E[S])^3},$$

a familiar renewal theory result that will become increasingly accurate for larger and larger J. Although we do not explore such approximations further at this point, it seems evident that for increasingly complex systems—those in which there are considerations of overhead, nonexponential distributions, and in which $J > 2$—the only practical route to understanding is through approximations and bounds. If simulations are undertaken, it is useful to have some idea of the variance of $C(t)$ so that run lengths may be established. Approximate variances are often adequate for such purposes.

6. Program termination and output. The previous development takes no account of the fact that individual programs actually terminate. In order to introduce this effect into the model, we can assume that each time a program leaves

the CPU stage one of two events occurs: (i) the program terminates or is completed, or (ii) the program experiences an honest page fault and must go to the DTU stage. Suppose that the choice of event (i) or (ii) is governed by a Bernoulli trials process so that with probability p the program terminates, and with probability $q = 1 - p$ the program continues to the DTU stage. In order to allow use of the previous analysis, we shall assume that in case event (i) occurs a *new* program is immediately introduced into the system at the DTU stage; the first pass through this stage may well represent I/0 activity on behalf of this newest program.

Let $M(t)$ represent individual program output over time t. Now given $C(t)$, $M(t)$ is conditionally binomial, with

$$E[M(t)|C(t)] = pC(t) \tag{6.1}$$

so

$$E[M(t)] = pE[C(t)] \sim p\mu t \equiv \mu_M t \tag{6.2}$$

as $t \to \infty$. Furthermore,

$$\begin{aligned} \operatorname{var}[M(t)] &= pqE[C(t)] + p^2 \operatorname{var}[C(t)] \\ &\sim pq\mu t + p^2\sigma^2 t \equiv \sigma_M^2 t, \end{aligned} \tag{6.3}$$

and since $M(t)$ is easily seen to be a cumulative process, the previously quoted theorem shows that the actual output of completed programs is approximately normal as $t \to \infty$.

REFERENCES

[1] D. R. COX, *Renewal Theory*, Methuen Monograph, London, 1962.

[2] D. P. GAVER, *Probability models for multiprogramming computer systems*, J. Assoc. Comput. Mach., 14 (1967), pp. 423–438.

[3] P. A. W. LEWIS AND G. S. SHEDLER, *A cyclic queue model of system overhead in multiprogrammed computer systems*, Ibid., 18 (1971), pp. 199–221.

[4] G. S. SHEDLER, *A cyclic queue model of a paging machine*, Research Rep. RC-2814, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., 1970.

[5] J. L. SMITH, *Multiprogramming under a page-on demand strategy*, Comm. ACM, 10 (1967), pp. 636–646.

[6] W. L. SMITH, *Renewal theory and its ramifications*, J. Roy. Statist. Soc. Ser. B, 20 (1958), pp. 243–301.

SIAM J. COMPUT.
Vol. 1, No. 2, June 1972

DEPTH-FIRST SEARCH AND LINEAR GRAPH ALGORITHMS*

ROBERT TARJAN†

Abstract. The value of depth-first search or "backtracking" as a technique for solving problems is illustrated by two examples. An improved version of an algorithm for finding the strongly connected components of a directed graph and an algorithm for finding the biconnected components of an undirect graph are presented. The space and time requirements of both algorithms are bounded by $k_1V + k_2E + k_3$ for some constants k_1, k_2, and k_3, where V is the number of vertices and E is the number of edges of the graph being examined.

Key words. Algorithm, backtracking, biconnectivity, connectivity, depth-first, graph, search, spanning tree, strong-connectivity.

1. Introduction. Consider a graph G, consisting of a set of vertices $\mathscr{V}$ and a set of edges $\mathscr{E}$. The graph may either be directed (the edges are ordered pairs (v, w) of vertices; v is the *tail* and w is the *head* of the edge) or undirected (the edges are unordered pairs of vertices, also represented as (v, w)). Graphs form a suitable abstraction for problems in many areas; chemistry, electrical engineering, and sociology, for example. Thus it is important to have the most economical algorithms for answering graph-theoretical questions.

In studying graph algorithms we cannot avoid at least a few definitions. These definitions are more-or-less standard in the literature. (See Harary [3], for instance.) If $G = (\mathscr{V}, \mathscr{E})$ is a graph, a *path* $p: v \overset{*}{\Rightarrow} w$ in G is a sequence of vertices and edges leading from v to w. A path is *simple* if all its vertices are distinct. A path $p: v \overset{*}{\Rightarrow} v$ is called a *closed path*. A closed path $p: v \overset{*}{\Rightarrow} v$ is a *cycle* if all its edges are distinct and the only vertex to occur twice in p is v, which occurs exactly twice. Two cycles which are cyclic permutations of each other are considered to be the same cycle. The *undirected version* of a directed graph is the graph formed by converting each edge of the directed graph into an undirected edge and removing duplicate edges. An undirected graph is *connected* if there is a path between every pair of vertices.

A (directed rooted) *tree* T is a directed graph whose undirected version is connected, having one vertex which is the head of no edges (called the *root*), and such that all vertices except the root are the head of exactly one edge. The relation "(v, w) is an edge of T" is denoted by $v \rightarrow w$. The relation "There is a path from v to w in T" is denoted by $v \overset{*}{\rightarrow} w$. If $v \rightarrow w$, v is the *father* of w and w is a *son* of v. If $v \overset{*}{\rightarrow} w$, v is an *ancestor* of w and w is a *descendant* of v. Every vertex is an ancestor and a descendant of itself. If v is a vertex in a tree T, T_v is the subtree of T having as vertices all the descendants of v in T. If G is a directed graph, a tree T is a *spanning tree* of G if T is a subgraph of G and T contains all the vertices of G.

If R and S are binary relations, R^* is the transitive closure of R, R^{-1} is the inverse of R, and

$$RS = \{(u, w) | \exists v((u, v) \in R \,\&\, (v, w) \in S)\}.$$

* Received by the editors August 30, 1971, and in revised form March 9, 1972.

† Department of Computer Science, Cornell University, Ithaca, New York 14850. This research was supported by the Hertz Foundation and the National Science Foundation under Grant GJ-992.

If $f, f_1, \cdots, f_n$ are functions of $x_1, \cdots, x_n$, we say f is $O(f_1, \cdots, f_n)$ if

$$|f(x_1, \cdots, x_n)| \leqq k_0 + k_1|f_1(x_1, \cdots, x_n)| + \cdots + k_n|f_n(x_1, \cdots, x_n)|$$

for all x_i and some constants $k_0, \cdots, k_n$. We shall assume a random-access computer model.

2. Depth-first search. Backtracking, or depth-first search, is a technique which has been widely used for finding solutions to problems in combinatorial theory and artificial intelligence [2], [11] but whose properties have not been widely analyzed. Suppose G is a graph which we wish to explore. Initially all the vertices of G are unexplored. We start from some vertex of G and choose an edge to follow. Traversing the edge leads to a new vertex. We continue in this way; at each step we select an unexplored edge leading from a vertex already reached and we traverse this edge. The edge leads to some vertex, either new or already reached. Whenever we run out of edges leading from old vertices, we choose some unreached vertex, if any exists, and begin a new exploration from this point. Eventually we will traverse all the edges of G, each exactly once. Such a process is called a *search* of G.

There are many ways of searching a graph, depending upon the way in which edges to search are selected. Consider the following choice rule: when selecting an edge to traverse, always choose an edge emanating from the vertex most recently reached which still has unexplored edges. A search which uses this rule is called a *depth-first search*. The set of old vertices with possibly unexplored edges may be stored on a stack. Thus a depth-first search is very easy to program either iteratively or recursively, provided we have a suitable computer representation of a graph.

DEFINITION 1. Let $G = (\mathscr{V}, \mathscr{E})$ be a graph. For each vertex $v \in \mathscr{V}$ we may construct a list containing all vertices w such that $(v, w) \in \mathscr{E}$. Such a list is called an *adjacency list* for vertex v. A set of such lists, one for each vertex in G, is called an *adjacency structure* for G.

If the graph G is undirected, each edge (v, w) is represented twice in an adjacency structure; once for v and once for w. If G is directed, each edge (v, w) is represented once: vertex w appears in the adjacency list of vertex v. A single graph may have many adjacency structures; in fact, each ordering of the edges around the vertices of G gives a unique adjacency structure, and each adjacency structure corresponds to a unique ordering of the edges at each vertex. Using an adjacency structure for a graph, we can perform depth-first searches in a very efficient manner, as we shall see.

Suppose G is a connected undirected graph. A search of G imposes a direction on each edge of G given by the direction in which the edge is traversed when the search is performed. Thus G is converted into a directed graph G'. The set of edges which lead to a new vertex when traversed during the search defines a spanning tree of G'. In general, the arcs of G' which are not part of the spanning tree interconnect the paths in the tree. However, if the search is depth-first, each edge (v, w) not in the spanning tree connects vertex v with one of its ancestors w.

DEFINITION 2. Let P be a directed graph, consisting of two disjoint sets of edges, denoted by $v \rightarrow w$ and $v \dashrightarrow w$ respectively. Suppose P satisfies the following properties:

(i) The subgraph T containing the edges $v \rightarrow w$ is a spanning tree of P.

(ii) We have $\dashrightarrow \subseteq (\xrightarrow{*})^{-1}$, where "$\rightarrow$" and "$\dashrightarrow$" denote the relations defined by the corresponding set of edges. That is, each edge which is not in the spanning tree T of P connects a vertex with one of its ancestors in T.

Then P is called a *palm tree*. The edges $v \dashrightarrow w$ are called the *fronds* of P.

THEOREM 1. *Let P be the directed graph generated by a depth-first search of a connected graph G. Then P is a palm tree. Conversely, let P be any palm tree. Then P is generated by some depth-first search of the undirected version of P.*

Proof. Consider the program listed below, which carries out a depth-first search of a connected graph, starting at vertex s, using an adjacency structure of the graph to be searched. The program numbers the vertices of the graph in the order they are reached during the search and constructs the directed graph (P) generated by the search.

```
BEGIN
    INTEGER i;
    PROCEDURE DFS(v, u); COMMENT vertex u is the father of
                    vertex v in the spanning tree being constructed;
        BEGIN
            NUMBER (v) := i := i + 1;
            FOR w in the adjacency list of v DO
                BEGIN
                    IF w is not yet numbered THEN
                        BEGIN
                            construct arc v → w in P;
                            DFS(w, v);
                        END
                    ELSE IF NUMBER (w) < NUMBER (v) and w ¬= u
                        THEN construct arc v -→ w in p;
                END;
        END;
    i := 0;
    DFS(s, 0);
END;
```

Figure 1 gives an example of the application of DFS to a graph. Suppose $P = (\mathscr{V}, \mathscr{E})$ is the directed graph generated by a depth-first search of some connected graph G, and assume that the search begins at vertex s. Examine the procedure DFS. The algorithm clearly terminates because each vertex can only be numbered once. Furthermore, each edge in the graph is examined exactly twice. Therefore the time required by the search is linear in V and E.

For any vertices v and w, let $d(v, w)$ be the length of the shortest path between v and w in G. Since G is connected, all distances are finite. Suppose that some vertex remains unnumbered by the search. Let v be an unnumbered vertex such that $d(s, v)$ is minimal. Then there is a vertex w such that w is adjacent to v and $d(s, w) < d(s, v)$. Thus w is numbered. But v will also be numbered, since it is adjacent to w. This means that all vertices are numbered during the search.

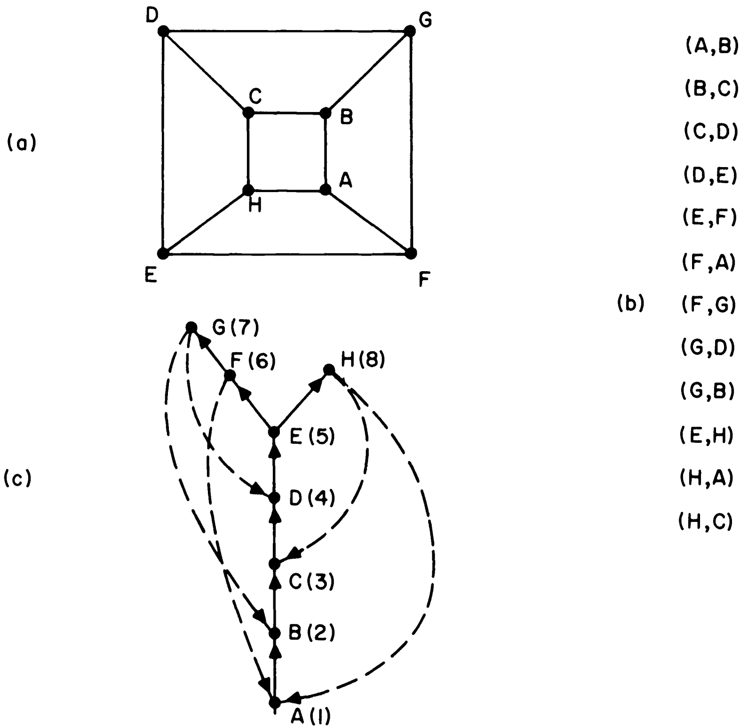


FIG. 1. *Application of depth-first search to a graph*

(a) *The graph*
(b) *Order of edge exploration*
(c) *Generated palm tree, with numbers of vertices in* ( )

The vertex s is the head of no edge $w \to s$. Each other vertex v is the head of exactly one edge $w \to v$. The subgraph T of P defined by the edges $v \to w$ is obviously connected. Thus T is a spanning tree of P.

Each arc of the original graph is directed in at least one direction; if (v, w) does not become an arc of the spanning tree T, either $v \dashrightarrow w$ or $w \dashrightarrow v$ must be constructed since both v and w are numbered whenever edge (v, w) is inspected and either NUMBER $(v) <$ NUMBER (w) or NUMBER $(v) >$ NUMBER (w).

The arcs $v \to w$ run from smaller numbered points to larger numbered points. The arcs $v \dashrightarrow w$ run from larger numbered points to smaller numbered points. If arc $v \dashrightarrow w$ is constructed, arc $w \to v$ is not constructed later because v is numbered. If arc $w \to v$ is constructed, arc $v \dashrightarrow w$ is not later constructed, because of the test "$w \neg = u$" in procedure DFS. Thus each edge in the original graph is directed in one and only one direction.

Consider an arc $v \dashrightarrow w$. We have NUMBER $(w) <$ NUMBER (v). Thus w is numbered before v. Since $v \dashrightarrow w$ is constructed and not $w \to v$, v must be numbered before edge (w, v) is inspected. Thus v must be numbered during execution

of DFS $(w, -)$. But all vertices numbered during execution of DFS $(w, -)$ are descendants of w. This means that $w \overset{*}{\rightarrow} v$, and G is a palm tree.

Let us prove the converse part of the theorem. Suppose that P is a palm tree, with spanning tree T and undirected version G. Construct an adjacency structure of G in which all the edges of T appear before the other edges of G in the adjacency lists. Starting with the root of T, perform a depth-first search using this adjacency structure. The search will traverse the edges of T preferentially and will generate the palm tree P; it is easy to see that each edge is directed correctly. This completes the proof of the theorem.

We may state Theorem 1 nonconstructively as the following corollary.

COROLLARY 2. *Let G be any undirected graph. Then G can be converted into a palm tree by directing its edges in a suitable manner.*

3. Biconnectivity. The value of depth-first search follows from the simple structure of a palm tree. Let us consider a problem in which this structure is useful.

DEFINITION 3. Let $G = (\mathscr{V}, \mathscr{E})$ be an undirected graph. Suppose that for each triple of distinct vertices v, w, a in $\mathscr{V}$, there is a path $p : v \overset{*}{\Rightarrow} w$ such that a is not on the path p. Then G is *biconnected.* (Other equivalent definitions of biconnectedness exist.) If, on the other hand, there is a triple of distinct vertices v, w, a in $\mathscr{V}$ such that a is on any path $p : v \overset{*}{\Rightarrow} w$, and there exists at least one such path, then a is called an *articulation point* of G.

LEMMA 3. *Let $G = (\mathscr{V}, \mathscr{E})$ be an undirected graph. We may define an equivalence relation on the set of edges as follows: two edges are equivalent if and only if they belong to a common cycle. Let the distinct equivalence classes under this relation be $\mathscr{E}_i$, $1 \leqq i \leqq n$, and let $G_i = (\mathscr{V}_i, \mathscr{E}_i)$, where $\mathscr{V}_i$ is the set of vertices incident to the edges of $\mathscr{E}_i$; $\mathscr{V}_i = \{v | \exists w((v, w) \in \mathscr{E}_i)\}$. Then:*

(i) *G_i is biconnected, for each $1 \leqq i \leqq n$.*

(ii) *No G_i is a proper subgraph of a biconnected subgraph of G.*

(iii) *Each articulation point of G occurs more than once among the $\mathscr{V}_i$, $1 \leqq i \leqq n$. Each nonarticulation point of G occurs exactly once among the $\mathscr{V}_i$, $1 \leqq i \leqq n$.*

(iv) *The set $\mathscr{V}_i \cap \mathscr{V}_j$ contains at most one point, for any $1 \leqq i, j \leqq n$. Such a point of intersection is an articulation point of the graph.*

The subgraphs G_i of G are called the *biconnected components of* G.

Suppose we wish to determine the biconnected components of an undirected graph G. Common algorithms for this purpose, for instance, Shirey's [14], test each vertex in turn to discover if it is an articulation point. Such algorithms require time proportional to $V \cdot E$, where V is the number of vertices and E is the number of edges of the graph. A more efficient algorithm uses depth-first search. Let P be a palm tree generated by a depth-first search. Suppose the vertices of P are numbered in the order they are reached during the search (as is done by the procedure DFS above). We shall refer to vertices by their numbers. If u is an ancestor of v in the spanning tree T of P, then $u < v$. For any vertex v in P, let LOWPT (v) be the smallest vertex in the set $\{v\} \cup \{w | v \overset{*}{\rightarrow} \dashrightarrow w\}$. That is, LOWPT (v) is the smallest vertex reachable from v by traversing zero or more tree arcs followed by at most one frond. The following results form the basis of an efficient algorithm

for finding biconnected components. This algorithm was discovered by Hopcroft and Tarjan [4]. Paton [12] describes a similar algorithm.

LEMMA 4. *Let G be an undirected graph and let P be a palm tree formed by directing the edges of G. Let T be the spanning tree of P. Suppose $p: v \overset{*}{\Rightarrow} w$ is any path in G. Then p contains a point which is an ancestor of both v and w in T.*

Proof. Let T_u with root u be the smallest subtree of T containing all vertices on the path p. If $u = v$ or $u = w$, the lemma is immediate. Otherwise, let T_{u_1} and T_{u_2} be two distinct subtrees containing points on p such that $u \to u_1$ and $u \to u_2$. If only one such subtree exists, then u is on p since T_u is minimal. If two such subtrees exist, path p can only get from T_{u_1} to T_{u_2} by passing through vertex u, since no point in one of these trees is an ancestor of any point in the other, while both $\to$ and $\dashrightarrow$ connect only ancestors in a palm tree. Since u is an ancestor of both v and w, the lemma holds.

LEMMA 5. *Let G be a connected undirected graph. Let P be a palm tree formed by directing the edges of G, and let T be the spanning tree of P. Suppose a, v, w are distinct vertices of G such that $(a, v) \in T$, and suppose w is not a descendant of v in T. (That is, $\neg(v \overset{*}{\to} w)$ in T.) If* LOWPT $(v) \geqq a$, *then a is an articulation point of P and removal of a disconnects v and w. Conversely, if a is an articulation point of G, then there exist vertices v and w which satisfy the properties above.*

Proof. If $a \to v$ and LOWPT $(v) \geqq a$, then any path from v not passing through a remains in the subtree T_v, and this subtree does not contain the point w. This gives the first part of the lemma.

To prove the converse, let a be an articulation point of G. If a is the root of P, then at least two tree arcs must emanate from a. Let v be the head of one such arc and let w be the head of another such arc. Then $a \to v$, LOWPT $(v) \geqq a$, and w is not a descendant of v. If a is not the root of P, consider the connected components formed by deleting a from G. One component must consist only of descendants of a. Such a component can contain only one son of v by Lemma 4. Let v be such a son of a. Let w be any proper ancestor of a. Then $a \to v$, LOWPT (v) $\geqq a$, and w is not a descendant of v. Thus the converse part of the lemma is true.

COROLLARY 6. *Let G be a connected undirected graph, and let P be a palm tree formed by directing the edges of G. Suppose that P has a spanning tree T. If C is a biconnected component of G, then the vertices of C define a subtree of T, and the root of this subtree is either an articulation point of G or is the root of T.*

Figure 2 shows a graph, its LOWPT values, articulation points, and biconnected components. The LOWPT values of all the vertices of a palm tree P may be calculated during a single depth-first search, since

$$\text{LOWPT}(v) = \min(\{\text{NUMBER}(v)\} \cup \{\text{LOWPT}(w) \mid v \to w\} \cup \{\text{NUMBER}(w) \mid v \dashrightarrow w\}).$$

On the basis of such a calculation, the articulation points and the biconnected components may be determined, all during one search. The biconnectivity algorithm is presented below. The program will compute the biconnected components

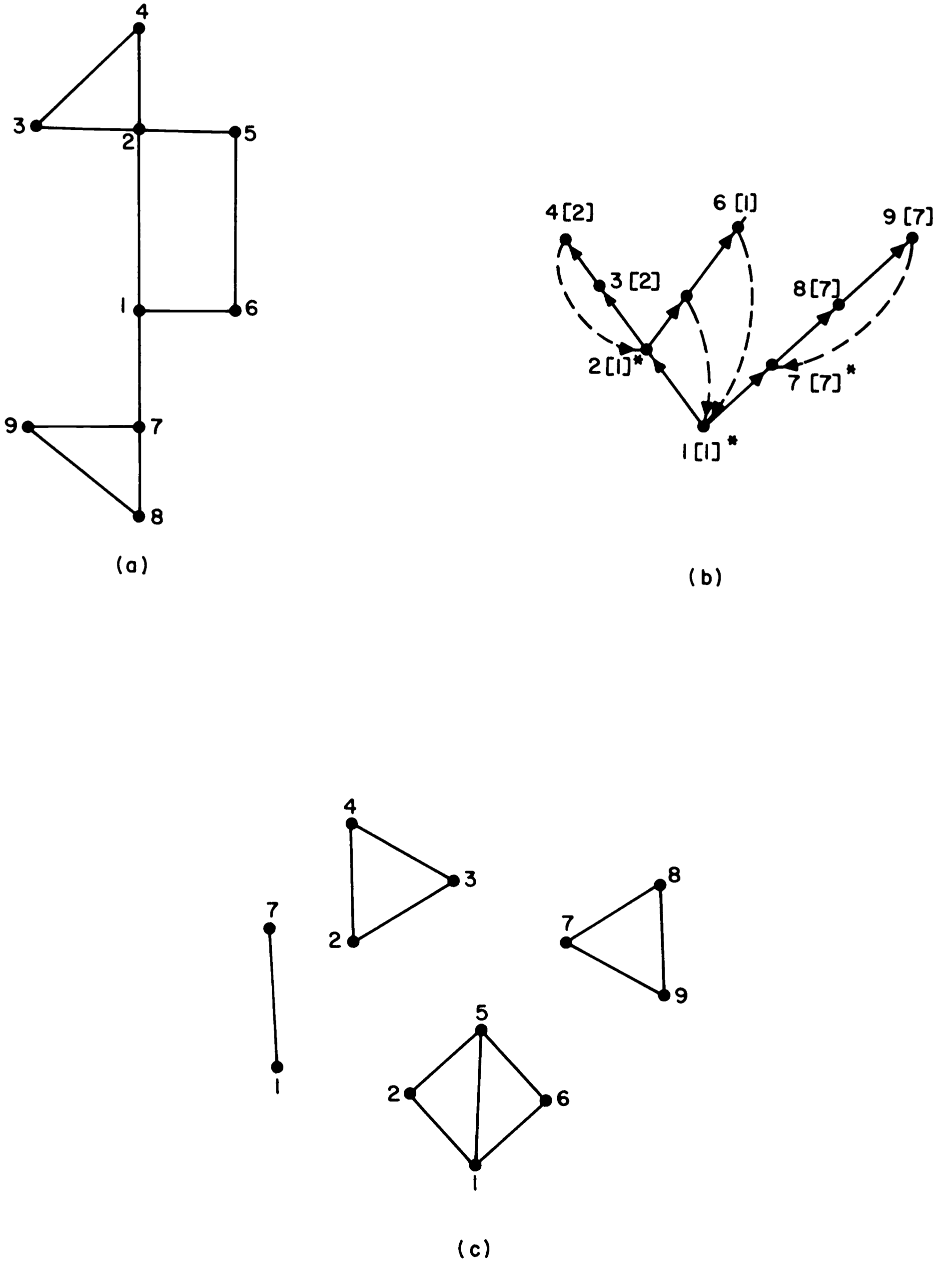


FIG. 2. *A graph and its biconnected components*

(a) *Graph*

(b) *A palm tree with* LOWPT *values in* [ ], *articulation points marked with* *

(c) *Biconnected components*

of a graph G, starting from vertex s.

```
BEGIN
    INTEGER i;
    procedure BICONNECT (v, u);
      BEGIN
          NUMBER (v) := i := i + 1;
          LOWPT (v) = NUMBER (v);
          FOR w in the adjacency list of v DO
              BEGIN
                  IF w is not yet numbered THEN
                    BEGIN
                        add (v, w) to stack of edges;
                        BICONNECT (w, v)
                        LOWPT (v) := min (LOWPT (v), LOWPT (w));
                        IF LOWPT (w) ≧ NUMBER (v) THEN
                          BEGIN
                              start new biconnected component;
                              WHILE top edge e = (u1, u2) on edge
                                      stack has NUMBER (u1)
                                      ≧ NUMBER (w) DO
                                  delete (u1, u2) from edge stack and
                                      add it to current component;
                              delete (v, w) from edge stack and add
                                  it to current component;
                          END;
                    END
                  ELSE IF (NUMBER (w) < NUMBER (v)) and
                        (w¬ = u) THEN
                    BEGIN
                        add (v, w) to edge stack;
                        LOWPT(v) := min(LOWPT(v), NUMBER(w));
                    END;
              END;
      END;
     i := 0;
     empty the edge stack;
     FOR w a vertex DO IF w is not yet numbered THEN BICONNECT (w, 0);
  END;
```

The edges of G are placed on a stack as they are traversed; when an articulation point is found the corresponding edges are all on top of the stack. (If $(v, w) \in T$ and LOWPT $(w) \geqq$ LOWPT (v), then the corresponding biconnected component contains the edges in $\{(u_1, u_2) | w \overset{*}{\to} u_1\} \cup \{(v, w)\}$ which are on the edge stack.) A single search on each connected component of a graph G will give us all the connected components of G.

THEOREM 7. *The biconnectivity algorithm requires $O(V, E)$ space and time when applied to a graph with V vertices and E edges.*

Proof. The algorithm clearly requires space bounded by $k_1V + k_2E + k_3$, for some constants k_1, k_2, and k_3. The algorithm is an elaboration of the depth-first search procedure DFS. During the search, LOWPT values are calculated and each edge is placed on the edge stack once and removed from the edge stack once. The amount of extra time required by these operations is proportional to E. Thus BICONNECT has a time bound linear in V and E.

THEOREM 8. *The biconnectivity algorithm correctly gives the biconnected components of any undirected graph G.*

Proof. The actual depth-first search undertaken by the algorithm depends on the adjacency structure chosen to represent G; we shall prove that the algorithm is correct for all adjacency structures. Notice first that the biconnectivity algorithm analyzes each connected component of G separately to find its biconnected components, applying one depth-first search to each connected component. Thus we need only prove that the biconnectivity algorithm works correctly on connected graphs G.

The correctness proof is by induction on the number of edges in G. Suppose G is connected and contains no edges. G either is empty or consists of a single point. The algorithm will terminate after examining G and listing no components. Thus the algorithm operates correctly in this case. Now suppose that the algorithm works correctly on all connected graphs with $E - 1$ or fewer edges. Consider applying the algorithm to a connected graph G with E edges.

Each edge placed on the stack of edges is eventually removed and added to a component since everything on the edge stack is removed whenever the search returns to the root of the palm tree of G. Consider the situation when the first component G' is formed. Suppose that this component does not include all the edges of G. Then the vertex v currently being examined is an articulation point of the graph and separates the edges in the component from the other edges in the graph by Lemma 5.

Consider only the set of edges in the component. If BICONNECT $(v, 0)$ is executed, using the graph G' as data, the steps taken by the algorithm are the same as those taken during the analysis of the edges of G' when the data consists of the entire graph G. Since G' contains fewer edges than G, the algorithm operates correctly on G' and G' must be biconnected. If we delete the edges of G' from G, we get another subgraph G'' with fewer edges than G since G' is not empty. The algorithm operates correctly on G'' by the induction assumption. The behavior of the algorithm on G is simply a composite of its behavior on G' and on G''; thus the algorithm must operate correctly on G.

Now suppose that only one component is found. We want to show that in this case G is biconnected. Suppose that G is not biconnected. Then G has an articulation point a. By Lemma 5, LOWPT $(v) \geqq a$ for some son v of a. But the articulation point test in the program will succeed when the edge (a, v) is examined, and more than one biconnected component will be generated. This contradiction shows that G is biconnected, and the algorithm works correctly in this case.

By induction, the biconnectivity algorithm gives the correct components when applied to any connected graph, and hence when applied to any graph.

4. Strong connectivity. The biconnectivity algorithm shows how useful depth-first search can be when applied to undirected graphs. However, when a directed graph is searched in a depth-first manner, a simple palm tree structure does not result, because the direction of search on each edge is fixed. The more complicated structure which results in this case is still simple enough to prove useful in at least one application.

DEFINITION 4. Let G be a directed graph. Suppose that for each pair of vertices v, w in G, there exist paths $p_1 : v \overset{*}{\Rightarrow} w$ and $p_2 : w \overset{*}{\Rightarrow} v$. Then G is said to be *strongly connected.*

LEMMA 9. *Let $G = (\mathscr{V}, \mathscr{E})$ be a directed graph. We may define an equivalence relation on the set of vertices as follows: two vertices v and w are equivalent if there is a closed path $p : v \overset{*}{\Rightarrow} v$ which contains w. Let the distinct equivalence classes under this relation be $\mathscr{V}_i$, $1 \leqq i \leqq n$. Let $G_i = (\mathscr{V}_i, \mathscr{E}_i)$, where $\mathscr{E}_i = \{(v, w) \in \mathscr{E} | v, w \in \mathscr{V}_i\}$. Then:*

(i) *Each G_i is strongly connected.*

(ii) *No G_i is a proper subgraph of a strongly connected subgraph of G.*

The subgraphs G_i are called the *strongly connected components* of G.

Suppose we wish to determine the strongly connected components of a directed graph. This problem is related to the problem of determining the ergodic subchains and transient states of a Markov chain. Fox and Landy [1] give an algorithm for solving the latter problem; Purdom [13] and Munro [10] present virtually identical methods for solving the former problem. These algorithms use depth-first search. Purdom claims a time bound of kV^2 for his algorithm; Munro claims $k \max(E, V \log V)$, where the graph has V vertices and E edges. Their algorithm attempts to construct a cycle by starting from a point and beginning a depth-first search. When a cycle is found, the vertices on the cycle are marked as being in the same strongly connected component and the process is repeated. The algorithm has the disadvantage that two small strongly connected components may be collapsed into a bigger one; the resultant extra work in relabeling may contribute V^2 steps using a simple approach, or $V \log V$ steps if a more sophisticated approach is used (see Munro [10]). In fact, the time bound may be reduced further if an efficient list merging algorithm [9] is used. However, a more careful study of what a depth-first search does to a directed graph reveals that an $O(V, E)$ algorithm which requires *no* merging of components may be devised.

Consider what happens when a depth-first search is performed on a directed graph G. The set of edges which lead to a new vertex when traversed during the search form a tree. The other edges fall into three classes. Some are edges running from ancestors to descendants in the tree. These edges may be ignored, because they do not affect the strongly connected components of G. Some edges run from descendants to ancestors in the tree; these we may call *fronds* as above. Other edges run from one subtree to another in the tree. These, we call *cross-links.* It is easy to verify that if the vertices of the tree are numbered in the order they are reached during the search, a cross-link (v, w) always has NUMBER $(v) >$ NUMBER (w). We shall denote tree edges by $v \rightarrow w$, and fronds and cross-links by $v \dashrightarrow w$.

Suppose G is a directed graph, to which a depth-first search algorithm is applied repeatedly until all the edges are explored. The process will create a set of trees which contain all the vertices of G, called the *spanning forest* F of G, and

sets of fronds and cross-links. (Other edges are thrown away.) A directed graph consisting of a spanning forest and sets of fronds and cross-links is called a *jungle*. Suppose the vertices are numbered in the order they are reached during the search and that we refer to vertices by their number. Then we have the following results.

LEMMA 10. *Let v and w be vertices in G which lie in the same strongly connected component. Let F be a spanning forest of G generated by repeated depth-first search. Then v and w have a common ancestor in F. Further, if u is the highest numbered common ancestor of v and w, then u lies in the same strongly connected component as v and w.*

Proof. Without loss of generality we may assume $v \leqq w$. Let p be a path from v to w in G. Let T_u with root u be the smallest subtree of a tree in F containing all the vertices in p. There must be such a tree, since p can pass from one tree in F to another tree with smaller numbered vertices but p can never lead to a tree with larger numbered vertices. If p were contained in two or more trees of F, it could not end at w, since $v \leqq w$.

Thus T_u exists, and v and w have a common ancestor in F. In fact, p must pass through vertex u, by a proof similar to the proof of Lemma 4, and u, v, w must all be in the same strongly connected component. This gives the lemma.

COROLLARY 11. *Let C be a strongly connected component in G. Then the vertices of C define a subtree of a tree in F, the spanning forest of G. The root of this subtree is called the* root *of the strongly connected component C.*

The problem of finding the strongly connected components of a graph G thus reduces to the problem of finding the roots of the strongly connected components, just as the problem of finding the biconnected components of an undirected graph reduces to the problem of finding the articulation points of the graph. We can construct a simple test to determine if a vertex is the root of a strongly connected component. Let

$$\text{LOWLINK}(v) = \min(\{v\} \cup \{w \mid v \overset{*}{\rightarrow} \dashrightarrow w \;\&\; \exists u (u \overset{*}{\rightarrow} v \;\&\; u \overset{*}{\rightarrow} w \;\&\; u \text{ and } w \text{ are in the same strongly connected component of } G)\}).$$

That is, LOWLINK (v) is the smallest vertex which is in the same component as v and is reachable by traversing zero or more tree arcs followed by at most one frond or cross-link.

LEMMA 12. *Let G be a directed graph with* LOWLINK *defined as above relative to some spanning forest F of G generated by depth-first search. Then v is the root of some strongly connected component of G if and only if* LOWLINK $(v) = v$.

Proof. Obviously, if v is the root of a strongly connected component C of G, then LOWLINK $(v) = v$, since if LOWLINK$(v) < v$, some proper ancestor of v would be in C and v could not be the root of C.

Consider the converse. Suppose u is the root of a strongly connected component C of G, and v is a vertex in C different from u. There must be a path $p : v \overset{*}{\Rightarrow} u$. Consider the first edge on this path which leads to a vertex w not in the subtree T_v. This edge is either a vine or a cross-link, and we must have LOWLINK $(v) \leqq w < v$, since the highest numbered common ancestor of v and w is in C.

Figure 3 shows a directed graph, its LOWLINK values, and its strongly connected components. LOWLINK may be calculated using depth-first search.

An algorithm for computing the strongly connected components of a directed graph in $O(V, E)$ time may be based on such a calculation. An implementation of such an algorithm is presented below. The points which have been reached during the search but which have not yet been placed in a component are stored on a stack. This stack is analogous to the stack of edges used by the biconnectivity algorithm.

```
BEGIN
    INTEGER i;
    PROCEDURE STRONGCONNECT (v);
        BEGIN
            LOWLINK (v) := NUMBER (v) := i := i + 1;
            put v on stack of points;
            FOR w in the adjacency list of v DO
                BEGIN
                    IF w is not yet numbered THEN
                      BEGIN comment (v, w) is a tree arc;
                          STRONGCONNECT (w);
                          LOWLINK (v) := min (LOWLINK (v),
                                             LOWLINK (w));
                      END
                    ELSE IF NUMBER (w) < NUMBER (v) DO
                       BEGIN comment (v, w) is a frond or cross-link;
                           if w is on stack of points THEN
                             LOWLINK (v) := min (LOWLINK (v),
                                               NUMBER (w));
                       END;
                END;
            If (LOWLINK (v) = NUMBER (v)) THEN
               BEGIN comment v is the root of a component;
                   start new strongly connected component;
                   WHILE w on top of point stack satisfies
                            NUMBER (w) ≧ NUMBER (v) DO
                        delete w from point stack and put w in
                             current component;
               END;
        END;
     i := 0;
     empty stack of points;
     FOR w a vertex IF w is not yet numbered THEN STRONGCONNECT (w);
  END;
```

THEOREM 13. *The algorithm for finding strongly connected components requires $O(V, E)$ space and time.*

Proof. The algorithm clearly requires space bounded by $k_1V + k_2E + k_3$, for some constants k_1, k_2, and k_3. The algorithm is an elaboration of the depth-first search procedure DFS, modified to apply to directed graphs. During the search, LOWLINK values are calculated, each point is placed on the stack of

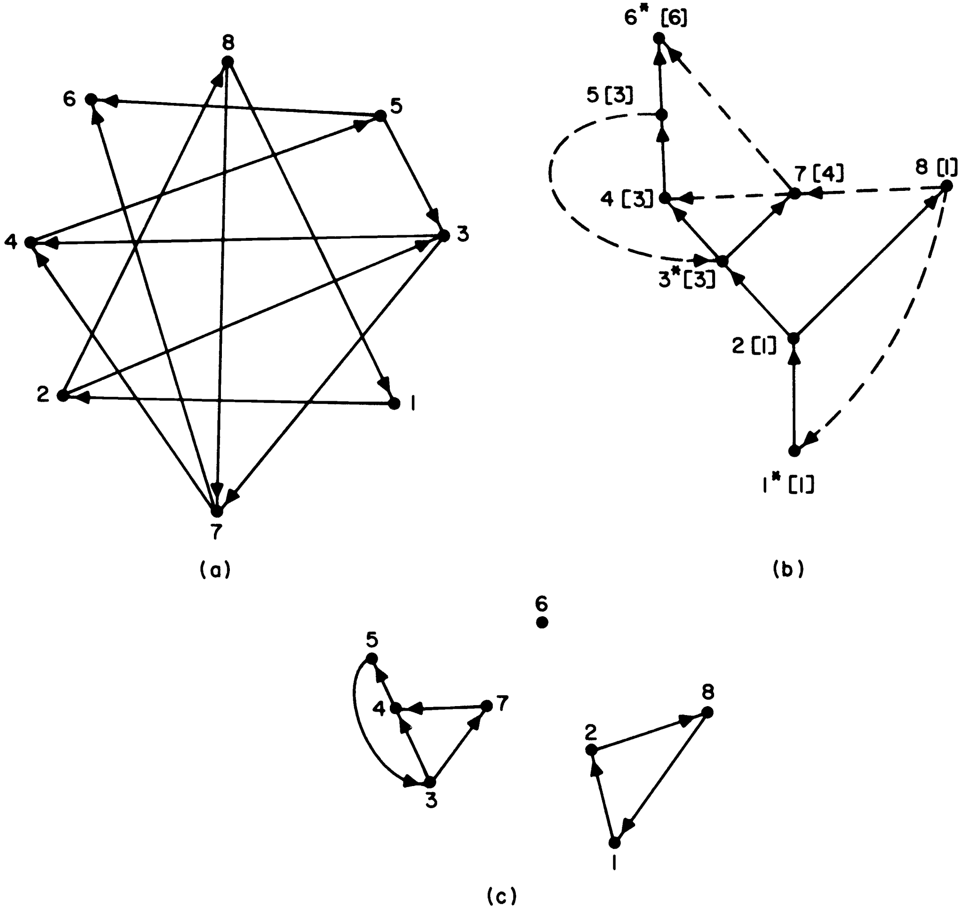


FIG. 3. *A graph and its strongly connected components*

(a) *Graph*

(b) *Jungle generated by depth-first search with* LOWLINK *values in* [ ], *roots of components marked with* *

(c) *Strongly connected components*

points once, and each point is removed from the stack of points once. Testing to see if a vertex is on the point stack can be done in a fixed time if a Boolean array is kept which answers this question for each vertex. The amount of extra time required by these operations is linear in V and E. Thus STRONGCONNECT has a time bound linear in V and E.

THEOREM 14. *The algorithm for finding strongly connected components works correctly on any directed graph G.*

Proof. We prove by induction that the calculation of LOWLINK (v) is correct. Suppose as the induction hypothesis that for all vertices v such that v is a proper descendant of vertex k or $v < k$, LOWLINK (v) is computed correctly. This means that the test to determine if v is the root of a component is performed

correctly for all such vertices v. The reader may verify that this somewhat strange induction hypothesis corresponds to considering vertices in the order they are examined for the *last* time during the depth-first search process.

Consider vertex $v = k$. Let $v \overset{*}{\rightarrow} w_1$ and let $w_1 \dashrightarrow w_2$ be a vine or cross-link such that $w_2 < v$. If vertices v and w_2 have no common ancestor, then before vertex v is reached during the search, vertex w_2 must have been removed from the stack of points and placed in a component. (The smallest numbered ancestor of vertex w_2 must be a component root.) Thus edge $w_1 \dashrightarrow w_2$ does not enter into the calculation of LOWLINK (v).

Otherwise, let u be the highest common ancestor of v and w_2. Vertex v is also the highest common ancestor of w_1 and w_2. If u is not in the same strongly connected component as w_2, then there must be a strongly connected component root on the tree path $u \overset{*}{\rightarrow} w_2$. Since $w_2 < v$, this root was discovered and w_2 was removed from the stack of points and placed in a component before the edge $w_1 \dashrightarrow w_2$ is traversed during the search. Thus $w_1 \dashrightarrow w_2$ will not enter into the calculation of LOWLINK (v). (This can only happen if $w_1 \dashrightarrow w_2$ is a cross-link.) On the other hand, if u is in the same strongly connected component as w_2, there is no component root $r \neg = u$ on the branch $u \overset{*}{\rightarrow} w_2$, and $v \dashrightarrow w_2$ will be used to calculate LOWLINK (w_2), and also LOWLINK (v), as desired. Thus LOWLINK (v) is calculated correctly, and by induction LOWLINK is calculated correctly for all vertices.

Since the algorithm correctly calculates LOWLINK, it correctly identifies the roots of the strongly connected components. If such a root u is found, the corresponding component contains all the descendants of u which are on the stack of points when u is discovered. These vertices are all on top of the stack of points, and are all put into a component by STRONGCONNECT. Thus STRONGCONNECT works correctly.

5. Further applications. We have seen how the depth-first search method may be used in the construction of very efficient graph algorithms. The two algorithms presented here are in fact optimal to within a constant factor, since every edge and vertex of a graph must be examined to determine a solution to one of the problems. (Given a suitable theoretical framework, this statement may be proved rigorously.) The similarity between biconnectivity and strong connectivity revealed by the depth-first search approach is striking. The possible uses of depth-first search are very general, and are certainly not limited to the examples presented. Hopcroft and Tarjan have constructed an algorithm for finding triconnected components in $O(V, E)$ time by extending the biconnectivity algorithm [8]. An algorithm for testing the planarity of a graph in $O(V)$ time [15] is also based on depth-first search. Combining the connectivity algorithms, the planarity algorithm, and an algorithm for testing isomorphism of triconnected planar graphs [7], we may construct an algorithm to test isomorphism of arbitrary planar graphs in $O(V \log V)$ time [8]. Depth-first search is a powerful technique with many applications.

REFERENCES

[1] B. L. FOX AND D. M. LANDY, *An algorithm for identifying the ergodic subchains and transient states of a stochastic matrix*, Comm. ACM, 11 (1968), pp. 619–621.

[2] S. W. GOLOMB AND L. D. BAUMERT, *Backtrack programming*, J. ACM, 12 (1965), pp. 516–524.

[3] F. HARARY, *Graph Theory*, Addison-Wesley, Reading, Mass., 1969.

[4] J. HOPCROFT AND R. TARJAN, *Efficient algorithms for graph manipulation*, Tech. Rep. 207, Computer Science Department, Stanford University, Stanford, Calif., 1971.

[5] ———, *A V^2 algorithm for determining isomorphism of planar graphs*, Information Processing Letters, 1 (1971), pp. 32–34.

[6] ———, *Planarity testing in V log V steps: Extended abstract*, Tech. Rep. 201, Computer Science Department, Stanford University, Stanford, Calif., 1971.

[7] J. HOPCROFT, *An N log N algorithm for isomorphism of planar triply connected graphs*, Tech. Rep. 192, Computer Science Department, Stanford University, Stanford, Calif., 1971.

[8] J. HOPCROFT AND R. TARJAN, *Isomorphism of planar graphs*, IBM Symposium on Complexity of Computer Computations, Yorktown Heights, N.Y., March, 1972, to be published by Plenum Press.

[9] J. HOPCROFT AND J. ULLMAN, *A linear list-merging algorithm*, Tech. Rep. 71-111, Cornell University Computer Science Department, Ithaca, N.Y., 1971.

[10] I. MUNRO, *Efficient determination of the strongly connected components and transitive closure of a directed graph*, Department of Computer Science, University of Toronto, 1971.

[11] N. J. NILSON, *Problem Solving Methods in Artificial Intelligence*, McGraw-Hill, New York, 1971.

[12] K. PATON, *An algorithm for the blocks and cutnades of a graph*, Comm. ACM, 14 (1971), pp. 468–475.

[13] P. W. PURDOM, *A transitive closure algorithm*, Tech. Rep. 33, Computer Sciences Department, University of Wisconsin, Madison, 1968.

[14] R. W. SHIREY, *Implementation and analysis of efficient graph planarity testing algorithms*, Doctoral thesis, Computer Sciences Department, University of Wisconsin, Madison, 1969.

[15] R. TARJAN, *An efficient planarity algorithm*, Tech. Rep. 244, Computer Science Department, Stanford University, Stanford, Calif., 1971.

SIAM J. COMPUT.
Vol. 1, No. 2, June 1972

A NOTE ON MERGING*

ALAN G. KONHEIM†

Abstract. This paper studies the merging operation when data is stored on the surfaces of a disk. The data consists of q lists (each of n numbers) stored on a disk. Each list is stored on one surface of a disk. The time needed to merge into a single (sorted) list, T, is a function of the required motion of the reading head. In this paper we calculate the expected value of T.

Key words. Merging, sorting.

In this note we shall calculate the "time" needed to carry out a merging operation. It represents an idealization of the type of operation employed in a computer in the sorting of numerical data.

Let $\{X_{i,j}: i = 1, \cdots, q, j = 1, \cdots, n\}$ be $N = nq$ numbers which are "stored" as q lists (of n numbers each). We assume that each of the q sequences $X_{i,1}, \cdots, X_{i,n}$ are in their natural order and that all of the numbers $\{X_{i,j}\}$ are distinct. To merge the q lists is to form the sequence $\{X_i: i = 1, \cdots, N\}$ obtained by arranging in natural order the numbers $\{X_{i,j}\}$. We define the position vector $\mathbf{v}_i = (v_{i,1}, v_{i,2})$ by

$$X_i = X_{v_{i,1}, v_{i,2}},$$

where $v_{i,1}$ is the *number* of the list containing the entry X_i and $v_{i,2}$ is the rank of X_i within this list. The "time" needed to merge these q lists is defined as

$$\tau = \sum_{i=1}^{N-1} |v_{i,2} - v_{(i+1),2}|.$$

In order to see the relationship between τ and time, we digress briefly in order to describe the physical nature of the storage and merging process.

The q lists are stored on the surfaces of a device called a *disk pack* (Fig. 1). This device consists of a number of storage surfaces which are divided into tracks. Information is written magnetically on these tracks. The basic unit of data is a *record* consisting of a numerical *key* (our numbers $\{X_{i,j}\}$) together with supplementary data. In this analysis we assume:

(i) one record is stored per track;
(ii) the records corresponding to the keys in one list are stored on consecutive tracks on the same storage surface; and
(iii) distinct lists are stored on different storage surfaces.

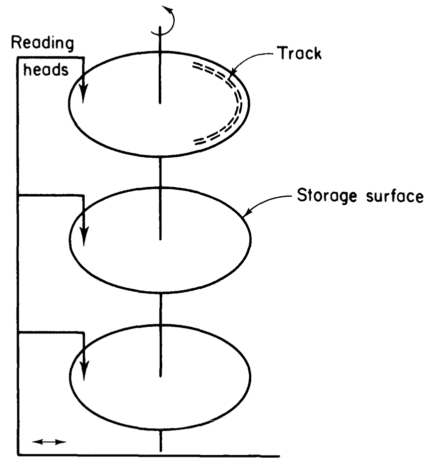
Access to the contents of a disk pack is provided through a collection of reading heads, with one head for each storage surface of the disk. The heads move together radially so that they can read from the corresponding track on any storage surface. Only one of the heads can be active (and read) at a time. To read from a different track requires the radial motion of the entire collection of heads.

The sorting of records consists of two phases:

(a) the production of strings of records, whose keys are in their natural order, and the writing of these strings on a single surface of the disk pack;

* Received by the editors June 2, 1970, and in revised form October 9, 1970.

† Thomas J. Watson Research Center, IBM Corporation, Yorktown Heights, New York 10598.
This research was supported in part by the United States Air Force under Contract AF 49(638)-1682.



DISK PACK

FIG. 1

(b) the merging of the resulting strings.

Our analysis is concerned with the second phase of the sorting operation. Note that we have made the simplifying assumption that the strings have a common length.

The coordinates of the position vector $\mathbf{v}_i$ give the number of the surface on which the record with key X_i is stored and the corresponding track number. The quantity τ corresponds to the total (radial) movement of the reading heads.

Since only the relative ordering of the numbers $\{X_{i,j}\}$ is relevant to the analysis, we may assume that $\{X_{i,j} = 1, 2, \cdots, N\}$. We need to specify now the process by which the q lists are formed. The q lists are determined by a chance experiment. We choose with the uniform distribution q subsets of size n from the set $\{1, 2, \cdots, N\}$. There are $(qn)!/(n!)^q$ such subdivisions and we postulate that all are equally probable.

LEMMA 1.

$$\Pr\{v_{i,1} = \mu, v_{(i+1),1} = \zeta\} = \begin{cases} \dfrac{n}{q(qn-1)} & \text{if } \mu \neq \zeta, \\ \dfrac{n-1}{q(qn-1)} & \text{if } \mu = \zeta. \end{cases}$$

Proof. Suppose first that $\mu \neq \zeta$; the μth list, $\mathscr{L}_\mu$, contains $n - 1$ elements different from i and $i + 1$ and these may be selected in $\binom{qn-2}{n-1}$ ways. Similarly, $\mathscr{L}_\zeta$ contains $n - 1$ elements different from i and $i + 1$ and these may be chosen from the remaining $(q - 1)n - 1$ elements in $\binom{q-n-1}{n-1}$ ways. But the number of ways of selecting two subsets of size n from a set of nq elements is $\binom{qn}{n}\binom{q-n}{n}$

and this yields the assertion for $\mu \neq \zeta$. Finally we note that

$$q(q-1)\frac{n}{q(qn-1)} + q \Pr\{v_{i,1} = v_{(i+1),1} = 1\} = 1,$$

which completes the proof.

LEMMA 2. *If $\mu \neq \zeta$, then the conditional probability*

$$\Pr\{v_{i,2} - v_{(i+1),2} = -m/v_{i,1} = \mu, v_{(i+1),1} = \zeta\}$$

is given by

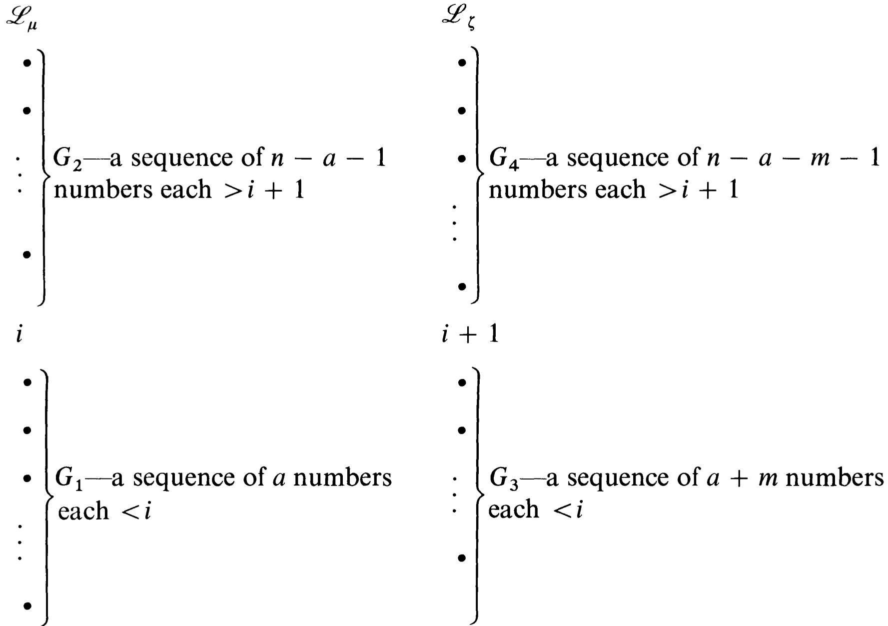
$$\sum_a f(i,a,m)\binom{qn-2}{n-1}^{-1}\binom{qn-n-1}{n-1}^{-1}$$

with

$$f(i,a,m) = \binom{i-1}{a}\binom{i-a-1}{a+m}\binom{qn-i-1}{n-a-1}\binom{qn-n-i+a}{n-a-m-1}$$

and where the summation is over all integers a for which all binomial coefficients have nonnegative arguments.

Proof. We can picture the situation as follows:



We then

(i) choose the a numbers in G_1 in $\binom{i-1}{a}$ ways; thereafter

(ii) choose the $a+m$ numbers in G_3 in $\binom{i-a-1}{a+m}$ ways; thereafter

(iii) choose the $n - a - 1$ numbers in G_2 in $\binom{qn - i - 1}{n - a - 1}$ ways; and finally,

(iv) choose the $n - a - m - 1$ numbers in G_4 in $\binom{qn - n - i + a}{n - a - m - 1}$ ways.

Thus $f(i, a, m)$ is the number of such configurations with the value a. Finally, there are $\binom{qn - 2}{n - 1}$ ways of choosing the elements of $\mathscr{L}_\mu$ and thereafter $\binom{qn - n - 1}{n - 1}$ ways of choosing the elements of $\mathscr{L}_\zeta$.

We note for further reference the following lemma.

LEMMA 3.

$$\Pr\{v_{i,2} - v_{(i+1),2} = -m/v_{i,1} = v_{(i+1),1} = \mu\} = \delta_{m,1}.$$

THEOREM.

$$E(\tau) = (qn - 1)\left[\frac{n - 1}{qn - 1} + \frac{qn - n}{qn - 1}\binom{n - 1}{2n - 1}\frac{2^{2n-3}}{\binom{2n - 2}{n - 1}}\right]$$

$$\sim \frac{q - 1}{4} n\sqrt{\pi n} \quad \textit{as } n \to \infty.$$

Proof. We shall first evaluate the sum $\sum_i \sum_m \sum_a f(i, a, m)|m|$ by interchanging the order of summation. First we write $f(i, a, m)$ in the equivalent form

$$f(i, a, m) = \binom{2a + m}{a}\binom{2n - 2a - m - 2}{n - a - m - 1}\binom{i - 1}{2a + m}\binom{qn - i - 1}{2n - 2a - m - 2}.$$

We first sum over i. We may assume that $m \geqq 0$ noting that under the condition $v_{i,1} \neq v_{(i+1),1}$, the random variable $v_{i,2} - v_{(i+1),2}$ has a symmetric distribution. We use the formula [1, p. 35]

$$\sum_{j=0}^{t-s} \binom{j}{r}\binom{t - j}{s} = \binom{t + 1}{r + s + 1} \tag{1}$$

which is a little known variant of the Vandermonde convolution. From (1) we find

$$\sum_i f(i, a, m) = \binom{2a + m}{a}\binom{2n - 2a - m - 2}{n - a - m - 1}\binom{qn - 1}{2n - 1}.$$

Next we use a formula of Jensen [1, pp. 148, 170]:

$$\sum_{j=0}^{t} \binom{x + jz}{j}\binom{y - jz}{t - j} = \sum_{j=0}^{t} \binom{x + y - j}{t - j} z^j, \tag{2}$$

obtaining

$$\sum_a \sum_i f(i, a, m) = \binom{qn - 1}{2n - 1} \sum_{j=0}^{n-m-1} \binom{2n - j - 2}{n - m - j - 1} 2^j.$$

This latter sum can be written in a different form; we use the formula

$$\sum_{j=0}^{t}\binom{z+1}{j}x^{n-j} = \sum_{j=0}^{t}\binom{z-j}{t-j}(x+1)^{j} \tag{3}$$

which is a direct consequence of (1) (if we expand $(x+1)^j$ by the binomial theorem). This yields

$$\sum_a\sum_i f(i,a,m) = \binom{qn-1}{2n-1}\sum_{j=0}^{n-m-1}\binom{2n-1}{j}.$$

It remains to multiply by m and carry out the summation over m. We have

$$\sum_{m=0}^{n-1} m\sum_a\sum_i f(i,a,m) = \binom{qn-1}{2n-1}\sum_{j=0}^{n-1}\binom{2n-1}{j}\binom{n-j}{2} = \binom{qn-1}{2n-1}\xi_n.$$

Now

$$2\xi_n = \tfrac{1}{2}n(n-1)\eta_n(1) - \tfrac{1}{2}(2n-1)\eta_n'(1) + \tfrac{1}{2}[\eta_n''(1) + \eta_n'(1)],$$

where

$$\eta_n(x) = \sum_{j=0}^{2n-1}\binom{2n-1}{j}x^j = (1+x)^{2n-1}$$

and thus

$$\sum_m |m|\sum_i\sum_a f(i,a,m) = 2^{2n-3}(n-1)\binom{qn-1}{2n-1}.$$

Finally we have by Lemma 1,

$$E\left\{\sum_i |v_{i,2} - v_{(i+1),2}|\,;\, v_{i,1} \neq v_{(i+1),1}\right\} = (q-1)n\binom{n-1}{2n-1}2^{2n-3}\frac{1}{\binom{2n-1}{n-1}}.$$

Adding to this the term

$$E\left\{\sum_i |v_{i,2} - v_{(i+1),2}|\,;\, v_{i,1} = v_{(i+1),1}\right\}$$

(which is $n-1$ by Lemmas 1 and 3), we finally obtain

$$E\left\{\sum_i |v_{i,2} - v_{(i+1),2}|\right\} = (n-1)\left[1 + \frac{qn-n}{2n-1}2^{2n-3}\frac{1}{\binom{2n-2}{n-1}}\right].$$

Using the asymptotic estimate

$$\binom{2r}{r} \sim 2^{2r}(\pi r)^{-1/2} \quad \text{as } r \to \infty,$$

we get

$$E\left\{\sum_i |v_{i,2} - v_{(i+1),2}|\right\} \sim \left(\frac{q-1}{4}\right)n\sqrt{\pi n} \quad \text{as } n \to \infty.$$

Acknowledgment. We would like to thank Mr. Luther Woodrum for suggesting this problem and the referee for several suggestions concerning the presentation.

REFERENCE

[1] JOHN RIORDAN, *Combinatorial Identities*, John Wiley, New York, 1968.

SIAM J. COMPUT.
Vol. 1, No. 2, June 1972

COMPUTATIONAL COMPLEXITY OF ITERATIVE PROCESSES*

J. F. TRAUB†

Abstract. The theory of optimal algorithmic processes is part of computational complexity. This paper deals with *analytic computational complexity*. The relation between the goodness of an iteration algorithm and its new function evaluation and memory requirements are analyzed. A new conjecture is stated.

Key words. computational complexity, optimal algorithm, optimal iteration, numerical mathematics, iteration theory.

1. Introduction. Computational complexity is one of the foundations of theoretical computer science. The phrase computational complexity seems to have been first used by Hartmanis and Stearns [12] in 1965 although the first papers belonging to the field are those of Rabin [28], [29] in 1959 and 1960.

One of its important components is the theory of optimal algorithmic processes. We distinguish between optimality theory for algebraic (or combinatorial) processes, which we call *algebraic computational complexity*, and optimality theory for analytic (or continuous) processes, which we call *analytic computational complexity*.

The last few years have witnessed striking developments in algebraic computational complexity; for example, the multiplication of numbers (Cook [6], Schönhage and Strassen [31]), the multiplication of matrices (Winograd [41], Strassen [32], Hopcroft and Kerr [14]), polynomial evaluation (Winograd [41]), median of a set of numbers (Floyd [10]), graph planarity (Hopcroft and Tarjan [15]). Surveys may be found in Knuth [19], Borodin [1], and Minsky [24].

Research on analytic computational complexity dates to the early sixties (Traub [33]–[39]) and predates most of the algebraic results. More specifically, the work on analytic computational complexity to date has concerned optimal iteration. Recent results are due to Brent [2], Cohen [3], Cohen and Varaiya [4], Feldstein [7], Feldstein and Firestone [8], [9], Hindmarsh [13], Jarratt [16], King [18], Kung [21], Miller [22], [23], Paterson [27], Rissanen [30], and Winograd and Wolfe [42], [43]. (Kung and Paterson's results are summarized at the end of § 2.)

In this paper we define basic concepts and pose some fundamental questions in optimal iteration. In the terminology of Knuth [20] we perform a Type B analysis. That is, we consider a family of algorithms for solving a particular problem and select the "best possible." We survey earlier work, report recent progress, and state a new conjecture. Since the field is changing rapidly, some of the results cited have not yet appeared in the open literature. An abbreviated version of this material was presented (Traub [40]) at the IFIP 71 Congress, with somewhat different terminology and notation.

* Received by the editors March 8, 1972.

† Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213. This research was supported in part by the Office of Naval Research under Contract N 00014-67-A-0314-0010, NR 044-422, and by the Advanced Research Projects Agency under Contract F44620-70-C-0107.

This paper is intended for the nonspecialist in iteration theory and therefore some precision in definitions and some generality in the models of iteration algorithms are sacrificed.

2. Basic concepts. We begin by specifying the problem. Let F denote the class of infinitely differentiable real functions defined on the real line. We assume that if $f \in F$, then f has at least one simple zero α, that is, a number α such that $f(\alpha) = 0$, $f'(\alpha) \neq 0$. The assumption of infinite differentiability is for simplicity. For any algorithm we shall discuss, f need only have a small number of derivatives on a finite interval.

Our problem is to approximate α for $f \in F$. This zero-finding problem may seem rather specialized, but, in fact, it is equivalent to the fixed-point problem of calculating a number α such that $\alpha = g(\alpha)$, a ubiquitous problem in mathematics and applied mathematics. It may be formulated in an abstract setting and covers partial differential equations, integral equations, boundary value problems for ordinary differential equations as well as many other important problems (Collatz [5]).

We consider iterative algorithms for the approximation of α. A sequence of approximating iterates $\{x_i\}$ is generated by an iteration function. We shall not give a formal definition of iteration algorithm. The interested reader may consult Ortega and Rheinboldt [25] and Cohen and Varaiya [4].

Our aim is to discuss optimal iteration algorithms. There are a number of measures we could optimize. For example, we could minimize the total number of arithmetic operations needed to approximate α to within an error ε. This measure is strongly dependent on the particular f in question. For our current purpose, we prefer a measure which is not so dependent on f and which is easier to calculate. (At the end of this section we report recent optimality results which optimize arithmetic operations.)

We introduce general measures of cost and goodness. The cost consists of two parts: the new evaluation cost e and the memory cost m.

DEFINITION. The *new evaluation cost* e is defined as the number of new function evaluations required.

This definition is motivated by the following considerations. An iteration step consists of two parts.

1. Calculate new function values.
2. Combine the data to calculate the next iterate.

Since the evaluation of functions requires invocation of subroutines whereas the calculation of the next iterate requires only a few arithmetic operations, we neglect the latter.

A function evaluation is the calculation of f or one of its derivatives. Thus if $f(x_i)$ and $f'(x_i)$ are required, $e = 2$. We could assign a new evaluation cost of θ_j for the evaluation of $f^{(j)}$ (Traub [39, p. 262]), but this would make the measure f-dependent.

We turn to the second component of the cost.

DEFINITION. If previous function evaluations at $x_{i-1}, \cdots, x_{i-m}$ are used to calculate x_{i+1}, then we define m as the *memory cost of the iteration.*

Another component of the cost is not included in this paper. An iteration such as the secant iteration involves the subtraction of quantities which are close

together, and to maintain accuracy, more precision should perhaps be carried. The theory should be extended to include this cost.

We turn now to a measure of the goodness of an iteration. Let $x_i \to \alpha$.

DEFINITION. If there exists a number p such that

$$\lim_{i\to\infty} \frac{|x_{i+1} - \alpha|}{|x_i - \alpha|^p} = A \neq 0, \infty,$$

then p is called the *order of the iteration.*

This definition of order will serve for our purposes. For other definitions of order the reader is referred to Ortega and Rheinboldt [25] and Cohen and Varaiya [4].

This is a reasonable measure of goodness since if x_i is near α, then x_{i+1} has about p times as many significant figures as x_i. A discussion may be found in Traub [39, Appendix C].

The order has two additional properties which make it useful for our purposes. It depends primarily on the algorithm and only weakly on f and it is fairly easy to calculate. For example, for all twice continuously differentiable functions f for which $f''(\alpha) \neq 0$, Newton iteration (see Example 1 below) has order $p = 2$. (Recall we are assuming throughout this paper that $f'(\alpha) \neq 0$.) Under the same conditions, the secant iteration has order $p = \frac{1}{2}(1 + \sqrt{5}) \doteq 1.62$.

Two widely known iteration algorithms may serve to illuminate these definitions. We shall use them to introduce data flow charts which are a convenient way to describe algorithms from our point of view.

Example 1. *Newton iteration.* Let x_0 be given. Define

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = \Phi[x_i, f(x_i), f'(x_i)].$$

The data flow chart of Fig. 1 exhibits the process at step i. For Newton iteration,

$$e = 2, \quad m = 0, \quad p = 2 \quad (\text{if } f''(\alpha) \neq 0).$$

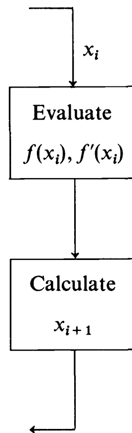


FIG. 1. *Data flow chart for Newton iteration*

Example 2. *Secant iteration.* Let x_0, x_1, be given. Define

$$x_{i+1} = x_i - f(x_i)\frac{(x_i - x_{i-1})}{f(x_i) - f(x_{i-1})}$$
$$= \Phi[x_i, x_{i-1}, f(x_i), f(x_{i-1})].$$

The data flow chart of Fig. 2 exhibits the process at step i. For secant iteration,

$$e = 1, \quad m = 1, \quad p = \tfrac{1}{2}(1 + \sqrt{5}) \doteq 1.62 \quad (\text{if } f''(\alpha) \neq 0).$$

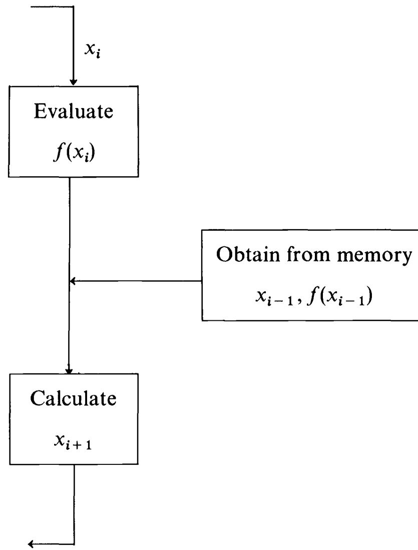


FIG. 2. *Data flow chart for secant iteration*

We now pose the following optimality questions which will be our focus for the remainder of this paper. Other optimality problems will be discussed at the end of this section.

2.1. Two optimality questions.

1. *What is the maximal order $P_{e,m}$ which can be achieved for iterations which use e new function evaluations and have memory m?*
2. *What is the most that memory m is worth? That is, what is $P_{e,m} - P_{e,0}$?*

The answers depend on the class of iterations under study. Traub [39, § 1.22] introduced four classes depending on the function evaluation and memory requirements of the algorithms. These classes are:

one-point;
one-point with memory;
multipoint;
multipoint with memory.

We shall discuss optimality results for only the first three classes in this paper.

These classes model algorithms appropriate for *stationary iterations* on *sequential machines.* An iteration rule is stationary if it does not change from step to step. A formal definition may be found in Ortega and Rheinboldt [25]. Because of the assumption of sequential machine, the definition of one-point iteration with memory (§ 5) uses the same number of derivatives at each point. On parallel machines we may want to vary the number of derivatives at each point. The case where the number of derivatives varies is studied by Traub [39, pp. 60–65] and Feldstein and Firestone [8].

Besides those posed earlier, we discuss some additional optimality questions. An important measure of the goodness of an algorithm is the efficiency index defined by

$$E = (\log_2 p)/e.$$

This measure is defined without motivation by Ostrowski [26, Chap. 3]. A derivation may be found in Traub [39, Appendix C]. Gentleman [11] gives an axiomatic treatment. A study of iterations with high values of the efficiency index is reported by Feldstein and Firestone [9].

When we consider algorithms for a fixed problem, it becomes meaningful to optimize relative to the number of arithmetic statements. Paterson [27] takes for his efficiency measure

$$\bar{E} = (\log_2 p)/\bar{M},$$

where p denotes the order and $\bar{M}$ denotes the number of multiplications or divisions. He excludes from $\bar{M}$ multiplication or division by a constant. Paterson considers iterations ϕ which have the following properties:

(i) ϕ is a rational function;
(ii) ϕ is univariate;
(iii) $\lim_{i\to\infty} x_i$ is an algebraic number;
(iv) ϕ has rational coefficients.

Under these conditions, Paterson proves that $\bar{E} \leqq 1$.

The Newton iteration for the problem $f = x^2 - A$ (which converges to $\sqrt{A}$),

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{A}{x_i}\right),$$

has $p = 2$, $\bar{M} = 1$. Hence $\bar{E} = 1$ which shows the result is sharp.

Kung [21] defines the *multiplication efficiency*

$$E = (\log_2 p)/M,$$

where M is the total number of multiplications or divisions required. He removes all restrictions on ϕ except condition (i). *Kung shows that* $E \leqq 1$. Since condition (i) is not a restriction for a computer algorithm, this is a very general result.

For the iteration

$$x_{i+1} = x_i^2 + x_i - \tfrac{1}{4},$$

which converges to $-\frac{1}{2}$, $p = 2$, $M = 1$. Hence $E = 1$ which shows the bound is sharp. On the other hand, $E = \frac{1}{2}$ for the Newton iteration for the square root in Kung's measure.

Kung derives another interesting result. Let P_M denote the maximal order for a sequence generated by an iteration with M multiplications. Then $P_M \leqq 2^M$ for all positive integers M. Moreover, this bound is sharp.

3. Interpolatory iteration. Before discussing optimality results for classes of iterations, we discuss particular families of iterations which play a special role in the theory, the interpolatory iteration algorithms $I_{e,m}$ introduced and analyzed by Traub [38], [39]. For our purpose here, we need not know how formulas for interpolatory iteration are derived. Indeed, there are two families of interpolatory iterations derived from direct and inverse iteration. Both families have the same order for a given e and m and we shall not distinguish between them. In both families, $I_{2,0}$ is Newton iteration and $I_{1,1}$ is secant iteration.

For interpolatory iterations we have a complete theory relating order to evaluation and memory costs. Let $q_{e,m}$ denote the order of an interpolatory iteration $I_{e,m}$. Then we have the following basic result.

THEOREM (Traub [39, § 3.3 and § 6.1]). *$q_{e,0} = e$. For all finite e and $m > 0$, $e < q_{e,m} < e + 1$. For e fixed, $q_{e,m}$ is a strictly increasing function of m and*

$$\lim_{m \to \infty} q_{e,m} = e + 1.$$

This is a very satisfying result. It says that for interpolatory iteration, increasing memory while keeping the number of new evaluations fixed always increases the order.

The following is an important corollary.

COROLLARY (Traub [39, § 6.1]). *For all finite m, $q_{e,m} - q_{e,0} < 1$.*

Thus for interpolatory iterations memory adds less than unity to the order.

Upper and lower bounds on the order are given by the following theorems. Let

$$\delta_{e,m} = e + 1 - q_{e,m},$$

and let ε denote the base of natural logarithms.

THEOREM (Traub [39, § 3.3]).

$$\frac{e}{(e+1)^m} < \delta_{e,m} < \frac{\varepsilon e}{(e+1)^m}.$$

A sharper result is given by the next theorem.

THEOREM (Kahan [17]).

$$\frac{e}{(e+1)^{m+1} - 1} \leqq \delta_{e,m} \leqq \frac{e}{(e+1)^{m+1} - 1 - em/(e+1)}.$$

Values of $q_{e,m}$ for small values of e and m may be found in Table 1.

4. One-point iteration.

DEFINITION. An iteration function belongs to the class of *one-point iterations* if all new function evaluations are at the point x_i and if its memory $m = 0$.

Thus

$$x_{i+1} = \Phi_{e,0}[x_i, f(x_i), \cdots, f^{(e-1)}(x_i)].$$

The data flow chart for a one-point iteration is given in Fig. 3.

TABLE 1
Values of $q_{e,m}$

m \ e	1	2	3
0	1.000	2.000	3.000
1	1.618	2.732	3.791
2	1.839	2.920	3.951
3	1.928	2.974	3.988

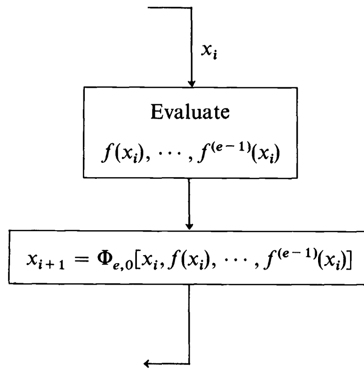


FIG. 3. *Data flow chart for one-point iteration*

For one-point iterations the first optimality question is settled by the theorem below. Recall that $P_{e,m}$ is the optimal order for an iteration characterized by new function evaluations e and memory m.

THEOREM (Traub [33], [39, § 5.4]).

$$P_{e,0} = e.$$

5. One-point iteration with memory.

DEFINITION. An iteration function belongs to the class of *one-point iterations with memory* if all new function evaluations are at the point x_i and if its memory $m > 0$.

Thus

$$x_{i+1} = \Phi_{e,m}[x_i, f(x_i), \cdots, f^{(e-1)}(x_i); x_{i-1}, f(x_{i-1}), \cdots, f^{(e-1)}(x_{i-1}), \cdots, x_{i-m}, f(x_{i-m}), \cdots, f^{(e-1)}(x_{i-m})].$$

The semicolon separates new function evaluations from those recovered from memory. The data flow chart for a one-point iteration with memory is given in Fig. 4.

The initial conjecture on optimality for this class was reported at the 1961 National ACM Conference.

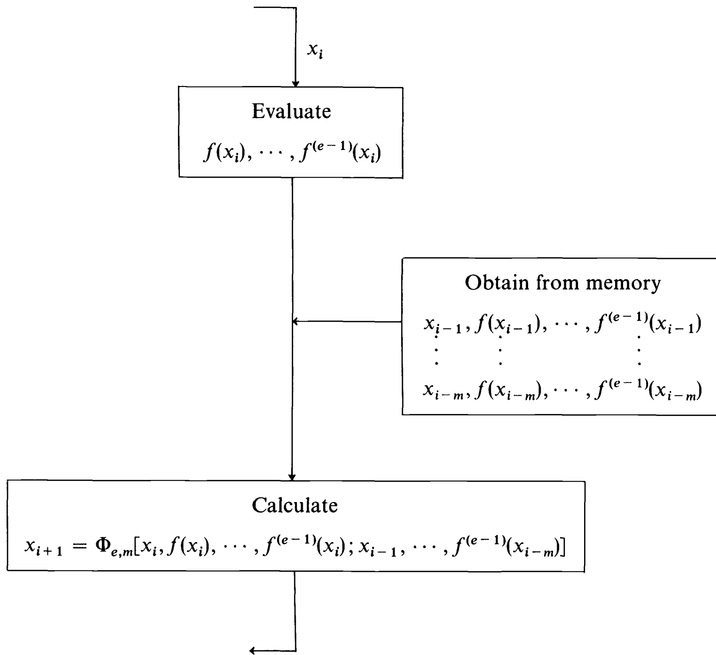


FIG. 4. *Data flow chart for one-point iteration with memory*

CONJECTURE (Traub [33], [39, § 6.3]). *For all one-point iterations with finite memory m,*

$$P_{e,m} - P_{e,0} < 1.$$

Until the late sixties no progress was reported, but there have been exciting recent results. The matter has been investigated by Winograd and Wolfe [42] who assert a stronger result. Under weak conditions on the admissible iteration functions, interpolatory iteration $I_{e,m}$ is optimal among all iterations characterized by new function evaluations e and memory m. The truth of the conjecture then follows from the corollary in § 3.

Winograd and Wolfe [42] have pointed out an ambiguity in the notion of memory since instead of using memory explicitly at each step, one can use it implicitly by encoding it in other data. Cohen and Varaiya [4] cite an example of such an encoding. Cohen and Varaiya deal with the ambiguity by adding a condition to the definition of order which insures that encoding does not increase the rate. Winograd and Wolfe [43] deal with the case where all past points are remembered. This side-steps the encoding issues.

Rissanen [30] resolves the ambiguity by imposing a smoothness condition on admissible algorithms. He proves that then the secant iteration (that is, the interpolatory iteration $I_{1,1}$) has maximal order among all algorithms one with $e = 1$, $m = 1$.

6. Multipoint iteration. We summarize the situation for one-point iterations with or without memory. A one-point iteration with e new function evaluations (and therefore $e - 1$ derivatives) is of order at most e. A one-point iteration with memory with e new function evaluations (and therefore $e - 1$ derivatives) is of order less than $e + 1$. Table 2 summarizes the situation.

TABLE 2
Summary of facts about iteration functions

	New function evaluations	Highest derivative	Optimal order
One-point	e	$e-1$	e
One-point with memory	e	$e-1$	$<e+1$

Is there a class of iteration algorithms for which these restrictions do not hold? An affirmative answer is provided by multipoint iterations (Traub [37], [39, § 1.2]).

DEFINITION. An iteration function belongs to the class of *multipoint iterations* if new function evaluations are made at more than one point and if its memory $m = 0$.

We shall confine ourselves to giving a general prescription and a data flow chart of a multipoint iteration only for the case of a two-point iteration. Then

$$z_i = \phi[x_i, f(x_i), \cdots, f^{(e_1-1)}(x_i)],$$
$$x_{i+1} = \psi[x_i, f(x_i), \cdots, f^{(e_1-1)}(x_i), z_i, f(z_i), \cdots, f^{(e_2-1)}(z_i)].$$

The data flow chart is given by Fig. 5.

A fourth class of iterations, multipoint with memory, is defined by Traub [39, § 1.2]. We shall not discuss multipoint iteration with memory here.

Table 2 lists two types of requirements, one on the total number of new function evaluations and a second on the highest derivative required. First we give examples to show that the restriction on derivatives need not apply for multipoint iterations.

Example 3.

$$x_{i+1} = \psi(x_i) = \frac{x_i\phi(\phi(x_i)) - \phi^2(x_i)}{x_i - 2\phi(x_i) + \phi(\phi(x_i))},$$
$$\phi(x_i) = x_i - f(x_i).$$

This is a particular case of the Steffensen–Householder–Ostrowski iteration (Traub [39, Appendix D]). Note that *no* derivatives are used. Yet if $f'(\alpha) \neq 1$, $p = 2$.

Example 4. Let $L \geqq 3$ be fixed and let

$$x_{i+1} = \phi[x_i, f(x_i), f'(x_i), f(\lambda_2), \cdots, f(\lambda_{L-1})],$$

where

$$\lambda_j = \lambda_j(x_i) = \lambda_{j-1}(x_i) - \frac{f(\lambda_{j-1}(x_i))}{f'(x_i)}, \qquad j = 2, \cdots, L-1,$$

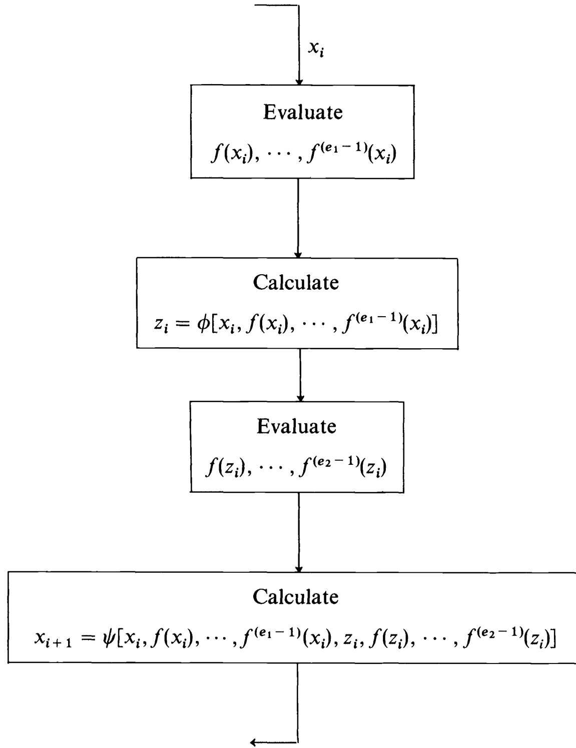


FIG. 5. *Data flow chart for two-point iteration*

and

$$\lambda_1(x_i) = x_i.$$

This is a multipoint iteration based on $L - 1$ points. The new function evaluations are $L - 1$ evaluations of f and one of f'. For all twice continuously differentiable f for which $f''(\alpha) \neq 0$, this iteration is of order L (Traub [39, § 8.34]).

These two examples show that for multipoint iterations there is no connection between the highest derivative required and the order.

For these two examples, the order equals the number of new function evaluations. Since we proved this was always the case for one-point iterations, we might be tempted to suppose that this result holds for multipoint iterations also. That this is not the case is shown by the following example.

Example 5.

$$z_i = x_i - \frac{f(x_i)}{f'(x_i)},$$

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}\left[\frac{f(z_i) - f(x_i)}{2f(z_i) - f(x_i)}\right].$$

The data flow chart is given in Fig. 6 and a picture in Fig. 7.

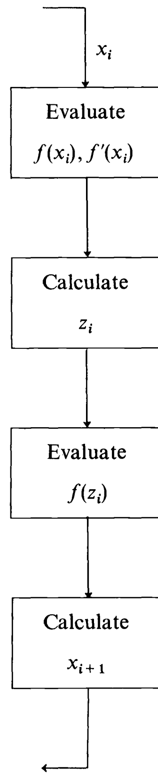


FIG. 6. *An example of a multipoint iteration*

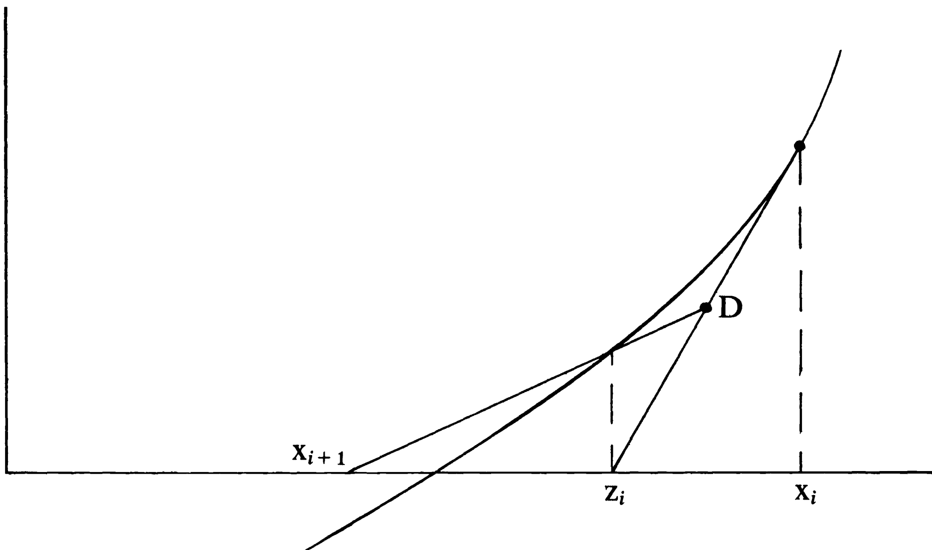


FIG. 7. *Geometric interpretation. D is the midpoint of the line between* $(z_i, 0)$ *and* $(x_i, f(x_i))$.

This iteration uses two evaluations of f and one of f' and is of order 4. Jarratt [16] has constructed a fourth order iteration using just two evaluations of f' and one of f. King [18] constructs a family of fourth order methods which use two values of f and one value of f'.

We turn to optimality considerations for multipoint iterations. As before let $P_{e,0}$ denote the maximal order for an iteration with new function evaluations e and no memory. If we permit only one-point or multipoint iterations (no memory), we know that $P_{2,0} \geqq 2$ (Newton iteration) and $P_{3,0} \geqq 4$ (Example 5 above).

NEW CONJECTURE. *For all one-point or multipoint iterations without memory,*

$$P_{2,0} = 2, \quad P_{3,0} = 4.$$

REFERENCES

[1] A. BORODIN, *Computational complexity—a survey*, Proc. Fourth Annual Princeton Conference on Information Sciences and Systems, Princeton, N.J., 1970, pp. 257–262.
[2] R. BRENT, *On maximizing the efficiency for solving systems of nonlinear equations*, to appear.
[3] A. I. COHEN, *Rate of convergence and optimality conditions of root finding and optimization algorithms*, Doctoral thesis, University of California, Berkeley, 1970.
[4] A. I. COHEN AND P. VARAIYA, *Rate of convergence and optimality conditions of root finding*, to appear.
[5] L. COLLATZ, *Functional Analysis and Numerical Mathematics*, Academic Press, New York, 1964.
[6] S. A. COOK, *On the minimum computation time for multiplication*, Doctoral thesis, Harvard University, Cambridge, 1966.
[7] A. FELDSTEIN, *Bounds on order and Ostrowski efficiency for interpolatory iteration algorithms*, Lawrence Radiation Laboratory, University of California, Livermore, 1969.
[8] A. FELDSTEIN AND R. M. FIRESTONE, *Hermite interpolatory iteration theory and parallel numerical theory*. Rep., Division of Applied Mathematics, Brown University, Providence, 1967.
[9] ———, *A study of Ostrowski efficiency for composite iteration functions*, Proc. ACM National Conference, 1969, pp. 147–155.
[10] R. W. FLOYD, *Notes on computing medians and percentiles*, to appear.
[11] W. M. GENTLEMAN, Private communication, 1970.
[12] J. HARTMANIS AND R. E. STEARNS, *Computational complexity of recursive sequences*, IEEE Proc. Fifth Annual Symposium on Switching Circuit Theory and Logical Design, 1964, pp. 82–90.
[13] A. C. HINDMARSH, *Optimality in a class of rootfinding algorithms*, SIAM J. Numer. Anal., 9 (1972).
[14] J. E. HOPCROFT AND L. E. KERR, *On minimizing the number of multiplications necessary for matrix multiplication*, SIAM J. Appl. Math., 20 (1971), pp. 30–36.
[15] J. HOPCROFT AND R. TARJAN, *Planarity testing in V log V steps: Extended abstract*, Proc. IFIP Congress, Booklet TA-2, 1971, pp. 18–22.
[16] P. JARRATT, *Some efficient fourth order multipoint methods for solving equations*, BIT, 9 (1969), pp. 119–124.
[17] W. KAHAN, Private communication, 1969.
[18] R. F. KING, *A family of fourth-order methods for nonlinear equations*, to appear.
[19] D. KNUTH, *The Art of Computer Programming*, vol. 2, Seminumerical Algorithms, Addison-Wesley, Reading, Mass., 1969.
[20] ———, *Mathematical analysis of algorithms*, Proc. IFIP Congress Booklet, I 1971, pp. 135–143.
[21] H. T. KUNG, *A bound on the multiplication efficiency of iteration*, Computer Science Dept. Rep., Carnegie-Mellon University, Pittsburgh, Pa., 1972.
[22] W. Miller, *Toward abstract numerical analysis*, Doctoral thesis, University of Washington, Seattle, 1969.
[23] ———, *Unsolvable problems with differentiability hypotheses*, Proc. Fourth Annual Princeton Conference on Information Sciences and Systems, Princeton, N.J., 1970, pp. 480–482.
[24] M. MINSKY, *Form and computer science*, J. Assoc. Comput. Mach., 17 (1970), pp. 197–215.
[25] J. M. ORTEGA AND W. C. RHEINBOLDT, *Iterative Solution of Nonlinear Equations in Several Variables*, Academic Press, New York, 1970.

[26] A. M. Ostrowski, *Solution of Equations and Systems of Equations*, 2nd ed., Academic Press, New York, 1966.

[27] M. S. Paterson, *Optimality for square root algorithms*, Private communication, 1971.

[28] M. O. Rabin, *Speed of computation of functions and classification of recursive sets*, Proc. Third Convention of Scientific Societies, Israel, 1959, pp. 1–2.

[29] ———, *Degrees of difficulty of computing a function and a partial ordering of recursive sets*, Tech. Rep. 2, Hebrew University, Jerusalem, 1960.

[30] J. Rissanen, *On optimum root-finding algorithms*, IBM Rep. RJ726, San Jose, Calif., 1970.

[31] A. Schönhage and V. Strassen, *Schnelle Multiplication Grosser Zahlen*, 1970.

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

[33] J. F. Traub, *On functional iteration and the calculation of roots*, Proc. 16th National ACM Conference, 5A–1, Los Angeles, Calif., 1961, pp. 1–4.

[34] ———, *On functional iteration and the calculation of roots*, Unpublished Rep., 1961, 142 pp.

[35] ———, *Optimal m-invariant iteration functions*, Notices Amer. Math. Soc., 9 (1962), p. 122.

[36] ———, *On the informational efficiency of iteration functions*, International Congress of Mathematicians, Stockholm, 1962.

[37] ———, *The theory of multipoint iteration functions*, Proc. 17th National ACM Conference, Syracuse, N.Y., 1962, pp. 80–81.

[38] ———, *Interpolatively generated iteration functions*, Proc. 18th National ACM Conference, 1963.

[39] ———, *Iterative Methods for the Solution of Equations*, Prentice-Hall, Englewood Cliffs, N.J., 1964.

[40] ———, *Optimal iterative processes: Theorems and conjectures*, Proc. IFIP Congress, Booklet TA–1, 1971, pp. 29–32.

[41] S. Winograd, *The number of multiplications involved in computing certain functions*, Proc. IFIP Congress, Booklet A, 1968, pp. A128–A130.

[42] S. Winograd and P. Wolfe, Private communication, 1969.

[43] ———, *Optimal iterative processes*, IBM Rep. RC 3511, Yorktown Heights, N.Y., 1971.

SIAM J. COMPUT.
Vol. 1, No. 2, June 1972

ALGORITHMS FOR MINIMUM COLORING, MAXIMUM CLIQUE, MINIMUM COVERING BY CLIQUES, AND MAXIMUM INDEPENDENT SET OF A CHORDAL GRAPH*

FĂNICĂ GAVRIL†

Abstract. A finite undirected graph is called chordal if every simple circuit has a chord. Given a chordal graph, we present ways for constructing efficient algorithms for finding a minimum coloring, a minimum covering by cliques, a maximum clique, and a maximum independent set. The proofs are based on a theorem of D. Rose [3] that a finite graph is chordal if and only if it has some special orientation called an R-orientation. In the last part of this paper we prove that an infinite graph is chordal if and only if it has an R-orientation.

Key words. Chordal graph, maximum clique, maximum independent set, minimum coloring, R-orientation.

1. Introduction. Let G be a finite undirected graph with no parallel edges and no self-loops. A set of vertices in the graph is called independent if no two elements of it are adjacent. A maximum independent set is one with the largest number of vertices of all independent sets. A clique is a maximal completely connected set of vertices; a maximum clique is one with a maximum size. If two vertices u, v are connected, we denote this by u—v, and if not, by $u \not- v$.

A graph is called chordal if every simple circuit $v_1—v_2—v_3— \cdots —v_l—v_1$, with $l > 3$, has a chord; that is, there exists an edge, not of the circuit, which connects two of the circuit's vertices. Some call the chordal graphs "triangulated." However, the term has a somewhat similar, but different meaning in the theory of planar graphs. If this terminology were accepted, it would allow valid statements like: "Some planar triangulated graphs are not triangulated."

Chordal graphs arise in many contexts. Consider a finite family of intervals on a linearly ordered set, and draw a graph in the following way: the vertices represent the intervals, and two vertices are joined by an edge if the two corresponding intervals intersect. Such a graph is called an interval graph. Hajós [6] first put the problem of characterizing an interval graph. Gilmore and Hoffman [7] gave a complete characterization of interval graphs by showing that the interval graphs are a special class of chordal graphs. A first problem is to find a maximum size set of intervals, such that no two intervals intersect. We find this by taking a maximum independent set of vertices in the corresponding interval graph. Another problem is to color the intervals with a minimum number of colors such that no two intersecting intervals have the same color. We do this by taking a minimum coloring of the interval graph.

Other families of chordal graphs are the cactus graphs, which are the connected graphs that do not possess any cycle of length greater than three, and the family of their adjoint graphs. For more detailed information on the properties and applications of the chordal graphs, see also [1], [2] and [9].

We must remark that chordality and transitive orientability of graphs (dealt with in [4]) are two independent properties. A quadrilateral without diagonals is

* Received by the editors November 1, 1971, and in revised form March 15, 1972.

† Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel.

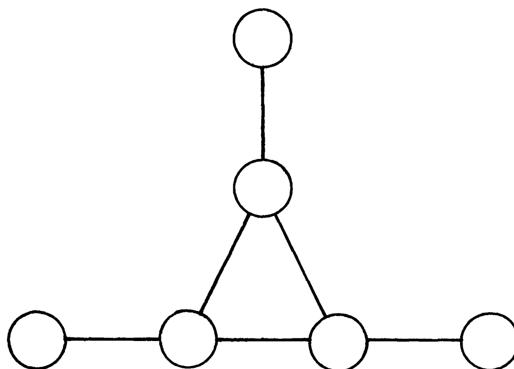


FIG. 1

transitively orientable and it is not chordal; on the other hand, the graph in Fig. 1 is chordal but it is not transitively orientable.

It is known that finite chordal graphs are perfect (see [1] and [2]). However, we do not know of any previous study of efficient algorithms for finding a minimum coloring, a maximum clique, a minimum covering by cliques, or a maximum independent set of such graphs. Our first aim is to present an approach for construction of algorithms for these four tasks. The algorithms used in the paper are not necessarily most efficient but are simply being used to provide upper bounds on the difficulty of algorithms resulting from the ideas in the paper.

The second aim is to generalize a theorem of Rose, by proving that an infinite graph is chordal if and only if it has an R-orientation. An orientation of the graph's edges is called an *R-orientation* if the following two conditions are satisfied:

(i) The resulting directed graph has no directed circuits.

(ii) If $b \to a$ and $c \to a$, then b—c; that is, either $b \to c$ or $c \to b$.

Rose calls it "monotone transitive orientation." A vertex of an oriented graph is called a "sink" if all the edges incident to the vertex enter it. It is clear that a finite oriented graph without directed circuits has a sink.

Dirac [8], and later Rose [3], proved that every chordal graph has a vertex such that the set of vertices adjacent to it form a completely connected set. Based on this theorem, Rose showed that a finite graph is chordal if and only if it has an R-orientation.

Consider any sink s of an R-oriented finite graph. Since all the edges which are incident to s enter s, all the adjacent vertices are connected to each other by edges. If we remove s and all the incident edges, the resulting graph is still R-oriented, and if its underlying undirected graph is chordal, the one for the original graph is chordal too. Thus, by induction on the number of vertices, R-orientability implies chordality. Conversely, if the graph is chordal, then it has a vertex v such that the set of its adjacent vertices form a completely connected set. By induction, the graph obtainined by eliminating v and all its incident edges has an R-orientation. Now return the vertex v to the graph and orient its incident edges towards it; we thus obtain an R-orientation of the given graph.

Based on this proof, Fulkerson [9] and Rose [3] showed a simple way of constructing an R-orientation, if one exists, and thus provides an easy test for

chordality. It runs as follows: Search for a vertex v such that all the vertices adjacent to it are connected to each other (or form a completely connected set). Eliminate v and all the edges incident to it. Repeat the same step on the remaining graph until only one vertex remains. If along the way, before the number of vertices is reduced to one, no such eliminatable vertex is found, the graph is not chordal. If the process may be completed, then return the vertices one by one in the reverse order, rebuilding the graph, and orient all edges incident to the reconstructed vertex towards it. It is clear that the resulting orientation is an R-orientation. The process is demonstrated on the following example.

Consider the graph described in Fig. 2. First, a can be eliminated, since all its neighbors are connected to each other. Next, c can be eliminated, then, in order, d, f, b, e, g. This is not the only order, and there is always more than one possible order, if any order exists. Upon returning in order g, e, b, f, d, c, a, and directing the reconstructed edges towards the new vertex, we get the graph of Fig. 3.

Let d be the maximum degree and n the number of G vertices. In this algorithm we test on $n(n+1)/2$ vertices if the adjacent vertices of any vertex form a completely connected set. Hence the bound number of steps to finish the algorithm is

$$\frac{n(n+1)}{2}\frac{d(d-1)}{2} = \frac{nd(n+1)(d-1)}{4}.$$

The vertices of every directed graph with no directed circuits can be numbered $1, 2, \cdots, n$, where n is the number of vertices, in such a way that all edges will be directed from low to high. This can be done by successive elimination of sinks, calling the first n, the second $n-1$, etc. Clearly, this can be done simultaneously with the R-orientation. Our example, after renaming the vertices, is shown in Fig. 4.

We shall assume, henceforth, that the given finite chordal graph has been R-oriented, that its vertices are $1, 2, \cdots, n$, and that all its edges are directed from

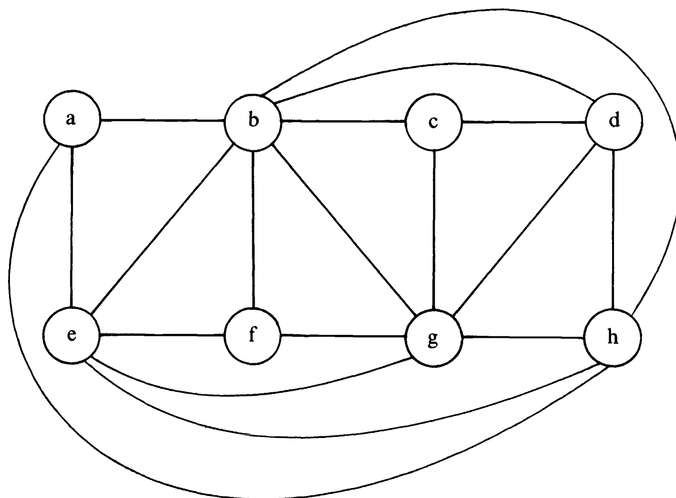


FIG. 2

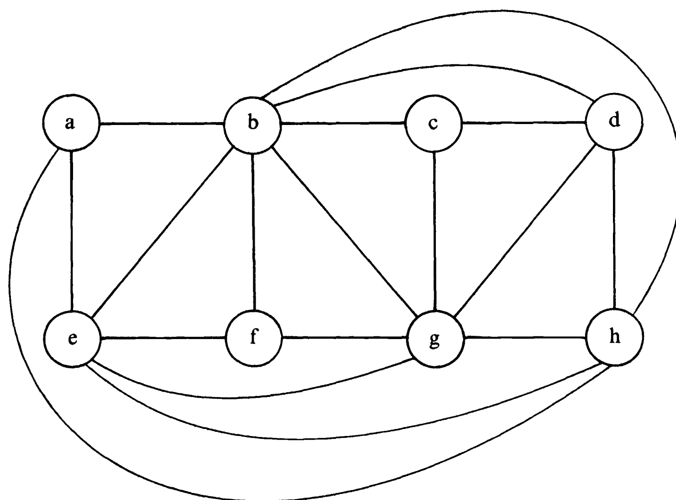


FIG. 3

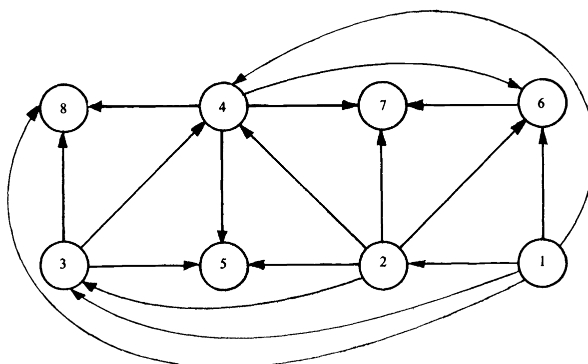


FIG. 4

low to high. We shall make use of these properties to answer questions about the underlying undirected graph.

2. Algorithms for minimum coloration and maximum clique. Let i be a vertex of the graph. Denote by J_i the set of all vertices j such that $j \to i$. Since G is chordal it is clear that $J_i \cup \{i\}$ is a completely connected set of vertices. Furthermore, if S is a completely connected set of vertices, then $S = J_i \cup \{i\}$ for some i; for let i be the greatest vertex in S, it follows that for every $j \in J_i$, $j \to i$.

The remark made above implies that every clique of G is of the form $J_i \cup \{i\}$, where i is the greatest vertex of the clique. *Thus, the number of cliques a finite chordal graph can have is at most n.* By taking only the maximals among the completely connected sets $J_i \cup \{i\}$, we obtain the set of all the cliques of G. Also, a maximum clique is easily found by finding a vertex i with the largest in-degree $d_I(i)$ ($d_I(i)$ is the number of edges which enter i). For every vertex we count the number of entering edges; hence the number of steps needed to find a maximum clique is at most

$n \cdot \max_{i \in V} d_I(i)$. In our example (Fig. 4), $\{1, 2, 3, 4\}$, $\{2, 3, 4, 5\}$, $\{1, 2, 4, 6\}$, $\{2, 4, 6, 7\}$, $\{1, 3, 4, 8\}$ are all the cliques. The vertices 4, 5, 6, 7 and 8 all have in-degree equal to three, and there is no vertex with a higher in-degree. Thus, all the cliques of our example are maximum.

This algorithm for finding a maximum clique is easily extended to a weighted graph; namely, one in which each vertex i has a weight ω_i. The maximum weight clique is found by computing for each vertex i the value w_i defined by

$$W_i = \sum_{j \in (J_i \cup \{i\})} \omega_j .$$

A vertex i, for which W_i is maximum, indicates a maximum weight clique, $J_i \cup \{i\}$.

It is interesting that the algorithm for minimum coloration of transitive graphs given in [4] is valid for R-oriented graphs too, but the proof of its validity is different. The algorithm runs as follows:

In the kth stage we generate a minimum coloration of the vertex subgraph G^k. (This is sometimes called the section subgraph; its set of vertices is $\{1, 2, \cdots, k\}$, and its edges are those edges of G which connect vertices i and j of this set.) Let this minimum coloration be $D_k = (A_1^k, A_2^k, \cdots, A_{m_k}^k)$, where $A_i^k \cap A_j^k = \varnothing$ if $i \neq j$ and $\bigcup_{i=1}^{m_k} A_i^k = \{1, 2, \cdots, k\}$. Clearly, $D_1 = (\{1\})$. We add vertex $k + 1$ to this coloration by the following rule: Find the first A_i^k to which $k + 1$ can be added without destroying its independence. If one is found, add $k + 1$ to it to form D_{k+1}; if none is found, add a new set, $A_{m_k+1}^{k+1} = \{k + 1\}$ to form D_{k+1} (clearly $m_{k+1} = m_k + 1$). D_n is a minimum coloration of G. (This will be proved shortly.) Consider our example of Fig. 4. We get:

$$\begin{aligned}
D_1 &= (\{1\}), \\
D_2 &= (\{1\}, \{2\}), \\
D_3 &= (\{1\}, \{2\}, \{3\}), \\
D_4 &= (\{1\}, \{2\}, \{3\}, \{4\}), \\
D_5 &= (\{1, 5\}, \{2\}, \{3\}, \{4\}), \\
D_6 &= (\{1, 5\}, \{2\}, \{3, 6\}, \{4\}), \\
D_7 &= (\{1, 5, 7\}, \{2\}, \{3, 6\}, \{4\}), \\
D_8 &= (\{1, 5, 7), \{2, 8\}, \{3, 6\}, \{4\}).
\end{aligned}$$

To establish the validity of the algorithm, consider the first vertex i to enter $A_{m_n}^n$. It was put there because it was connected to (lower vertices) $v_j \in A_j^i$ for $j = 1, 2, \cdots, m_{i-1}$, where $m_{i-1} + 1 = m_n$. Since all these edges are directed $v_j \to i$, the set $\{v_1, v_2, \cdots, v_{m_{i-1}}, i\}$ is a clique whose size is m_n. Therefore, any coloration must have at least m_n sets and hence D_n is a minimum coloration.

In the $k + 1$ stage we search for every A_i^k to see whether it has a vertex connected to the vertex $k + 1$; but the number of all the vertices in $A_1^k, \cdots, A_{m_k}^k$ is exactly k, and therefore we need k steps to do it. Hence for a minimum coloration we have at most $n(n - 1)/2$ steps. By the algorithm the maximum clique size is equal to the minimum coloration size.

3. Algorithms for minimum covering by cliques and maximum independent set. Let us define inductively a sequence of vertices $n_1, n_2, \cdots, n_t$ in the following way: $n_1 = n$; n_k is the highest vertex smaller than n_{k-1} and which is not in $J_{n_1} \cup J_{n_2} \cup \cdots \cup J_{n_{k-1}}$; all vertices smaller than n_t are in $J_{n_1} \cup \cdots \cup J_{n_t}$. Hence $\{n_1, n_2, \cdots, n_t\} \cup J_{n_1} \cup J_{n_2} \cup \cdots \cup J_{n_t}$ is the set of all vertices of G.

The set $\{n_1, n_2, \cdots, n_t\}$ is clearly an independent set, and hence a minimum covering by cliques must have at least t sets. On the other hand, for every i, $1 \leqq i \leqq t$, $S_{n_i} = J_{n_i} \cup \{n_i\}$ is a clique and also $(S_{n_1}, \cdots, S_{n_t})$ is a covering by cliques of G. Therefore, this is a minimum covering by cliques and $\{n_1, \cdots, n_t\}$ is a maximum independent set of G. By the algorithm, the size of the maximum independent set is equal to the size of the minimum covering by cliques. In the example of Fig. 3, we have: $n_1 = 8$; $n_2 = 7$; $n_3 = 5$. Therefore, $\{8, 7, 6\}$ is a maximum independent set, and $(\{1, 3, 4, 8\}, \{2, 4, 6, 7\}, \{2, 3, 4, 5\})$ is a minimum covering by cliques. For finding the highest vertex smaller than n_{k-1} which is not in $J_{n_1} \cup \cdots \cup J_{n_{k-1}}$ we need $(n - k - 1) \cdot (k - 1)$ steps. Hence, the maximum number of steps for finding a minimum covering by cliques and a maximum independent set is

$$\sum_{k=1}^{t} (n - k - 1)(k - 1) = \frac{t(t - 1)(3n - 2t + 1)}{6}.$$

4. Infinite chordal graphs. In a finite graph, the independence number is defined as the size of the maximum independent set. In the case of an infinite graph G, we must define the independence number $\alpha(G)$ as the superior limit of all independent set cardinalities. This is because an independent set whose cardinality is equal to this superior limit may not exist; for example, see the graph in Fig. 5. In this graph, the vertices are v_i^j, where i, j go to infinity. For every i, $\{v_i^j\}_{j=1}^{i}$ is an independent set, and if $i \neq k$, then v_i^j is connected to v_k^r by an edge. In this graph, $\alpha(G) = \aleph_0$, but there is no independent set with cardinality $\aleph_0$. Also, we define $\beta(G)$ as the superior limit of cardinalities of all the cliques.

For any set M, denote its cardinality by $|M|$. Let V_G be the set of vertices of G. For a vertex $v \in V_G$, denote the set of all vertices connected to v by Γ_v and hence $d(v) = |\Gamma_v|$ is the degree of v. Also denote by $\bar{d}(v)$ the number of vertices to which v is not connected.

Suppose G is an infinite undirected graph (not necessarily chordal) with $|V_G| > \sup_{v \in V_G} d(v)$. Let A be a maximal independent set, not necessarily of maximum size. Hence $|A| \leqq \alpha(G)$. Because of the maximality of A, every vertex of G is in A or is connected to a vertex in A. Hence $V_G = (\bigcup_{v \in A} \Gamma_v) \cup A$ and therefore

$$|V_G| \leqq |A|(\sup_{v \in A} |\Gamma_v| + 1) \leqq \alpha(G) \sup_{v \in V_G} (d(v) + 1).$$

But $\sup_{v \in V_G} d(v) < |V_G|$, and $|V_G|$ is an infinite cardinality; hence $\alpha(G) = |V_G|$. On the other hand, a minimal covering by cliques of G must have at least $\alpha(G)$ cliques, and at most $|V_G|$. Hence the cardinality of the minimum covering by cliques of G is $|V_G|$.

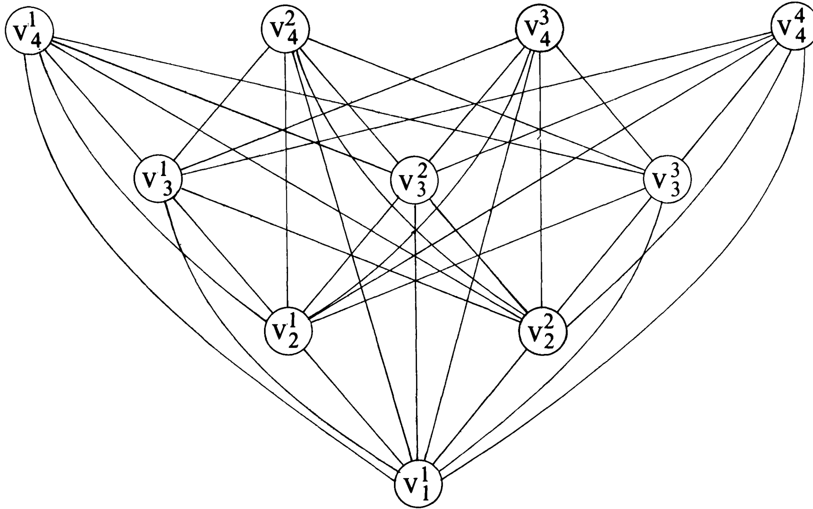


FIG. 5

Therefore, if G is an infinite graph such that $|V_G| > \sup_{v \in V_G} d(v)$, then $|V_G|$ is the cardinality of the minimum covering by cliques and also is the independence number of G.

By a similar proof we obtain: if G is an infinite graph such that $|V_G| > \sup_{v \in V_G} \bar{d}(v)$, then $|V_G|$ is the cardinality of the minimum coloring and is equal to $\beta(G)$.

The above remarks give us, in a special case, the critical numbers of an infinite graph.

Let us now return to the chordal graphs. For infinite graphs the first condition for an R-orientation is that the graphs have no finite directed circuits. Rose proved in [3] that a finite graph is chordal if and only if it has an R-orientation. We want to generalize this theorem by proving that an infinite graph is chordal if and only if it has an R-orientation.

If G is an infinite R-oriented graph, then every finite vertex subgraph is R-oriented and hence every circuit $v_1—v_2— \cdots —v_l—v_1$, $l > 3$, has a chord. Therefore if G is an infinite R-oriented graph, then G is chordal.

Now we must prove that if G is an infinite chordal graph, then it has an R-orientation. The proof is by a method of mathematical logic given by Beth which appears in [5]. The method presumes the existence of the axiom of choice; the symbols and terms are those of [5].

We construct the following generalized first order theory T with equality. There are three binary predicate letters: the equality "$=$", A_1 (corresponding to the relation of the graph G) and A_2 (corresponding to the R-orientation). Also, for every vertex v in G, there is an individual constant a_v. The axioms of T, in addition to the equality axioms, are:

1. $\sim A_1(x, x)$ (x is not connected to itself in the graph).
2. $A_1(x, y) \supset A_1(y, x)$ (symmetry of the graph).
3. For any two distinct vertices v, u of G, $a_v \neq a_u$.

4. If v is connected to u in G, then $A_1(a_v, a_u)$, and if v is not connected to u in G, then $\sim A_1(a_v, a_u)$.
5. $A_2(x, y) \supset A_1(x, y)$ (if there is an oriented edge from x to y, then x and y are connected in the underlying undirected graph).
6. $A_1(x, y) \supset (A_2(x, y) \vee A_2(y, x))$ (every edge is oriented).
7. $A_2(x, y) \supset \sim A_2(y, x)$ (every edge is directed in only one direction).
8. $A_2(y, x) \wedge A_2(z, x) \supset A_1(y, z)$ (the condition that A_2 represents the R-orientation).
9. For every finite number $n \geqq 3$,

$$\sim \exists x_1 x_2 \cdots x_n (A_2(x_1, x_2) \wedge A_2(x_2, x_3) \wedge \cdots \wedge A_2(x_n, x_1))$$

(the orientation represented by A_2 has no directed circuits).

Any finite set of these axioms involves only a finite number of individual constants $a_{v_1}, \cdots, a_{v_r}$. The corresponding subgraph of G on the vertices $\{v_1, \cdots, v_r\}$ is a finite chordal graph and hence it has an R-orientation. Hence, this finite graph is a model of the given finite set of axioms. Therefore any finite set of axioms of T is consistent and therefore, by the compactness theorem, T is consistent. It follows that T has a normal model, that is, a model in which "$=$" is the identity relation. This model is an R-oriented graph because of axioms 8, 9, and by axioms 3, 4 it has G as a vertex subgraph. Therefore G is R-oriented. *Thus a graph G (finite or infinite) is chordal if and only if it has an R-orientation.*

Consider a (finite or infinite) chordal graph G such that the degree of every vertex of G is finite. G has an R-orientation. Because the degree of every vertex is finite, it follows that all cliques of G are of the form $J_v \cup \{v\}$, where J_v is the set of all vertices u such that $u \to v$.

Therefore, if G is a (finite or infinite) chordal graph and the degree of every vertex is finite, then the cardinality of the set of all G cliques is less than or equal to $|V_G|$.

Acknowledgment. I wish to express my thanks to Professor S. Even whose kind help made this paper possible.

REFERENCES

[1] A. HAJNAL AND T. SURANYI, *Über die Auflösung von Graphen in Voltständige Teilgraphen*, Ann. Univ. Sci. Budapest. Eötvös Sect. Math., 1 (1958), pp. 113–121.
[2] C. BERGE, *Some Classes of Perfect Graphs*, *Graph Theory and Theoretical Physics*, F. Harary, ed., Academic Press, New York, 1967.
[3] D. J. ROSE, *Triangulated graphs and the elimination process*, J. Math. Anal. Appl., 32 (1970), pp. 597–609.
[4] S. EVEN, A. LEMPEL AND A. PNUELI, *Permutation graphs and transitive graphs*, Proc. Fourth Annual Princeton Conference on Information Sciences and Systems, Princeton Univ., Princeton, N.J., 1970, pp. 463–467; J. ACM, to appear.
[5] E. MENDELSON, *Introduction to Mathematical Logic*, Van Nostrand, Princeton, N.J., 1964.
[6] G. HAJÓS, *Über eine Art von Graphen*, Internat. Math. Nachr., 11 (1957), p. 65.
[7] P. C. GILMORE AND A. J. HOFFMAN, *A characterization of comparability graph and of interval graphs*, Canad. J. Math., 16 (1964), pp. 539–548.
[8] G. A. DIRAC, *On rigid circuit graphs*, Abh. Math. Sem. Univ. Hamburg, 25 (1961), pp. 71–76.
[9] D. R. FULKERSON AND O. A. GROSS, *Incidence matrices and interval graphs*, Pacific J. Math., 15 (1965), pp. 835–855.

SIAM J. COMPUT.
Vol. 1, No. 2, June 1972

FLOW GRAPH REDUCIBILITY*

MATTHEW S. HECHT AND JEFFREY D. ULLMAN†

Abstract. The structure of programs can often be described by a technique called "interval analysis" on their flow graphs. Here, we characterize the set of flow graphs that can be analyzed in this way in terms of two very simple transformations on graphs. We then give a necessary and sufficient condition for analyzability and apply it to "goto-less programs," showing that they all meet the criterion.

Key words. Code optimization, flow graph, interval analysis, reducibility, goto-less program, Church–Rosser system.

1. Introduction. The application of many code improvement techniques depends on globally modeling a program by a directed graph called a "flow graph." This model provides a comprehensive view of the control flow of a program. Examples of improvement possible by flow graph analysis are the detection and removal of useless and redundant statements and the moving of loop independent computations outside loops. Much of the analysis for this type of improvement hinges on the property of a flow graph called "reducibility," e.g., [1]–[5].

In this paper the "interval" analysis technique of Cocke and Allen [1], [6] is reviewed and reducibility is defined. Next, we present a new technique for treating flow graph reducibility, called "collapsibility," and show it equivalent to reducibility. Finally, we give a structural characterization of nonreducible flow graphs and use this characterization to obtain an interesting result about flow graphs for "goto-less programs."

2. Necessary concepts from graph theory. In this section we present the concepts from graph theory which are used in this paper.

DEFINITION. A *directed graph* G is a pair (N, E), where N is a set and E is a relation on N. The elements of N are called *nodes*, and the ordered pairs in E are called *edges*.

Let $G = (N, E)$ be a graph. A graph $G' = (N', E')$ is said to be a *subgraph* of G if $N' \subseteq N$ and $E' \subseteq E \cap (N' \times N')$.

Edge (n, m) is said to *leave* node n and *enter* node m. We say n is a *predecessor* of m, and m is a *successor* of n. The *in-degree* of a node is the number of edges entering and the *out-degree* is the number of edges leaving.

A sequence of nodes $(n_0, n_1, \cdots, n_k)$, $k \geqq 0$, is a *path of length k* from node n_0 to node n_k if there is an edge which leaves node n_{i-1} and enters node n_i for $1 \leqq i \leqq k$. A *cycle* is a path $(n_0, n_1, \cdots, n_k)$ in which $n_0 = n_k$. If $k = 1$, then the cycle is a *loop*. If (n, n) is an edge, we say node n *has a loop*.

A graph is *rooted* if there exists at least one node r such that there is a path to all nodes from r. The node r is called a *root* of the graph.

* Received by the editors August 26, 1971, and in revised form March 6, 1972.

† Department of Electrical Engineering, Princeton University, Princeton, New Jersey 08540.
This work was supported by the National Science Foundation under Grant GJ-1052.

It is often useful to attach certain information to the nodes of a graph. Let (N, E) be a graph. A *node labeling* of the graph is a function from N to a set A of node labels.

A *tree* T is a graph $G = (N, E)$ with a specified node r in N such that:

(a) node r has in-degree zero;

(b) node r is a root of T;

(c) all other nodes of T have in-degree one.

An *ordered tree* is a tree with a linear order on the successors of each node.

We follow the convention of drawing trees with the root on top and having all edges directed downward. The successors of a node of an ordered tree are always linearly ordered from left to right in a diagram.

A *spanning tree* of graph G is a subgraph of G which is a tree and contains all nodes in the graph.

A *flow graph* is a 3-tuple $F = (N, E, i)$, where (N, E) is a finite graph and i is a root of (N, E), called the *initial node*.

Example 1. Figure 1(a) shows a flow graph with node 1 as the initial node. Figure 1(b) cannot be a flow graph, since it has no root.

3. Reducibility. A flow graph may be analyzed by constructs called "intervals."

DEFINITION. Let G be a flow graph and n a node of G. The *interval with header* n, denoted $I(n)$, is constructed by the following algorithm.

ALGORITHM A (Cocke and Allen) *Interval construction.*

Input: Flow graph G and designated node n.

Output: $I(n)$.

Method:

A1. Place n in $I(n)$.

A2. If n' is a node not yet in $I(n)$, n' is not the initial node, and all edges entering n' leave nodes in $I(n)$, add n' to $I(n)$.

A3. Repeat step A2 until no more nodes can be added to $I(n)$. □

It should be observed that although n' in step A2 may not be well-determined, $I(n)$ does not depend on the order in which candidates for n' are chosen. A candidate at one iteration of A2 will, if it is not chosen, still be a candidate at the next iteration.

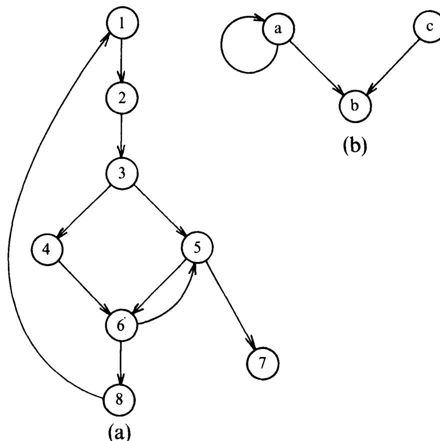


FIG. 1. *Examples of graphs*

The next algorithm partitions a flow graph uniquely into disjoint intervals.

ALGORITHM B (Cocke and Allen) *Partition of a flow graph into intervals.*

Input: A flow graph $G = (N, E, i)$.

Output: A set of disjoint intervals $I_1, \cdots, I_k$, whose union is N.

Method:

B1. Establish a list H of header nodes and a list L of intervals. Initially, H consists only of i, and L is empty.

B2. If H is empty, halt; L is the desired list of intervals.

B3. Otherwise, choose n on H, and compute $I(n)$ by Algorithm A.

B4. Add $I(n)$ to L. Delete n from H, but add to H any node which has a predecessor in $I(n)$, but which is not already in H or in one of the intervals on L. Return to B2.

Example 2. Let us consider the flow graph of Fig. 1(a). We begin with node 1, the initial node, on list H. Algorithm A tells us to add node 2 to $I(1)$, then to add nodes 3 and 4. No further nodes can be added to $I(1)$. For example, node 5 has an edge entering from 6, which is not currently in $I(1)$, and 6 has an edge entering from 5.

We therefore place $I(1) = \{1, 2, 3, 4\}$ on L, and add 5 and 6 to H. Then, we compute $I(5) = \{5, 7\}$ and $I(6) = \{6, 8\}$. Note that 1 is not added to $I(6)$, because it is the initial node.

Two important properties of intervals [1], [3], [4] are:

(i) every cycle within the interval includes the interval header; and

(ii) every edge entering a node of the interval from the outside enters the header.

Evidently, the intervals of one flow graph can be considered the nodes of another flow graph in which there is an edge between intervals I_1 and I_2 if and only if $I_1 \neq I_2$, and there is an edge from a node in I_1 to the header of I_2. Furthermore, this process may be performed repeatedly.

DEFINITION. Let G be a flow graph. Then $I(G)$, the *derived graph* of G, is defined as follows:

(a) The nodes of $I(G)$ are the intervals of G.

(b) There is an edge from the node representing interval I_1 to that representing I_2 if there is any edge from a node in I_1 to the header of I_2, and $I_1 \neq I_2$.

(c) The initial node of $I(G)$ is the interval containing the initial node of G.

Flow graph G is called *irreducible* if and only if $I(G) = G$. The sequence $G = G_0, G_1, G_2, \cdots, G_n$ is called the *derived sequence* for G if $G_{i+1} = I(G_i)$, and G_n is irreducible. G_n is called the *limit* flow graph of G and is denoted by $\hat{I}(G)$.

Flow graph G is called *reducible* if and only if $\hat{I}(G)$ is a single node with no loop. Otherwise, it is called *nonreducible.*

Example 3. Let G_0 be the graph of Fig. 1(a). Then $G_1 = I(G_0)$ has three nodes, corresponding to the three intervals $\{1, 2, 3, 4\}$, $\{5, 7\}$ and $\{6, 8\}$ of G_0. Let these nodes be n_1, n_2 and n_3, respectively. Then G_1 is shown in Fig. 2. There is an edge from n_1 to n_2, for example, because of the edge in G_0 from node 3 to node 5.

4. Collapsibility. We shall define a pair of simple transformations that together have the same effect on flow graphs as the interval construction does. Moreover, it will be apparent that the data flow analysis suggested in [1], [3], [4], [6], using the interval construction, could be equally well done if construction of

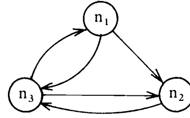


FIG. 2. *Derived flow graph*

the derived sequence of a graph G were replaced by repeated application of our transformations.

There are various advantages to the approach taken here, compared with the interval analysis approach. For example, [7] gives an $O(n \log n)$ algorithm to determine whether a flow graph is reducible. In comparison, the straightforward technique of constructing the derived sequence can take $O(n^2)$ steps if performed in the obvious way. Consider, for example, the flow graph of n nodes of Fig. 3. Also, [8] gives an algorithm taking $O(n \log n)$ bit vector operations to find common subexpressions in a reducible graph. In comparison, the techniques of [1], [4] can require $O(n^2)$ bit vector operations. (Fig. 3 again suffices.)

Moreover, these transformations seem to characterize the set of reducible flow graphs in a nice way, and they lead to a further characterization of reducibility that makes it clear in many cases that the control flow structure of a given programming language will yield only reducible flow graphs. For example, the D-charts developed from "goto-less programs" [16] are all reducible. We now give the definitions of the two transformations.

DEFINITION. Let G be a flow graph. Suppose n is a node in G with a loop, that is, an edge from n to itself. Transformation T_1 on node n is removal of this loop.

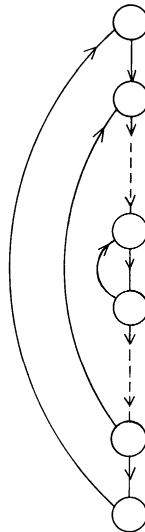


FIG. 3. *Flow graph requiring $O(n^2)$ steps for interval analysis*

Let n_1 and n_2 be nodes in G such that n_1 is the unique predecessor of n_2, and n_2 is not the initial node. The transformation T_2 on node pair (n_1, n_2) is merging nodes n_1 and n_2 to one node, which we may call n_1/n_2, and deleting the unique edge between them. Let $n \neq n_1$ and $n \neq n_2$. There is an edge from node n to n_1/n_2 if there was previously an edge from n to n_1 (there cannot be one from n to n_2), and there is an edge from n_1/n_2 to n if there was previously one to n from either n_1 or n_2 or both. n_1/n_2 has a loop if there was an edge from n_2 to n_1.

Example 4. Figure 4 shows a flow graph which is transformed into a single node by one application of T_1 and two of T_2. Although T_2 is not applicable to the original graph, it becomes applicable after use of T_1.

Various authors have considered similar transformations, but from the point of view of generating graphs rather than analyzing (i.e., reducing) them. Cooper [9] considers three generating rules, one of which is the inverse of T_1 (i.e., addition of loops). The other two together are equivalent to the inverse of T_2. It is shown in [9] that together with a construction which is the inverse of "node splitting" [10], these generating rules are capable of building an arbitrary flow graph.

Engeler [11], [12] considers "normal form flow charts," which are built by two generating rules, one of the inverse of T_1 and the other equivalent to the inverse of T_2, restricted so that the two nodes involved have disjoint sets of successors. Thus, the normal form flow charts are a subset of the reducible graphs. They are characterized as trees with additional back edges.

We now proceed to develop useful properties of the transformations T_1 and T_2.

DEFINITION. A flow graph is called *collapsible* if and only if it can be transformed into a single node with no loop by repeated application of T_1 and T_2. Otherwise, it is called *noncollapsible*.

Example 5. The flow graph of Fig. 2 is noncollapsible. There are no loops, and no node but the initial node has a unique entering edge, so neither T_1 nor T_2 is applicable. On the other hand, the flow graph of Fig. 4 is collapsible.

T_1 and T_2 have a useful property; they form a "finite Church–Rosser" transformation [13].

DEFINITION. Let R be a relation on a set S. Let xRy denote $(x, y) \in R$. The *inverse* of R, R^{-1}, is $\{(y, x) | (x, y) \in R\}$. R is *symmetric* if $R = R^{-1}$. R is *reflexive* if $(x, x) \in R$ for all $x \in S$. R is *transitive* if xRy and yRz imply xRz for all x, y, z in S.

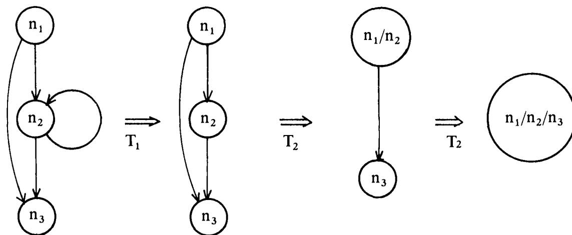


FIG. 4. *Applications of* T_1 *and* T_2

If R_1 and R_2 are relations on S, then the *composition* of R_1 and R_2, denoted R_1R_2, is $\{(x, z)|$ for some y in S, xR_1y and $yR_2z\}$. The *reflexive closure* of R, denoted $R^{\#}$, is $R \cup \{(x, x)|x \in S\}$. The *transitive closure* of R, denoted R^+, is $R^1 \cup R^2 \cup R^3 \cup \cdots$, where $R^1 = R$ and $R^i = RR^{i-1}$ for $i \geqq 2$. The *reflexive transitive closure* of R, denoted R^* is $R^{\#} \cup R^+$. The *completion* of R, denoted $\hat{R}$, is $\{(x, y)|xR^*y$ and there is no z such that $yRz\}$.

A pair $(S, \Rightarrow)$, where S is a set and $\Rightarrow$ is a relation on S, is said to be *finite* if for each p in S, there is a constant k_p such that if[1] $p \overset{i}{\Rightarrow} q$, then $i \leqq k_p$. That is, there is a bound on the number of times $\Rightarrow$ can be applied in succession, beginning with any element p. We say $(S, \Rightarrow)$ is *finite Church–Rosser* (FCR) if it is finite, and $\overset{\wedge}{\Rightarrow}$ is a function, i.e., $p \overset{\wedge}{\Rightarrow} q$ and $p \overset{\wedge}{\Rightarrow} r$ implies $q = r$. If set S is understood, $\Rightarrow$ is called an *FCR transformation.*

The following theorem gives a test for the FCR property which is simpler to apply than the definition. It is proved in [13].

THEOREM 1. *Let $\Rightarrow$ be a relation on set S. Then $(S, \Rightarrow)$ is FCR if and only if it is finite, and for all p in S, if $p \Rightarrow p_1$ and $p \Rightarrow p_2$, then there is some q such that $p_1 \overset{*}{\Rightarrow} q$ and $p_2 \overset{*}{\Rightarrow} q$.*

DEFINITION. Let S be the set of flow graphs. We define the relation $\underset{i}{\Rightarrow}$, $i = 1$ or 2, by $G \underset{i}{\Rightarrow} G'$ if and only if G can be transformed into G' by an application of T_i. Let $\Rightarrow$ denote the union of $\underset{1}{\Rightarrow}$ and $\underset{2}{\Rightarrow}$. The reflexive closure, k-fold product, transitive closure, reflexive transitive closure, and the completion of $\Rightarrow$ are respectively given by $\overset{\#}{\Rightarrow}$, $\overset{k}{\Rightarrow}$, $\overset{+}{\Rightarrow}$, $\overset{*}{\Rightarrow}$ and $\overset{\wedge}{\Rightarrow}$.

THEOREM 2. *$(S, \Rightarrow)$ is FCR.*

Proof. We use Theorem 1 and note that in this case, we shall always be able to find q such that $p_1 \overset{\#}{\Rightarrow} q$ and $p_2 \overset{\#}{\Rightarrow} q$.

Finiteness property. Let G be a flow graph with n nodes. Each application of T_1 or T_2 deletes at least one edge. Thus, $\Rightarrow$ is finite.

Commutativity property. Suppose $G \underset{i}{\Rightarrow} G_1$ and $G \underset{j}{\Rightarrow} G_2$, where $i, j \in \{1, 2\}$. There are three distinct cases to consider.

Case 1. $i = j = 1$. Suppose T_1 is applied to node n_1 to yield G_1 and to node n_2 to yield G_2. If $n_1 = n_2$, then $G_1 = G_2$. If $n_1 \neq n_2$, then T_1 may be performed on n_2 in G_1 and on n_1 in G_2 to yield equal graphs.

Thus, $G \Rightarrow G_1 \overset{\#}{\Rightarrow} H$ and $G \Rightarrow G_2 \overset{\#}{\Rightarrow} H$, where H is the graph resulting after applying T_1 to nodes n_1 and n_2 in G.

Case 2. $i = j = 2$. Suppose T_2 is applied to node pair (n_1, n_2) in G to yield G_1, and to node pair (n_3, n_4) in G to yield G_2. If $n_1 = n_3$ and $n_2 = n_4$, then $G_1 = G_2$. If all four nodes are distinct, then apply T_2 to (n_3, n_4) in G_1, and apply T_2 to (n_1, n_2) in G_2 to yield equal graphs. Now suppose neither of the previous subcases holds. If $n_1 = n_3$ and no other equalities hold, then Fig. 5 shows the subgraph of interest. Otherwise, if $n_2 = n_3$ and no other equalities hold, then Fig. 6 shows the subgraph of interest. Thus, $G \Rightarrow G_1 \overset{\#}{\Rightarrow} H$ and $G \Rightarrow G_2 \overset{\#}{\Rightarrow} H$, where H is the graph resulting after applying T_2 to (n_1, n_2) and to (n_3, n_4) in G.

[1] We place the symbols $\hat{}$, $\#$, $*$, $+$ and i above the relation symbol $\Rightarrow$ instead of at the upper right corner, as indicated for relation R in the above definition.

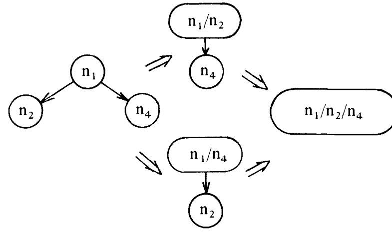


FIG. 5. *Applications of* T_2

The case in which $n_1 = n_4$ and no other equalities hold is symmetric to the case $n_2 = n_3$ above. The case $n_1 = n_4$ and $n_2 = n_3$ is impossible, because then the flow graph has two isolated nodes, and hence must consist of only n_1 and n_2. But one of these must be the initial node, and T_2 is thus either not applicable to (n_1, n_2) or not applicable to (n_3, n_4). Since we have assumed $n_1 \neq n_2$ and $n_3 \neq n_4$, and n_2 may not be n_4 unless $n_1 = n_3$, we have considered all possibilities.

Case 3. $i \neq j$. Suppose T_2 is applied to node pair (n_1, n_2) in G to yield G_1, and T_1 is applied to node n_3 in G to yield G_2. Clearly, $n_2 \neq n_3$. Consequently, T_1 and T_2 do not "interfere"; T_1 may be applied to node n_3 in G_1, and T_2 may be applied to node pair (n_1, n_2) in G_2 to yield equal graphs. Thus, $G \Rightarrow G_1 \Rightarrow H$ and $G \Rightarrow G_2 \Rightarrow H$, where H is the result of applying T_2 to (n_1, n_2) and T_1 to n_3.

5. Equivalence of reducibility and collapsibility. Theorems 3 and 4 establish that a flow graph is reducible if and only if it is collapsible.

DEFINITION. Let the first n nodes added to an interval $I(h)$ in Algorithm A be called a *partial interval*. We assume, of course, that the interval $I(h)$ has at least n nodes, and $n \geqq 1$.

LEMMA 1. *Let G be a flow graph. Then* $G \stackrel{*}{\Rightarrow} I(G)$.

Proof. It suffices to show that a partial interval is collapsible to its header, and that connections (edges) between a partial interval and the other nodes in the flow graph are maintained. Thus, constructing the derived graph $I(G)$ of flow graph G corresponds exactly to collapsing the intervals of G.

Inductive hypothesis. A partial interval of n nodes is collapsible to its header, and edges between the partial interval and the other nodes of the flow graph are preserved. That is, edges leaving the partial interval to another node outside the partial interval remain. The header will have no loops.

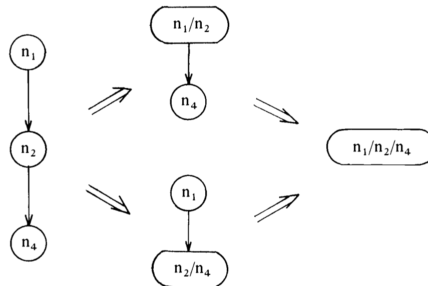


FIG. 6. *Applications of* T_2

Basis. The first node added to an interval is the header node. The only collapsing possible is removal of a loop if present. This possible application of T_1 will not destroy any edge to another node in the graph outside the partial interval.

Inductive step. Assume that the inductive hypothesis is true for a partial interval of n nodes, and consider the addition of another node m to the partial interval. This new node only has edges entering it from nodes in the partial interval. Since the first n nodes of the partial interval are collapsible by the induction hypothesis, there will be exactly one edge from the collapsed partial interval to m. Thus, T_2 is applicable. Edges from m to nodes outside the partial interval now leave the node representing the collapsed partial interval. If there is a loop introduced by the application of T_2, it can be removed by T_1.

As an immediate consequence of Lemma 1, we have the following.

THEOREM 3. *If a flow graph is reducible, then it is collapsible.*

Proof. If $\hat{I}(G) = 0$,[2] then $G \overset{*}{\Rightarrow} 0$ by Lemma 1, iterated.

The converse of Theorem 3 is easy to prove.

THEOREM 4. *If a flow graph is collapsible, then it is reducible.*

Proof. Suppose $G \hat{\Rightarrow} 0$, and let $\hat{I}(G) = G'$. By Lemma 1 iterated, $G \overset{*}{\Rightarrow} G'$. We must have $G' \hat{\Rightarrow} 0$. (For if $G' \hat{\Rightarrow} G''$, then $G \hat{\Rightarrow} G''$. Since $\hat{\Rightarrow}$ is a function, and $G \hat{\Rightarrow} 0$, we have $G'' = 0$.)

If $G' \neq 0$, then since $G' \hat{\Rightarrow} 0$, T_1 or T_2 is applicable to G'. We have assumed $I(G') = G'$, so every node appears on the header list when Algorithm B is applied to G'. If T_1 is applicable to node n, then $I(n)$ does not have a loop in $I(G')$, so $I(G') \neq G'$. If T_2 is applicable to node pair (n_1, n_2), then n_2 is in $I(n_1)$, so again, $I(G') \neq I(G)$. We conclude that $G' = 0$.

6. Characterization theorem for nonreducible flow graphs. We shall now show the existence of a certain subgraph in all and only the nonreducible flow graphs. Prior to showing this result, we present the concept of a "depth-first spanning tree" of a flow graph.

DEFINITION. A *depth-first spanning tree* (DFST) of a flow graph G is a spanning tree that is constructed by Algorithm C.

Algorithm C. *DFST of a flow graph.*

Input: Flow graph G.

Output: DFST of G.

Method:

C1. The root of the DFST is the initial node of G. Let this node be the node n "under consideration."

C2. Perform step C3 until it is no longer applicable. Then perform C4 and C5.

C3. If the node n under consideration has a successor x not already on the DFST, we select node x as the right-most successor of n found so far in the spanning tree. If this step is successful, node x becomes the node n under consideration.

C4. If the node under consideration is the root, then halt.

C5. Otherwise, back up the DFST one node toward the root and consider this node by going to step C2.

[2] Let 0 represent the graph with one node and no edges.

DEFINITION. We define the *spine* of a DFST T to be the sequence of nodes $(n_1, n_2, \cdots, n_k)$ such that n_1 is the root of T, n_{i+1} is the right-most successor of n_i, $1 \leqq i \leqq k-1$, and n_k has no successors.

We can add to the DFST T of a flow graph G the edges of G which are not edges of T. Conventionally, we show edges of T as solid lines and edges of G not in T by dashed lines. An important property of DFST's is the following.

LEMMA 2 [14]. *Let $G = (N, E, i)$ be a flow graph and $T = (N, E')$ one of its DFST's. If there is an edge (n_1, n_2) in $E - E'$, then either*:

(i) *there is a path from n_1 to n_2 in T;*

(ii) *there is a path from n_2 to n_1 in T;*

(iii) $n_1 = n_2$; *or*

(iv) *n_1 is to the right of n_2 in T.*[3]

Example 6. Let G be the flow graph of Fig. 7(a). If we consider nodes in the order 1, 2, 3, 4, then back to 3, then to 5, we obtain the DFST of Fig. 7(b). The spine is 1, 2, 3, 5.

DEFINITION. Let (*) denote any of the graphs represented in Fig. 8, where the wiggly lines denote node disjoint (except for the endpoints, of course) paths; a, b, c and i, the initial node, are distinct, except that a and i may be the same.

LEMMA 3. *The absence of subgraph* (*) *in a flow graph is preserved by T_1 and T_2.*

Proof. Let G be a flow graph and let n_1 and n_2 be any two nodes in G. We observe that if a path does not exist between n_1 and n_2, then neither T_1 nor T_2 will create such a path; neither will they make two paths be node disjoint if they were not so already.

THEOREM 5. *If a flow graph is nonreducible, then it has a subgraph of form* (*).

Proof. We prove the theorem by induction on n, the number of nodes of G.

Inductive hypothesis. Flow graph G with n nodes has a subgraph of form (*).

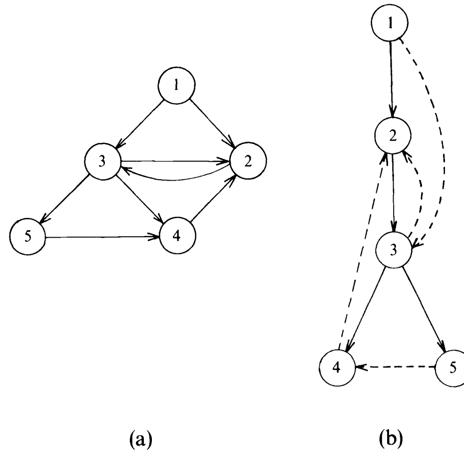


FIG. 7. *Example of Algorithm* C

[3] The notion of "to the right" has only been defined for nodes with the same predecessor. We can extend it naturally by saying that if n is to the right of m, then all n's successors are to the right of all of m's successors.

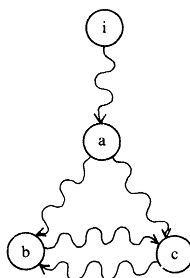


FIG. 8. *The graph* (*)

Basis. $n = 3$. This is an elementary consideration of the three cases in Fig. 9 with the initial nodes at the top.

Inductive step. $n > 3$. Assume that the inductive hypothesis is true, and consider a nonreducible flow graph G with n nodes. By Lemma 3, we may assume without loss of generality that T_1 is not applicable to G. That is, if G can become G' under repeated application of T_1, and we can show that G' has (*), then we will also have shown that G has (*). By the inductive hypothesis and Lemma 3, it follows that T_2 is not applicable to G. Thus, we may assume that G is irreducible. Let T be a DFST for G, and let the spine of T be $(n_1, n_2, \cdots, n_k)$.

We claim that $k \geqq 3$. The initial node n_1 is on the spine. Now consider the right-most successor of the root, namely n_2. Surely n_2 exists, since $n > 1$. Node n_2 must have at least two entering edges in G, since G is irreducible (else T_2 would be applicable). By Lemma 2, other entering edges must come from nodes having a path from n_2 in the tree. Thus, n_2 must have at least one successor, n_3.

Now find the highest number d, such that n_d has an edge (in G but not T) to some $n_i \neq n_1$ on the spine, with $i < d$. n_d always exists because, in particular, n_2 has such an edge entering. Let b be the largest number in the range $1 < b < d$, such that there is an edge from n_d to n_b in G.

Find (if possible) the first node n_a on the spine starting from the root with a forward edge (in G but not in T) entering a node n_c, such that n_c is below n_b on the spine and equal to or below n_d. Figure 10 depicts this situation. Notice that nodes n_a, n_b, and n_c correspond to nodes a, b, and c in (*), and n_1 corresponds to i.

Suppose that there is no such edge (n_a, n_c) in G. Let us consider the subgraph H of G consisting of the nodes on the spine from n_b to n_d, together with their connecting edges in G. There are no edges of G entering a node in H from above other than n_b by assumption, and there are no edges of G entering a node in H below n_d on the spine since (n_d, n_b) is the "lowest" backward edge. Furthermore,

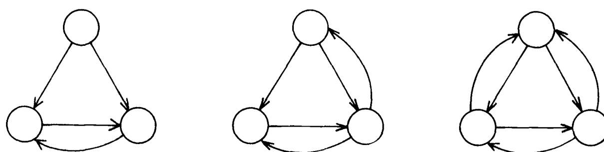


FIG. 9. 3-*Node irreducible flow graphs*

FIG. 10. *Paths in spanning tree*

by Lemma 2 no other edges enter nodes in H. Thus, any reduction by T_1 or T_2 taking place in H, with n_b treated as the initial node, will also be a valid reduction in G. Since G is irreducible, we conclude that H is likewise irreducible. Finally, since $b > 1$, the induction hypothesis applies to H. This ends the induction.

But, since H has a subgraph of form (*) with initial node n_b, it is easy to show that G has a subgraph (*) with initial node n_1 by adding the path from n_1 to n_b.

COROLLARY. *If G is irreducible, then it has a subgraph (*) in which the path from a to c is a single edge.*

Theorem 5 is stronger than a previously known result [4], [15], which states that every nonreducible graph has a double entry cycle. For example, Fig. 11 shows a graph which has a double entry cycle, but which is a "D-chart." In the next section we use Theorem 5 to prove that all D-charts are reducible.

THEOREM 6. *If a flow graph G has a subgraph (*), then G is nonreducible.*

Proof. We proceed by induction on the number of nodes, n, in G. The basis is again trivial. For the induction, suppose that G of $n > 3$ nodes is reducible, but has a subgraph (*). Let G' be the graph formed by applying T_1 to G until no longer possible. It is easy to see that G' also contains (*), and by Theorem 2 is reducible. Therefore T_2 is applicable to some node pair (n_1, n_2) of G'. Let n_2 be the successor of n_1, and let G'' be the result of applying T_2 to G'. We consider cases, depending on the relation of n_2 to (*).

Case 1. n_2 is not one of the nodes represented by (*), including the paths shown. It is straightforward in this case to show that (*) is present in G''.

Case 2. n_2 is a of (*). Then n_1 must be the predecessor of a on the path from i to a. Again, (*) exists in G''.

Case 3. n_2 is b or c. Since b and c each have at least two distinct predecessors, this case is impossible.

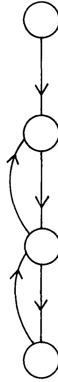


FIG. 11. *Reducible flow graph with double entry cycle*

Case 4. n_2 is a node on one of the paths of (*). Then n_1 is on the same path (possibly an endpoint). (*) clearly exists in G''.

Since G'' has one fewer node than G, the inductive hypothesis applies to G''. Therefore G'' is nonreducible. But by Theorem 2, since $G \overset{*}{\Rightarrow} G''$, and $G \overset{\wedge}{\Rightarrow} 0$, it follows that $G'' \overset{\wedge}{\Rightarrow} 0$, i.e., G'' is reducible. We have a contradiction, and conclude that G is nonreducible.

7. Applications of the characterization theorem. *D*-charts [16]–[19] or "block form programs" [20] are a restricted class of flow charts which can be implemented by a programming language having no explicit "goto" statements. They are as powerful as general flow charts provided additional variables called "flags" are introduced to represent a history of control flow [17].

We define *D*-charts by informal "graph grammars." (See [21], for example.) The graph grammars we use are similar to the grammars for formal languages, except that the production rules indicate the replacement of nodes in a labeled graph by subgraphs. For example, Fig. 12 presents a simple definition of *D*-charts. The start symbol is ⟨block⟩. Rule (3) in Fig. 12 shows that a ⟨block⟩ may be replaced by an "iteration" structure, (while-do), and rule (2) enables possible replacement of a ⟨block⟩ by an "if-then-else" structure.

DEFINITION. A *D-chart* is a flow graph which can be produced by the following rules.

1. Begin with a single node, the initial node, labeled ⟨block⟩.
2. Replace, at will, a node n, labeled ⟨block⟩, by one of the structures on the right of the → in Fig. 12. Edges entering n now enter the highest node in each of the replacement structures. Edges leaving n now leave the lowest node in structures 1, 2 and 4 and the higher node in structure 3.
3. If the node replaced is the initial node, the highest replacing node becomes initial.
4. Terminate the generation process if there are no nodes labeled ⟨block⟩. Otherwise return to step 2.

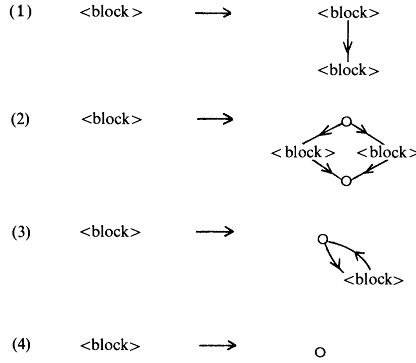


FIG. 12. *Graph grammar for D-charts*

Example 7. The sequence of graphs shown in Fig. 13 illustrate the generation of a *D*-chart. Figs. 13(b), (c) and (d) are produced by rules 2, 1 and 3, respectively. Figure 13(e), the *D*-chart, is produced by three applications of rule 4.

THEOREM 7. *Every D-chart is reducible.*

Proof. We shall use Theorem 5 and show that (*) cannot appear in a *D*-chart. If (*) does appear, then node *a*, which has at least two direct descendants, must be created as the highest node in one of the replacement structures of rules (2) and (3) in Fig. 12. These possibilities are shown in Figs. 14(a) and (b) respectively.

In Fig. 14(a), regions R_1 and R_2 are the sets of nodes generated by the two nodes labeled ⟨block⟩ in Fig. 12 (2). Since paths in (*) are node disjoint, nodes *b*

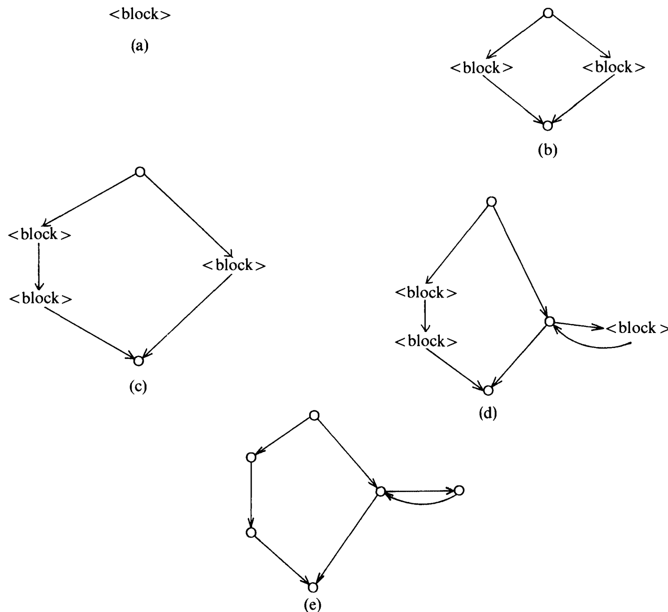


FIG. 13. *Generation of a D-chart*

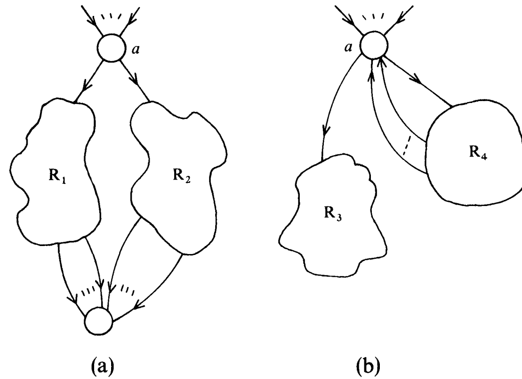


FIG. 14. *Portions of a D-chart*

and c must be found in R_1 and R_2, respectively. But it is elementary that there can be no paths from R_1 to R_2 that do not pass through a. Thus, no (*) exists in this case.

In Fig. 14(b), region R_4 represents the nodes generated by the node ⟨block⟩ in Fig. 12 (3), and R_3 represents the nodes accessible from a without entering R_4. We note that any node labeled ⟨block⟩ in the generation scheme for D-charts has out-degree at most one. Thus, b and c of (*) must appear in R_3 and R_4, respectively. Again, we observe that a path from b to c must pass through a, and we conclude the theorem.

Another simple example of the application of Theorem 5 is the following.

THEOREM 8. *The flow graphs of those* FORTRAN *programs whose transfers to previous statements are all caused by the normal termination of* DO *loops are reducible.*

Proof. If the flow graph for such a program had subgraph (*), then the loop between nodes b and c would be part of a DO loop, and the paths from a to b and c could not be part of that DO loop. Since DO loops may be entered at only one point, we would conclude that b and c are the same node. Thus, (*) does not appear in such a flow graph.

8. Conclusions. We have demonstrated that interval analysis is a special case of a more general reduction technique. This technique, the application of two transformations:

T_1 = removal of loops,

T_2 = merging of a node with a unique predecessor with that predecessor,

can be used for data flow analysis exactly as interval analysis is used.

We then showed that all and only the nonreducible flow graphs have a subgraph (*) consisting of at least three nodes, a, b and c, with node disjoint paths from the initial node to a, from a to b and c and from b to c and back. (a may be the initial node.) We used this result to prove that certain kinds of programs have reducible flow graphs.

REFERENCES

[1] F. E. ALLEN, *Control flow analysis*, SIGPLAN Notices, 5 (1970), pp. 1–19.

[2] ———, *Program Optimization*, Annual Review in Automatic Programming, vol. 5, Pergamon Press, New York, 1969.

[3] A. V. AHO AND J. D. ULLMAN, *The Theory of Parsing, Translation and Compiling*, vol. II, Compiling, Prentice-Hall, Englewood Cliffs, N.J., to appear.

[4] M. SCHAEFER, *A mathematical theory of global program analysis*, unpublished notes, System Devel. Corp., Santa Monica, Calif., 1971.

[5] K. KENNEDY, *A global flow analysis algorithm*, Internat. J. Computer Math., 3 (1971), pp. 5–15.

[6] J. COCKE, *Global common subexpression elimination*, SIGPLAN Notices, 5 (1970), pp. 20–24.

[7] J. E. HOPCROFT AND J. D. ULLMAN, *An n log n algorithm for detecting reducible graphs*, Proc. 6th Annual Princeton Conference on Information Sciences and Systems, Princeton, N.J., 1972.

[8] J. D. ULLMAN, *Fast algorithms for the elimination of common subexpressions*, TR 106, Computer Science Laboratory, Princeton Univ., Princeton, 1972.

[9] D. E. COOPER, *Programs for mechanical program verification*, Machine Intelligence 6, American Elsvier, New York, 1971, pp. 43–62.

[10] J. COCKE AND R. E. MILLER, *Some analysis techniques for optimizing computer programs*, Proc. 2nd Internat. Conf. of System Sciences, Hawaii, 1969.

[11] E. ENGLER, *Structure and meaning of elementary programs*, Symposium on Semantics of Algorithmic Languages, E. Engeler, ed., Springer-Verlag, New York, 1971, pp. 89–101.

[12] ———, *Algorithmic approximations*, J. Comput. System Sci., 5 (1971), pp. 61–82.

[13] A. V. AHO, R. SETHI and J. D. ULLMAN, *Code optimization and finite Church–Rosser systems*, TR 92, Computer Science Laboratory, Princeton University, Princeton, N.J., 1971.

[14] R. TARJAN, *Depth first search and linear graph algorithms*, Proc. IEEE 12th Annual Symposium on Switching and Automata Theory, 1971.

[15] F. E. ALLEN, private communication.

[16] E. W. DIJKSTRA, *GOTO statement considered harmful*, Comm. ACM, 11 (1968), pp. 147–148.

[17] J. BRUNO AND K. STEIGLITZ, *The expression of algorithms by charts*, TR 88, Computer Science Laboratory, Princeton University, Princeton, 1971.

[18] D. E. KNUTH AND R. W. FLOYD, *Notes on avoiding "GOTO" statements*, TR-CS 148, Computer Science Department, Stanford University, Stanford, Calif., 1970.

[19] J. R. RICE, *The GOTO statement reconsidered*, Comm. ACM, 11 (1968), p. 538, and reply by E. W. Dijkstra, p. 538 and p. 541, 11 (1968).

[20] D. C. COOPER, *Some transformations and standard forms of graphs, with applications to computer programs*, Machine Intelligence 2, American Elsevier, New York, 1968, pp. 21–32.

[21] T. PAVLIDIS, *Linear and Context Free Graph Grammars*, J. Assoc. Comput. Mach., 19 (1972), pp. 11–22.

SIAM J. COMPUT.
Vol. 1, No. 3, September 1972

EXPECTATIONS OF FUNCTIONS OF SEQUENCES OVER FINITE ALPHABETS WITH GIVEN TRANSITION PROBABILITIES BY METHODS INDEPENDENT OF SEQUENCE LENGTH*

D. M. JACKSON†

Abstract. A special case of the problem discussed in this paper occurs in connection with nonparametric classification and is introduced from this point of view. The special case concerns the computation of expectations of statistical functions of the "distance" between pairs of fixed-length sequences over a binary alphabet with given a priori state transition probabilities. The general problem involves an extension to alphabets of arbitrary order and the comparison of an arbitrary number of fixed-length sequences. Given a set of sequences, it is shown that for a large class of functions exact computation may be carried out by an algorithm whose computation time is independent of the length of the sequences. It is further shown that results for all functions of this class may be derived from a small number of *basis functions*. Two methods for computing basis functions are given. Basis functions for the commonly encountered special case involving pairs of binary sequences are given explicitly.

Key words and phrases. Nonparametric classification, generating functions, algorithms, combinatorial functions.

1. Introduction. The work presented here was motivated by a problem in nonparametric classification theory. Accordingly, a brief description of the problem is now given.

Nonparametric classification theory deals with the construction of classifications of populations of objects when the underlying statistical distribution is the unknown "parameter." The objective of classification in this case is to derive a decomposition of the data into classes which will reveal the structure of the population, and to generate predictions and hypotheses about the population which may be verified independently. Methods of factor analysis, linear and nonlinear discrimination and cluster analysis of various types are instances of nonparametric techniques (Good [1], Friedman and Rubin [2], Zadeh [3], Cover and Hart [4], Haralick [5], Dempster [6], inter al.). The methods have been used in a broad range of applications including pattern recognition, in which recognition and identification are of prime importance (Nagy [7], Becker [8], inter al.), taxonomy in which the discovery of phylogenetic relationships in evolutionary populations is of prime importance (Cormack [9], Jardine and Sibson [10], inter al.), and information retrieval in which the characterization of linguistic or semantic relationships is of prime importance (Jackson [11], inter al.).

In a typical situation the objects of the population are characterized by a set of vectors whose components specify the observed value or state of an attribute. The vectors are of fixed length. A classification algorithm develops classes of objects which are similar to each other while remaining well-differentiated from

* Received by the editors March 28, 1972.

† Faculty of Mathematics, Department of Combinatorics and Optimization, University of Waterloo, Waterloo, Ontario, Canada. This work was supported in part by the National Science Foundation under Grant GN-27224, and in part by a University of Waterloo Research Grant, and by a grant from the National Research Council of Canada.

other classes. Similarity may be regarded as a generalized distance function on the attribute space. The most frequently encountered problems involve binary attributes and similarity is a distance function for pairs of fixed-length binary vectors.

Estimates of the effect of errors in the data on the determination of the distance between pairs of objects is a necessary adjunct to the construction of a classification. In cluster analysis, for example, the construction of a putative class may proceed by accretion. If estimates of the expected error in distance are available, the assignment of objects to classes can be modified in a way which reduces the chance of spurious assignment due to faulty data (Jackson and White [12]). The usefulness of statistical functions (mean, variance, etc.) of the distance between objects in the presence of errors extends to general nonparametric methods.

In this paper, attention is confined to the case in which the quantities, whose expectations are to be determined, are expressible as multivariate polynomials, either directly or by approximation. Ito [13], for example, gives a complete orthonormal basis for functions of binary variables and demonstrates its relationship to finite Walsh functions (Golomb [14]) and Lazarsfeld–Bahadur functions (Bahadur [15]). In addition, it will be assumed that the random errors are independent and have a common distribution. The case of systematic or propagated errors is not treated. The assumption is valid in a variety of applications, in particular those in which the attribute values are determined by independent measurements. Ecological classification, information retrieval and taxonomy are examples of such applications.

The computation of these expectations is made difficult by the combinatorial nature of the problem. The problem is particularly acute when the number of attributes is too small to derive sufficiently accurate approximations using the central limit theorem. In this paper it is shown that, for a reasonably broad class of distance functions which includes the Hamming distance function, the computation for determining expectations of statistical functions of distance can be carried out very rapidly. More precisely, if the distance function is representable as a multivariate polynomial, then the dependence of computation time on the number of attributes is reduced from polynomial to constant. Moreover, each expectation is expressible in terms of a small number of *basis functions*. Since the latter are multivariate polynomials in the a priori transition probability matrix and the arguments of the distance functions, the determination for different error models or for different pairs of objects is reduced to direct polynomial evaluation. Rational preconditioning (Pan [16]) may be used to speed up the polynomial evaluation.

Although the binary problem, involving pairs of sequences over a binary alphabet, is our main concern, the more general problem involving w-tuples of sequences of length N over an alphabet of order k is treated, since this may be done without additional complication. The general problem is reminiscent of the generalized matching problems (Barton [17]) with the additional property that the kind of an object may be misrecognized. It is anticipated that the general problem is of use in evaluating the results of classification by nonparametric methods. More specifically, the admission of the extended alphabet facilitates the examination of circumstances in which the attributes are not dichotomous.

The purpose of this paper is twofold: first, to determine exactly, in an analytic form, the expectations of statistical functions of the distance between pairs of objects when observations on discrete attributes are susceptible to error; second, to determine, in a form convenient for subsequent numerical computation, explicit expressions for the expectations of statistical functions for the important special case concerning pairs of sequences over the binary alphabet (0, 1). Expressions will be derived for arbitrary state transition probabilities so that specialization to a particular case, for example, equiprobability of error, is immediate. For reference purposes, the algebraic forms of these basis functions are given in Appendix B for the commonly encountered special case involving pairs of fixed-length sequences over the binary alphabet.

Section 2 contains a definition of the general problem, a special case of which is the comparison of pairs of binary sequences. The computational order of an algorithm for solving the problem is found to be polynomial in the length of the sequences. Section 3 shows that the computational order of the process is independent of the length of the sequences for a class of distance functions which includes Hamming distance. In addition, it is shown that a solution of the problem may be given in terms of the joint moments of the error distribution and that these may be reduced to multivariate polynomials in the arguments of the distance function and the a priori state transition probability matrix. Section 4 gives a number of recurrence relations for reducing the computation time further. A characterization of the joint moments of arbitrary order is given in terms of the basis functions, which are shown to be incomplete joint factorial moments (incomplete in the sense that the terms of the basis functions are among those of the corresponding joint factorial moments). Section 5 gives an alternative method for constructing the joint moments. Examples of the application of the two methods are given in Section 6 and a tabulation of certain basis functions in Appendix B.

2. Preliminary analysis. Suppose that U is a set of sequences of length N composed of symbols from a finite alphabet of order k, and that $(A_1, A_2, \cdots, A_w)$ is an arbitrary w-tuple of sequences from U. The symbols of the alphabet label the states of the attributes and the sequences are characterizations of the objects. $\Lambda \equiv \{u_i\}$ is a λ-partition of the set $X \equiv \{x_i\}$ of all distinct w-tuples of symbols of the alphabet. In general Λ serves as a device for collecting together separate frequency counts on prescribed configurations of attributes to form, by linear combination, a set of variables which may be more convenient in specific realizations of the problem. $\mathbf{n}$ is a vector whose ith component gives the number of symbol positions in which the configuration u_i occurs.

$f(\mathbf{n})$ is a given real-valued function (distance function) of $\mathbf{n}$ and ϕ is a statistical function of f. The *error matrix* $\mathbf{\Phi}$ is an array giving the expectations of ϕ for all w-tuples of sequences when the symbols of the sequences are susceptible to error independently with a given a priori probability $\mathbf{H}$. The $\lambda \times \lambda$ transition probability matrix $\mathbf{P}$ giving the probability of transitions $u_j \rightarrow u_i$ may be derived straightforwardly from $\mathbf{H}$ and is column stochastic. Efficient computation of the error matrix $\mathbf{\Phi}$ when

$$\phi(f(\mathbf{m})) \in \mathbb{P}_\lambda \tag{2.1}$$

(the class of multivariate polynomials in λ variables) is the principal concern of this paper.

For brevity we shall refer to the construction of $\mathbf{\Phi}$ as the (w, k, N)-problem, since these quantities characterize the problem. Since the (w, k, N)-problem has λ variables we shall also refer to it as the λ-variable problem.

In practice, the $(2, 2, N)$-problem, involving the comparison of pairs of binary sequences, is the most frequently encountered. The transition probability matrix for this case is given by

$$\mathbf{H} = \begin{bmatrix} s_+ & r_- \\ r_+ & s_- \end{bmatrix}, \quad \text{where } s_+ + r_+ = s_- + r_- = 1$$

and $r_+ = \text{prob}\,(0 \to 1)$, $r_- = \text{prob}\,(1 \to 0)$.

Since pairs of object characterizations are involved,

$$X \equiv (x_1, x_2, x_3, x_4) = ((0,0), (0,1), (1,0), (1,1)).$$

The independence of f on the order in which objects are compared induces a partition on X given by:

$$\Lambda \equiv (u_1, u_2, u_3) = (x_1, (x_2, x_3), x_4) \quad \text{so } \lambda = 3.$$

The transition probability matrix for the elements of Λ may be determined from Λ and $\mathbf{H}$, and is given by

$$\begin{bmatrix} s_+^2 & r_-s_+ & r_-^2 \\ 2r_+s_+ & s_-s_+ + r_-r_+ & 2r_-s_- \\ r_+^2 & r_+s_- & s_-^2 \end{bmatrix}.$$

It has been shown (Jackson and White [18]) that the order of the computation may be reduced in the $(2, 2, N)$ case with $r_+ = r_-$ provided that each element of $\mathbf{\Phi}$ is required only to within prescribed accuracy ε. A conservative upper estimate on the order of computation in this case is $N^2k_\varepsilon^6$, where $k_\varepsilon \ll 2N$. Complete recomputation of $\mathbf{\Phi}$, however, is necessary if N, f, ϕ or $\mathbf{H}$ are altered. In practice, it is commonly useful to compute $\mathbf{\Phi}$ for a variety of N and $\mathbf{H}$ as part of the classification procedure. In § 3 it is shown that repetition of the computations for changes in these quantities may be considerably reduced for the class of ϕ and f which satisfies (2.1).

To simplify the subsequent analysis, a convention is adopted for abbreviating power products, products of multinomial coefficients and products of summation operators. The convention is summarized in Appendix A.

3. Analysis. If ϕ and f satisfy (2.1), then $\psi(\mathbf{m}) \equiv \phi(f)$ may be expressed in the form

$$\psi(\mathbf{m}) = \sum_{\mathbf{i}} a_{\mathbf{i}}\mathbf{m}^{\mathbf{i}}.$$

If ϕ and f do not satisfy (2.1), then the values of $a_{\mathbf{i}}$ and the admissible values of $\mathbf{i}$ are determined by polynomial approximation to $\phi(f)$.

Since

$$\phi^*(\mathbf{n}) = \mathscr{E}(\psi|\mathbf{n}),$$

then

$$\phi^*(\mathbf{n}) = \sum_{\mathbf{i}} a_{\mathbf{i}} \cdot B_{\mathbf{i}}(\mathbf{n}, \mathbf{P}).$$

For independently occurring errors, the probability distribution of $\mathbf{m}$ conditional on $\mathbf{n}$ is given by

$$P(\mathbf{m}|\mathbf{n}) = \sum_{\mathbf{I}} \begin{bmatrix} \mathbf{n} \\ \mathbf{I} \end{bmatrix}_{\rho} \cdot \mathbf{P}^{\mathbf{I}} \quad (\boldsymbol{\rho}(\mathbf{I}) = \mathbf{n}, \boldsymbol{\sigma}(\mathbf{I}) = \mathbf{m}), \tag{3.1}$$

where $\mathbf{I}$ is a $\lambda \times \lambda$ matrix with nonnegative integer elements giving the number of transitions $u_j \to u_i$. Thus $B_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$ are identified as the joint moments of the distribution $P(\mathbf{m}|\mathbf{n})$. The high order joint moments are of value if the distance function is well-approximated by a polynomial.

The computation time for $\mathbf{\Phi}$ may be crudely bounded above by the time taken to compute all the matrices $\mathbf{I}$ whose elements sum to N. For the λ-variable problem there are $\binom{N + \lambda^2 - 1}{\lambda^2 - 1} = O(N^{\lambda^2 - 1})$ such matrices.

The following lemma may be used to reduce this estimate.

LEMMA 3.1. *The probability generating function for* $P(\mathbf{m}|\mathbf{n})$ *is* $G(\mathbf{x}, \mathbf{n}, \mathbf{P}) \equiv (\mathbf{x} \cdot \mathbf{P})^{\mathbf{n}}$.

Proof. The proof is direct from (3.1).

The computation time for $\mathbf{\Phi}$ may be bounded above by the time for computing the sequences $\{P(\mathbf{m}|\mathbf{n})|\sigma(\mathbf{m}) = N\}$ for $\sigma(\mathbf{n}) = N$, since a particular $\phi^*(\mathbf{n})$ is a linear combination of the elements of one of these sequences. There are $N^{\lambda - 1}$ such sequences. From Lemma 3.1, the sequence $\{P(\mathbf{m}|\mathbf{n})|\boldsymbol{\sigma}(\mathbf{m}) = N\}$ is formed by the coefficients of $\mathbf{x}^{\mathbf{i}}$ in $G(\mathbf{x}, \mathbf{n}, \mathbf{P})$. The computation of these coefficients may be regarded as an interpolatory problem which may be carried out in time $O(N^{\lambda - 1} \cdot \log N)$ by fast discrete multivariate Fourier transforms (Cochran et al. [19]). Computation time for $\mathbf{\Phi}$ is therefore bounded above by $O(N^{2\lambda - 2} \log N)$.

Advantage may be taken of the particular form of ψ treated here to carry out a further reduction. The reduction is carried out in Theorem 3.1 and is stated in Corollary 3.1. As a preliminary, the following lemma is needed.

LEMMA 3.2. *The probability generating function for* $B_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$ *is* $G'(\mathbf{x}, \mathbf{n}, \mathbf{P}) \equiv G(\mathbf{e}(\mathbf{x}), \mathbf{n}, \mathbf{P})$, *where* $\mathbf{e}(\mathbf{x}) \equiv (e^{x_1}, e^{x_2}, \cdots, e^{x_\lambda})$.

Proof. See, for example, Moran [20]. The $B_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$ have been identified as joint moments.

Remark. Technically, $G'(\mathbf{x}, \mathbf{n}, \mathbf{P})$ is the probability generating function for $(1/\mathbf{i}!)B_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$.

THEOREM 3.1 (Main theorem). *Let* $B_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$ *be the joint moments of the error distribution given in* (3.1). *Then*

$$B_{\mathbf{i}}(\mathbf{n}, \mathbf{P}) = \sum_{\boldsymbol{\sigma}(\boldsymbol{\gamma}) = \theta(\mathbf{i})}^{\mathbf{i}} (\mathbf{n})_{\boldsymbol{\rho}(\boldsymbol{\gamma})} \cdot \mathbf{P}^{\boldsymbol{\gamma}} \begin{bmatrix} \boldsymbol{\sigma}(\boldsymbol{\gamma}) \\ \boldsymbol{\gamma} \end{bmatrix}_{\sigma} S_{\mathbf{i}}^{(\boldsymbol{\sigma}(\boldsymbol{\gamma}))}.$$

Proof. We shall use the following relationship which results from the column stochastic property of $\mathbf{P}$:

$$\mathbf{e}(\mathbf{x}) \cdot \mathbf{P} = (\mathbf{e}(\mathbf{x}) - \mathbf{1}) \cdot \mathbf{P} + \mathbf{1}.$$

Exponentiating both sides, expanding the right-hand side by the binomial theorem and using Lemma 3.2, we obtain

$$G'(\mathbf{x}, \mathbf{n}, \mathbf{P}) = \sum_{\substack{\mathbf{m} \\ \mathbf{m} \leqq \mathbf{n}}} \binom{\mathbf{n}}{\mathbf{m}} ((\mathbf{e}(\mathbf{x}) - \mathbf{1}) \cdot \mathbf{P})^{\mathbf{m}}.$$

Expansion by the multinomial theorem yields

$$G'(\mathbf{x}, \mathbf{n}, \mathbf{P}) = \sum_{\substack{\mathbf{m} \\ \mathbf{m} \leqq \mathbf{n}}} \binom{\mathbf{n}}{\mathbf{m}} \sum_{\substack{\gamma \\ \rho(\gamma) = \mathbf{m}}} \begin{bmatrix} \mathbf{m} \\ \gamma \end{bmatrix}_\rho \cdot \mathbf{P}^\gamma (\mathbf{e}(\mathbf{x}) - \mathbf{1})^{\sigma(\gamma)}$$

whence, upon rearrangement,

$$G'(\mathbf{x}, \mathbf{n}, \mathbf{P}) = \sum_{\substack{\gamma \\ \rho(\gamma) \leqq \mathbf{n}}} (\mathbf{n})_{\rho(\gamma)} \cdot \mathbf{P}^\gamma \begin{bmatrix} \sigma(\gamma) \\ \gamma \end{bmatrix} (\mathbf{e}(\mathbf{x}) - \mathbf{1})^{\sigma(\gamma)} \frac{1}{(\sigma(\gamma))!}.$$

Using the generating function for Stirling numbers of the second kind, we may reduce this to

$$G'(\mathbf{x}, \mathbf{n}, \mathbf{P}) = \sum_{\substack{\gamma \\ \rho(\gamma) \leqq \mathbf{n}}} (\mathbf{n})_{\sigma(\gamma)} \cdot \mathbf{P}^\gamma \begin{bmatrix} \sigma(\gamma) \\ \gamma \end{bmatrix} \sum_{\substack{\mathbf{i} \\ \mathbf{i} \geqq \sigma(\gamma)}} S_{\mathbf{i}}^{(\sigma(\gamma))} \frac{\mathbf{x}^{\mathbf{i}}}{\mathbf{i}!}.$$

The result follows by reversal of the order of summation and by equating coefficients of $\mathbf{x}^{\mathbf{i}}$ on either side, using Lemma 3.2.

COROLLARY 3.1.

(i) *The time taken to compute* $B_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$ *is independent of* N *and* $\mathbf{n}$.

(ii) *The degree of* $B_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$ *as a polynomial in* $\mathbf{n}$ *and* $\mathbf{P}$ *is independent of* N *and* $\mathbf{n}$.

Proof. The proof follows directly from Theorem 3.1.

A convenient tabulation of Stirling numbers and multinomial coefficients is given by Abramovitz and Stegun [21, Tables 24.2 and 24.4].

Specialization of Theorem 3.1 to the univariate case of a single variable distributed binomially yields

$$m_s = \sum_{x=1}^{s} (n)_x p^x S_s^{(x)}$$

which is an expression for the sth moment given by Riordan [22].

4. Methods of expansion. I. A number of relationships between the $B_i(\mathbf{n}, \mathbf{P})$ are now derived to simplify the construction of explicit polynomials for $B_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$. The following definition is needed.

DEFINITION 4.1 (The substitution convention).

(i) The *substitution convention* (Jackson [23]) $\mathfrak{Q}(x)$ is defined by:

(a) $\mathfrak{Q}(x): x^i \to (x)_i$, i.e., $(x^i)|_{\mathfrak{Q}(x)} = (x)_i$;

(b) $\mathfrak{Q}^{-1}(x): (x)_i \to x^i$.

(ii) The simultaneous application of conventions $\mathfrak{Q}(x)$ and $\mathfrak{Q}(y)$ is denoted by $\mathfrak{Q}(\mathbf{z})$, where $\mathbf{z} = (x, y)$.

(iii) The sequential application of conventions $\mathfrak{Q}(x)$ and $\mathfrak{Q}(y)$ from left to right is denoted by $(\mathfrak{Q}(x), \mathfrak{Q}(y))$.

(iv) The pair of inverses (see Riordan [24]):

(a) $x^n = \sum_{i=1}^{n} S_n^{(i)}(x)_i$, where $S_n^{(i)}$ are Stirling numbers of the second kind,

(b) $(x)_n = \sum_{i=1}^{n} s_n^{(i)} x^i$, where $s_n^{(i)}$ are Stirling numbers of the first kind,

are algebraic devices for manipulating expressions into a form suitable for transformation under one of the conventions $\mathfrak{Q}$ or $\mathfrak{Q}^{-1}$.

The following theorem simplifies the form of $B_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$ given in Theorem 3.1.

THEOREM 4.1. *Let* $\mathfrak{D} \equiv (\mathfrak{Q}^{-1}(\mathbf{P} \cdot \mathbf{n}), \mathfrak{Q}(\mathbf{n}))$. *Then* $B_{\mathbf{i}}(\mathbf{n}, \mathbf{P}) = (\mathbf{P} \cdot \mathbf{n})^{\mathbf{i}}|_{\mathfrak{D}}$.

Proof. From Theorem 3.1,

$$B_{\mathbf{i}}(\mathbf{n}, P) = \sum_{\mathbf{j}=\theta(\mathbf{i})}^{\mathbf{i}} S_{\mathbf{i}}^{(\mathbf{j})} \sum_{\substack{\gamma \\ \boldsymbol{\sigma}(\gamma)=\mathbf{j}}} (\mathbf{n})_{\boldsymbol{\rho}(\gamma)} \cdot \mathbf{P}^{\gamma} \begin{bmatrix} \boldsymbol{\sigma}(\gamma) \\ \gamma \end{bmatrix}_{\sigma}.$$

Thus from Definition 4.1 (i) (a) and the multinomial theorem,

$$B_{\mathbf{i}}(\mathbf{n}, \mathbf{P}) = \sum_{\mathbf{j}=\theta(\mathbf{i})} S_{\mathbf{i}}^{(\mathbf{j})} (\mathbf{P} \cdot \mathbf{n})^{\mathbf{j}}|_{\mathfrak{Q}(\mathbf{n})} = \sum_{j=\theta(\mathbf{i})} S_{\mathbf{i}}^{(\mathbf{j})} (\mathbf{P} \cdot \mathbf{n})_{\mathbf{j}}|_{\mathfrak{D}}$$

whence the result follows by Definition 4.1 (iv) (a).

Certain relationships between the $B_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$ now become clear. The results are incorporated into the next theorem.

THEOREM 4.2 (Permutation theorem). *Let* π_σ *and* π_ρ *be row and column suffix permutations, respectively, on the set of integers* $(1, 2, \cdots, \lambda)$. *Then*

(i) $B_{\pi_\sigma(\mathbf{i})}(\mathbf{n}, \mathbf{P}) = B_{\mathbf{i}}(\mathbf{n}, \pi_\sigma^{-1}(\mathbf{P}))$;

(ii) $B_{\mathbf{i}}(\mathbf{n}, \mathbf{P}) = B_{\mathbf{i}}(\pi_\rho(\mathbf{n}), \pi_\rho(\mathbf{P}))$.

Proof. From Theorem 4.1,

(i) $B_{\pi_\sigma(\mathbf{i})}(\mathbf{n}, \mathbf{P}) = (\mathbf{P} \cdot \mathbf{n})^{\pi_\sigma(\mathbf{i})}|_{\mathfrak{D}} = (\pi_\sigma^{-1}(\mathbf{P}) \cdot \mathbf{n})^{\mathbf{i}}|_{\mathfrak{D}} = B_{\mathbf{i}}(\mathbf{n}, \pi_\sigma^{-1}(\mathbf{P}))$;

(ii) $B_{\mathbf{i}}(\mathbf{n}, \mathbf{P}) = (\mathbf{P} \cdot \mathbf{n})^{\mathbf{i}}|_{\mathfrak{D}} = (\pi_\rho(\mathbf{P}) \cdot \pi_\rho(\mathbf{n}))^{\mathbf{i}}|_{\mathfrak{D}} = B_{\mathbf{i}}(\pi_\rho(\mathbf{n}), \pi_\rho(\mathbf{P}))$.

Theorem 4.2 is of importance in the computation of the $B_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$'s as a set of multivariate polynomials in $\mathbf{n}$ and $\mathbf{P}$. Part (i) of the theorem has two consequences. First, it gives a rule for deriving $B_{\mathbf{j}}(\mathbf{n}, \mathbf{P})$ from $B_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$ if $\mathbf{j}$ is a permutation of $\mathbf{i}$. Second, it provides a means of generating some of the terms of $B_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$ from others of its terms. Part (ii) of the theorem also accomplishes the latter. The results are incorporated into the next theorem which may be regarded as a description of an algorithm for determining $B_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$.

DEFINITION 4.2 (Basis functions). A *basis function* $\Psi_{\mathbf{j}}(\mathbf{n}, \mathbf{P})$ is a multivariate polynomial in $\mathbf{P}$ and $\mathbf{n}$ such that

(i) $\Psi_{\mathbf{j}}(\mathbf{n}, \mathbf{P}) = \sum \begin{bmatrix} \boldsymbol{\sigma}(\gamma) \\ \gamma \end{bmatrix}_{\sigma} (\mathbf{n})_{\boldsymbol{\rho}(\gamma)} \cdot \mathbf{P}^{\gamma}$, where the summation is over all γ such that $\boldsymbol{\sigma}(\gamma) = \mathbf{j}$, $\rho_1(\gamma) \geqq \rho_2(\gamma) \geqq \cdots \geqq \rho_\lambda(\gamma)$.

(ii) If γ and γ' satisfy the summation conditions, and if $\boldsymbol{\sigma}(\gamma) = \boldsymbol{\sigma}(\gamma')$ where $\gamma' = \pi_\sigma(\gamma)$; or if $\boldsymbol{\rho}(\gamma) = \boldsymbol{\rho}(\gamma')$ where $\gamma' = \pi_\rho(\gamma)$, then one of the matrices γ, γ' is eliminated from the summation.

THEOREM 4.3 (First construction theorem). *Let R be the set of all permutations on the integers $(1, 2, \cdots, \lambda)$. Let $R^{(\mathbf{i})}$ be the subset of permutations in R such that $\pi(\mathbf{i}) = \mathbf{i}$, excluding those permutations, except the identity, which exchange zero elements. If $i_1 \geqq i_2 \geqq \cdots \geqq i_\lambda$, then*

$$B_{\mathbf{i}}(\mathbf{n}, \mathbf{P}) = \sum_{\pi_\sigma, \pi_\rho}^{*} B_{\mathbf{i}}^{*}(\pi_\rho(\mathbf{n}), \pi_\rho \pi_\sigma(\mathbf{P})), \quad \pi_\rho \in R, \quad \pi_\sigma \in R^{(\mathbf{i})},$$

where (i)

$$B_{\mathbf{i}}^{*}(\mathbf{n}, \mathbf{P}) = \sum_{\substack{\mathbf{j} = \theta(\mathbf{i}) \\ j_1 \geqq j_2 \geqq \cdots \geqq j_\lambda}}^{\mathbf{i}} S_{\mathbf{i}}^{(\mathbf{j})} \Psi_{\mathbf{j}}(\mathbf{n}, \mathbf{P})$$

and (ii) $\sum^*$ *denotes summation of terms* $\mathbf{n}$ *and* $\mathbf{P}$ *which are algebraically distinct.*

Proof. The proof follows directly from Theorems 3.1 and 4.2.

Remark. If the summation over $R^{(\mathbf{i})}$ is carried out first, then each term of $B_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$ may be expressed as a product of n's multiplied by a sum of products of P's. Thus some factorization of the terms may be achieved.

If values for $\mathbf{P}$ are substituted before those of $\mathbf{n}$, then a reduction in the number of multiplications needed to compute $B_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$ can be obtained.

THEOREM 4.4 (Extension theorem). *If $\Psi_{\mathbf{j}}(\mathbf{n}, \mathbf{P})$ is a basis function, then $\Psi_{\mathbf{j}}(\mathbf{n}, \mathbf{P})$ is independent of λ if $\lambda \geqq \sigma(\mathbf{j})$.*

Proof. If $\lambda \geqq \sigma(\mathbf{j})$, then a vector of length λ can accommodate any set of column sums which sum to a number less than or equal to λ. Addition of a row and column of zeros to the matrix $\boldsymbol{\gamma}$ (see Definition 4.2) does not affect the basis functions.

An example of the use of Theorem 4.3 is given in § 6.

A final observation may be made in connection with Theorem 4.3 to identify the basis functions. Note (i) of Definition 4.2 suggests that the basis functions are related to the joint factorial moments of the error distribution. The relationship is characterized by the following theorem.

THEOREM 4.5. *Let $B_{(\mathbf{i})}(\mathbf{n}, \mathbf{P}) = \sum_{\pi_\sigma, \pi_\rho}^{*} \Psi_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$, $\pi_\rho \in R$ and $\pi_\sigma \in R^{(\mathbf{i})}$. Then the $B_{(\mathbf{i})}(\mathbf{n}, \mathbf{P})$ are the joint factorial moments of the error distribution.*

Proof. The proof consists of identifying the generating function for $B_{(\mathbf{i})}(\mathbf{n}, \mathbf{P})$ as the joint factorial moment generating function.

From Definition 4.2, the generating function for $B_{(\mathbf{i})}(\mathbf{n}, \mathbf{P})$ is

$$G''(\mathbf{x}, \mathbf{n}, \mathbf{P}) = \sum_{\mathbf{i}} \frac{\mathbf{x}^{\mathbf{i}}}{\mathbf{i}!} \sum_{\substack{\boldsymbol{\gamma} \\ \boldsymbol{\sigma}(\boldsymbol{\gamma}) = \mathbf{i}}} (\mathbf{n})_{\boldsymbol{\rho}(\boldsymbol{\gamma})} \begin{bmatrix} \boldsymbol{\sigma}(\boldsymbol{\gamma}) \\ \boldsymbol{\gamma} \end{bmatrix}_{\sigma} \mathbf{P}^{\boldsymbol{\gamma}}.$$

Rearrangement of this expression yields

$$G''(\mathbf{x}, \mathbf{n}, \mathbf{P}) = \sum_{\mathbf{i}} \binom{\mathbf{n}}{\mathbf{i}} \sum_{\substack{\boldsymbol{\gamma} \\ \boldsymbol{\rho}(\boldsymbol{\gamma}) = \mathbf{i}}} \begin{bmatrix} \mathbf{i} \\ \boldsymbol{\gamma} \end{bmatrix}_{\rho} \mathbf{P}^{\boldsymbol{\gamma}} \mathbf{x}^{\boldsymbol{\sigma}(\boldsymbol{\gamma})},$$

which is reduced, by the multinomial theorem, to

$$G''(\mathbf{x}, \mathbf{n}, \mathbf{P}) = \sum_{\mathbf{i}} \binom{\mathbf{n}}{\mathbf{i}} (\mathbf{x} \cdot \mathbf{P})^{\mathbf{i}}.$$

From Lemma 3.1 and the column stochastic property of P, we have

$$G''(\mathbf{x}, \mathbf{n}, \mathbf{P}) = G(\mathbf{x} + \mathbf{1}, \mathbf{n}, \mathbf{P}).$$

Thus, $G''(\mathbf{x}, \mathbf{n}, \mathbf{P})$ is the joint factorial moment generating function and the theorem follows.

For the relationship between moments and factorial moments in the univariate case, see, for example, Moran [20].

5. Methods of expansion. II. Theorem 4.1 may be used to provide an alternative method for constructing the $B_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$. The following definition is needed.

DEFINITION 5.1.

(i) Multiplication with respect to $\mathfrak{Q}(x)$ is defined as follows: Let $g_1(x)$ and $g_2(x)$ be arbitrary functions. Then $g_1(x) \times_{\mathfrak{Q}(x)} g_2(x) \equiv (g_1(x)|_{\mathfrak{Q}^{-1}(x)} \times g_2(x)|_{\mathfrak{Q}^{-1}(x)})|_{\mathfrak{Q}(x)}$.

(ii) Iterated multiplication with respect to $\mathfrak{Q}(x)$ is denoted by $\prod_{\mathfrak{Q}(x)}$.

We shall use $\times_{\mathfrak{D}}$ for computing the joint moments from the moments. The following lemma is required.

LEMMA 5.1. *Let U and V be two expressions. Then $U \times_{\mathfrak{Q}(x,y)} V = U \times_{\mathfrak{Q}(y)} V$ if not both U and V depend on x.*

Proof. The proof is straightforward.

THEOREM 5.1 (Recurrence relation).

$$B_{\mathbf{i}+\mathbf{i}'}(\mathbf{n}, \mathbf{P}) = B_{\mathbf{i}}(\mathbf{n}, \mathbf{P}) \times_{\mathfrak{D}} B_{\mathbf{i}'}(\mathbf{n}, \mathbf{P}).$$

Proof. From Theorem 4.1,

$$\begin{aligned} B_{\mathbf{i}+\mathbf{i}'}(\mathbf{n}, \mathbf{P}) &= ((\mathbf{P} \cdot \mathbf{n})^{\mathbf{i}} \times (\mathbf{P} \cdot \mathbf{n})^{\mathbf{i}'})|_{\mathfrak{D}} \\ &= ((\mathbf{P} \cdot \mathbf{n})^{\mathbf{i}}|_{(\mathfrak{D}, \mathfrak{D}^{-1})} \times (\mathbf{P} \cdot \mathbf{n})^{\mathbf{i}'}|_{(\mathfrak{D}, \mathfrak{D}^{-1})})|_{\mathfrak{D}} \end{aligned}$$

since the convention $(\mathfrak{D}, \mathfrak{D}^{-1})$ has null effect.The result follows immediately.

The following theorem provides a construction for $B_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$ based on the previous theorem.

THEOREM 5.2 (Second construction theorem). *Let $\mathbf{i}(j)$ denote the vector derived from $\mathbf{i}$ as follows:*

$$i_k(j) = \begin{cases} 0 & \text{if } k \neq j, \\ 1 & \text{otherwise}. \end{cases}$$

Then (i) $B_{\mathbf{i}}(\mathbf{n}, \mathbf{P}) = \prod_{j=1\ \mathfrak{Q}(\mathbf{n})}^{\lambda} B_{\mathbf{i}(j)}(\mathbf{n}, \mathbf{P})$;

(ii) $B_{\mathbf{i}(j)}(\mathbf{n}, \mathbf{P}) = \sum_{k=1}^{i_j} S_k^{(i_j)}((\mathbf{P} \cdot \mathbf{n})^{\mathbf{1}(j)})^k|_{\mathfrak{Q}(\mathbf{n})}$.

Proof. (i) Let $\mathbf{x} = \mathbf{P} \cdot \mathbf{n}$, and suppose that $j \neq k$. Then from Theorem 4.1,

$$\begin{aligned} B_{\mathbf{i}(j)}(\mathbf{n}, \mathbf{P}) \times_{\mathfrak{D}} B_{\mathbf{i}(k)}(\mathbf{n}, \mathbf{P}) &= x_j^{i_j}|_{\mathfrak{D}} \times_{\mathfrak{Q}^{-1}(x_j, x_k), \mathfrak{Q}(\mathbf{n})} x_k^{i_k}|_{\mathfrak{D}} \\ &= x_j^{i_j}|_{\mathfrak{D}} \times_{\mathfrak{Q}(\mathbf{n})} x_k^{i_k} \quad \text{(from Lemma 5.1, since } j \neq k) \\ &= B_{\mathbf{i}(j)}(\mathbf{n}, \mathbf{P}) \times_{\mathfrak{Q}(\mathbf{n})} B_{\mathbf{i}(k)}(\mathbf{n}, \mathbf{P}). \end{aligned}$$

The theorem follows by application of this result to the components of **i** and from Theorem 5.1.

(ii)
$$B_{\mathbf{i}(j)} = (x_j^{i_j})|_{(\mathfrak{Q}^{-1}(x_j),\mathfrak{Q}(\mathbf{n}))}$$
$$= \sum_{k=0}^{i_j} S_k^{(i_j)}(x_j)_k|_{(\mathfrak{Q}^{-1}(x_j),\mathfrak{Q}(\mathbf{n}))} \quad \text{(from Definition 4.1 (iv) (a))}$$
$$= \sum_{k=0}^{i_j} S_k^{(i_j)} x_j^k|_{\mathfrak{Q}(\mathbf{n})}.$$

The result follows since $x_j = (\mathbf{P}\cdot\mathbf{n})^{\mathbf{1}(j)}$.

It follows from Theorem 5.2 that each $B_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$ may be computed from the set of polynomials $\{B^k_{\mathbf{1}(j)}(\mathbf{n}, \mathbf{P})\}$; $k = 1, 2, \cdots, i_j$, $j = 1, 2, \cdots, \lambda$. These polynomials are exponentiated multilinear forms. Theorem 4.2 (permutation theorem) shows that this set may be reduced to $\{B^k_{\mathbf{1}(1)}(\mathbf{n}, \mathbf{P})\}$, $k = 1, 2, \cdots, \max(i_1, i_2, \cdots, i_\lambda)$ by suffix permutation. An example of the use of Theorem 5.2 is given in § 6.

6. Examples of the algorithms.

Example 1. Suppose that $f(\mathbf{m}) = m_1 m_2^{1/2}$ and that $(f^2|\mathbf{n})$ is to be computed. Now $(f^2|\mathbf{n}) = B_{210}(\mathbf{n}, \mathbf{P})$. The set R of permutations is given by

$$R = \{(1,2,3),(2,1,3),(3,1,2),(1,3,2),(2,3,1),(3,2,1)\}$$
$$\equiv \{\pi_1, \pi_2, \pi_3, \pi_4, \pi_5, \pi_6\}.$$

In addition, $R^{(2,1,0)} = \{\pi_1\}$. From the first construction theorem (Theorem 4.3),

$$B_{210}(\mathbf{n}, \mathbf{P}) = \sum_{\substack{\pi_\rho \\ \pi_\rho \in R}}^{*} (\Psi_{210}(\pi_\rho(\mathbf{n}), \pi_\rho(\mathbf{P})) + \Psi_{110}(\pi_\rho(\mathbf{n}), \pi_\rho(\mathbf{P}))).$$

The construction of $B_{210}(\mathbf{n}, \mathbf{P})$ involves six matrices. These are tabulated in Table 1 together with the terms in **n** and **P** which they generate.

The factorization mentioned in the remark following Theorem 4.3 is observed in Table 2 and may be carried out by constructing all matrices with the same column sums, for given row sums. If **P** is given numerically, then this factorization results in a reduction of time when the multivariate polynomials are evaluated for a number of **n**'s. Table 2 tabulates the action of the operator $\sum^*$.

Thus $B_{210}(\mathbf{n}, \mathbf{P})$ is equal to the sum of the terms contained in Table 2. Theorem 4.2 allows us to compute $B_{120}(\mathbf{n}, \mathbf{P})$, $B_{201}(\mathbf{n}, \mathbf{P})$, $B_{102}(\mathbf{n}, \mathbf{P})$, $B_{012}(\mathbf{n}, \mathbf{P})$ and $B_{021}(\mathbf{n}, \mathbf{P})$ from $B_{210}(\mathbf{n}, \mathbf{P})$ by row suffix permutation.

The tables of Appendix B give the basis functions for the case of the binary alphabet. It has been shown in § 2 that three variables ($\lambda = 3$) are involved in this problem. Table B.1 gives $B^*_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$ in terms of the basis functions, while Table B.2 expresses the basis functions as explicit polynomials in **n** and **P**. The form of the latter matrix, for the binary case, has been given in § 2 in terms of the a priori transition probability between the symbols of the alphabet. Provision of Tables B.1 and B.2 removes the necessity of reconstructing basis functions since they can be stored as initial data. Expansion of the compact $B^*_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$ by row and column suffix permutations to form $B_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$ is a straightforward task.

TABLE 1
Tabulation of matrices involved in the construction of $B_{210}(\mathbf{n}, \mathbf{P})$

Matrix γ	Column Sum Vector $\rho(\gamma)$	Row Sum Vector $\sigma(\gamma)$	Constant $\begin{bmatrix}\sigma(\gamma)\\ \gamma\end{bmatrix}_\sigma$	Algebraic Term $(\mathbf{n})_{\rho(\gamma)}\mathbf{P}^\gamma$
$\begin{bmatrix}2 & \cdot & \cdot\\ 1 & \cdot & \cdot\\ \cdot & \cdot & \cdot\end{bmatrix}$	(3, 0, 0)	(2, 1, 0)	1	$(n_1)_3P_{11}^2P_{21}$
$\begin{bmatrix}2 & \cdot & \cdot\\ \cdot & 1 & \cdot\\ \cdot & \cdot & \cdot\end{bmatrix}$	(2, 1, 0)	(2, 1, 0)	1	$(n_1)_2(n_2)_1P_{11}^2P_{22}$
$\begin{bmatrix}1 & 1 & \cdot\\ 1 & \cdot & \cdot\\ \cdot & \cdot & \cdot\end{bmatrix}$	(2, 1, 0)	(2, 1, 0)	2	$(n_1)_2(n_2)_1P_{11}P_{12}P_{11}$
$\begin{bmatrix}\cdot & 1 & 1\\ 1 & \cdot & \cdot\\ \cdot & \cdot & \cdot\end{bmatrix}$	(1, 1, 1)	(2, 1, 0)	2	$(n_1)_1(n_2)_1(n_3)_1P_{12}P_{13}P_{21}$
$\begin{bmatrix}1 & \cdot & \cdot\\ 1 & \cdot & \cdot\\ \cdot & \cdot & \cdot\end{bmatrix}$	(2, 0, 0)	(1, 1, 0)	1	$(n_1)_2P_{11}P_{21}$
$\begin{bmatrix}1 & \cdot & \cdot\\ \cdot & 1 & \cdot\\ \cdot & \cdot & \cdot\end{bmatrix}$	(1, 1, 0)	(1, 1, 0)	1	$(n_1)_1(n_2)_1P_{11}P_{22}$

Example 2. We shall now determine $B_{210}(\mathbf{n}, \mathbf{P})$ using the second construction theorem (Theorem 5.2).

From Theorem 5.2, $B_{210}(\mathbf{n}, \mathbf{P})$ may be decomposed as follows:

$$B_{210}(\mathbf{n}, \mathbf{P}) = B_{200}(\mathbf{n}, \mathbf{P}) \times_{\mathfrak{Q}(\mathbf{n})} B_{010}(\mathbf{n}, \mathbf{P}).$$

Let $\mathbf{x} = (\mathbf{P} \cdot \mathbf{n})$. Then from Theorem 5.2 (ii),

$$B_{200}(\mathbf{n}, \mathbf{P}) = (x_1^2 + x_1)|_{\mathfrak{Q}(\mathbf{n})} \quad \text{since } S_1^{(2)} = S_2^{(2)} = 1,$$

$$B_{010}(\mathbf{n}, \mathbf{P}) = (x_2)|_{\mathfrak{Q}(\mathbf{n})} \quad \text{since } S_1^{(1)} = 1.$$

Thus

$$\begin{aligned} B_{210}(\mathbf{n}, \mathbf{P}) &= ((x_1^2 + x_1)x_2)|_{\mathfrak{Q}(\mathbf{n})} \\ &= ((P_{11}n_1 + P_{12}n_2 + P_{13}n_3)^2 \\ &\quad +(P_{11}n_1 + P_{12}n_2 + P_{13}n_3))(P_{21}n_1 + P_{22}n_2 + P_{23}n_3)|_{\mathfrak{Q}(\mathbf{n})}. \end{aligned}$$

TABLE 2
Evaluation of $\sum^*_{\substack{\pi_\rho \\ \pi_\rho \in R}} (\Psi_{210}(\pi_\rho(\mathbf{n}), \pi_\rho(\mathbf{P})) + \Psi_{110}(\pi_\rho(\mathbf{n}), \pi_\rho(\mathbf{P})))$

Permutation[1]	Contribution[2] of $B^*_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$ to $B_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$
π_1	$(n_1)_3P_{11}^2P_{21} + (n_1)_2(n_2)_1(P_{11}^2P_{21} + 2P_{11}P_{12}P_{21})$ $+2(n_1)_1(n_2)_1(n_3)_1P_{12}P_{13}P_{21} + (n_1)_2P_{11}P_{21} + (n_1)_1(n_2)_1P_{11}P_{22}$
π_2	$(n_2)_3P_{12}^2P_{22} + (n_2)_2(n_1)_1(P_{12}^2P_{22} + 2P_{12}P_{11}P_{22})$ $+(\text{———}) + (n_2)_2P_{12}P_{22} + (n_2)_1(n_1)_1P_{12}P_{21}$
π_3	$(n_3)_3P_{13}^2P_{23} + (n_3)_2(n_1)_1(P_{13}^2P_{23} + 2P_{13}P_{11}P_{23})$ $+2(n_3)_1(n_1)_1(n_2)_1P_{11}P_{12}P_{23} + (n_3)_2P_{13}P_{23} + (n_3)_1(n_1)_1P_{13}P_{21}$
π_4	$(\text{———}) + (n_1)_2(n_3)_1(P_{11}^2P_{21} + 2P_{11}P_{13}P_{21})$ $+(\text{———}) + (\text{———}) + (n_1)_1(n_3)_1P_{11}P_{13}$
π_5	$(\text{———}) + (n_2)_2(n_3)_1(P_{12}^2P_{22} + 2P_{12}P_{13}P_{22})$ $+2(n_2)_1(n_3)_1(n_1)_1P_{13}P_{11}P_{22} + (\text{———}) + (n_2)_1(n_3)_1P_{12}P_{23}$
π_6	$(\text{———}) + (n_3)_2(n_1)_1(P_{13}^2P_{23} + 2P_{13}P_{12}P_{23})$ $+(\text{———}) + (\text{———}) + (n_3)_1(n_2)_1P_{13}P_{12}$

1. π_i is applied to the polynomial given by the terms in Table 1.
2. (————) indicates a term which has been removed since it was identical algebraically to a term already present.

This can be expanded straightforwardly to give the sum of the terms in Table 2. For example, $B_{210}(\mathbf{n}, \mathbf{P})$ contains the product

$$(P_{11}n_1 + P_{12}n_2 + P_{13}n_3)(P_{21}n_1 + P_{22}n_2 + P_{23}n_3)|_{\mathfrak{Q}(\mathbf{n})}$$

which may be expanded readily into

$$(n_1)_2P_{11}P_{21} + (n_2)_2P_{12}P_{22} + (n_3)_2P_{13}P_{23} + (n_1)_1(n_2)_1(P_{12}P_{21} + P_{11}P_{22})$$
$$+ (n_2)_1(n_3)_1(P_{12}P_{23} + P_{22}P_{13}) + (n_3)_1(n_1)_1(P_{11}P_{23} + P_{13}P_{21}).$$

It is important to note that the application of $\mathfrak{Q}(\mathbf{n})$ at the final stage implies that the variables $\{x_i\}$ may not be allowed to assume numerical values at the start of the computation, even if $\mathbf{P}$ and $\mathbf{n}$ are given.

Although the first construction theorem (Theorem 4.3) leads to a more indirect method for computing $B_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$, the abbreviation obtained by expressing the $B^*_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$ in terms of the basis functions $\Psi_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$ is particularly valuable. Moreover, the expansion of $B^*_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$ to the full form $B_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$ involves only suffix permutation and can be carried out rapidly in a linear scan of the stored forms of the basis functions.

The second construction theorem (Theorem 5.2) was used to verify that the results agreed with those already obtained by the previous method.

7. Summary. The polynomials given in Appendix B may be used in an important special case of the (w, k, N)-problem, namely that of determining the expectations ϕ^* of statistical functions ϕ of the distance function f for pairs of binary sequences of fixed length when the digits are susceptible to error with given probability. It has been seen that this is a 3-variable problem. Appendix B contains the basis functions for computing the expectations for all $\mathbf{i}$ with $\sigma(\mathbf{i}) \leqq 4$. Extension to higher $\sigma(\mathbf{i})$ has not been given for the sake of brevity. The probability matrix $\mathbf{P}$

for the problem is determined straightforwardly from the given a priori state transition probability matrix. For an arbitrary pair of sequences, it has been proved as a special case of Theorem 3.1, that the time taken to determine the expectations of $\phi(f)$ is independent of the length of the sequences.

Appendix A. The implied product convention. The purpose of the implied product convention is to simplify expressions involving products of multinomial coefficients, exponentiated variables and multiple summations. In the subsequent analysis, vectors and matrices appear in boldface. The elements of all vectors and matrices, with the exception of those which are probabilistic, are nonnegative integers. It will be assumed throughout the text that the dimensions of the vectors and matrices are such that the expressions in which they occur are well-defined. Under this assumption, explicit discussion of the sizes of vectors and matrices is largely avoided.

(i) *Inequalities.* Inequalities between matrices operate elementwise. For example,

$$\mathbf{x} > \mathbf{y} \Leftrightarrow x_{ij} > y_{ij} \quad \text{for } 1 \leqq i,j \leqq \lambda, \text{ where } \mathbf{x} \text{ and } \mathbf{y} \text{ are } \lambda \times \lambda.$$

(ii) *Rows and columns of matrices.*

(a) $\boldsymbol{\sigma}(\mathbf{M}) \equiv \{\sigma_i(\mathbf{M})\} \equiv \{\sum_j M_{ij}\}$; row sums of $\mathbf{M}$.

(b) $\boldsymbol{\rho}(\mathbf{M}) \equiv \{\rho_j(\mathbf{M})\} \equiv \{\sum_i M_{ij}\}$; column sums of $\mathbf{M}$.

(iii) *Implied products.*

(a) For matrices:

$$\mathbf{M}^{\mathbf{I}} \equiv \prod_{i,j} M_{ij}^{I_{ij}}, \text{ where } \mathbf{M} \text{ and } \mathbf{I} \text{ are } \lambda \times \lambda.$$

(b) For summation operators:

$$\sum_{\mathbf{i}=\mathbf{a}}^{\mathbf{I}} \equiv \prod_{p,q} \sum_{i_{pq}=a_{pq}}^{I_{pq}}, \text{ where } \mathbf{I}, \mathbf{i} \text{ and } \mathbf{a} \text{ are } \lambda \times \lambda.$$

(c) For multinomial coefficients:

$$\begin{bmatrix} \mathbf{i} \\ \mathbf{M} \end{bmatrix}_\sigma \equiv \prod_{j=1}^{\lambda} \begin{bmatrix} i_j \\ {}_j\mathbf{M} \end{bmatrix} \quad \text{if } \mathbf{i} = \boldsymbol{\sigma}(\mathbf{M}) \text{ and } {}_j\mathbf{M} \text{ is the } j\text{th row of } \mathbf{M}.$$

$$\begin{bmatrix} \mathbf{i} \\ \mathbf{M} \end{bmatrix}_\rho \equiv \prod_{j=1}^{\lambda} \begin{bmatrix} i_j \\ \mathbf{M}_j \end{bmatrix} \quad \text{if } \mathbf{i} = \boldsymbol{\rho}(\mathbf{M}) \text{ and } \mathbf{M}_j \text{ is the } j\text{th column of } \mathbf{M},$$

where $\begin{bmatrix} i \\ \mathbf{u} \end{bmatrix} = \dfrac{i!}{u_1!u_2!\cdots u_\lambda!}$.

(d) For Stirling numbers:

$$S_{\mathbf{I}}^{(\mathbf{J})} \equiv \prod_{i,j} S_{I_{ij}}^{(J_{ij})}.$$

(e) For falling factorials:

$$(\mathbf{n})_{\mathbf{i}} \equiv \prod_j (n_j)_{i_j}.$$

(f) For factorials:

$$\mathbf{i}! \equiv \prod_j i_j!.$$

(g) For binomial coefficients:

$$\binom{\mathbf{n}}{\mathbf{m}} \equiv \prod_i \binom{n_i}{m_i}.$$

(iv) *Special constants.*

(a) $\mathbf{1}$ denotes the unit vector with λ components.

(b) $\mathbf{0}$ denotes the zero vector with λ components.

(c) $\theta(i) = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{otherwise}. \end{cases}$

(d) $\theta(\mathbf{n}) = (\theta(n_1), \theta(n_2), \cdots, \theta(n_\lambda))$.

Throughout the text it will be assumed that summation over a matrix index is carried out for nonnegative integer values of the elements.

Appendix B. The following tables give the forms of $B^*_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$ and $\Psi_{\mathbf{j}}(\mathbf{n}, \mathbf{P})$ for the 3-variable problem for a number of different $\mathbf{i}$. The arguments of these quantities are suppressed for brevity.

TABLE B.1

Expansion of $B^*_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$ in terms of basis functions $\Psi_j(\mathbf{n}, \mathbf{P})$

$B^*_{100} = \Psi_{100}$	$B^*_{110} = \Psi_{110}$
$B^*_{200} = \Psi_{200} + \Psi_{100}$	$B^*_{210} = \Psi_{210} + \Psi_{110}$
$B^*_{300} = \Psi_{300} + 3\Psi_{200} + \Psi_{100}$	$B^*_{220} = \Psi_{220} + \Psi_{210} + \Psi_{110}$
$B^*_{400} = \Psi_{400} + 6\Psi_{300} + 7\Psi_{200} + \Psi_{100}$	$B^*_{111} = \Psi_{111}$
$B^*_{310} = \Psi_{310} + 3\Psi_{210} + \Psi_{110}$	$B^*_{211} + \Psi_{211} + \Psi_{111}$

TABLE B.2

Expansion of the basis functions $\Psi_{\mathbf{i}}(\mathbf{n}, \mathbf{P})$ as polynomials in $\mathbf{n}$ and $\mathbf{P}$

$$\Psi_{100} = (n_1)_1 P_{11}$$

$$\Psi_{200} = (n_1)_2 P_{11}^2 + 2(n_1)_1(n_2)_1 P_{11} P_{12}$$

$$\Psi_{300} = (n_1)_3 P_{11}^3 + 3(n_1)_2(n_2)_1 P_{11}^2 P_{12} + 6(n_1)_1(n_3)_1 P_{11} P_{12} P_{13}$$

$$\Psi_{400} = (n_1)_4 P_{11}^4 + 4(n_1)_3(n_2)_1 P_{11}^3 P_{12} + 6(n_1)_2(n_2)_2 P_{11}^2 P_{12}^2 + 12(n_1)_2(n_2)_1(n_3)_1 P_{11}^2 P_{12} P_{13}$$

$$\Psi_{110} = (n_1)_2 P_{11} P_{21} + (n_1)_1(n_2)_1 P_{11} P_{22}$$

$$\Psi_{210} = (n_1)_3 P_{11}^2 P_{21} + (n_1)_2(n_2)_1(P_{11}^2 P_{22} + 2P_{11} P_{12} P_{21}) + 2(n_1)_1(n_2)_1(n_3)_1 P_{12} P_{13} P_{21}$$

$$\Psi_{220} = (n_1)_4 P_{11}^2 P_{21}^2 + 2(n_1)_3(n_2)_1 P_{11} P_{21}^2 P_{12} + (n_1)_2(n_2)_2(4P_{11} P_{12} P_{21} P_{22} + P_{11}^2 P_{22}^2) + (n_1)_2(n_2)_1(n_3)_1(4P_{11} P_{12} P_{21} P_{23} + 2P_{11}^2 P_{22} P_{23})$$

$$\Psi_{310} = (n_1)_4 P_{11}^3 P_{21} + (n_1)_3(n_2)_1(3P_{11}^2 P_{12} P_{21} + P_{11}^3 P_{21}) + 3(n_1)_2(n_2)_2 P_{11}^2 P_{12} P_{22} + (n_1)_2(n_2)_1(n_3)_1(6P_{11} P_{12} P_{13} P_{21} + 3P_{11}^2 P_{12} P_{23})$$

$$\Psi_{111} = (n_1)_3 P_{11} P_{21} P_{31} + (n_1)_2(n_2)_1 P_{11} P_{21} P_{31} + (n_1)_1(n_2)_1(n_3)_1 P_{11} P_{22} P_{33}$$

$$\Psi_{211} = (n_1)_4 P_{11}^2 P_{21} P_{31} + (n_1)_3(n_2)_1(P_{11}^2 P_{21} P_{32} + 2P_{11} P_{12} P_{21} P_{31}) + (n_1)_2(n_2)_1(n_3)_1(P_{11}^2 P_{22} P_{33} + 2P_{11} P_{21} P_{12} P_{33} + 2P_{12} P_{13} P_{21} P_{31}) + (n_1)_2(n_2)_2(P_{11}^2 P_{22} P_{32} + 2P_{11} P_{12} P_{21} P_{32})$$

Acknowledgments. The author thanks Mr. S. K. Sahni and Mr. J. K. Wong of the Department of Computer Science, Cornell University, for their participation in this work.

REFERENCES

[1] I. J. GOOD, *The estimation of probabilities: An essay on modern Bayesian methods*, Research Monograph 30, MIT Press, Cambridge, Mass., 1965.

[2] H. P. FRIEDMAN AND J. RUBIN, *On some invariant criteria for grouping data*, J. Amer. Statist. Assoc., 62 (1967), pp. 1159–1178.

[3] L. A. ZADEH, *Fuzzy sets*, Information and Control, 8 (1965) pp. 338–353.

[4] T. M. COVER AND P. E. HART, *Nearest neighbour pattern classification*, IEEE Trans. Information Theory, IT-13 (1967), pp. 21–27.

[5] R. M. HARALICK AND I. DINSTEIN, *An iterative clustering procedure*, IEEE Trans. Systems, Man and Cybernetics, SMC-1 (1971), pp. 275–289.

[6] A. P. DEMPSTER, *An overview of multivariate data analysis*, J. Multivariate Analysis, 1 (1971), pp. 316–346.

[7] G. NAGY, *State of the art in pattern recognition*, Proc. IEEE, 56 (1968), pp. 836–862.

[8] P. W. BECKER, *Recognition of Patterns*, Polyteknísk Forlag, København, 1968.

[9] R. M. CORMACK, *A review of classification*, J. Roy. Statist. Soc. Ser. A, 134 (1971), pp. 321–353.

[10] N. JARDINE AND R. SIBSON, *Mathematical Taxonomy*, John Wiley, New York, 1971.

[11] D. M. JACKSON, *Classification, relevance and information retrieval*, Advances in Computers, vol. 11, M. C. Yovits, ed., Academic Press, New York, 1971, pp. 59–125.

[12] D. M. JACKSON AND L. J. WHITE, *The weakening of taxonomic inferences by homological errors*, Math. Biosci., 10 (1971), pp. 63–89.

[13] T. ITO, *A note on a general expansion of functions of binary attributes*, Information and Control, 12 (1968), pp. 206–211.

[14] S. W. GOLOMB, *On the classification of Boolean functions*, IRE Trans. Circuit Theory, CT-6 (1959), pp. 176–186.

[15] R. R. BAHADUR, *On classification based on responses to n dichotomous items*, Studies in Item Analysis, H. Solomon, ed., Stanford University Press, Stanford, Calif., 1961, pp. 169–186.

[16] V. YA. PAN, *Methods of computing values of polynomials*, Russian Math. Surveys, 21 (1966), no. 1, pp. 105–136.

[17] D. E. BARTON, *The matching distributions: Poisson limiting forms and derived methods of approximation*, J. Roy. Statist. Soc. Ser. B, 20 (1958), pp. 73–92.

[18] D. M. JACKSON AND L. J. WHITE, *Stability problems in non-statistical classification theory*, Comput. J., to appear.

[19] W. T. COCHRAN, J. W. COOLEY, D. L. FAVIN, H. D. HELMS, R. A. KAENEL, W. W. LANG, G. C. MALING, JR., D. E. NELSON, C. M. READER AND P. M. WELSH, *What is the fast Fourier transform?*, Proc. IEEE, 55 (1967), pp. 1664–1674.

[20] A. P. MORAN, *An Introduction to Probability Theory*, Clarendon Press, Oxford, 1968.

[21] M. ABRAMOVITZ AND J. A. STEGUN, *Handbook of Mathematical Functions*, Dover, New York, 1965.

[22] J. RIORDAN, *Moment recurrence relations for binomial, Poisson and hypergeometric frequency distributions*, Ann. Math. Statist., 8 (1937), pp. 103–111.

[23] D. M. JACKSON, *Closed form approximations for random errors in distance functions*, Comput. J., to appear.

[24] J. RIORDAN, *Combinatorial Identities*, John Wiley, New York, 1968.

SIAM J. COMPUT.
Vol. 1, No. 3, September 1972

LOCALITY IN PAGE REFERENCE STRINGS*

G. S. SHEDLER AND C. TUNG†

Abstract. A probabilistic model is presented of program material in a paging machine. The sequences of page references in the model are associated with certain sequences of LRU stack distances and have reference patterns formalizing a notion of "locality" of reference. Values for parameters of the model can be chosen to make the page-exception characteristics of the generated sequences of page references consistent with those of actual program traces.

The statistical properties of the execution intervals (times between page-exception) for sequences of references in the model are derived, and an application of these results is made to a queuing analysis of a simple multiprogrammed paging system. Some numerical results pertaining to the program model and the queuing analysis are given.

1. Introduction. Although several authors (see Lewis and Shedler [1], Gaver and Shedler [2], [3], and others) have recognized that characteristics of multiprogrammed paging systems may be obtained by analysis of cyclic-queues, little attention has been paid to the matter of obtaining an appropriate representation of the program load in such studies. Almost invariably it has been the case in the queuing analyses to which we refer that the aspect of program behavior that is required is a characterization of the times to page fault, or execution intervals, and that for reasons of mathematical necessity, execution intervals *for the entire multiprogrammed load* have been taken to be independent, identically (often exponentially) distributed random variables. Although in some cases it can be argued that such a representation is adequate in terms of the response variables in the queuing model that were studied, it appears that a less gross representation of program behavior would be preferable. This paper is an attempt to formulate a representation of the page reference sequences of individual programs, in such a way as to make explicit the memory-size dependence of the program's page-exception characteristics and to formalize the notion of locality of reference.

Most of the previous attempts to model program behavior known to the authors (e.g., Denning, Chen and Shedler [4], Aho, Denning and Ullman [5]) have been primarily addressed to questions of optimality of the replacement algorithm for a single sequence of references rather than to questions of performance in a multiprogramming environment. The exception to this is the study of memory contention in a paging machine of Oden and Shedler [6], in which a model of program behavior in the spirit of this paper is presented. Both the Oden–Shedler model and the model presented in this paper draw on the ideas of stack algorithms and stack distances defined and studied by Mattson, Gecsei, Slutz and Traiger [7].

In § 2 we give a description of the model of program behavior that we propose along with some discussion of locality of reference. We then derive in § 3 the structural properties of sequences of execution intervals in the model. In § 4 some matters are discussed concerning the choice of values for parameters in the model.

* Received by the editors November 2, 1971, and in final revised form May 31, 1972.

† IBM Research Laboratory, San Jose, California 95114.

The next section contains an application of the program behavior model to a queuing analysis of a simple multiprogrammed paging machine. Some numerical results are given in the final section.

2. Definition of the program behavior model. The program model which we propose is based on the notion of LRU[1] "stack distance," as defined in [7]. There is defined in [7] a class of replacement algorithms called "stack algorithms," which have the following property: Given a replacement algorithm in the class, there corresponds to each reference that a program makes during the course of its execution a "stack," which is a list of the referenced pages of the program. This stack has the property that the contents of memory just after the reference is precisely the first b pages in the stack, b being the (fixed) memory capacity. Several well-known replacement algorithms including LRU are stack algorithms. Given the stacks, one may speak of the "stack distance" of a reference, which is defined to be the position that the referenced page occupies in the stack corresponding to the previous reference. Thus a *reference causes a page exception if and only if its stack distance exceeds the number of pages of the program in memory.* It follows that a sequence of stack distances (to be referred to as a distance string) provides sufficient information in itself to determine (as a function of memory size) when page exceptions occur. For a particular stack algorithm and a reference string (i.e., the time sequence of referenced pages), the corresponding distance string is well-defined. Conversely, it can be shown that given an LRU distance string, the associated referenced string is unique up to naming of the program's pages. For a thorough treatment of these ideas, the reader is referred to [7].

The model of program behavior presented here is an attempt to formalize the notion of "locality" in page-reference strings. By so doing, we hope to provide a parametrised mechanism for generating "program-like" reference strings, exhibiting a broad range of program behavior. We anticipate that apart from analytic studies this will be useful in obtaining a representation of a program load for detailed (simulation) models of computer systems or subsystems.

An intuitive statement of the "locality" of reference with which we deal might be the following: During the course of its execution, there is a subset of a program's pages that is favored in the sense that pages in the subset are more heavily referenced than those outside the subset. It is generally the case that both the contents and the cardinality of this "favored set" change slowly during the course of the execution of the program.

The model of program behavior adopted here rests upon the definition of a certain class of finite-state first order Markov chains for the generation of sequences of positive integers. The reference strings of the model are defined implicitly by interpreting the output of such a Markov chain as a sequence of LRU stack distances.

It is convenient to introduce the Markov chains referred to above by means of a set of directed graphs, each Markov chain being associated with a unique graph in the set. For positive integers n and f such that $1 \leqq f \leqq n$, we define the (n, f) *graph* to be a directed graph having n vertices and $f^2 + 3(n - f) - 1$

[1] LRU is the replacement algorithm that chooses for replacement that page (among those currently in memory) least recently referenced.

directed edges. Denoting the vertices of the graph by $\{1, 2, \cdots, n\}$, the set of directed edges in the graph is defined by (i)–(iv), where $i \to j$ means that there exists a directed edge from vertex i to vertex j.

(i) $i \to j$ and $j \to i$, $1 \leqq i, j \leqq f$;
(ii) $i \to 1$ and $1 \to i$, $f < i \leqq n$;
(iii) $i \to i + 1$, $f < i \leqq n - 1$;
(iv) all the edges in the (n, f)-graph are specified by (i)–(iii).

Note that (i)–(iv) above amount to the assertion of the full graph on the vertex set $\{1, 2, \cdots, f\}$, that each vertex of the set $\{f + 1, f + 2, \cdots, n\}$ is an immediate successor and predecessor of vertex 1, and that the only other immediate successor of vertex $i \in \{f + 1, f + 2, \cdots, n - 1\}$ is $i + 1$. Note also that the exclusion of additional possible edges in an (n, f)-graph represents an approximation to reality.

We interpret the (n, f)-graphs as state-transition diagrams for Markov chains, existence of a directed edge indicating a *positive* one-step transition probability, absence of a directed edge indicating a one-step transition probability equal to zero. When a set of transition probabilities such that the sum of the probabilities out of any vertex equals unity has been specified, we refer to an (n, f)-graph as a *labeled* (n, f) *graph*, and to the corresponding Markov chain as an (n, f)*-chain*. We shall refer to the transition matrix of an (n, f)-chain as the *matrix* of the labeled (n, f)-graph. Thus, for example, a labeled (5, 2)-graph is shown in Fig. 1.

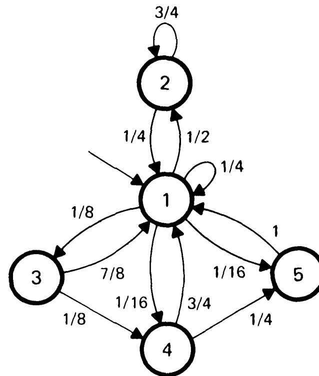


FIG. 1. *A labeled* (5, 2)*-graph*

As indicated above a reference string is the time sequence of page references of a program. More precisely, suppose that with respect to a particular page size, a program has n pages named $a_1, a_2, \cdots, a_n$. Then a *reference string* R of the program is an infinite string over the alphabet $A = \{a_1, a_2, \cdots, a_n\}$. We shall assume an initial LRU stack S_0 to be given and shall consider the (LRU) *distance string* D of the program to be an infinite string over the alphabet $\{1, 2, \cdots, n\}$, according to the definition given in [7]. In the model we obtain reference strings for a program having n pages from an (n, f)-chain by interpreting the states of the Markov chain as LRU stack distances and constructing the reference string associated with a generated distance string as follows. Assuming the initial stack (from "top" to "bottom") to be $a_1, a_2, \cdots, a_n$ and the corres-

ponding initial state in the chain to be state 1, we shall denote a distance string that is generable in the chain by $D = d_1d_2d_3 \cdots$, the sequence of LRU stacks by $S_0S_1S_2 \cdots$, and the elements of the ith stack by $S_i = S_i(1), S_i(2), \cdots, S_i(n)$. Then, by the definition of the stack, the reference string $R = r_1r_2r_3 \cdots$, associated with the distance string D, is the sequence of first elements in the stacks, i.e.,

$$r_i = S_i(1), \qquad i \geqq 1,$$

where the sequence of LRU stacks is defined by

$$S_0 = a_1, a_2, \cdots, a_n,$$

and for $i \geqq 0$,

$$S_{i+1} = \begin{cases} S_i(1), S_i(2), \cdots, S_i(n) = S_i, & \text{if } d_{i+1} = 1, \\ S_i(j), S_i(1), \cdots, S_i(j-1), S_i(j+1), \cdots, S_i(n), & \text{if } d_{i+1} = j, \\ & 1 < j \leqq n-1, \\ S_i(n), S_i(1), \cdots, S_i(n-1), & \text{if } d_{i+1} = n. \end{cases}$$

It is important to note that although the distance strings D have a first order Markov structure (by assumption), the reference strings R do *not* have this structure. Apart from degenerate cases, the reference strings are not of order k for any finite k. In fact, there does not appear to be an explicit characterization of the nature of the dependence in the reference string process. As an example of the reference string construction, consider the (7, 3)-graph of Fig. 2 and the distance string illustrated in Fig. 3 in which we take the alphabet of page names to be $\{a, b, c, d, e, f, g\}$.

As just described, reference strings in the model arise from distance strings generated by the Markov chain of a labeled (n, f)-graph, n being taken as the number of pages in the program. We now wish to indicate an interpretation of the

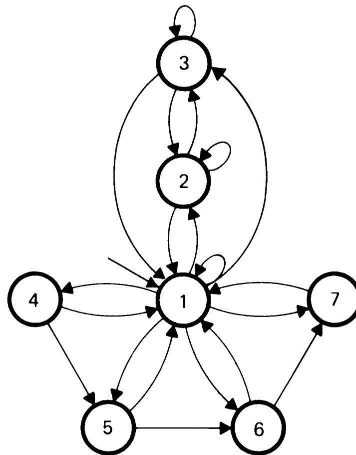


FIG. 2. *The* (7, 3)-*graph*

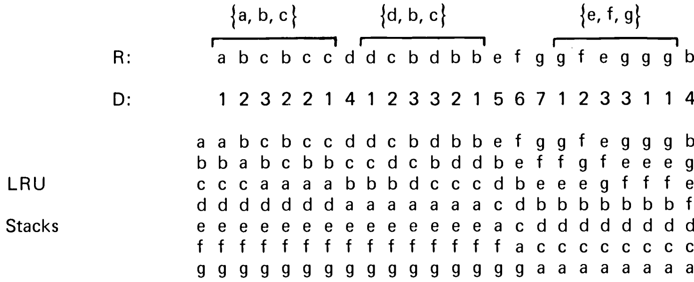


FIG. 3. *Reference string construction*

parameter f and the sense in which reference strings in the model have locality of reference.

We consider f to be the size of a program's "favored set" of pages and consider references corresponding to LRU stack distances $1, 2, \cdots, f$ to be references to the favored set. Similarly, we consider references corresponding to distances greater than f to be references outside the favored set. Thus, given an initial stack, an initial favored set of pages is defined, i.e., the f top-most pages in the stack. Then, assuming an initial distance of 1, there will occur a run of references to the favored set before a reference outside of the favored set occurs. Note that the edge-defining properties (i)–(iii) of the labeled (n, f)-graph permit a reference to any page in the favored set to be followed by a reference to any other page in the favored set.

When, after a random number of references within the favored set, a reference to a page outside the favored set occurs (distance greater than f), a run of references of up to $n - f$ such pages occur before a run of references to pages in the favored set (distances less than or equal to f) again occurs. Note that under our interpretation of favored set, each instance of a distance greater than f during this transition period alters the composition of the set of favored pages, and thus the favored set changes in time. Note also that by the edge-defining properties of the graph, each run of distances greater than f is immediately preceded by, and immediately followed by, a distance equal to 1. Thus, the length of the path of distances taken from distance 1 back to distance 1 through distances greater than f determines the size of the intersection of two successive favored sets. These ideas are illustrated in Fig. 3 where the successive favored sets are indicated in braces and the runs of references in the favored set are delineated by brackets.

Our notion of locality is related qualitatively to the length of runs of references to the favored set, in that we consider reference strings with relatively long runs of references to the favored set to have greater locality of reference than reference strings in which these runs are relatively short.

With respect to this notion of locality, we adopt the following as a measure of the locality of the reference strings generated by a labeled (n, f)-graph.

DEFINITION. Let a labeled (n, f)-graph be given, and denote by F the (random) number of references in a run of references to the favored set. Then the *measure of locality* is defined to be $E(F)$, the expected value of the number of references in a run of references to the favored set.

By this definition the extent to which a labeled (n, f)-graph gives rise to reference strings with locality is determined by the matrix $\mathbf{P}$ of the distance string chain.

We close this section with an observation concerning the page-exception characteristics of reference strings generated in the model. It is clear that the Markov chain of any labeled (n, f)-graph is irreducible and aperiodic. Therefore, if $\mathbf{P} = (p_{ij})$ is the transition matrix of the chain, there exists a unique probability vector $\boldsymbol{\pi} = (\pi_1, \pi_2, \cdots, \pi_n)$ satisfying $\boldsymbol{\pi}\mathbf{P} = \boldsymbol{\pi}$ such that in a realization D, π_i is the long-run fraction of stack distances generated that are equal to i. It follows therefore that in a memory of size b, the long-run fraction of references in a realization which are page exceptions will be $\sum_{i=b+1}^{n} \pi_i$.

3. The semi-Markov process of execution intervals. In this section we derive the structural properties of the sequence of execution intervals, i.e., times between page-exceptions, that result when a reference string in the model is processed under demand paging in a constant number of page-frames, using LRU replacement.

Consider a labeled (n, f)-graph and let $\mathbf{P} = (p_{ij})$ be the transition matrix of the associated (n, f)-chain. Thus, for $1 \leqq i, j \leqq n$,

$$p_{ij} = \Pr\{d_{k+1} = j | d_k = i\}, \qquad k \geqq 1.$$

We shall denote by b $(1 \leqq b < n)$ the number of page-frames of memory in which a reference string $R = r_1 r_2 \cdots$ associated with the (n, f)-chain is processed. Now consider R and define the sequence of epochs $\{t_k\}$, $t_k > t_{k-1}$, as the epochs of (discrete) time at which a page exception occurs, given that the reference string R is processed in b page-frames of memory.

These epochs $\{t_k\}$ define the sequence of execution intervals $I_1, I_2, \cdots$, of the reference string R as follows:

$$\begin{aligned} I_1 &= r_1 r_2 \cdots r_{t_1}, \\ I_2 &= r_{t_1+1} \cdots r_{t_2}, \\ &\vdots \\ I_k &= r_{t_k+1} \cdots r_{t_{k+1}}, \\ &\vdots \end{aligned}$$

Thus an *execution interval* (in memory of size b) is the sequence of references between successive page exceptions, and is specified by a closed interval of the form $[r_{t_k+1}, r_{t_{k+1}}]$ or by the degenerate interval $[r_{t_k}]$.

In the development below it will be useful to keep in mind that since we assume LRU replacement and that the distance strings D are LRU distance strings, references r_k causes a page exception in memory of size b if and only if distance $d_k > b$. It follows that we can consider a notion of the "type" of an execution interval in terms of distances. Thus we make the following definition.

DEFINITION. The execution interval $[r_{t_k+1}, r_{t_{k+1}}]$ is defined to be of *type* $[i; j]$ if $d_{t_k+1} = i$ and $d_{t_{k+1}} = j$. The execution interval $[r_{t_k}]$ is defined to be of *type* $[i]$ if $d_{t_k} = i$.

Thus, for example, in the case of a labeled (5, 2)-graph with $b = 2$ there are five types of execution intervals: $[1;3], [1;4], [1;5], [4], [5]$.

We seek a characterization of $Z_b(t)$, the process of execution intervals in memory of size b. It will be convenient to work with the distance string rather than directly with the reference string. It is easily verified (see, for example, Fabens [8]) that $Z_b(t)$ is a discrete-time finite-state semi-Markov process (SMP) if the state of the process at epoch t is taken to be the type of execution interval in progress at epoch t, as defined above. Note that state changes occur at the epochs $\{t_k\}$ (i.e., whenever a page exception occurs), and the sequence of states of $Z_b(t_k)$ is a Markov chain.

We shall denote the number of states in the semi-Markov process $Z_b(t)$ by N_b, and by a suitable encoding, identify the states with the integers $1, 2, \cdots, N_b$. Then, suppressing the dependence on b, we shall denote by q_{ij} the transition probabilities of the imbedded Markov chain, i.e., for $1 \leqq i, j \leqq N_b$,

$$q_{ij} = \Pr\{Z_b(t_k) = j | Z_b(t_{k-1}) = i\}.$$

The length of time T_{ij} spent by the process $Z_b(t)$ in state i before the next transition, given that the next transition is into state j, is a random variable with a distribution which we denote by $W_{ij}(t)$, i.e.,

$$W_{ij}(t) = \Pr\{T_{ij} \leqq t\}.$$

The mean of this distribution will be denoted by μ_{ij} and the unconditional holding time distribution $W_i(t)$ in state i is defined by

$$W_i(t) = \sum_{j=1}^{N_b} q_{ij} W_{ij}(t)$$

with mean

$$\mu_i = \sum_{j=1}^{N_b} q_{ij} \mu_{ij}.$$

It is important to note that in the SMP of execution intervals, *the holding times T_{ij} are independent of j*, i.e., $W_{ij}(t) = W_i(t)$. Note also that the matrix $\mathbf{Q} = (q_{ij})$ of transition probabilities, and the distributions of the holding times $T_i \equiv T_{ij}$ can be computed from the elements of the matrix $\mathbf{P} = (p_{ij})$ of transition probabilities of the underlying Markov-chain which generates the distance string.

By way of illustration we again consider a labeled (5, 2)-graph with $b = 2$. Then the SMP of execution intervals is a five-state process. Using the encoding,

$$[1;3] \leftrightarrow 1,$$

$$[1;4] \leftrightarrow 2,$$

$$[1;5] \leftrightarrow 3,$$

$$[4] \leftrightarrow 4,$$

$$[5] \leftrightarrow 5,$$

the form of $\mathbf{Q}$ is given by

$$\mathbf{Q} = \begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} & 0 \\ q_{21} & q_{22} & q_{23} & 0 & q_{25} \\ q_{31} & q_{32} & q_{33} & 0 & 0 \\ q_{41} & q_{42} & q_{43} & 0 & q_{45} \\ q_{51} & q_{52} & q_{53} & 0 & 0 \end{bmatrix}.$$

Note that $T_4 = T_5 \equiv 1$, and that $T_i \geqq 2$ for $i = 1, 2, 3$.

Clearly the imbedded chain of the execution interval process is irreducible and thus has a stationary distribution, i.e., there exists a unique probability vector $\boldsymbol{\alpha}$ such that $\boldsymbol{\alpha}\mathbf{Q} = \boldsymbol{\alpha}$.

4. Discussion. It seems likely that, in practice, one would want to specify the size of a program along with particular page-exception characteristics (i.e., fraction of references that are page exceptions as a function of memory size) and then choose values for the remaining parameters in the model so as to be able to generate "program-like" reference strings exhibiting a range of program behavior, but consistent with the specified page-exception characteristics. In this section we give a result which indicates how this can be done.

Given a value for n, the size of the program in pages, we suppose that a vector $\hat{\boldsymbol{\pi}} = (\hat{\pi}_1, \hat{\pi}_2, \cdots, \hat{\pi}_n)$ is specified, $\hat{\pi}_i$ being the fraction of distances that are to be equal to i.

For $f = n$ it is clear that for any vector $\hat{\boldsymbol{\pi}}$ with all $\hat{\pi}_i > 0$ there is an (n, f)-graph whose matrix $\mathbf{P}$ satisfies $\hat{\boldsymbol{\pi}}\mathbf{P} = \hat{\boldsymbol{\pi}}$ (take $p_{ij} = \hat{\pi}_j$, for all i, j). For $f < n$, however, additional constraints on the $\hat{\boldsymbol{\pi}}$-vector are needed to ensure the existence of a labeled (n, f)-graph whose long-run behavior is consistent with $\hat{\boldsymbol{\pi}}$. The following proposition gives a set of sufficient conditions. A proof of the proposition is given in the Appendix.

PROPOSITION 1. *Let n be a positive integer and let $1 \leqq f < n$ be given. Let $\hat{\boldsymbol{\pi}} = (\hat{\pi}_1, \hat{\pi}_2, \cdots, \hat{\pi}_n)$ be a probability vector with all positive components such that*

(i) $\hat{\pi}_1 > \hat{\pi}_j, \qquad f + 1 \leqq j \leqq n,$

(ii) $\dfrac{\hat{\pi}_1}{\hat{\pi}_1 + \cdots + \hat{\pi}_f} > \dfrac{\hat{\pi}_{f+1}}{\hat{\pi}_1},$

(iii) $\dfrac{\hat{\pi}_{f+j} - \hat{\pi}_{f+j-1}}{\hat{\pi}_1} < \dfrac{\hat{\pi}_1}{\hat{\pi}_1 + \cdots + \hat{\pi}_f} - \dfrac{\hat{\pi}_{f+1}}{\hat{\pi}_1}, \qquad 2 \leqq j \leqq n - f.$

Then there exists a labeled (n, f)-graph with matrix $\mathbf{P}$, such that $\hat{\boldsymbol{\pi}}\mathbf{P} = \hat{\boldsymbol{\pi}}$.

The following corollary is easily established.

COROLLARY 1. *Let n be a positive integer and let $\hat{\boldsymbol{\pi}} = (\hat{\pi}_1, \hat{\pi}_2, \cdots, \hat{\pi}_n)$ be a probability vector such that*

(i) $\hat{\pi}_j > 0, \qquad 1 \leqq j \leqq n,$

(ii) $\hat{\pi}_j > \hat{\pi}_{j+1}, \qquad 1 \leqq j \leqq n - 1,$

(iii) $\dfrac{\hat{\pi}_1}{\hat{\pi}_1 + \cdots + \hat{\pi}_{n-j}} > \dfrac{\hat{\pi}_{n-j+1}}{\hat{\pi}_1}, \qquad 1 \leqq j \leqq n - 2.$

Then for all f $(1 \leqq f \leqq n)$ there exists a labeled (n, f)-graph with matrix $\mathbf{P}$ such that $\hat{\boldsymbol{\pi}}\mathbf{P} = \hat{\boldsymbol{\pi}}$.

The following result relates the elements of a matrix of an (n, f)-graph to the measure of locality defined in § 2.

PROPOSITION 2. *Let n be a positive integer and let $1 \leqq f \leqq n$ be given. Let $\hat{\boldsymbol{\pi}} = (\hat{\pi}_1, \cdots, \hat{\pi}_n)$ be a probability vector for which there exists a matrix $\mathbf{P} = (p_{ij})$ of the (n, f)-graph such that $\hat{\boldsymbol{\pi}}\mathbf{P} = \hat{\boldsymbol{\pi}}$. Then*

$$E(F) = \frac{(\hat{\pi}_1 + \hat{\pi}_2 + \cdots + \hat{\pi}_f)}{\hat{\pi}_{f+1} + \hat{\pi}_1(p_{1,f+2} + \cdots + p_{1,n})} + 1.$$

Proof. Consider the semi-Markov process of execution intervals for memory of size f. This process has $2(n - f) - 1$ states corresponding to execution intervals of type $[1, f + 1], [1, f + 2], \cdots, [1, n], [f + 2], \cdots, [n]$. We shall denote the states in this order by $1, 2, \cdots, 2(n - f) - 1$. Consideration of conditional expectations reveals that $E(F)$ satisfies the equation

$$\begin{aligned} E(F) \cdot (\alpha_1 + \alpha_2 + \cdots + \alpha_{n-f}) + 1 \cdot (\alpha_{n-f+1} + \cdots + \alpha_{2(n-f)-1}) \\ = \frac{1}{\hat{\pi}_{f+1} + \cdots + \hat{\pi}_n}, \end{aligned} \tag{1}$$

the right-hand side being the (stationary) mean execution interval in memory of size f, and $\boldsymbol{\alpha} = (\alpha_1, \cdots, \alpha_{2(n-f)-1})$ being the stationary vector of the imbedded Markov chain in the semi-Markov process of execution intervals. Denoting the matrix of this imbedded chain by $\mathbf{Q}$, it follows from the equation $\boldsymbol{\alpha}\mathbf{Q} = \boldsymbol{\alpha}$ that

$$\begin{aligned} \alpha_1 p_{f+1,f+2} &= \alpha_{n-f+1}, \\ \alpha_j p_{f+j,f+j+1} + \alpha_{n-f+j-1} p_{f+j,f+j+1} &= \alpha_{n-f+j}, \qquad 2 \leqq j \leqq n - f - 1. \end{aligned} \tag{2}$$

In addition we have the equations

$$\begin{aligned} \hat{\pi}_{f+1} &= \alpha_1(\hat{\pi}_{f+1} + \hat{\pi}_{f+2} + \cdots + \hat{\pi}_n), \\ \hat{\pi}_{f+j} &= (\alpha_j + \alpha_{n-f+j-1})(\hat{\pi}_{f+1} + \cdots + \hat{\pi}_n), \qquad 2 \leqq j \leqq n - f. \end{aligned} \tag{3}$$

The $2(n - f) - 1$ equations of (2) and (3) imply

$$\begin{aligned} \alpha_1 &= \frac{\hat{\pi}_{f+1}}{\hat{\pi}_{f+1} + \cdots + \hat{\pi}_n}, \\ \alpha_j &= \frac{\hat{\pi}_{f+j}}{\hat{\pi}_{f+1} + \cdots + \hat{\pi}_n} - \frac{\hat{\pi}_{f+j-1} p_{f+j-1,f+j}}{\hat{\pi}_{f+1} + \cdots + \hat{\pi}_n}, \qquad 2 \leqq j \leqq n - f, \\ \alpha_{n-f+j-1} &= \frac{\hat{\pi}_{f+j-1} p_{f+j-1,f+j}}{\hat{\pi}_{f+1} + \cdots + \hat{\pi}_n}, \qquad 2 \leqq j \leqq n - f. \end{aligned} \tag{4}$$

Substituting the values of α_i given by (4) into (1) yields

$$E(F) = \frac{\hat{\pi}_1 + \cdots + \hat{\pi}_f}{(\hat{\pi}_{f+1} + \cdots + \hat{\pi}_n - \hat{\pi}_{f+1} p_{f+1,f+2} - \cdots - \hat{\pi}_{n-1} p_{n-1,n})} + 1.$$

Finally, note that by virtue of $\hat{\pi}\mathbf{P} = \hat{\pi}$, we have

$$\hat{\pi}_1 p_{1,f+j} + \hat{\pi}_{f+j-1} p_{f+j-1,f+j} = \hat{\pi}_{f+j}, \qquad 2 \leqq j \leqq n - f,$$

and the proposition follows.

We close this section by noting an application of the result of Proposition 2. The expression for $E(F)$ given in Proposition 2 reveals that, given a vector π and a matrix $\mathbf{P} = (p_{ij})$ such that $\pi\mathbf{P} = \pi$, the measure of locality $E(F)$ is determined by the sum $p_{1,f+2} + \cdots + p_{1,n}$. In view of this result, given π, an algorithm for determining the elements of matrices $\mathbf{P}$ such that $\pi\mathbf{P} = \pi$ for a range of the measure of locality is derivable directly from the construction given in the proof of Proposition 1.

5. Queuing analysis of a multiprogrammed paging machine. In this section we apply the results of § 3 to the study of certain aspects of memory management in a multiprogrammed paging machine. The essential components of the hardware configuration which we consider are shown in Fig. 4. The main memory contains

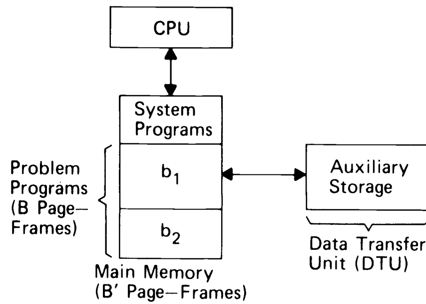


FIG. 4. *System configuration*

B' page-frames and there exists auxiliary storage large enough to store all information needed. We assume that two problem programs are being run in the system. A part of the main memory is used as the residence of system (control) programs. Of the remaining B page-frames of main memory, b_i page-frames are allocated to problem program i. Clearly we want

(i) $b_1 + b_2 = B$ and if n_i is the number of pages in program i, the case of interest is that

(ii) $1 \leqq b_i < n_i$ for $i = 1, 2$.

Under the multiprogramming assumption there is more than one program resident in the main memory, giving rise to contention for processing resources. Hence a conceptual queue is formed for processing services to be provided by the central processor unit (CPU). Whenever a program which is receiving processing service from the CPU references a page which is not in main memory a request for data transfer service is made by the CPU to move the referenced page from auxiliary storage to main memory so as to be available for processing. Having initiated this request the CPU is free to render service to the next available program. Since we have assumed multiprogramming there can be more than one

request wanting the service of data transfer. Thus a second conceptual queue is formed for data transfer services to be provided by the data transfer unit (DTU), which consists of the auxiliary storage and the associated necessary control. As soon as a referenced page is moved from auxiliary storage to main memory, the requesting program (logically) is again available for processing. It is assumed that the CPU can be operated concurrently with the DTU. Thus in multiprogramming mode, the CPU can process one program while the DTU is processing page requests for other programs.

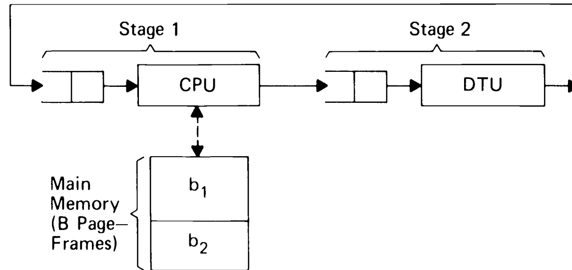


FIG. 5. *Paging machine queuing model*

The model we consider (depicted in Fig. 5) consists of two sequential stages, each stage acting as a single server. The system serves two programs, each of which goes through both stages in sequence and then returns to the first stage, this process being repeated continuously. It is assumed that after completion of CPU service a program moves instantaneously from stage 1 to the queue in stage 2 and after DTU service at that stage back to the queue in stage 1. For $i = 1, 2$ we suppose that program i in the system has a reference string generated by a labeled (n_i, f_i)-graph.

The analysis that we give is in discrete time (reference time) and is under the following assumptions:

(i) Service times at the DTU are independent of service times at the CPU.

(ii) The successive DTU service times are assumed to be independently and identically distributed as a random variable T with finite mean but otherwise arbitrary distribution specified by $f_T(k)$, where

$$f_T(k) = \Pr\{T = k\}, \qquad k \geqq 1,$$

such that $\sum_k kf_T(k) < \infty$. In that which follows we exclude the degenerate case $f_T(1) \equiv 1$, noting that this case can be handled by similar methods.

The question to which we address ourselves is the determination of the CPU utilization for the multiprogrammed system. The analysis which follows draws upon the characterization given in § 3 of the sequence of execution intervals of a program in the system as a finite-state semi-Markov process. We shall denote the number of states in the execution interval SMP of program i by J_i and note that J_i is a function of n_i, f_i, and b_i. Since no confusion will result, we index the states of the two semi-Markov processes by $\{1, 2, \cdots, J_1\}$ and $\{1, 2, \cdots, J_2\}$ respectively.

We shall denote the transition matrices of the execution interval semi-Markov processes by $\mathbf{Q}^{(1)} = (q_{ij}^{(1)})$ $(1 \leqq i,j \leqq J_1)$ and $\mathbf{Q}^{(2)} = (q_{ij}^{(2)})$ $(1 \leqq i,j \leqq J_2)$ respectively. We denote the holding times in state j by $X_j^{(1)}$ $(1 \leqq j \leqq J_1)$ and $X_j^{(2)}$ $(1 \leqq j \leqq J_2)$ respectively and denote their mean values by $\mu_j^{(1)}$ and $\mu_j^{(2)}$. Finally we suppose that when observation of the system begins, program 1 is queued for service at the DTU, program 2 is queued for service at the CPU, and that the state of each of the semi-Markov processes of execution intervals is 1.

The model is analyzed by concentrating on particular epochs $\{\tau_k\}$ as defined below at which changes in the state of the system occur. The system state changes at these particular epochs and the successive times between the changes are a semi-Markov process [8]. This means that the changes in state are generated by a one-step Markov chain with matrix say $\mathbf{R}$, and the times between changes, given the initial and terminal states, are independent of the previous history of the process. The matrix $\mathbf{R}$ is derived from the matrices $\mathbf{Q}^{(1)}$ and $\mathbf{Q}^{(2)}$ of the execution interval processes of the individual programs. A subsequence of the $\{\tau_k\}$ are regeneration points in the process and the mean time between these regeneration points will also be derived. Finally, CPU utilization, that is, the long-run expected fraction of time that the CPU is busy, is obtained from these quantities.

Consider the sequence of times $\{\tau_k\}$, $k = 0, 1, 2, \cdots$, with $\tau_k > \tau_{k-1}$, at which either (i) the CPU is idle, a DTU service has just been completed and the served program has moved to the CPU stage queue, or (ii) the DTU is idle, a CPU service has just been completed and the served program has moved to the DTU stage queue.

It is easily seen that the state of the system at the times $\{\tau_k\}$ constitutes a Markov chain C if the state of the system is defined at the times $\{\tau_k\}$ by the name of the program at the CPU and the state of each of the execution interval processes.

DEFINITION. For $i = 1, 2$ and $j_1 = 1, 2, \cdots, J_1$, and $j_2 = 1, 2, \cdots, J_2$ the system is in state $(i; j_1, j_2)$ at epoch τ_k if program i is at the CPU, the state of the execution interval process of program 1 is j_1, and the state of the execution interval process of program 2 is j_2.

The imbedded chain C is a finite chain having $r = 2J_1J_2$ states. In the sequel for typographical convenience we shall enumerate the states $\{i; j_1, j_2\}$ in lexicographic order. We thus rename the states $1, 2, \cdots, r$, according to

$$\begin{aligned}
(1;1,1) &\leftrightarrow 1,\\
(1;1,2) &\leftrightarrow 2,\\
&\vdots\\
(1;1,J_2) &\leftrightarrow J_2,\\
(1;2,1) &\rightarrow J_2 + 1,\\
&\vdots\\
(2;J_1,J_2) &\leftrightarrow r = 2J_1J_2.
\end{aligned}$$

The matrix $\mathbf{R}$ is of the form

$$\mathbf{R} = \begin{bmatrix} 0 & \mathbf{R}_1 \\ \mathbf{R}_2 & 0 \end{bmatrix},$$

where the matrix $\mathbf{R}_1$ has the block form

$$\mathbf{R}_1 = \begin{bmatrix} \mathbf{D}_{11} & \mathbf{D}_{12} & \cdots & \mathbf{D}_{1J_1} \\ \mathbf{D}_{21} & \mathbf{D}_{22} & \cdots & \mathbf{D}_{2J_1} \\ \vdots & \vdots & & \vdots \\ \mathbf{D}_{J_11} & \mathbf{D}_{J_12} & \cdots & \mathbf{D}_{J_1J_1} \end{bmatrix},$$

the blocks $\mathbf{D}_{ij}$ being $J_2 \times J_2$ diagonal matrices of the form

$$\mathbf{D}_{ij} = \begin{bmatrix} q_{ij}^{(1)} & & 0 \\ & \ddots & \\ 0 & & q_{ij}^{(1)} \end{bmatrix},$$

and the matrix $\mathbf{R}_2$ has the block diagonal form

$$\mathbf{R}_2 = \begin{bmatrix} \mathbf{Q}^{(2)} & & 0 \\ & \ddots & \\ 0 & & \mathbf{Q}^{(2)} \end{bmatrix}.$$

As an example, suppose that program 1 has a reference string generated by a (4, 2)-graph and that program 2 has a reference string generated by a (5, 2)-graph. Suppose further that program 1 executes in 2 page-frames of memory and program 2 executes in 3 page-frames of memory. Thus $n_1 = 4, f_1 = 2, b_1 = 2, n_2 = 5, f_2 = 2, b_2 = 3$. Then $J_1 = 3$ (execution intervals of type $[1;3], [1;4], [4]$) and $J_2 = 3$ (execution intervals of type $[1;4], [1;5], [5]$). Then the matrix $\mathbf{Q}^{(1)}$ is a 3×3 matrix of the form

$$\mathbf{Q}^{(1)} = \begin{bmatrix} q_{11}^{(1)} & q_{12}^{(1)} & q_{13}^{(1)} \\ q_{21}^{(1)} & q_{22}^{(1)} & 0 \\ q_{31}^{(1)} & q_{32}^{(1)} & 0 \end{bmatrix}$$

and the matrix $\mathbf{Q}^{(2)}$ is a 3×3 matrix of the form

$$\mathbf{Q}^{(2)} = \begin{bmatrix} q_{11}^{(2)} & q_{12}^{(2)} & q_{13}^{(1)} \\ q_{21}^{(2)} & q_{22}^{(2)} & 0 \\ q_{31}^{(2)} & q_{32}^{(2)} & 0 \end{bmatrix}.$$

The imbedded chain C has $r = 18$ states and the matrix $\mathbf{R}$ is of the form given in Diag. 1, which displays the block structure given above.

Since the imbedded chain of each of the execution interval processes is irreducible, it follows that the Markov chain C is irreducible, and thus has a stationary distribution, i.e., there exists a unique probability vector $\boldsymbol{\beta} = (\beta_1, \beta_2, \cdots, \beta_r)$ such that $\boldsymbol{\beta}\mathbf{R} = \boldsymbol{\beta}$.

We now consider the semi-Markov process of times between the epochs $\{\tau_k\}$ and the renewal process consisting of returns to state 1 (i.e., $(1;1,1)$). It is easy to check from the assumptions that the time between an epoch τ_k and τ_{k+1} is, given the state of the system at these epochs, a random variable whose distribu-

DIAG. 1

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$1 \leftrightarrow 1; 1, 1$										$q_{11}^{(1)}$	0	0	$q_{12}^{(1)}$	0	0	$q_{13}^{(1)}$	0	0
$2 \leftrightarrow 1; 1, 2$		0			0			0		0	$q_{11}^{(1)}$	0	0	$q_{12}^{(1)}$	0	0	$q_{13}^{(1)}$	0
$3 \leftrightarrow 1; 1, 3$										0	0	$q_{11}^{(1)}$	0	0	$q_{12}^{(1)}$	0	0	$q_{13}^{(1)}$
$4 \leftrightarrow 1; 2, 1$										$q_{21}^{(1)}$	0	0	$q_{22}^{(1)}$	0	0	0	0	0
$5 \leftrightarrow 1; 2, 2$		0			0			0		0	$q_{21}^{(1)}$	0	0	$q_{22}^{(1)}$	0	0	0	0
$6 \leftrightarrow 1; 2, 3$												$q_{21}^{(1)}$	0	0	$q_{22}^{(1)}$	0	0	0
$7 \leftrightarrow 1; 3, 1$										$q_{31}^{(1)}$	0	0	$q_{32}^{(1)}$	0	0	0	0	0
$8 \leftrightarrow 1; 3, 2$		0			0			0		0	$q_{31}^{(1)}$	0	0	$q_{32}^{(1)}$	0	0	0	0
$9 \leftrightarrow 1; 3, 3$										0	0	$q_{31}^{(1)}$	0	0	$q_{32}^{(1)}$	0	0	0
$10 \leftrightarrow 2; 1, 1$	$q_{11}^{(2)}$	$q_{12}^{(2)}$	$q_{13}^{(2)}$															
$11 \leftrightarrow 2; 1, 2$	$q_{21}^{(2)}$	$q_{22}^{(2)}$	0		0			0			0			0			0	
$12 \leftrightarrow 2; 1, 3$	$q_{31}^{(2)}$	$q_{32}^{(2)}$	0															
$13 \leftrightarrow 2; 2, 1$				$q_{11}^{(2)}$	$q_{12}^{(2)}$	$q_{13}^{(2)}$												
$14 \leftrightarrow 2; 2, 2$		0		$q_{21}^{(2)}$	$q_{22}^{(2)}$	0		0			0			0			0	
$15 \leftrightarrow 2; 2, 3$				$q_{31}^{(2)}$	$q_{32}^{(2)}$	0												
$16 \leftrightarrow 2; 3, 1$							$q_{11}^{(2)}$	$q_{12}^{(2)}$	$q_{13}^{(1)}$									
$17 \leftrightarrow 2; 3, 2$		0			0		$q_{21}^{(2)}$	$q_{22}^{(2)}$	0		0			0			0	
$18 \leftrightarrow 2; 3, 3$							$q_{31}^{(2)}$	$q_{32}^{(2)}$	0									

tion does not depend on the times between previous epochs $\tau_l, l \leqq k$, or the state of the system at the epochs $\tau_m, m < k$. Thus this transition process is an r-state semi-Markov process. It is also easy to verify that for all $i = 1, 2, \cdots, r$ the distribution of the holding time in state i, given that the next transition is to state j, is independent of j. Accordingly we denote the mean holding time in state i by λ_i and let $\boldsymbol{\lambda} = (\lambda_1, \lambda_2, \cdots, \lambda_r)$. These mean values are given by

$$\lambda_i = \begin{cases} E\{\max(X_{j_1}^{(1)}, T)\} & \text{if } i \leftrightarrow (1; j_1, j_2) \text{ for all } j_2, \\ E\{\max(X_{j_2}^{(2)}, T)\} & \text{if } i \leftrightarrow (2; j_1, j_2) \text{ for all } j_1. \end{cases} \tag{5}$$

Now consider the subsequence $\{\tau_k'\}$ of the sequence of Markov epochs $\{\tau_k\}$ consisting of the τ_k at which the system enters state 1. These are regeneration points in the process and the times between them which we call $\{Y_k\}$ are independent, identically distributed random variables. A standard mean first passage time calculation in the r-state semi-Markov process yields

$$E(Y) = \frac{1}{\beta_1} \sum_{i=1}^{r} \beta_i \lambda_i. \tag{6}$$

Let us define $U(t)$ to be the total amount of CPU service which occurs in the system during the time interval $(0, t)$, and note that $U(\tau_{k+1}') - U(\tau_k')$, $k = 0, 1, \cdots$, is a sequence of independent, identically distributed (positive) random variables. Moreover, the distributions of these increments between renewal points are computable from the fact that the increments between Markov points form a semi-Markov process. The expected value of the total amount of CPU service between epochs τ_k' and τ_{k+1}' can be found by obtaining the expected amount of this service time between Markov epochs τ_k and τ_{k+1}, given the states at τ_k and τ_{k+1}.

Define random variables $X_1, X_2, \cdots, X_r$ by

$$X_i = \begin{cases} X_{j_1}^{(1)} & \text{if } i \leftrightarrow (1; j_1, j_2) \text{ for all } j_2, \\ X_{j_2}^{(2)} & \text{if } i \leftrightarrow (2; j_1, j_2) \text{ for all } j_1. \end{cases} \tag{7}$$

Then by a first passage decomposition using the equation $\boldsymbol{\beta}\mathbf{R} = \boldsymbol{\beta}$ (cf. [9, pp. 132–133]) it can be shown that

$$E\{U(\tau_{k+1}') - U(\tau_k')\} = \frac{1}{\beta_1} \sum_{i=1}^{r} \beta_i E(X_i). \tag{8}$$

It then follows from (5)–(8) by familiar arguments (see Lewis and Shedler [1]) that

$$\begin{aligned} \text{CPU Utilization} &= \lim_{t \to \infty} \frac{E\{U(t)\}}{t} \\ &= \frac{E\{U(\tau_{k+1}') - U(\tau_k')\}}{E(Y)} \\ &= \frac{\sum_{i=1}^{r} \beta_i E(X_i)}{\sum_{i=1}^{r} \beta_i E\{\max(X_i, T)\}}. \end{aligned} \tag{9}$$

As a final step, we observe that if $\boldsymbol{\alpha}^1 = (\alpha_1^{(1)}, \cdots, \alpha_{J_1}^{(1)})$ satisfies $\boldsymbol{\alpha}^{(1)}\mathbf{Q}^1 = \boldsymbol{\alpha}^{(1)}$ and $\boldsymbol{\alpha}^{(2)} = (\alpha_1^{(2)}, \cdots, \alpha_{J_2}^{(2)})$ satisfies $\boldsymbol{\alpha}^{(2)}\mathbf{Q}^{(2)} = \boldsymbol{\alpha}^{(2)}$, then drawing on the block structure of $\mathbf{R}$ it can be shown that

$$\sum \beta_k = \frac{\alpha_i^{(1)}}{2}, \qquad 1 \leqq i \leqq J_1,$$

where the sum is over all $k \leftrightarrow (1; i, j)$. In addition, it can be shown that

$$\sum \beta_k = \frac{\alpha_j^{(2)}}{2}, \qquad 1 \leqq j \leqq J_2,$$

where the sum is over all $k \leftrightarrow (2; i, j)$.

Therefore by virtue of (7) and (9) we can conclude that

$$\text{CPU Utilization} = \frac{\sum_{j=1}^{J_1} \alpha_j^{(1)} E(X_j^{(1)}) + \sum_{j=1}^{J_2} \alpha_j^{(2)} E(X_j^{(2)})}{\sum_{j=1}^{J_1} \alpha_j^{(1)} E\{\max(X_j^{(1)}, T)\} + \sum_{j=1}^{J_2} \alpha_j^{(2)} E\{\max(X_j^{(2)}, T)\}} \tag{10}$$
$$= \frac{\sum_{j=1}^{J_1} \alpha_j^{(1)} \mu_j^{(1)} + \sum_{j=1}^{J_2} \alpha_j^{(2)} \mu_j^{(2)}}{\sum_{j=1}^{J_1} \alpha_j^{(1)} E\{\max(X_j^{(1)}, T)\} + \sum_{j=1}^{J_2} \alpha_j^{(2)} E\{\max(X_j^{(2)}, T)\}}.$$

6. Numerical results. In this section we present some results of numerical studies concerned with the relationship between locality in page reference strings and the resulting execution interval processes, along with some implications of this relationship for multiprogramming.

Two different $\boldsymbol{\pi}$-vectors have been used in the numerical studies reported in this section, the first being derived from "hit ratios" obtained by LRU stack processing techniques from a trace of the execution of a COBOL program (COBOL), the second being derived similarly from the execution of a sort-merge (SM) program. The two "success functions" ($\pi_1 + \cdots + \pi_b$ vs. b) are displayed in Fig. 6.

Given vector $\boldsymbol{\pi} = (\pi_1, \pi_2, \cdots, \pi_n)$, a value for f and a matrix $\mathbf{P}$ of the (n, f)-graph such that $\boldsymbol{\pi}\mathbf{P} = \boldsymbol{\pi}$, the (stationary) *mean* execution interval in a given amount of memory b is constant (equal to $(\pi_{b+1} + \cdots + \pi_n)^{-1}$), independent of the size of the favored set f or the value of the measure of locality $E(F)$. The *distribution* of the stationary execution interval, of course, is dependent on f and $E(F)$. In Table 1, for the COBOL trace, the variance, coefficient of variation, and coefficient of skewness are displayed for several typical stationary distributions of execution intervals. For each value of f (equal to 2 or 5) the distribution summarized is for memory size equal to f. In each case, a matrix $\mathbf{P}$ was obtained according to the construction of Proposition 1 in § 4. It can be shown for such a choice of the matrix $\mathbf{P}$ that the distribution is determined by the value of the measure of locality.

From Table 1, it appears that for fixed f, the variance of execution intervals increases as the measure of locality increases. The stationary distributions of

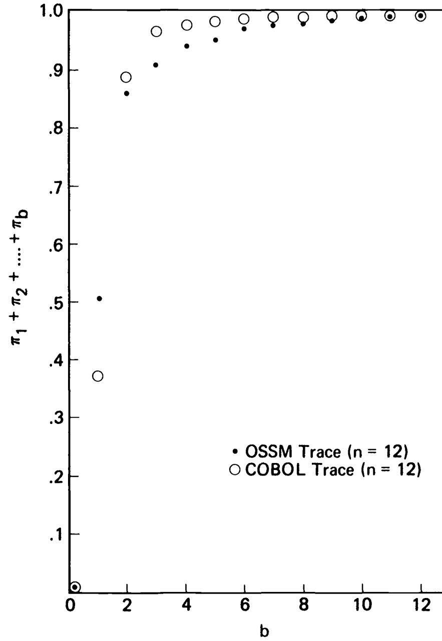


FIG. 6. *"Hit ratio" curves*

execution intervals are positively skewed, being longer tailed as the measure of locality increases. Note also that in every case shown but one, the coefficient of variation is at least equal to one, with somewhat greater coefficients of variation observed over the range of the measure of locality for the larger value of f.

Taking CPU utilization as a measure of the effectiveness of multiprogramming, the queuing analysis of §5 makes it possible to begin to assess the

TABLE 1
Stationary distribution of execution intervals X_f in memory size f (COBOL trace)

f	$E(F)$	$\mu = E(X_f)$	$\sigma^2 = E((X_f - \mu)^2)$	$C = \frac{\sigma^2}{\mu^2}$	$\gamma = \frac{E((X_f - \mu)^3)}{\sigma^3}$
2	9.02	9.02	78.97	0.972	2.11
2	9.23	9.02	82.34	1.01	2.12
2	10.33	9.02	99.95	1.23	2.18
2	11.73	9.02	122.41	1.51	2.30
2	12.06	9.02	127.73	1.57	2.33
2	12.20	9.02	128.43	1.58	2.33
2	12.29	9.02	129.11	1.59	2.34
5	78.41	78.41	6168.00	1.00	1.99
5	81.27	78.41	6610.00	1.08	2.00
5	97.26	78.41	9082.00	1.48	2.11
5	120.5	78.41	12682.00	2.06	2.33
5	126.63	78.41	13621.00	2.22	2.40
5	127.44	78.41	13745.00	2.24	2.41
5	128.28	78.41	13870.00	2.26	2.41

TABLE 2
Values of CPU utilization with multiprogramming, identical programs — SM trace, equal partitions
DTU service times constant = 10

f	$E(F)$	Total Amount of Memory B										
		2	4	6	8	10	12	14	16	18	20	22
2	7.43	.204	.612	.758	.858	.905	.954	.977	.994	.998	.999	1.000
2	9.75	.204	.579	.710	.796	.853	.909	.952	.982	.991	.996	1.000
2	14.52	.204	.538	.658	.735	.804	.867	.929	.971	.984	.994	1.000
2	16.42	.204	.527	.645	.721	.793	.857	.924	.968	.982	.993	1.000
5	21.36	.204	.619	.767	.866	.903	.953	.976	.994	.998	.999	1.000
5	27.92	.204	.609	.742	.823	.851	.908	.952	.982	.991	.996	1.000
5	39.32	.204	.598	.716	.781	.800	.865	.928	.970	.984	.994	1.000
5	44.87	.204	.594	.710	.771	.789	.855	.923	.968	.983	.993	1.000
8	118.03	.204	.615	.764	.870	.914	.962	.980	.994	.998	.999	1.000
8	143.16	.204	.613	.761	.864	.907	.953	.970	.982	.991	.996	1.000
8	180.96	.204	.612	.757	.858	.900	.944	.959	.970	.984	.994	1.000
8	192.40	.204	.611	.757	.857	.898	.941	.957	.967	.983	.993	1.000

implications for multiprogramming of the observed relationship between locality and skewness of the execution interval distributions. Tables 2–4 display values of CPU utilization with multiprogramming for the case of (statistically) identical SM programs, under a memory management policy of equal partitions. In Tables 5–7 corresponding values are given for the case of (statistically) identical COBOL programs. In each case reported, the DTU service times were taken to be constant.

These numerical studies suggest that lower CPU utilization results from a set of programs having a greater measure of locality than from a set of programs (with the same success functions) but a smaller measure of locality. Note also

TABLE 3
Values of CPU utilization with multiprogramming, identical programs — SM trace, equal partitions
DTU service times constant = 100

f	$E(F)$	Total Amount of Memory B										
		2	4	6	8	10	12	14	16	18	20	22
2	7.43	.020	.071	.106	.158	.203	.319	.442	.766	.933	.977	.997
2	9.75	.020	.071	.106	.158	.202	.313	.433	.728	.885	.952	.997
2	14.52	.020	.071	.106	.157	.200	.303	.420	.691	.839	.928	.997
2	16.42	.020	.071	.106	.156	.199	.299	.417	.682	.829	.923	.997
5	21.36	.020	.071	.106	.158	.203	.319	.442	.765	.933	.977	.997
5	27.92	.020	.071	.106	.158	.202	.313	.432	.728	.885	.952	.997
5	39.32	.020	.071	.106	.158	.200	.302	.419	.690	.839	.928	.997
5	44.87	.020	.071	.106	.158	.199	.298	.416	.681	.829	.923	.997
8	118.03	.020	.071	.106	.158	.203	.319	.444	.765	.933	.977	.997
8	143.16	.020	.071	.106	.158	.203	.319	.440	.727	.885	.952	.997
8	180.96	.020	.071	.106	.158	.203	.318	.436	.688	.838	.928	.997
8	192.40	.020	.071	.106	.158	.203	.318	.434	.679	.828	.923	.997

TABLE 4
Values of CPU utilization with multiprogramming, identical programs — SM trace, equal partitions
DTU service times constant = 1000

f	$E(F)$	Total Amount of Memory B										
		2	4	6	8	10	12	14	16	18	20	22
2	7.43	.002	.007	.011	.016	.020	.032	.047	.114	.271	.470	.805
2	9.75	.002	.007	.011	.016	.020	.032	.047	.114	.268	.458	.805
2	14.52	.002	.007	.011	.016	.020	.032	.047	.114	.262	.443	.806
2	16.42	.002	.007	.011	.016	.020	.032	.047	.114	.260	.439	.807
5	21.36	.002	.007	.011	.016	.020	.032	.047	.114	.271	.470	.805
5	27.92	.002	.007	.011	.016	.020	.032	.047	.114	.268	.458	.805
5	39.32	.002	.007	.011	.016	.020	.032	.047	.114	.262	.443	.806
5	44.87	.002	.007	.011	.016	.020	.032	.047	.114	.260	.439	.806
8	118.03	.002	.007	.011	.016	.020	.032	.047	.114	.271	.470	.805
8	143.16	.002	.007	.011	.016	.020	.032	.047	.114	.268	.458	.805
8	180.96	.002	.007	.011	.016	.020	.032	.047	.114	.262	.442	.806
8	192.40	.002	.007	.011	.016	.020	.032	.047	.114	.260	.438	.806

that for large values of DTU service time and equal partitions, values of CPU utilization show little sensitivity to change in f.

The skewness of the distributions of execution intervals that has been observed suggests that unequal partitions could give rise to a wide range of values of CPU utilization, and in particular to values of CPU utilization substantially higher than those realized by equal partitions. That this does in fact occur is illustrated by the graphs of Figs. 7 and 8, in which, for a fixed (total) amount of memory B, the maximum and minimum (over all partitions of memory of size B) values of CPU utilization are plotted as functions of B. Additional data, not repro-

TABLE 5
Values of CPU utilization with multiprogramming, identical programs — COBOL trace, equal partitions
DTU service times constant = 10

f	$E(F)$	Total Amount of Memory B										
		2	4	6	8	10	12	14	16	18	20	22
2	9.23	.158	.694	.948	.975	.989	.995	.996	.998	.999	.999	1.000
2	10.33	.158	.677	.903	.942	.974	.981	.985	.991	.994	.997	1.000
2	11.73	.158	.659	.859	.909	.959	.968	.975	.983	.990	.995	1.000
2	12.11	.158	.654	.849	.902	.956	.965	.972	.982	.989	.995	1.000
5	81.26	.158	.709	.959	.981	.989	.995	.996	.998	.999	.999	1.000
5	97.25	.158	.705	.947	.966	.973	.981	.985	.991	.994	.997	1.000
5	120.54	.158	.702	.934	.952	.958	.968	.974	.983	.990	.995	1.000
5	127.42	.158	.701	.932	.949	.955	.965	.972	.982	.989	.995	1.000
8	304.26	.158	.706	.959	.982	.992	.997	.997	.998	.999	.999	1.000
8	409.01	.158	.705	.954	.976	.985	.990	.990	.991	.994	.997	1.000
8	617.09	.158	.703	.949	.970	.979	.983	.983	.983	.990	.995	1.000
8	696.96	.158	.703	.948	.969	.977	.981	.982	.982	.989	.995	1.000

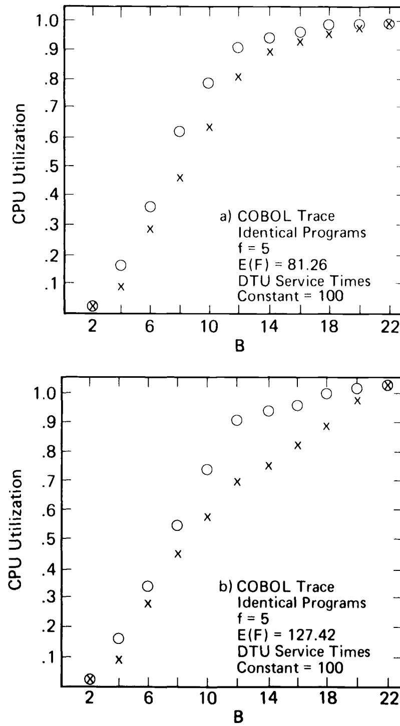


FIG. 7. *Extreme values of CPU utilization (COBOL)*

duced here, suggests that this range of CPU utilization is influenced by the value of f, smaller values of f giving rise generally to a greater range of CPU utilization.

TABLE 6
Values of CPU utilization with multiprogramming, identical programs — COBOL trace, equal partitions DTU service times constant = 100

f	$E(F)$	Total Amount of Memory B										
		2	4	6	8	10	12	14	16	18	20	22
2	9.23	.016	.090	.308	.466	.638	.861	.893	.937	.967	.985	.996
2	10.33	.016	.090	.303	.447	.614	.790	.828	.885	.931	.964	.996
2	11.73	.016	.090	.293	.420	.587	.722	.767	.836	.896	.945	.996
2	12.11	.016	.090	.290	.412	.581	.707	.754	.826	.888	.940	.997
5	81.26	.016	.090	.309	.468	.638	.861	.892	.937	.967	.985	.996
5	97.25	.016	.090	.308	.461	.612	.790	.828	.885	.931	.964	.996
5	120.54	0.16	.090	.307	.454	.585	.722	.767	.836	.986	.945	.996
5	127.42	.016	.090	.307	.452	.578	.706	.753	.825	.888	.940	.996
8	304.26	.016	.090	.309	.469	.643	.870	.898	.937	.967	.985	.996
8	409.01	.016	.090	.309	.466	.633	.834	.856	.885	.931	.964	.996
8	617.09	.016	.090	.308	.464	.622	.798	.816	.836	.896	.944	.996
8	696.96	.016	.090	.308	.463	.620	.791	.807	.825	.888	.940	.996

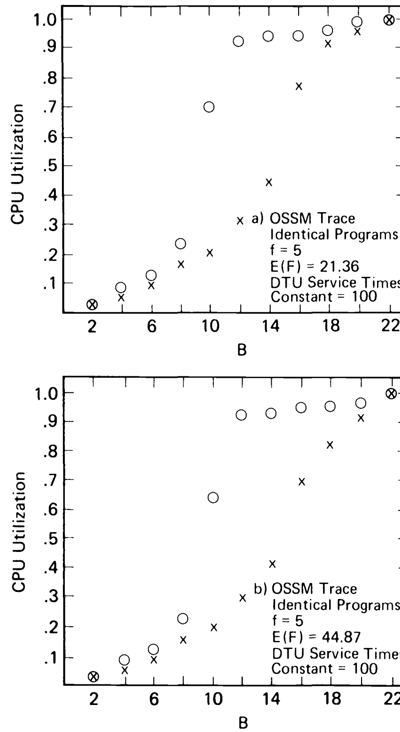


FIG. 8. *Extreme values of CPU utilization (SM)*

As a final remark, we note that the observed skewness of execution interval distributions suggests that the program model of this paper would predict values of CPU utilization in multiprogramming that are somewhat *lower* than those

TABLE 7
Values of CPU utilization with multiprogramming, identical programs — COBOL trace, equal partitions DTU service times constant = 1000

f	*E*(*F*)	Total Amount of Memory B										
		2	4	6	8	10	12	14	16	18	20	22
2	9.23	.002	.009	.031	.050	.078	.172	.203	.285	.408	.576	.769
2	10.33	.002	.009	.031	.050	.078	.172	.202	.281	.397	.554	.769
2	11.73	.002	.009	.031	.050	.078	.169	.199	.272	.380	.530	.769
2	12.11	.002	.009	.031	.050	.078	.167	.197	.270	.376	.524	.769
5	81.26	.002	.009	.031	.050	.078	.172	.203	.285	.408	.576	.768
5	97.25	.002	.009	.031	.050	.078	.172	.202	.281	.397	.554	.769
5	120.54	.002	.009	.031	.050	.078	.169	.199	.272	.380	.530	.769
5	127.42	.002	.009	.031	.050	.078	.167	.197	.269	.376	.524	.769
8	304.26	.002	.009	.031	.050	.078	.172	.203	.284	.408	.576	.768
8	409.01	.002	.009	.031	.050	.078	.172	.203	.281	.397	.554	.769
8	617.09	.002	.009	.031	.050	.078	.172	.202	.272	.380	.530	.769
8	696.96	.002	.009	.031	.050	.078	.172	.202	.269	.375	.524	.769

resulting from the assumption of geometric (memoryless) distributions of execution intervals. This has been verified numerically for the cases reported in this paper.

Appendix. Proof of Proposition 1. Any matrix $\mathbf{P} = (p_{ij})$ of a labeled (n,f)-graph is a stochastic matrix of the form

$$\mathbf{P} = \begin{bmatrix} \mathbf{P}_1 & \mathbf{P}_2 \\ \mathbf{P}_3 & \mathbf{P}_4 \end{bmatrix},$$

where the $f \times f$ matrix

$$\mathbf{P}_1 = \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1f} \\ p_{21} & p_{22} & \cdots & p_{2f} \\ \vdots & \vdots & & \vdots \\ p_{f1} & p_{f2} & \cdots & p_{ff} \end{bmatrix},$$

the $f \times (n-f)$ matrix

$$\mathbf{P}_2 = \begin{bmatrix} p_{1,f+1} & p_{1,f+2} & \cdots & p_{1n} \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix},$$

the $(n-f) \times f$ matrix

$$\mathbf{P}_3 = \begin{bmatrix} p_{f+1,1} & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ p_{n-1,1} & 0 & \cdots & \cdot \\ 1 & 0 & \cdots & 0 \end{bmatrix},$$

and the $(n-f) \times (n-f)$ matrix

$$\mathbf{P}_4 = \begin{bmatrix} 0 & p_{f+1,f+2} & 0 & \cdots & 0 \\ 0 & 0 & p_{f+2,f+3} & & \vdots \\ \vdots & & & \ddots & 0 \\ 0 & & & & p_{n-1,n} \\ 0 & 0 & \cdots & 0 & 0 \end{bmatrix},$$

where the indicated p_{ij} are all positive. Note that if $\mathbf{P}$ is to satisfy $\hat{\pi}\mathbf{P} = \hat{\pi}$, then

$$p_{1,f+1} = \hat{\pi}_{f+1}/\hat{\pi}_1, \tag{A.1}$$

and that if $p_{1,f+j}$ is chosen $(2 \leqq j \leqq n-f)$, then

$$p_{f+j-1,f+j} = \frac{\hat{\pi}_{f+j} - \hat{\pi}_1 p_{1,f+j}}{\hat{\pi}_{f+j-1}} \tag{A.2}$$

and

$$p_{f+j-1,1} = 1 - \left(\frac{\hat{\pi}_{f+j} - \hat{\pi}_1 p_{1,f+j}}{\hat{\pi}_{f+j-1}}\right)$$

are determined.

Thus it will be sufficient to show that there exist positive $p_{1,f+j} (2 \leqq j \leqq n - f)$ such that

$$0 < \frac{\hat{\pi}_{f+j} - \hat{\pi}_1 p_{1,f+j}}{\hat{\pi}_{f+j-1}} < 1 \tag{A.3}$$

and that there exist positive p_{ij} $(1 \leqq i, j \leqq f)$ such that

$$\begin{aligned} \sum_{j=1}^{f} p_{ij} &= 1, & 2 \leqq i \leqq f, \\ \sum_{i=1}^{f} \hat{\pi}_i p_{ij} &= \hat{\pi}_j, & 2 \leqq j \leqq f, \\ \sum_{j=1}^{f} p_{1j} &= 1 - \left(\frac{\hat{\pi}_{f+1}}{\hat{\pi}_1} + p_{1,f+2} + \cdots + p_{1,n}\right), & \\ \sum_{i=1}^{f} \hat{\pi}_i p_{i1} &= \hat{\pi}_1 - \hat{\pi}_{f+1} p_{f+1,1} - \hat{\pi}_{f+2} p_{f+2,1} - \cdots - \hat{\pi}_n p_{n,1}. & \end{aligned} \tag{A.4}$$

It is easily verified that a solution to (4) is given by

$$\begin{aligned} p_{11} &= 1 - \left(\frac{\hat{\pi}_2 + \cdots + \hat{\pi}_f}{\hat{\pi}_1 + \hat{\pi}_2 + \cdots + \hat{\pi}_f}\right) - \left(\frac{\hat{\pi}_{f+1}}{\hat{\pi}_1} + p_{1,f+2} + \cdots + p_{1,n}\right), & \\ p_{i1} &= \frac{\hat{\pi}_1}{\hat{\pi}_1 + \hat{\pi}_2 + \cdots + \hat{\pi}_f}, & 2 \leqq i \leqq f, \\ p_{ij} &= \frac{\hat{\pi}_j}{\hat{\pi}_1 + \hat{\pi}_2 + \cdots + \hat{\pi}_f}, & 1 \leqq i \leqq f, \ 2 \leqq j \leqq f, \end{aligned}$$

if we ensure that $p_{11} > 0$, i.e.,

$$1 - \left(\frac{\hat{\pi}_2 + \cdots + \hat{\pi}_f}{\hat{\pi}_1 + \hat{\pi}_2 + \cdots + \hat{\pi}_f}\right) - \left(\frac{\hat{\pi}_{f+1}}{\hat{\pi}_1} + p_{1,f+2} + \cdots + p_{1,n}\right) > 0, \tag{A.5}$$

and thus we must find positive $p_{1,f+j}$ $(2 \leqq j \leqq n - f)$ satisfying (3) and (5). Therefore, using condition (i) of the hypothesis, we obtain a matrix $\mathbf{P}$ as required if for $2 \leqq j \leqq n - f$ we choose $p_{1,f+j}$ such that

$$\max\left(0, \frac{\hat{\pi}_{f+j} - \hat{\pi}_{f+j-1}}{\hat{\pi}_1}\right) < p_{1,f+j} < \frac{\hat{\pi}_{f+j}}{\hat{\pi}_1} \tag{A.6}$$

and

$$p_{1,f+2} + \cdots + p_{1,n} < 1 - \left(\frac{\hat{\pi}_2 + \cdots + \hat{\pi}_f}{\hat{\pi}_1 + \cdots + \hat{\pi}_f}\right) - \frac{\hat{\pi}_{f+1}}{\hat{\pi}_1} = \frac{\hat{\pi}_1}{\hat{\pi}_1 + \cdots + \hat{\pi}_f} - \frac{\hat{\pi}_{f+1}}{\hat{\pi}_1}.$$

Conditions (ii) and (iii) of the hypothesis ensure that the two inequalities in (6) are consistent.

Acknowledgment. The authors are indebted to J-P. Jacob for several stimulating conversations during the course of this work. The method of calculating CPU utilization given in § 4 is based on an observation about cyclic queues by J. Gecsei.

REFERENCES

[1] 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.

[2] 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.

[3] ———, *Approximate models for multiprogramming computer systems*, 5th Annual IEEE Computer Conference, Boston, Sept. 1971.

[4] P. J. DENNING, Y. C. CHEN AND G. S. SHEDLER, *A model for program behavior under demand paging*, IBM Res. Rep. RC–2301, Yorktown Heights, N.Y., 1968.

[5] A. V. AHO, P. J. DENNING AND J. D. ULLMAN, *Principles of optimal page replacement*, J. Assoc. Comput. Mach., 18 (1971), pp. 80–93.

[6] P. H. ODEN AND G. S. SHEDLER, *A model of memory contention in a paging machine*, Comm. ACM, 15 (1972).

[7] R. L. MATTSON, J. GECSEI, D. R. SLUTZ AND I. L. TRAIGER, *Evaluation techniques for storage hierarchies*, IBM Systems J., 8 (1970), pp. 78–117.

[8] A. J. FABENS, *The solution of queueing and inventory models by semi-Markov processes*, J. Roy. Statist. Soc. Ser. B, 23 (1961), pp. 113–127.

[9] R. E. BARLOW AND F. PROSCHAN, *Mathematical Theory of Reliability*, John Wiley, New York, 1967.

SIAM J. COMPUT.
Vol. 1, No. 3, September 1972

PROGRAM SCHEMES WITH PUSHDOWN STORES*

STEVEN BROWN, DAVID GRIES AND THOMAS SZYMANSKI†

Abstract. We attempt to characterize classes of schemes allowing pushdown stores, building on an earlier work by Constable and Gries [1]. We study the effect (on the computational power) of allowing one, two, or more pushdown stores, both with and without the ability to detect when a pds is empty. A main result is that using one pds is computationally equivalent to allowing recursive functions.

We also study the effect of adding the ability to do integer arithmetic, and multidimensional arrays.

Key words. Program schemes, schemata, pushdown stores, stacks, recursion, programming languages.

1. Introduction. In Constable and Gries [1] the following classes of schemes were defined:

P = class of schemes using simple variables, with assignment, conditional, goto and while statements.

P_A = class of schemes P, with the additional feature of arrays of subscripted variables.

P_{Ae} = class of schemes P_A, with the additional feature of an equality test on subscript values.

P_R = class of schemes P, with the additional feature of ALGOL-like recursive procedures.

P_M = class of schemes P, with the additional feature of a finite number of distinguishable markers, or constants, allowed as values. (There may appear arbitrarily many instances of a marker.)

P_{pds} = class of schemes P, with the additional feature of pushdown stores.

$P_{\mathbb{N}}$ = class of schemes P, with the additional feature of integer arithmetic.

One could then build other classes. For example, P_{AM} is the class of schemes allowing arrays and markers. In particular, $P_{(m,n)}$ refers to the class of schemes allowing m pushdown stores and n markers.

In a sense, a scheme is an abstraction of a program, and by studying these classes of schemes we gain more understanding of the computational power of the different data structures and control mechanisms used in programming languages. A large part of a recent paper by Constable and Gries [1] was devoted to showing the following inclusions and equivalences, where, for example, $P < P_R$ means that for every scheme in P there exists an equivalent scheme in P_R but not conversely; and $P_{Ae} \equiv P_{AM}$ means that for every scheme in P_{Ae} there exists an equivalent scheme in P_{AM}, and conversely:

$$P < P_R \leqq P_{(1,0)} < P_A \equiv P_{Ae} \equiv P_{AM} \equiv P_{(2,1)} \equiv P_{(1,0)\mathbb{N}}.$$

Hence you can "do more" with arrays than you can with recursive procedures. It was claimed that P_{Ae} and equivalent classes are "universal." All the above inclusions and equivalences are effective, except for $P_A \equiv P_{Ae}$; for any scheme

* Received by the editors May 17, 1972.

† Department of Computer Science, Cornell University, Ithaca, New York 14850. This research was supported by the National Science Foundation under Grant GJ-28176.

$S \in P_{Ae}$ an equivalent scheme $S' \in P_A$ exists, but it cannot in general be constructed! In [1] it is assumed that all the basic functions and predicates are total.

This paper resolves some questions left open in [1], and discusses some more inclusions and equivalences of classes of schemes, mostly having to do with pushdown stores. Our results can be best given by the inclusion diagram of Fig. 1.

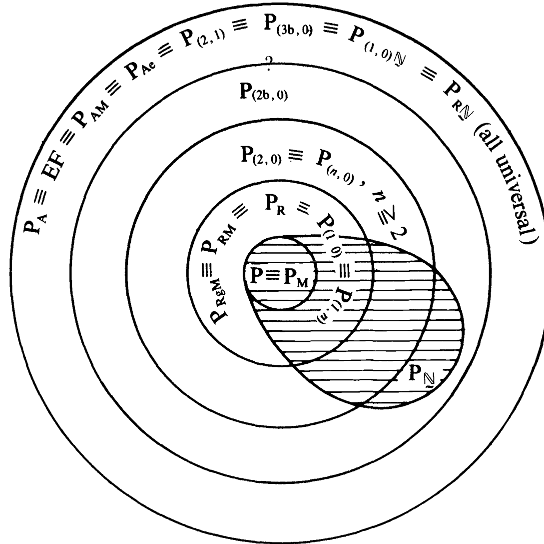


FIG. 1. *Inclusion diagram for classes of schemata*

Two new classes of schemes appear in the figure. P_{Rg} is the class of schemes P_R allowing the additional feature of global variables (as used in ALGOL). P_{pdsb} is the class of schemes P_{pds} with the additional feature of a test for the bottom of a pushdown store. (In P_{pds} execution of a pop instruction has absolutely no effect if the stack is empty.) Thus $P_{(2b,0)}$ allows 2 pushdown stores, tests on emptiness of these pds's, and no markers.

The question mark on the line above $P_{(2b,0)}$ indicates an unsolved problem; we do not know whether

$$P_{(2b,0)} < P_{(3b,0)} \quad \text{or} \quad P_{(2b,0)} \equiv P_{(3b,0)}.$$

The inclusion diagram brings out some interesting points. Oddly enough, adding the feature of markers adds nothing to the power of many classes; we have

$$P \equiv P_M, \quad P_R \equiv P_{RM}, \quad P_{(1,0)} \equiv P_{(1,n)} \quad \text{for } n \geqq 0, \quad \text{and} \quad P_A \equiv P_{AM}.$$

Only when adding markers to $P_{(2,0)}$ do we add computational power, and then only one marker is needed to achieve "universality."

Adding the ability to do integer arithmetic, however, has more of an effect on the computational power. Thus, adding integer arithmetic to P_R or $P_{(1,0)}$ yields the "universal" class of schemes $P_{R\mathbb{N}}$ or $P_{(1,0)\mathbb{N}}$. Of special interest in the diagram is $P_{\mathbb{N}}$. Note how it "contains a piece" of each of the other classes. According to Corollary 10.9 of [1], the characteristic property of this class is the following:

Let S be any scheme in any class. Then there exists an equivalent scheme $S' \in P_{\mathbb{N}}$ if and only if there is a bound n and an equivalent effective functional in which each expression and proposition can be evaluated using at most n variables. Thus the characteristic property is that the scheme really needs only a fixed, bounded number of variables, if it can internally perform integer arithmetic.

In [1], the pushdown store in P_{pds} was formulated so that a pop is a null operation if the pds is empty. This was done solely because it was the "cleanest" and easiest definition to work with. It is interesting to note that being able to test for the bottom of a pds is computationally important. Thus we have $P_{(2,0)} < P_{(2b,0)}$. Of course $P_{(1,0)} \equiv P_{(1b,0)}$, since $P_{(1,0)} \equiv P_{(1,n)}$ and we can simulate the test for the bottom of a stack by using a marker. Note also that $P_{(3b,0)}$ is universal and thus equivalent to $P_{(2,1)}$.

This paper is organized as follows. We assume the reader is familiar with [1] and refer to all the definitions and results given there, without repeating them here. The rest of this section is devoted to a few other necessary definitions and comments.

Section 2 discusses the equivalence of $P_{(1,0)}$ with P_R. This means that the data structure of a single stack is equivalent to the control mechanism of recursive procedures. In § 3 we relate $P_{(1,0)}$ to $P_{(2,0)}$ and $P_{(n,0)}$, and $P_{(n,0)}$ to P_{Ae} for $n > 2$. Section 4 discusses the use of the statement which tests for the emptiness of a pds, and relates classes $P_{(nb,0)}$ for $n \geqq 3$, with $P_{(2b,0)}$ and $P_{(2,0)}$.

In § 5 we show how $P_{\mathbb{N}}$ fits in. In the final section we solve another open problem of [1]; we show that adding multidimensional arrays to P_A adds no more computational power. All equivalences and inclusions shown in this paper are effective.

(1.1) DEFINITION. A scheme in the class P_{Rg} (Recursive functions allowing global variables) is a scheme in P_R (see [1, Def. 3.4]) with the following change: the function definition may also have the form

$$\langle\text{function def}\rangle ::= f(v_1, \cdots, v_{Rf})\ \mathbf{global}\ w\{, w\}; \langle\text{body}\rangle.$$

The *global variables* w_i in the statement "**global** $w_1, \cdots, w_n$" may not appear in the formal parameter list $v_1, \cdots, v_{Rf}$. $w_1, \cdots, w_n$ refer to the variables with the same names (if any) used in the main ⟨body⟩ of the programs and they are *not* initialized to Ω upon invocation of the function ⟨body⟩. Note that if two ⟨function def⟩s declare the same name to be global, then the names refer to the same variable.

(1.2) DEFINITION. A scheme in the class P_{pdsb} (or $P_{(1b,0)}, P_{(2b,0)}, \cdots$) is a scheme in the class P_{pds} (or $P_{(1,0)}, P_{(2,0)}, \cdots$) (see [1, Def. 4.7] and [1, § 7]), with the following additional statement type allowed:

$$\langle S\rangle ::= \mathbf{if}\ EMPTYPDS(s)\ \mathbf{then}\ [l:]\langle S\rangle_1\ \mathbf{else}\ [l:]\langle S\rangle_2$$

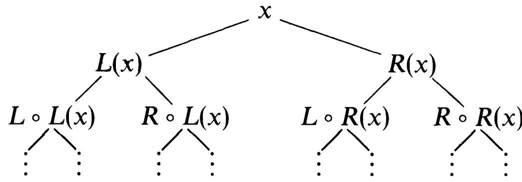
where s is a pushdown store.

We next define a functional which will be used frequently in this paper. Let

$$Leaftest(P, L, R, x): \mathscr{P}(D) \times \mathscr{F}(D) \times \mathscr{F}(D) \times D \to D$$

$$(1.3) \qquad = \begin{cases} x & \text{if there exists a sequence } f_1, f_2, \cdots, f_n \text{ where each } f_i \\ & \text{is either } L \text{ or } R \text{ and} \\ & P(f_n \circ f_{n-1} \circ \cdots \circ f_1(x)) = \mathbf{true}; \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Informally, *Leaftest* is a search performed on the following binary tree:



Leaftest searches this tree in an attempt to find a node whose value makes the predicate P true. If such a node is found, *Leaftest* returns x as its output value; otherwise, the search continues forever.

Leaftest has been an important functional in the brief history of "comparative schematology." Paterson and Hewitt [3] first used it as a scheme which could not be performed in $\mathrm{P_R}$. Gries and Constable [1] then gave a scheme in $\mathrm{P_A}$ for it, to help show that $\mathrm{P_R} < \mathrm{P_A}$. In this paper, *Leaftest* or variations of it are used to prove the inclusions

$$\mathrm{P}_{(1,0)} < \mathrm{P}_{(n,0)} \quad \text{for } n > 1, \quad \mathrm{P}_{(n,0)} < \mathrm{P_A}, \quad \mathrm{P}_{(n,0)} < \mathrm{P}_{(2b,0)}.$$

We shall also make use of "locators" in several proofs.

(1.4) DEFINITION. Given a scheme S (in any class), a *locator* S' for S is a scheme with the following properties:

(i) S and S' use the same input variables, basic functions and predicates.

(ii) When executing, S' attempts to find a predicate P_i of rank RP_i and two lists of argument values a_1 and a_2 such that $P_i(a_1) = \mathbf{true}$, $P_i(a_2) = \mathbf{false}$. If it finds them, S' puts the values of a_1 into variables $RT_1, \cdots, RT_{RP_i}$, puts the values of a_2 into variables $RF_1, \cdots, RF_{RP_i}$, and transfers control to a statement

$$BEGIN_i\colon \mathbf{halt}\ (OMEGA).$$

(iii) If S' does not find a predicate as in (ii), then

(a) if S executes infinitely long, then so does S';

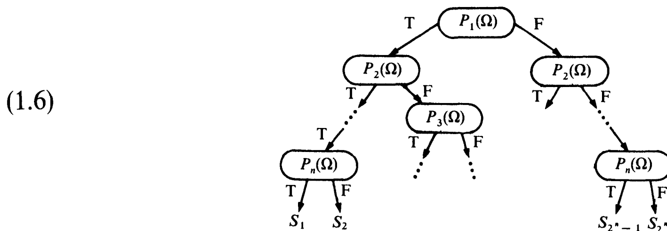
(b) if S halts with value V, then so does S'.

The chief use of a locator is in the construction of a scheme S' without markers equivalent to a scheme S which uses markers. Once the predicate is "located" as described in the definition above, the markers of S can be "simulated" in S' using a sequence of the argument lists a_1, a_2 as bits (see [1, Def. 5.1]). A main result which we shall use is the following rewording of Theorem 5.5 of [1].

(1.5) THEOREM. *Let S be a scheme in some class. Let S' in class P_2 be a locator for S. Suppose there exist P-simulators* (*see* [1, *Def.* 5.1]) *for S in P_2. The locator and P-simulators can be put together to form a scheme S'' in P_2 equivalent to S.*

Proof. Assume without loss of generality that the predicates $P_1, \cdots, P_n$ of a scheme S all have rank 1. Then the locator we construct has the following form:

(1.6)



where the S_i are statements. S_1 for example must do the following:

(1.7) S_1 must "simulate" the $v = (\mathbf{true}, \cdots, \mathbf{true})$-autonomous behavior of S until either

(i) it halts and outputs the same result that S would, or

(ii) a predicate P_i is evaluated with argument a_2 such that $P_i(a_2) = \mathbf{false}$. At this point RT is initialized to Ω, RF is set to a_2, and control is transferred to $BEGIN_i$.

This is a *very* brief introduction to locators and simulators, and the reader is encouraged to review § 5 and § 9 of [1].

Throughout the rest of this paper, all manipulations of pushdown stores will be written using the following notation:

PUSH(pd, V) when executed, places the value currently stored in the variable *V* on the top of the stack *pd*.

POP(pd, V) when executed, removes the top value from the stack *pd* and assigns it to the variable *V*. If the stack *pd* is empty when this statement is executed, then the operation is treated as a null operation.

2. The equivalence of P_R and $P_{(1,0)}$. Theorem 7.5 of [1] showed that $P_R \leqq P_{(1,0)}$. Here we prove that $P_{(1,0)} \leqq P_R$, yielding the equivalence of P_R and $P_{(1,0)}$. Hence a single stack is just as computationally powerful as recursive procedures. The proof is a series of lemmas establishing the following inclusions, in order:

$$P_{(1,n)} \leqq P_{RgM} \leqq P_{Rg} \leqq P_R \leqq P_{(1,0)} \quad \text{for } n \geqq 0. \tag{2.1}$$

An obvious by-product is that neither global variables nor markers add anything to the power of recursive procedures (P_R). A look at the proof of $P_{RgM} \leqq P_{Rg}$ (Theorem 2.3) will also convince the reader that $P_M \leqq P$ and thus $P_M \equiv P$.

Suppose we have a scheme $S \in P_R$. We can translate S into an equivalent scheme $S1 \in P_{(1,0)}$, then translate $S1$ into $S2 \in P_{RgM}$, into $S3 \in P_{Rg}$, and finally into $S4 \in P_R$, again. You will note by the constructions of the lemmas that $S4$ uses only *one* recursive procedure definition. Hence, for any scheme $S \in P_R$ which uses $n > 1$ recursive procedures we can construct an equivalent scheme $S4$ in P_R which uses only *one* recursive procedure.

Another interesting point concerns the class $P_{(1,0)}$. Given any scheme $S \in P_{(1,0)}$ we can construct an equivalent scheme $S1 \in P_{(1,0)}$ such that if $S1$ halts, its pds is empty. This is quite remarkable since in $P_{(1,0)}$ one cannot test to see if the pds is empty. This fact comes out easily from the constructions in the lemmas involved.

(2.2) LEMMA. $P_{(1,n)} \leqq P_{RgM}$ *for* $n \geqq 0$.

Proof. Given a scheme $S \in P_{(1,n)}$ which uses a single pds P, we construct an equivalent scheme $S2 \in P_{RgM}$. The basic idea is to define a function F which is essentially the same as the main scheme. The pds P becomes a simple variable P which is a formal parameter of F, and the pds is represented by the "stack" of invocations of F. Except for a second formal parameter, all other variables are global to F. This is illustrated in Fig. 2. When $S \in P_{(1,n)}$ executes the statement $PUSH(P, v)$ at $\langle S \rangle_1$, the scheme $S2$ executes a call of F, with the value of V as the argument. The main problem is that F should begin executing *not* at the first

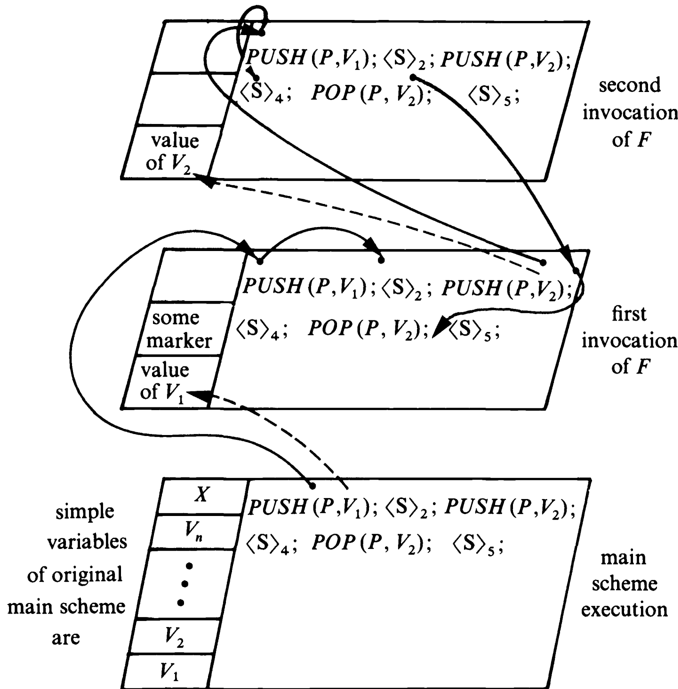


FIG. 2. *Representing a* pds *by function calls*

statement, but at statement $\langle S\rangle_2$ (see Fig. 2). We do this by passing a marker as a second argument to F to indicate where it should begin executing.

Similarly, a pop instruction $POP(P, w)$ is essentially a return instruction. Again, we must make sure that the calling invocation of F does not begin executing after its call (which was a push), but at the statement after this pop ($\langle S\rangle_5$ in Fig. 2). To do this, the value returned by F is also a special marker.

Normal **halts** in F and pops in the main scheme must be handled similarly. We leave the details to the Appendix.

(2.3) THEOREM. $\mathrm{P}_{\mathrm{RgM}} \leqq \mathrm{P}_{\mathrm{Rg}}$.

Proof. We first show in Lemma 2.4 that we can construct P-simulators $\in \mathrm{P}_{\mathrm{Rg}}$ for any scheme $\in \mathrm{P}_{\mathrm{RgM}}$ (see [1, Def. 5.1]). According to Theorem 1.5, we then need only show that for $S \in \mathrm{P}_{\mathrm{RgM}}$ we can construct a locator $\in \mathrm{P}_{\mathrm{Rg}}$. In Lemma 2.6 we establish the decidability of the finiteness of the v-autonomous behavior of any scheme $\in \mathrm{P}_{\mathrm{RgM}}$ (see [1, § 9]). This important fact, and the method of the decision, are used in Lemma 2.7 to build a locator $\in$ P (and thus $\in \mathrm{P}_{\mathrm{Rg}}$) for any scheme $\in \mathrm{P}_{\mathrm{RgM}}$.

(2.4) LEMMA. *Let* $S \in \mathrm{P}_{\mathrm{RgM}}$ *use a predicate* P. *Then we can construct a* P-*simulator* $S1 \in \mathrm{P}_{\mathrm{Rg}}$.

Proof. We proceed essentially as in the proof of Theorem 5.3 of [1]. Assume without loss of generality that P has rank 1, and that $P(RT) =$ **true**, $P(RF) =$ **false**, where RT contains the value rt and RT contains rf.

Suppose S uses markers $M_1, M_2, \cdots, M_k$. Each variable v of S is represented in $S1$ by variables $v, v^1, \cdots, v^k$. The following table indicates the correspondence

between values stored in v during execution of S, and in $v, v^1, \cdots, v^k$ during execution of $S1$:

variable v in S	*variables $v, v^1, \cdots, v^k$ in $S1$*
$\bar{v} \in D$	$\bar{v}, rf, \cdots, rf$
M_1	$\Omega, rt, rf, \cdots, rf$
$\vdots$	$\vdots$
M_k	$\Omega, rf, \cdots, rf, rt$

We leave it to the reader to show how to translate the statements $v \leftarrow f(\cdots)$ (where f is a basic function), $v \leftarrow w$, $v \leftarrow M_i$, **if** $p(\cdots)$ **then** $\cdots$, and **if** $v = M_i$ **then** $\cdots$, of S into equivalent statements for $S1$. The main problem is with calls and returns of recursive functions.

Each function definition $f(v_1, \cdots, v_n): \cdots$ is transformed into $f(v_1, v_1^1, \cdots, v_1^k, \cdots, v_n, v_n^1, \cdots, v_n^k): \cdots$, so that the parameters get passed properly. For a call of a recursive function

$$w \leftarrow f(v_1, \cdots, v_n) \tag{2.5}$$

in S, however, we must return values not only for w, but also for $w^1, \cdots, w^k$. These will be returned in new global variables $x^1, \cdots, x^k$. Add them to the list of global variables in each function definition. Now change each call (2.5) to

$$\textbf{begin} \quad w \leftarrow f(v_1, v_1^1, \cdots, v_1^k, \cdots, v_n, v_n^1, \cdots, v_n^k);$$
$$w^1 \leftarrow x^1; \cdots; w^k \leftarrow x^k;$$
$$\textbf{end}$$

and change each **halt**(v) within a function definition to

$$\textbf{begin} \quad x^1 \leftarrow v^1; \cdots; x^k \leftarrow v^k; \textbf{halt}(v)\ \textbf{end}$$

Q.E.D.

(2.6) LEMMA. *It is decidable whether the v-autonomous behavior of a scheme in P_{RgM} is finite or infinite.*

Proof. We can assume that the global variables of S are $V_1, \cdots, V_g$ and that by suitable renaming of variables, they are *not* used as local variables or formal parameters. Assume that S is completely labeled. Let r be the largest of the ranks of the recursive functions of S, and let S use markers $M_1, \cdots, M_{m-1}$. Let S use predicates $P_1, \cdots, P_n$.

Consider the v-autonomous behavior of S, described in [1, (9.9)]. This behavior does not depend on the input values, or on which value of the domain D is in any variable at any point. Using $\bar{v}$ to denote *any* value in D, the m possible values that can affect the behavior at some point are $\bar{v}, M_1, \cdots, M_{m-1}$.

If the v-autonomous behavior is infinite, then one of the two following things must happen: (i) the level of nesting of function invocations is infinite; or (ii) within the execution of a function (or main program), there must be an infinite loop. We now derive bounds on the nesting of function invocations and the

number of statements executed within a function which, if reached, indicate there is infinite behavior.

Suppose there is a call $v \leftarrow f(\cdots)$ of a recursive function. The behavior of the scheme while f is executing depends only on the *values* of the actual parameters and of the global variables $V_1, \cdots, V_g$. Hence there are at most

$$m(r + g) \quad \text{possible different behaviors.}$$

Thus, if a recursive function f is called recursively $m(r + g) + 1$ times (without returning), two calls on f have already occurred with the same actual parameter and global variable values $(\bar{v}, M_1, \cdots, M_{m-1})$. Neither of these two calls will finish and the scheme is in an infinite loop.

Second, consider the v-autonomous behavior within a recursive function f (of the main scheme). Suppose f has s statements and l local variables (including the formal parameters). Then we know the recursive function has infinite v-autonomous behavior if the behavior has as many as $s \cdot r \cdot (l + m) + 1$ labels in it. Q.E.D.

(2.7) LEMMA. *Every scheme S in P_{RgM} has a locator S' in P.*

Proof. The locator S' for S has the form shown in (1.6). We need only show how to construct the statements $S_1, \cdots, S_{2^n}$ described there. We outline in the Appendix the construction of S_1 only which simulates the v-autonomous behavior of S where $v = (\mathbf{true}, \cdots, \mathbf{true})$, as described in (1.7). The construction of the other S_i is similar. The important point to note is that we can effectively decide whether the v-autonomous behavior of S is finite or infinite (Lemma 2.6). Q.E.D.

(2.8) LEMMA. $P_{Rg} \leqq P_R$.

Proof. Suppose scheme $S \in P_{Rg}$ has function definitions for functions $F_1, \cdots, F_n$, and suppose that the variables used globally are $V_1, \cdots, V_m$. By suitably renaming the local variables we can make sure that $V_1, \cdots, V_m$ are used *only* as global variables, and we can assume S has the form

$$
\begin{array}{ll}
(v, \cdots, v)\colon \langle S\rangle; \cdots; \langle S\rangle & \\
F_1(v, \cdots, v)\colon \mathbf{global}\ V_1, \cdots, V_m; & \langle S\rangle; \cdots; \langle S\rangle \\
\vdots \qquad\qquad \vdots & \\
F_n(v, \cdots, v)\colon \mathbf{global}\ V_1, \cdots, V_m; & \langle S\rangle; \cdots; \langle S\rangle
\end{array}
\tag{2.9}
$$

We give in the Appendix a construction which reduces by one the number of global variables. By executing this construction m times, we arrive at an equivalent scheme in P_{RM}. What this construction does is make V_1 a parameter of each function. This creates the problem that we cannot return the value of V_1, so what we do first is call F_1 (say) to get the function value back, and then call a similar routine F'_1 which returns the value for V_1. Q.E.D.

3. Markerless pds schemes. In this section we show that

$$P_A > P_{(n,0)} \equiv P_{(2,0)} > P_{(1,0)} \quad \text{for } n \geqq 2.$$

The proper inclusions are both proved using the *Leaftest* scheme or a variation of it.

A second important idea is proved in Lemma 3.2; for any scheme $S \in P_{(n,0)}$ we can construct a locator in P. We use this to show the following result.

(3.1) THEOREM. $P_{(n,0)} \equiv P_{(2,0)}$ *for* $n \geqq 2$.

Proof. Lemma 3.2 shows how to construct a locator in P for $S \in P_{(n,0)}$; because of Theorem 1.5 we need only show how to construct P-simulators in $P_{(2,0)}$ for S. Consider S to be in P_{pdsM} rather than P_{pds} and use Theorem 7.3 of [1] to construct $S1 \in P_{(2,1)}$ equivalent to S.

We construct a simulator $S2 \in P_{(2,0)}$ for $S1$ (and thus for S) by simulating the single marker. We represent each simple variable V of S1 by variables V and V' and initially set each V' to rf.

To produce the P-simulator we make a copy S' of $S1$ and change it as follows (we assume without loss of generality that all predicates have rank 1):

(a) At the beginning of S' insert for every simple variable V the statement $V' \leftarrow RF$.

(b) For the pds's $PD1$ and $PD2$ add at the beginning of S'

$$PUSH(PD1, RF); \qquad PUSH(PD2, RF);$$

to indicate they are empty.

(c) Change each $PUSH(PDj, V)$ (except those inserted in (b)) to

$$\textbf{begin } PUSH(PDj, V); \qquad PUSH(PDj, V'); \qquad PUSH(PDj, RT) \textbf{ end}$$

(d) Change each $POP(PDj, V)$ to

begin $POP(PDj, X)$;
  if $P(X)$ **then**
    begin $POP(PDj, V')$; $POP(PDj, V)$ **end**
  else $PUSH(PDj, X)$
end

where X is a new temporary variable. This construction allows a pop of an empty pds to be treated as a null operation.

(e) Change each assignment $V \leftarrow W$ to

$$\textbf{begin } V \leftarrow W; \qquad V' \leftarrow W' \textbf{ end}$$

(f) Change each assignment $V \leftarrow M$ to

$$\textbf{begin } V \leftarrow OMEGA; \qquad V' \leftarrow RT \textbf{ end}$$

(g) Change each assignment $V \leftarrow f(\cdots)$ to

$$\textbf{begin } V \leftarrow f(\cdots); \qquad V' \leftarrow RF \textbf{ end}$$

(h) Change each test

$$\textbf{if } V = M \textbf{ then } \langle S_1 \rangle \textbf{ else } \langle S_2 \rangle$$

to

$$\textbf{if } P(V') \textbf{ then } \langle S_1 \rangle \textbf{ else } \langle S_2 \rangle$$

It should be clear from the construction that the modified S' runs in $P_{(2,0)}$ and simulates the behavior of S exactly. Q.E.D.

(3.2) LEMMA. *For any scheme* $S \in \mathrm{P}_{(n,0)}$ *we can construct a locator* $S' \in \mathrm{P}$.

Proof. The locator has the form given in (1.6). We show how to construct only statement S_1 of (1.6) as described in (1.7).

Assume that S has $|S|$ statements. We first show that under autonomous behavior the scheme references at most the top $|S|$ locations of any pds. With constant predicates, S executes l (say) statements, $l \leqq |S|$, and then halts (hence at most l locations of any pds can be referenced), or executes l different statements and then enters an infinite loop, where the loop consists of $r \leqq |S|$ statements.

If a pds has a net growth during execution of the r statements of the loop, then no element lower than $|S|/2$ from the top can be referenced. On the other hand, if a pds shrinks in size or remains the same during one execution of the loop, then the stack size is at most $l + r/2 \leqq |S|$.

We now show how to construct S_1. We generate $(|S| + 1)^n$ different copies of S (changing the labels so the copies are independent). Let the copies be denoted by $S'_{i_1 i_2 \cdots i_n}$, where each i_j denotes the number of occupied positions in simulated stack j. Clearly the initial "state" is $S'_{000\cdots 0}$. We shall assign new labels to every statement in every copy; the labels will be $l^j_{i_1 i_2 i_3 \cdots i_n}$, where $j = 1, \cdots, |S|$, and the i_m are keyed to the copy.

The copies are then altered and connected in the following way. Consider the pushdown stack m.

(a) In all copies $S'_{i_1 \cdots i_{m-1}, 0, i_{m+1} \cdots i_n}$, all statements popping stack m are replaced by the null statement.

(b) In all copies $S'_{i_1 \cdots i_m \cdots i_n}$, where $i_m < |S|$, after each *PUSH* statement labeled $l^j_{i_1 \cdots i_m \cdots i_n}$ for stack m we insert

$$\textbf{go to } l^{j+1}_{i_1 \cdots i_{m+1} \cdots i_n}$$

(c) In all copies $S'_{i_1 \cdots i_m \cdots i_n}$, $i_m > 0$, after each stack m *POP* statement labeled $l^j_{i_1 \cdots i_m \cdots i_n}$ we insert

$$\textbf{go to } l^{j+1}_{i_1 \cdots i_{m-1} \cdots i_n}$$

Most of this complexity is to guarantee that a null operation is performed if an empty stack is popped.

Assume now without loss of generality that all predicates are monadic and that we have $P_i(RT) = \textbf{true}$ for each predicate P_i. (We are creating S_1 of (1.6) only, now.) We represent each pds p by new simple variables $V_{p,1}, \cdots, V_{p,|S|}$. We modify all $PUSH(p, w)$ statements and all $POP(p, w)$ statements (in all copies of S) as follows:

(a) Change $PUSH(p, w)$ to

$$\textbf{begin } V_{p,|S|} \leftarrow V_{p,|S|-1}; \cdots ; V_{p,2} \leftarrow V_{p,1}; V_{p,1} \leftarrow W \textbf{ end}$$

(b) Change $POP(p, w)$ to

$$\textbf{begin } W \leftarrow V_{p,1}; V_{p,1} \leftarrow V_{p,2}; \cdots ; V_{p,|S|-1} \leftarrow V_{p,|S|} \textbf{ end}$$

We also replace each statement

$$\textbf{if } P_i(X) \textbf{ then } \langle \mathrm{S}_1 \rangle \textbf{ else } \langle \mathrm{S}_2 \rangle$$

by

if $P_i(X)$ **then** $\langle S_1 \rangle$

else begin $RF \leftarrow X$; **go to** $BEGIN_i$ **end**

and add statements

$BEGIN_i$: **halt** $(OMEGA)$;

at the end of the scheme. The result of these transformations is statement S_1. Q.E.D.

(3.3) LEMMA. *Leaftest cannot be computed in* $P_{(n,0)}$.

Proof. Suppose $S \in P_{(n,0)}$ computes *Leaftest* (P, L, R, X). Now consider the following scheme S':

$S'(P, L, R, X)$: $V \leftarrow X$;

if $P(X)$ **then go to** $BEGIN_1$;

Locator (S);

$BEGIN_1$: **halt**(V);

The notation "Locator (S)" refers to the body of the locator scheme for S constructed according to Lemma 3.2. Control is passed to the label $BEGIN_1$ by the locator only if Locator (S) has generated some value for which P is **true**, since P is the only predicate in S which can potentially take on both true and false values. By the construction of S' the only new values which S' can generate are concatenated applications of the functions L and R applied to the initial value X. By definition these are just node values in the binary tree generated by X, L and R. Hence, control is passed to $BEGIN_1$ only if a value is found for which P is **true**. It should be equally clear that if there is any value in the tree which makes P **true**, Locator (S) by hypothesis will eventually find it and will transfer control to $BEGIN_1$.

We must also consider the possibility that Locator (S) will stop on the value Ω, which can arise in several situations, according to the definitions of schemata behavior (see [1]). We can eliminate this case by observing that the value of the *Leaftest* functional is by definition independent of the truth value of $P(\Omega)$. Since Locator (S) is a P-scheme (Lemma 2.1), it has only a finite number of variables, and we can modify Locator (S) so as to keep track of which locations contain the value Ω. This is done by keeping many copies of the scheme, such that each copy corresponds to particular variables $V_1, \cdots, V_t$ containing Ω and all other variables containing computed values. By this means, therefore, we can force a false branch whenever $P(\Omega)$ is tested. Such a locator clearly performs the same locator tasks as the original one.

After having taken care of the Ω problem as above, we see that S' is equivalent to S. Referring once again to Lemma 2.1, we note that since the modified Locator (S) is a P-scheme, S' is also a P-scheme. But S must still be able to compute *Leaftest* in its full generality, and we therefore would have a P-scheme S' which computes *Leaftest*. But this contradicts the result of [4] in which it is shown that *Leaftest* cannot be computed in P_R (and hence not in P). Thus S could not have existed and *Leaftest* is not computable in $P_{(n,0)}$. Q.E.D.

(3.4) THEOREM. $P_{(n,0)} < P_A$.

Proof. Consider $S \in P_{(n,0)}$ to be in P_{pdsM}. By Theorem 8.2 of [1] we can construct an equivalent scheme $S1 \in P_{AM}$, and by Theorem 5.4 of [1] we can construct P-simulators for it in P_A. Second, by Lemma 3.2, we can construct a Locator in P (and hence in P_A) for S. We then apply Theorem 1.5.

Theorem 6.6 of [1] and Lemma 3.3 show that the containment is proper. Q.E.D.

(3.5) THEOREM. $P_{(1,0)} < P_{(2,0)}$.

Proof. Clearly $P_{(1,0)} \leqq P_{(2,0)}$; to show that the containment is proper we exhibit a function computable in $P_{(2,0)}$ but not in $P_{(1,0)}$. Consider the functional $f(P, L, R, X, Y, Z)$:

if $P(Y)$ *and* $\neg P(Z)$ **then** *Leaftest* (P, L, R, X) **else** X

First we show how to compute the above functional in $P_{(2,0)}$. Clearly we can write *Leaftest*(P, L, R, X) as a scheme in $P_{(2,1)}$ since we can do it in P_A and $P_A \equiv P_{(2,1)}$. Lemma 3.1 shows how to construct a P-simulator for *Leaftest* in $P_{(2,0)}$. The following scheme in $P_{(2,0)}$ then computes the above functional:

(X, Y, Z): **if** $P(Y)$ *and* $\neg P(Z)$ **then**

begin $RT \leftarrow Y$; $RF \leftarrow Z$;

{P-simulator in $P_{(2,0)}$ for *Leaftest*}

end

else halt (X);

Suppose now we have a scheme $S(P, L, R, X, Y, Z) \in P_{(1,0)}$ which computes the above functional f. From it we construct a scheme $S'(P, L, R, X) \in P_{(1,m)}$ which computes *Leaftest*(P, L, R, X). Since $S' \in P_{(1,m)} \equiv P_{(1,0)} \equiv P_R$ and *Leaftest*(P, L, R, X) cannot be performed in P_R ([1, Thm. 6.6]) we have a contradiction to the fact that a scheme to compute f existed in $P_{(1,0)}$.

To construct $S'(P, L, R, X)$ perform the following. Let M_1, M_2 be two markers, and let W be a new variable. Insert at the beginning of S the statements

$$Y \leftarrow M_1; \qquad Z \leftarrow M_2;$$

Then change each conditional

if $P(V)$ **then** S_1 **else** S_2

of S to

if $V = M_1$ **then** S_1

else if $V = M_2$ **then** S_2

else if $P(V)$ **then** S_1 **else** S_2

We must show that $S'(P, L, R, X) = $ *Leaftest*(P, L, R, X) for all domains D and all interpretations of P, L, R, and X. For any interpretation, consider the domain $D' = D \cup \{M_1, M_2\}$ (where $D \cap \{M_1, M_2\} = \varnothing$), predicate P', and functions

F', L', where

$$P'(d) = P(d) \quad \text{for } d \in D, \quad P'(M_1) = \textbf{true}, \quad P'(M_2) = \textbf{false},$$
$$L'(d) = L(d) \quad \text{for } d \in D, \quad L'(M_1) = M_1, \quad L'(M_2) = M_2,$$
$$R'(d) = R(d) \quad \text{for } d \in D, \quad R'(M_1) = M_1, \quad R'(M_2) = M_2.$$

A look at S and S' will show that

$$S(P', L', R', X, M_1, M_2) = S'(P, L, R, X) \quad \text{for } X \in D.$$

But, by definition of f we have $S(P', L', R', X) = S'(P', L', R', X) = \mathit{Leaftest}(P', L', R', X) = \mathit{Leaftest}(P, L, R, X)$ for $X \in D$. Q.E.D.

4. Bottom markers and pds's. A major drawback to programming in $P_{(n,0)}$ is the inability to locate the bottom of a pushdown store. This makes it impossible to perform such useful tasks as transferring the contents of one pds into another while perhaps performing some action on each value as it goes by. However, every "real" programming language incorporating stacks or pds's also contains primitives which allow either the trapping of an interrupt on pds underflow or else explicit testing for empty pds's.

Accordingly, we extend the class $P_{(n,0)}$ by adding to the language the construct

$$\textbf{if } \mathit{EMPTYPDS}(pd) \textbf{ then } \langle S_1 \rangle \textbf{ else } \langle S_2 \rangle$$

The semantics of this statement should be obvious. This new class will be called $P_{(nb,0)}$, where the b is intended to remind the reader that we now have the ability to find the *bottom* of the pds.

Intuitively, the ability to test for the bottom of a pds is less powerful than the ability to place markers in it. Classes utilizing markers are allowed an unbounded number of copies of the markers which can occur anywhere, whereas marking the bottom of each pds is equivalent to using only a fixed number of copies of each marker and requiring that the markers always appear in a certain relative position.

We prove in this section the expected result that $P_{(1b,0)} \equiv P_{(1,0)}$. We also show that $P_{(3b,0)}$ is effectively equivalent to the "universal" classes P_{AM}, P_{Ae} and $P_{(2,1)}$. As far as $P_{(2b,0)}$ is concerned, we show that

$$P_{(2b,0)} > P_{(2,0)} > P_{(1b,0)} \equiv P_{(1,0)}.$$

However, we do not know whether $P_{(2b,0)}$ is equivalent to $P_{(3b,0)}$ or not. This open problem will be discussed at the end of the section.

(4.1) LEMMA. $P_{(nb,0)} \leqq P_{(n,1)}$ *for any* $n \geqq 1$.

Proof. Given a scheme S in $P_{(nb,0)}$, we must construct an equivalent scheme S' in $P_{(n,1)}$. S' uses a marker M. We insert at the beginning of S statements to push M onto each pds. Next we replace all tests for an empty pds by tests for M at the top of the pds. This also requires changes in *POP* statements. We leave the details to the reader. Q.E.D.

(4.2) THEOREM. $P_{(1b,0)} \equiv P_{(1,1)} \equiv P_{(1,0)}$.

Proof. Clearly $P_{(1,0)} \leqq P_{(1b,0)}$. By Lemma 4.1 we know that $P_{(1b,0)} \leqq P_{(1,1)}$. By § 2, $P_{(1,1)} \leqq P_{(1,0)}$. Q.E.D.

(4.3) THEOREM. $P_{(nb,0)} \leqq P_{AM}$.

Proof. By Lemma 4.1 we have $P_{(nb,0)} \leqq P_{(n,1)}$ and by [1, Thm. 8.2], $P_{(n,1)} \leqq P_{AM}$. Q.E.D.

We are now ready to discuss the "universality" of $P_{(3b,0)}$. We do this in two parts. First of all, we show that for any scheme S in P_{Ae} there exists a scheme S' in $P_{(1,0)\underset{\sim}{N}}$ which does *not* store integer values on the stack. This result arises easily from some results in [1] concerning effective functionals and program schemes. Secondly, Lemma 4.5 will show how to construct a scheme in $P_{(3b,0)}$ equivalent to S'.

(4.4) LEMMA. *For any scheme S in P_{Ae} there exists an equivalent scheme S' in $P_{(1,0)\underset{\sim}{N}}$ which does not store integer values on its pds.*

Proof. Assertion 10.6 of [1] says that there exists an *effective functional F* equivalent to S (see [1, Def. 10.3]); Assertion 10.8 of [1] then states that there exists a scheme S' in $P_{(1,0)\underset{\sim}{N}}$ equivalent to F and thus equivalent to S. In the constructions in Assertions 10.6 and 10.8 of [1], S' has the form given in Fig. 3.

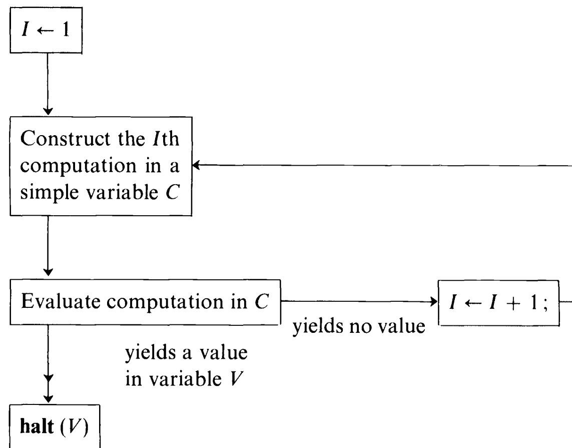


FIG. 3

This scheme S' satisfies the desired property; the pds is used only to hold temporary values in D occurring during the evaluation of computation C. Each computation is constructed in Polish postfix form encoded as an integer in a single variable, and uses only a finite number of simple variables. Q.E.D.

(4.5) LEMMA. *Let S be a scheme in $P_{(1,0)\underset{\sim}{N}}$ which never stores an integer value on its pds PD. Then we can find an equivalent scheme S' in $P_{(3b,0)}$.*

Proof. In addition to pds PD, S' uses two pds's $PD1$ and $PD2$ as counters to simulate the contents of the integer variables and arithmetic in the finite control of S. We first modify the scheme S so that its set of variables can be partitioned into a set $\{X_1, \cdots, X_k\}$ which is used only for manipulating domain values and a set $\{V_1, \cdots, V_m\}$ which is used only for holding integer values. This modification can easily be made by "splitting" each variable of the original scheme into two copies and adding some states to the finite control of S. The X_i and V_i work together in that whenever the simulated variable to which an (X_i, V_i) pair corresponds contains a domain element, X_i contains the element and $V_i = 0$; whenever the simulated variable contains an integer, $X_i = \Omega$ and V_i contains the integer.

At any point in simulated time, the height of pds $PD1$ of S' will be

$$p_1^{c(v_1)} \cdot p_2^{c(v_2)} \cdots p_m^{c(v_m)},$$

where the p_i's are distinct prime numbers and $C(V_i)$ represents the contents of variable V_i. We retain the simple variables X_i for holding domain values. Since all V_i contain 0 initially, we initialize $PD1$ to a height of 1 by pushing Ω into the stack. We must now show how to simulate the primitive arithmetic operations of $V \leftarrow V + 1$, $V \leftarrow V \dot{-} 1$ and $V \ominus 0$.

1. Replace each statement $V_i \leftarrow V_i + 1$ by a compound statement which "pours" the contents of pds $PD1$ into pds $PD2$, inserting $p_i - 1$ new elements into $PD2$ with each element that is transferred from $PD1$ to $PD2$. This multiplies the stack height by p_i. We can then restore the canonical state by pouring the elements back into $PD1$ from $PD2$.

2. Replace each test for $V_i \ominus 0$ by a compound statement which pours $PD1$ into $PD2$, computing $Z = |PD1| \bmod p_i$. Thus $V_i = 0$ if and only if $Z = 0$. We then restore the canonical state.

3. Replace each statement $V_i \leftarrow V_i \dot{-} 1$ by a compound statement which first tests for $V_i \ominus 0$ and does nothing further if true. Otherwise, we pour $PD1$ into $PD2$, pushing onto $PD2$ only one element for every p_i that is popped from $PD1$. We then restore the canonical state. Q.E.D.

(4.6) THEOREM. $P_{Ae} \equiv P_{(3b,0)}$.

Proof. Apply Lemmas 4.4 and 4.5 to get $P_{Ae} \leqq P_{(3b,0)}$. Apply Theorem 4.3 and the fact that $P_{AM} \equiv P_{Ae}$ ([1, Thm. 8.8]) to get $P_{(3b,0)} \leqq P_{Ae}$. Q.E.D.

(4.7) THEOREM. $P_{(2b,0)} > P_{(n,0)}$ *for* $n \geqq 1$.

Proof. The relations $P_{(2b,0)} \geqq P_{(2,0)} \equiv P_{(n,0)}$ from left to right are (i) obvious, and (ii) proved in Theorem 3.1. We need only find a functional which is $P_{(2b,0)}$ computable but not $P_{(n,0)}$ computable. *Leaftest* (see Introduction) is not $P_{(n,0)}$ computable by Lemma 3.4. We show it can be performed in $P_{(2b,0)}$ by the following algorithm (using pds's $PD1$ and $PD2$).

Step 1. Initialize $PD1$ to contain a copy of the input X.

Step 2. Transfer the contents of $PD1$ to $PD2$, applying the predicate P to each value moved. Halt if any value yields true.

Step 3. Compute $L(V)$ and $R(V)$ for each value V in $PD2$, storing the results in $PD1$ as they are computed. When $PD2$ becomes empty, return to Step 2.

The ability to test for an empty stack is crucial here, because it allows us to tell when all values have been transferred. Q.E.D.

We have carefully avoided discussing the class $P_{(2b,0)}$ in this section because this class has resisted our best attempts at characterization. Intuitively, two pds's seem to be adequate for control purposes, because such a configuration is essentially a two-counter machine [3] and has sufficient power to simulate any Turing machine. Moreover, even with one pushdown store available we have as much room for intermediate results as is necessary. Thus at first glance, it would seem likely that the operation of the two control stacks could be merged with that of the work stack and hence we could prove that $P_{(2b,0)}$ is also universal. However, none of our attempts to do this has been successful.

We shall now introduce a functional which is a generalization of *Leaftest* and which is pertinent to the discussion of the power of $P_{(2b,0)}$. Suppose we are

given a set of functions $\{F_1, \cdots, F_k\}$, a set of predicates $\{P_1, \cdots, P_m\}$ and a set of values $\{x_1, \cdots, x_n\}$. The class of all "arithmetic" expressions generable from these objects may be represented by the following context-free grammar:

$$E \to x_1 | \cdots | x_n$$

$$E \to F_1(\underbrace{E, \cdots, E}_{\substack{RF_1 \\ \text{times}}})$$

$$\vdots$$

$$E \to F_k(\underbrace{E, \cdots, E}_{\substack{RF_k \\ \text{times}}})$$

We now define:

(4.8) *Husearch* $(F_1, \cdots, F_k, P_1, \cdots, P_m, x_1, \cdots, x_n)$

$$= \begin{cases} x_1 & \text{if } \exists \text{ an integer } i \text{ and expressions} \\ & E_1 \text{ through } E_{RP_i} \text{ (as defined by } E \text{ above)} \\ & \ni P_i(E_1, \cdots, E_{RP_i}) = \textbf{true}; \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Thus, *Husearch* searches the Herbrand Universe generated by the F_i's and x_j's.

Constable and Gries [1, Construction 9.11] showed how to perform this search in P_A. A thorough discussion of the search program in P_A is given by Gries [2].

(4.9) THEOREM. $P_{(2b,0)} \equiv P_{(3b,0)}$ *if and only if the Husearch functional can be computed in* $P_{(2b,0)}$.

Proof. The "only if" portion follows immediately from [1, Construction 7.11] and Theorem 4.6. To prove the "if" part, we sketch how the *Husearch* computation can be used to construct a locator in $P_{(2b,0)}$ for a scheme $S \in P_{(3b,0)}$. Once we have such a locator we can use the values it generates to simulate markers and thus have universal power [1, Thm. 8.4].

Therefore, suppose we are given a scheme $S \in P_{(3b,0)}$. Let us assume autonomous behavior for S. Using the same approach as in the proof of Lemma 2.7, we first ascertain whether the autonomous behavior is finite or infinite. If it is finite, we can clearly construct a locator in P. If the autonomous behavior is infinite, we launch into the *Husearch* computation. If *Husearch* never halts, then S cannot halt (though the converse is clearly not true). If *Husearch* does halt, we can then simulate S directly with the values it returns. Q.E.D.

Notice that the construction outlined in this theorem is noneffective, because it is recursively unsolvable whether the autonomous behavior of an arbitrary $P_{(3b,0)}$ scheme is finite or infinite. Even if we could "program" the *Husearch* functional in $P_{(2b,0)}$, we still would have left as an open problem whether or not the two classes $P_{(2b,0)}$ and $P_{(3b,0)}$ are effectively equivalent.

We may note that in the simple case in which S is a scheme using only monadic functions and predicates of any rank then *Husearch* can be computed in $P_{(2b,0)}$ and there does exist $S' \in P_{(2b,0)}$ such that $S' = S$. However, all attempts to program

the general *Husearch* in $P_{(2b,0)}$ have so far failed, leading us to the following conjecture.

(4.10) CONJECTURE. $P_{(2b,0)} < P_{(3b,0)}$.

In some sense $P_{(2b,0)}$ is very "close" to the universal power of $P_{(3b,0)}$, because the slightest additional power given to $P_{(2b,0)}$ makes it universal. In particular, let us give $P_{(2b,0)}$ one "chip" which it can place anywhere in its stacks and for which it can test. Note that there is only one copy of this chip C, so if we execute the statements

$$V \leftarrow C;$$
$$PUSH(PD1, V);$$
$$\textbf{if } V = C \textbf{ then} \cdots$$

the predicate must be false. We assume that only the latest copy of C exists, and other instances (such as in V above) are replaced by Ω. For convenience, we assume that as long as the chip is in a simple variable it is "moved around" by assignments; that is, the sequence

$$V \leftarrow C;$$
$$W \leftarrow C;$$
$$X \leftarrow C;$$

results in V and W having the value Ω and X containing the chip. However, when the chip enters a data structure (such as a pushdown stack) it becomes inaccessible until it is later fetched by the data structure accessing primitives. Thus,

$$X \leftarrow C;$$
$$PUSH(PD1, X);$$
$$Y \leftarrow C;$$

while syntactically valid, results in both X and Y containing the value Ω and the chip being on the top of the pds. If a $POP(PD1, W)$ is executed, W then contains the chip. The concept of a chip is difficult to express clearly because it is antithetical to the usual notion of the contents of a variable.

(4.11) THEOREM. $P_{(2b,0)C} \equiv P_{(3b,0)}$, *where the equivalence is effective.*

Proof. (i) $P_{(3b,0)} \geqq P_{(2b,0)C}$. We know from Theorem 4.6 that $P_{(3b,0)}$ is universal, and therefore by Theorems 8.4 and 7.3 of [1],

$$P_{(3b,0)} \overset{\text{eff}}{\equiv} P_{(2,1)} \overset{\text{eff}}{\equiv} P_{(2,3)}$$

(two stacks and three markers).

Containment is obvious between $P_{(2,3)}$ and $P_{(2b,0)C}$.

(ii) $P_{(3b,0)} \leqq P_{(2b,0)C}$.

The argument in this direction depends on a Gödelization of the three pds's of an arbitrary scheme S in $P_{(3b,0)}$, so that these pds's can be represented in a new scheme S' in $P_{(2b,0)C}$. The method of pds storage, where $PD1$ contains values $a_1, a_2, \cdots$, $PD2$ contains $b_1, b_2, \cdots$, and $PD3$ contains $c_1, c_2, \cdots$, is shown in

Fig. 4. In the simulation given below we assume that we initialize $PD1$ by $PUSH(PD1, OMEGA)$; and that the resting configuration (between pds activity) is for the entire pds to be in $PD1$ and for $PD2$ to be empty.

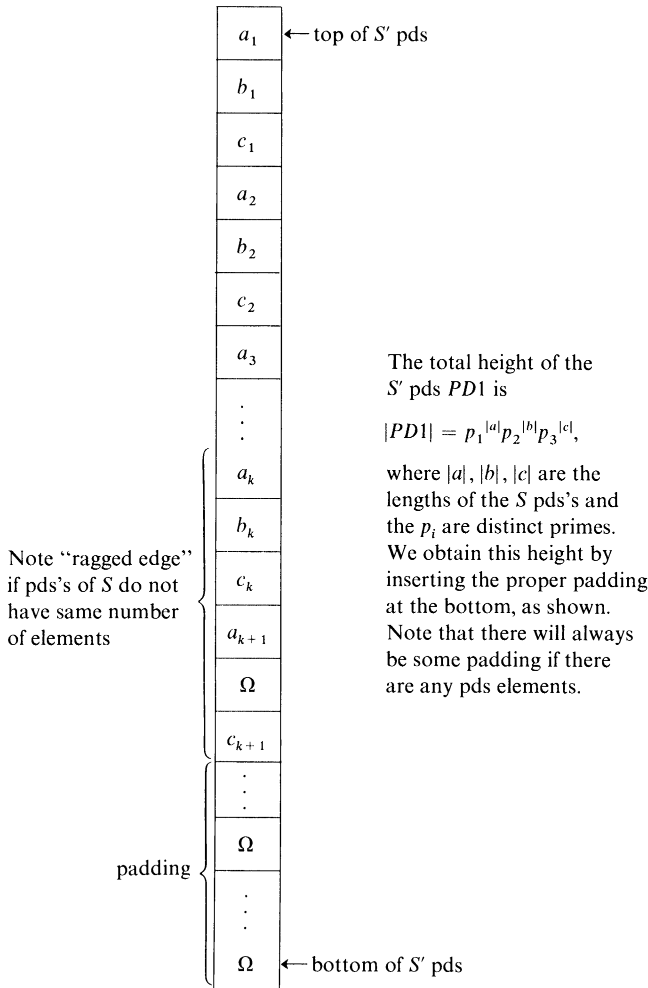


FIG. 4

Now we shall show how to push, pop, and test for emptiness any of the three pds's:

Push onto j-th pds. We need to multiply the pds length in S' by p_j, then move every element in the jth pds of S down 3 positions, and finally insert the new element in the jth position. We do this by

until $EMPTYPDS(PD1)$ **do**

begin $POP(PD1, V)$; $PUSH(PD2, V)$ **end**;

(thus moving the pds to $PD2$) (The section below uses the chip to multiply the pds

size by p_j, inserting the padding at the bottom of the pds.)

```
until EMPTYPDS(PD2) do
    begin
        POP(PD2, V); PUSH(PD2, C); PUSH(PD2, V);
        until EMPTYPDS(PD1) do
            begin POP(PD1, V); PUSH(PD2, V) end;
        PUSH(PD1, OMEGA); (repeated p_j − 1 times)
                 :
        POP(PD2, V);
        until V ⊜ C do
            begin PUSH(PD1, V); POP(PD2, V) end
    end
```

(Now we insert the new element and shift other elements of pds j down 3 positions.)

```
POP(PD1, V); PUSH(PD2, V); (repeated j − 1 times)
V1 ← new item;
until EMPTYPDS(PD1) do
    begin
        PUSH(PD2, V1); POP(PD1, V1);
        if ¬EMPTY PDS(PD1) do
            begin POP(PD1, V); PUSH(PD2, V) end;
        if ¬EMPTYPDS(PD1) do
            begin POP(PD1, V); PUSH(PD2, V) end;
    end;
```

(The element shifted off the bottom will be just padding.)

```
until EMPTYPDS(PD2) do begin POP(PD2, V); PUSH(PD1, V) end
```

(thus restoring $PD1$)

Test for emptiness of j-th pds. To simulate the statement

if $EMPTYPDS(PDS_j)$ **then** $\langle S_1 \rangle$ **else** $\langle S_2 \rangle$

we simply pour from $PD1$ to $PD2$, computing $Z = |PD1| \bmod p_j$. The jth pds is empty if and only if $Z \neq 0$. Details are left to the reader,

Pop from j-th pds. We first test the jth pds for emptiness and do nothing if it is empty. Otherwise, the behavior is analogous to that for the push; we take the jth element of $PD1$ as the one desired, percolate the $(3 + j)$th element to the jth position, etc., and divide $|PD1|$ by p_j. The division is accomplished by starting C

from the top of $PD1$ and moving it downward. At each move we cut off $p_j - 1$ elements from the bottom of $PD1$ by appropriate pouring manipulation. We terminate when C reaches the bottom of the shrinking stack.

To construct S' given S, we simply duplicate the body of S and substitute for each *PUSH*, *POP* and bottom test the code described above. That the scheme so created mimics S should be clear from the construction.

Hence, since we have shown $\mathrm{P}_{(3b,0)} \leqq \mathrm{P}_{(2b,0)C}$ and $\mathrm{P}_{(3b,0)} \geqq \mathrm{P}_{(2b,0)C}$ effectively, the theorem is established. Q.E.D.

5. Schemes and integer arithmetic. In this section we shall investigate the power of some classes of schemes whose control structures have been augmented by the ability to do integer arithmetic. Accordingly, we allow the statements $V \leftarrow 0$, $V \leftarrow V + 1$, $V \leftarrow V \dot{-} 1$ and the conditional statement **if** $V \ominus 0$ **then** $\langle S_1 \rangle$ **else** $\langle S_2 \rangle$. We leave it to the reader to show how more complicated statements, such as $V_i \leftarrow V_j$ or $V_i \leftarrow V_j \times V_k$, or indeed any computable function over the integers can be built up from these primitive statements. The formal definition of this new class $P_{\underset{\sim}{\mathbb{N}}}$ is given in [1, (4.9)].

It has already been shown in Theorem 10.10 of [1] that the class $\mathrm{P}_{(1,0)\underset{\sim}{\mathbb{N}}}$ is universal in the sense of being effectively equivalent to the classes P_{Ae}, $\mathrm{P}_{(3b,0)}$, etc. In the remaining portions of this section we characterize the class $\mathrm{P}_{\underset{\sim}{\mathbb{N}}}$ and find that it partially overlaps the classes P_{R} and $\mathrm{P}_{(2,0)}$ but is properly contained in $\mathrm{P}_{(2b,0)}$. The reason for this rather unusual property is the immense power of the control structure of a $\mathrm{P}_{\underset{\sim}{\mathbb{N}}}$ scheme (indeed, we have enough power to simulate an arbitrary Turing machine) coupled with the restriction of a fixed number of locations for computing results over the output domain.

Our basic tool here involves the functional *Evalcutset* first described in [4]:

$$Evalcutset\,(x) = \textbf{if } P(x) \textbf{ then } x \textbf{ else}$$

$$H(Evalcutset(L(x)), Evalcutset(R(x))).$$

Intuitively, *Evalcutset* does the following:

(a) examines the infinite binary tree formed by the monadic functions L and R operating on the value input in x;

(b) finds the (unique) minimal cutset of this tree such that all nodes in the cutset make P **true**;

(c) treats the portion of the tree above and including this cutset as a description of an arithmetic expression in H, L, R and x;

(d) evaluates the expression so defined.

In [4] it was shown that *Evalcutset* cannot be computed using a fixed number of variables because an unbounded number of temporary results will in general be necessary for this computation. This then implies that *Evalcutset* cannot be computed in $\mathrm{P}_{\underset{\sim}{\mathbb{N}}}$ and furthermore implies that any functional which requires an evaluation of *Evalcutset* independent of the other inputs to the functional cannot be computed in $\mathrm{P}_{\underset{\sim}{\mathbb{N}}}$ either.

We shall find the following fact concerning monadic functions useful.

(5.1) THEOREM. *Consider the restriction of schemes to monadic functions only (predicates may have any rank). Then* $\mathrm{P}_{\underset{\sim}{\mathbb{N}}} \equiv \mathrm{P}_{\mathrm{Ae}}$.

Proof. Clearly $\mathrm{P}_{\mathbb{N}} \leqq \mathrm{P}_{\mathrm{Ae}}$. Consider a scheme S in P_{Ae}, and construct an equivalent effective functional F ([1, Thm. 10.6]). Since all functions are monadic, all expressions in the computations of F have the form $f_1(f_2(\cdots(f_j(x))\cdots))$, where the f_i are function names and x is an input variable. Any expression can thus be evaluated using one variable. Any proposition $\mathrm{P}(e_1, \cdots, e_n)$ can hence be evaluated using n variables. By Corollary 10.9 of [1] we can construct an equivalent scheme in $\mathrm{P}_{\mathbb{N}}$. Q.E.D.

In order to characterize $\mathrm{P}_{\mathbb{N}}$, we now introduce 6 functionals, each using the monadic predicate P, the monadic functions L and R, the dynadic function H and the input variables w, x, y, z.

$$f_1 = \begin{cases} w & \text{if } Evalcutset(P, L, R, H, w) \text{ is defined,} \\ \text{undefined} & \text{otherwise;} \end{cases}$$

$$f_2 = \begin{cases} Evalcutset(P, L, R, H, w) & \text{if } Evalcutset(P, L, R, H, w) \text{ is defined,} \\ \text{undefined} & \text{otherwise;} \end{cases}$$

$$f_3 = \begin{cases} w & \text{if } P(x) \wedge \neg P(y) \wedge Leaftest(P, L, R, z) \text{ is defined,} \\ \text{undefined} & \text{otherwise;} \end{cases}$$

$$f_4 = \begin{cases} Evalcutset(P, L, R, H, w) & \text{if } P(x) \wedge \neg P(y) \wedge Leaftest(P, L, R, z) \text{ is defined,} \\ \text{undefined} & \text{otherwise;} \end{cases}$$

$$f_5 = \begin{cases} w & \text{if } Leaftest(P, L, R, z) \text{ is defined,} \\ \text{undefined} & \text{otherwise;} \end{cases}$$

$$f_6 = \begin{cases} Evalcutset(P, L, R, H, w) & \text{if } Leaftest(P, L, R, x) \text{ is defined,} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Clearly, f_2, f_4 and f_6 cannot be computed in $\mathrm{P}_{\mathbb{N}}$ since each of them must conditionally evaluate *Evalcutset* which needs an unbounded number of variables. On the other hand, consider functionals f_1, f_3, and f_5. We do not have to evaluate *Evalcutset*; we just have to know whether it is *defined*, and we can tell this by the following functional:

$$Evalcutsetdef(x) = \textbf{if } P(x) \textbf{ then true}$$
$$\textbf{else } Evalcutsetdef(L(x)) \textit{ and } Evalcutset(R(x))$$

Evalcutset and Leaftest can both be programmed in P_{Ae} using only monadic functions, and hence, by Theorem 5.1 can be computed in $\mathrm{P}_{\mathbb{N}}$. Hence, f_1, f_3, and f_5 are computable in $\mathrm{P}_{\mathbb{N}}$.

We exhibit in Fig. 5 a Venn diagram of the classes P, P_{R}, $\mathrm{P}_{(2,0)}$, $\mathrm{P}_{(2b,0)}$, $\mathrm{P}_{\mathbb{N}}$ and place each of the functionals f_i in the most restrictive class which permits its computation.

(5.2) LEMMA. *f_1 and f_2 can be computed in* P_{R}.

Proof. The proof is an obvious programming exercise.

(5.3) LEMMA. *f_3 and f_4 can be computed in* $\mathrm{P}_{(2,0)}$ *but not in* P_{R}.

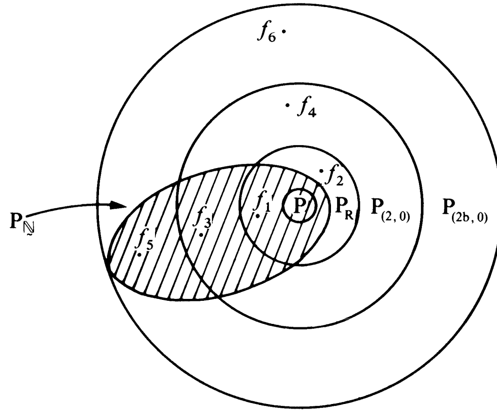


FIG. 5

Proof. The test $P(x) \wedge \neg P(y)$ gives us the necessary values with which to simulate markers. Once we have markers we essentially have a universal scheme. Finally, Theorem 3.5 shows why neither f_3 nor f_4 can be computed in $\mathrm{P_R}$.

(5.4) LEMMA. *f_5 and f_6 can be computed in* $\mathrm{P}_{(2b,0)}$ *but not in* $\mathrm{P}_{(2,0)}$.

Proof. It should be clear that given a $\mathrm{P}_{(2,0)}$ scheme to compute either f_5 or f_6 we can modify it to produce a $\mathrm{P}_{(2,0)}$ scheme which computes *Leaftest*. However, this is impossible by Lemma 3.3. Hence, neither f_5 nor f_6 is $\mathrm{P}_{(2,0)}$ computable. On the other hand, since *Leaftest* is $\mathrm{P}_{(2b,0)}$ computable and thus furnishes us with any necessary markers, we conclude that both f_5 and f_6 are $\mathrm{P}_{(2b,0)}$ computable.

The final result needed to complete the Venn diagram in Fig. 5 is the following theorem.

THEOREM. $\mathrm{P}_{\mathbb{N}} < \mathrm{P}_{(2b,0)}$.

Proof. The inclusion follows as a corollary to the proof of Lemma 4.5. The two pds's are used as counters holding the Gödelized contents of the variables of the $\mathrm{P}_{\mathbb{N}}$ scheme.

The fact that the inclusion is proper follows from the fact that f_6 is not $\mathrm{P}_{\mathbb{N}}$ computable. Q.E.D.

One final comment is in order here, namely that the *Husearch* functional of § 4 is not $\mathrm{P}_{\mathbb{N}}$ computable. This fact follows from Corollary 6.3 of [5].

6. Multidimensional arrays. Let us allow the use of an n-dimensional array $A, n > 1$. We use the obvious interpretation that $A[w_1, \cdots, w_n]$ is the same variable as $A[v_1, \cdots, v_n]$ if and only if $w_i \ominus v_i$ for $i = 1, \cdots, n$ (see [1, Def. 4.5]). The main result of this section is that adding n-dimensional arrays in $\mathrm{P_A}$ does not change the power of the class.

(6.1) THEOREM. *Let S be a scheme in* $\mathrm{P_A}$, *with the addition of n-dimensional arrays. We can construct an equivalent scheme S′ in* $\mathrm{P_A}$.

Proof. We show how to construct in $\mathrm{P_A}$ both a locator and a P-simulator for S. We combine these as described in the proof of Theorem 1.5 to form S' in $\mathrm{P_A}$ equivalent to S.

The locator is constructed as was the locator for a P_{AM} scheme in showing that $P_A \equiv P_{AM}$ [1, Thm. 9.14]. The locator is shown in (1.6), where the statements $S_1, \cdots, S_{2^n}$ have to be constructed.

If the v_i-autonomous behavior of S is finite, we construct S_i as in [1, (9.9)]. Note that since the behavior of S is finite, S_i will reference only a finite set of variables, and we can change all the referenced variables to simple variables. Thus S_i will clearly be in P_A.

If the v_i-autonomous behavior of S is infinite, we construct S_i as described in [1, (9.11)], and S_i is a statement of P_A.

Now note that, given S in P_A using multidimensional arrays, we can *effectively* decide whether the v-autonomous behavior of S is finite or infinite. Suppose S contains p statements. Begin recording the v-autonomous behavior; if $p + 1$ labels are recorded, there is a loop and the behavior is infinite. (This is the same process as deciding whether a scheme in P_A has finite or infinite v-autonomous behavior [1, Thm. 9.5].) Hence, we can *effectively* construct the locator.

To construct the P-simulators, consider scheme S to be in P_{Ae} with multi-dimensional arrays. Lemma 6.2 below shows how to construct an equivalent scheme S' in P_{Ae} using only one-dimensional arrays. Using Theorem 8.8 of [1] we can construct an equivalent scheme S'' in P_{AM}. Finally we use Theorem 5.4 of [1] to construct P-simulators in P_A for S'' and hence for S. Q.E.D.

(6.2) LEMMA. *Let S be a scheme in P_{Ae} which in addition uses an n-dimensional array A, $n > 1$. There exists an equivalent scheme S' in P_{Ae} which uses $n + 1$ one-dimensional arrays $B_0, B_1, \cdots, B_n$ in place of A.*

Proof. In addition to the arrays, S' uses an additional variable I, whose purpose is to indicate how many different elements $A[\cdots]$ (in S) have been assigned values. If in S, $A[w_1, \cdots, w_n]$ has been assigned a value v, then in S' for some j we have

$$B_1[j] = w_1, \cdots, B_n[j] = w_n, \quad B_0[j] = v.$$

Let J be a new variable, and let $COPY[J, w_1, \cdots, w_n]$ stand for the statement

begin $J \leftarrow 0$;
  until $I \ominus J$ **do**
    begin if $B_1[J] \ominus w_1$ *and* $\cdots$ *and* $B_n[J] \ominus w_n$
      then goto $FOUND$;
      $J \leftarrow J + 1$
    end;
  $B_1[J] \leftarrow w_1; \cdots; B_n[J] \leftarrow w_n$;
  $B_0[J] \leftarrow OMEGA; I \leftarrow I + 1$;
  ⋮
 $FOUND$;
 end;

This statement performs a linear search for an index J such that $B_1[J] = w_1, \cdots, B_n[J] = w_n$. If found, then $A[w_1, \cdots, w_n]$ in S is the location $B_0(J)$ in S'. If not found, it is added.

Now, to translate S into S', we

1. Add the following statement to the beginning of $S: I \leftarrow 0$.
2. Transform S so that the only reference to the array A is in statements

$$A[w_1, \cdots, w_n] \leftarrow v \quad \text{and} \quad v \leftarrow A[w_1, \cdots, w_n]$$

where w_i and v are simple variables.

3. Change each statement $A[w_1, \cdots, w_n] \leftarrow v$ to

$$\textbf{begin } COPY[J, w_1, \cdots, w_n];\ B_0[J] \leftarrow v \textbf{ end}$$

4. Change each statement $v \leftarrow A[w_1, \cdots, w_n]$ to

$$\textbf{begin } COPY[J, w_1, \cdots, w_n];\ v \leftarrow B_0[J] \textbf{ end}$$

Q.E.D.

Appendix. We give here the details of the proofs and constructions in some of the lemmas and theorems. Refer to the proper lemma in the paper for discussion.

(2.2) LEMMA. $P_{(1,n)} \leqq P_{RgM}$ *for* $n \geqq 0$.

Step 1. Given S in $P_{(1,n)}$, create $S1$ equivalent to S as follows. For each statement $PUSH(P, v)$ or $POP(P, v)$ in S, generate a new label L and replace the statement by

$$\textbf{begin } PUSH(P, v);\ L: \textbf{end}$$

or

$$\textbf{begin } POP(P, v);\ L: \textbf{end}$$

For each statement **halt**(v), generate a new label L and replace the statement by L: **halt**(v). Hence, each $PUSH$ and POP is followed by a labeled null statement, and each **halt** is labeled. Let these new unique labels be called $L_1, L_2, \cdots, L_l$.

Step 2. Create $S2 \equiv S1$ as follows. Let $S1$ have the form

$$(v, \cdots, v): S_1;\ S_2;\ \cdots;\ S_n$$

and suppose it uses simple variables $V_1, \cdots, V_k$. Then $S2$ is the scheme

$$(v, \cdots, v): S_1;\ S_2;\ \cdots;\ S_n$$
$$F(P, X): \textbf{global } V_1, \cdots, V_k;$$
$$S_1;\ S_2;\ \cdots;\ S_n$$

Note that the same labels are used in both the main scheme and in F. However, labels are local to the function in which they are used, and jumps out of functions are not allowed.

Step 3. Create $S3 \in P_{RgM}$, $S3 \equiv S2$. In $S3$, $PUSH$ statements in $S2$ are replaced by calls on F; POPs and **halt**s within F are replaced by returns; and POPs in the main scheme are deleted. The pds P is now a parameter variable of F.

(a) Let $\bar{L}_1, \cdots, \bar{L}_l$ be new unique markers corresponding to the labels

$L_1, \cdots, L_l$ introduced in Step 1. Insert just before statement S_1 of F the sequence

if $X = \bar{L}_1$ **then go to** L_1;

$\vdots$

if $X = \bar{L}_l$ **then go to** L_l;

(b) Change each $PUSH$ statement **begin** $PUSH(P, v)$; L_i: **end** to

begin $X \leftarrow F(v, \bar{L}_i)$;

if $X = \bar{L}_1$ **then go to** L_1;

$\vdots$

if $X = \bar{L}_l$ **then go to** L_l;

L_i: **end**

(c) Change each statement

L_i: **halt**(V) within F to L_i: **halt**($\bar{L}_i$).

Note that this construction causes all **halts** of the scheme to result in an empty pds at termination.

(d) Change each POP statement **begin** $POP(P, v)$; L_i: **end** within F to

begin $v \leftarrow P$; **halt**($\bar{L}_i$); L_i: **end**

(e) Replace each POP statement **begin** $POP(P, v)$; L_i: **end** within the main scheme by **begin** L_i: **end**. (Within the main scheme the pds is empty, and POP is a null instruction.) Q.E.D.

(2.7) LEMMA. *Every scheme S in $\mathrm{P}_{\mathrm{RgM}}$ has a locator S' in* P.

Proof. The locator S' for S has the form (1.6) and we must only show how to construct the statements $S_1, \cdots, S_{2^n}$ described there. We outline only the construction of S_1, which simulates the v-autonomous behavior of S where $v =$ (**true**, $\cdots$, **true**), as described by (1.7). We rely on the notation and results of Lemma 2.6.

• The first phase is to construct the v-autonomous behavior of S as described in [1, (9.10)], with the following changes and additions:

(i) With each label L_i of the behavior, keep (a) the statement it labels; (b) an indication of which function execution it occurs in (not only the function, but which call of the function it is); (c) the current values $(\bar{v}, M_1, \cdots, M_{m-1})$ of the global variables; and (d) the current values of the local variables of the function (or main program).

(ii) If a label is added which already occurs in the behavior for this particular function execution (say at position j), and if the values of the global and local variables are the same, then the behavior is infinite. Stop building the behavior and record with this last label the position j.

(iii) After a label L_i: $v \leftarrow f(\cdots)$ (where f is a recursive function) is added, perform the following. Check back to see if a call of f with the same argument *values* (not variables) and global values has occurred and is not yet finished (say

at position j of the behavior). If so, the behavior is infinite. Stop building the behavior and record with this last label the position j.

If no such previous call has occurred, then before proceeding expand the call $v \leftarrow f(\cdots)$ in S as described in [1,(3.7)]. This of course makes the call $v \leftarrow f(\cdots)$ superfluous, and it will be deleted later.

- The second phase is to construct the statement S_1 from the behavior constructed in Step 1. This behavior is of course just the partial behavior if we stopped building the behavior via Steps 2 or 3 above. (Because of Lemma 2.6 the construction must stop eventually, either with a **halt** or by Steps 2 or 3.)

Construct S_1 from the behavior with the following changes:

(i) Replace statements L_i: $v \leftarrow f(\cdots)$, where f is a recursive procedure, by null statements.

(ii) If the construction of the behavior was stopped by (ii) of phase 1, then generate a new label L, prefix it to the jth substatement within S_1 (using the j of (ii) of phase 1), and replace the last label of the behavior by **go to** L.

(iii) If the construction of the behavior was stopped by (iii) of phase 1, then generate a new label L and prefix it to the jth substatement of S_1 (using the j of (iii) of phase 1). Suppose this call at the jth position was originally $v \leftarrow f(v_1, \cdots, v_n)$ and suppose the last labeled statement of the behavior is $w \leftarrow f(w_1, \cdots, w_n)$. Then generate new variables $V_1, \cdots, V_n$ and add to S_1 the statements

$$V_1 \leftarrow w_1; \cdots; V_n \leftarrow w_n;$$
$$v_1 \leftarrow V_1; \cdots; v_n \leftarrow V_n; \textbf{go to } L; \qquad \text{Q.E.D.}$$

(2.8) LEMMA. $P_{RgM} \leqq P_{RM}$, $P_{Rg} \leqq P_R$.

Proof. Suppose scheme $S \in P_{RgM}$ has function definitions for functions $F_1, \cdots, F_n$, and suppose that the variables used globally are $V_1, \cdots, V_m$. By suitably renaming the local variables we can make sure that $V_1, \cdots, V_m$ are used *only* as global variables, and hence we can assume S has the form

$$
\begin{array}{ll}
 & (v, \cdots, v): \langle S \rangle; \cdots; \langle S \rangle \\
 & F_1(v, \cdots, v): \textbf{global } V_1, \cdots, V_m; \langle S \rangle; \cdots; \langle S \rangle \\
(2.9) & \vdots \qquad\qquad\qquad\qquad \vdots \\
 & F_n(v, \cdots, v): \textbf{global } V_1, \cdots, V_m; \langle S \rangle; \cdots; \langle S \rangle
\end{array}
$$

We give a construction which reduces by one the number of global variables. If the construction is executed m times we arrive at an equivalent scheme in P_{RM}. Let us now show how to eliminate the need for V_1 to be global.

Note that execution of a function F_i may change the value of V_1. We must therefore find a way of transmitting this change back to the calling function or main program. To do this, we replace each call of F_i by *two* calls; one call to F_i returns the normal value, and the second call to a new function $\bar{F}_i$. $\bar{F}_i$ executes exactly the way F_i does, but just returns a different value.

Step 1. For each function definition F_i in (2.9) insert a new function definition

$$\bar{F}_i(v, \cdots, v); \qquad \textbf{global } V_1, \cdots, V_m; \qquad \langle S \rangle; \cdots; \langle S \rangle$$

where $\bar{F}_i$ looks exactly like F_i except that each **halt**(w) has been replaced by **halt**(V_1). The resulting scheme $S1$ is equivalent to S, since all we have done is add function definitions.

Step 2. Replace $S1$ by the following scheme $S2$:

$$(v, \cdots, v)\colon \langle \mathrm{S} \rangle\,; \cdots ; \langle \mathrm{S} \rangle$$

$$F_1(v, \cdots, v, V_1)\colon \mathbf{global}\ V_2, \cdots, V_m; \langle \mathrm{S} \rangle\,; \cdots ; \langle \mathrm{S} \rangle$$

$$\vdots \qquad \vdots \qquad \vdots$$

$$F_n(v, \cdots, v, V_1)\colon \mathbf{global}\ V_2, \cdots, V_m; \langle \mathrm{S} \rangle\,; \cdots ; \langle \mathrm{S} \rangle$$

$$\bar{F}_1(v, \cdots, v, V_1)\colon \mathbf{global}\ V_2, \cdots, V_m; \langle \mathrm{S} \rangle\,; \cdots ; \langle \mathrm{S} \rangle$$

$$\vdots \qquad \vdots \qquad \vdots$$

$$\bar{F}_n(v, \cdots, v, V_1)\colon \mathbf{global}\ V_2, \cdots, V_m; \langle \mathrm{S} \rangle\,; \cdots ; \langle \mathrm{S} \rangle$$

$S2$ executes as $S1$ does, except for the fact that if during execution of a function V_1 is changed, this change is not transmitted back to the variable V_1 local to the point of call. The final Step 3 translates $S2$ into $S3$, where $S3$ is equivalent to $S1$ and thus S.

Step 3. Each of the functions F_i and $\bar{F}_i$ and the main program uses new variables $V_0, \bar{V}_2, \cdots, \bar{V}_m$ which are *local* to the function or main program. Replace each call

$$w \leftarrow F_i(v_1, \cdots, v_n, V_1)$$

by

begin $\bar{V}_2 \leftarrow V_2; \cdots ; \bar{V}_m \leftarrow V_m;$	(Save global values)
$V_0 \leftarrow F_i(v_1, \cdots, v_m, V_1);$	(Call F_i to get normal result into V_0)
$V_2 \leftarrow \bar{V}_2; \cdots ; V_m \leftarrow \bar{V}_m;$	(Restore global values)
$V_1 \leftarrow \bar{F}_i(v_1, \cdots, v_m, V_1);$	(Call $\bar{F}_i$ to execute as F_i did but return the value of V_1)
$w \leftarrow V_0;$	(Put result into variable w)
end	

Q.E.D.

REFERENCES

[1] ROBERT L. CONSTABLE AND DAVID GRIES, *On classes of program schemata*, this Journal, 1 (1972), pp. 66–118.

[2] DAVID J. GRIES, *Programming by induction*, Information Processing Letters, 1 (1972), pp. 100–107.

[3] JOHN E. HOPCROFT AND J. D. ULLMAN, *Formal Languages and Their Relation to Automata*, Addison-Wesley, London, 1969.

[4] M. S. PATERSON AND C. E. HEWITT, *Comparative schematology*, Conference Record of Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, New York, 1970, pp. 119–128.

[5] H. R. STRONG, *Translating recursion equations into flow charts*, J. Comput. System Sci., 5 (1971), pp. 254–284.

SIAM J. COMPUT.
Vol. 1, No. 3, September 1972

ANALYSIS OF SCANNING POLICIES FOR REDUCING DISK SEEK TIMES*

E. G. COFFMAN, L. A. KLIMKO, AND BARBARA RYAN†

Abstract. A number of recent studies have examined techniques for sequencing disk accesses to minimize or reduce seek times. The principal methods proposed have been called scanning policies. In this paper we formulate and analyze simple mathematical models of head motion in disk systems in which two different scanning policies are implemented. Expressions for response times are derived, and the properties they imply are discussed.

1. Introduction. Frequently, the major factor determining overall performance in a multiprogramming or multiprocessing system is the rate at which information can be transferred between main memory and magnetic disk units being used in the role of auxiliary storage. As a result, considerable effort has gone into the study of methods for increasing this transfer rate. For the most part these methods amount to easily implemented rules for efficiently sequencing queues of requests for disk access [1], [2], [3].

In general, the total transfer time between core memory and disks involves three components: (i) a seek time (head positioning delay), (ii) a rotational latency delay, and (iii) a transfer time. From the operating system's point of view we must normally assume that transfer rates are not under our control. However, we can institute various techniques for reducing the first two delays. Seek time usually dominates these two delays and forms the particular subject of this paper. Methods of reducing rotational latency which are independent of procedures for scheduling seeks are discussed in [1], [4].

In the next section we discuss briefly a number of methods for scheduling disk-access requests to reduce seek times. We then describe a simple mathematical model which is used in the subsequent section to analyze two well-known scanning rules for servicing requests. Our principal goal is the insight into general behavior obtainable through mathematical modeling. For simulations and approximate methods applied to models incorporating more system detail we refer to [1], [2], [3].

2. Sequencing rules. For our purposes it is convenient to adopt the simple model pictured in Fig. 1, in which a single head and a single disk with N tracks (or cylinders) are shown. Without any particular concern for operating efficiency, the first discipline normally considered is FIFO (first-in-first-out) sequencing, simply because it is usually the easiest to implement. Specifically, requests for information to be read or written on disk are stored in a FIFO queue; at each decision point the head of the queue is selected and the disk head moved (if necessary) to the addressed track. After this seek time a latency delay is incurred while the start of the addressed record is rotating to a position under the head, at which point the actual transfer of information commences. In general, in a system with multiple disk units there may also be a delay (prior to the latency delay) waiting for the availability of an input-output channel on which to transfer

* Received by the editors February 14, 1972, and in revised form May 15, 1972.

† Computer Science Department, Pennsylvania State University, University Park, Pennsylvania 16802.

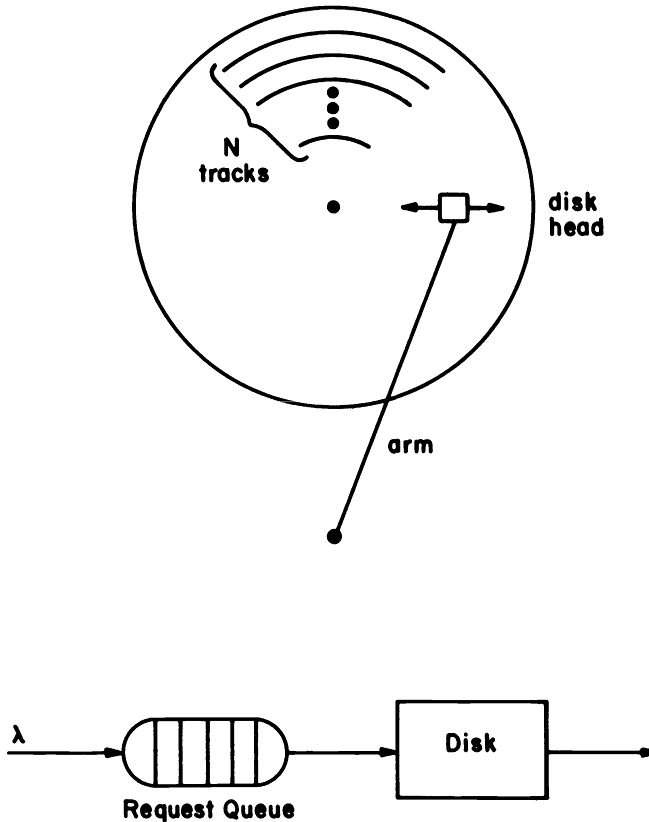


FIG. 1. *Disk system model*

the information requested. In the mathematical model presented later the delays subsequent to seek times are all lumped into a single parameter.

Analyses of the FIFO system have been carried out by a number of authors, by applying known results for tandem and parallel queues. Our interest focuses on the more efficient rules described next. First, however, we should note for future comparison the expected delay incurred by seeks in the FIFO system. Suppose there are N tracks and that the track addresses of successive requests are chosen independently and at random from the set $\{1, 2, \cdots, N\}$. Then it is easily shown that the mean number of tracks over which the head must move in servicing a request is asymptotically $N/3$.

By making use of track addresses assumed known for the waiting requests we can reduce substantially this initial delay. A possible method for improving service is the so-called shortest-seek-time-first (SSTF) procedure [1]. At each decision point with SSTF sequencing the next request selected for service is the one having a track address closest to the current position of the head. A little reflection reveals an apparent drawback of the SSTF rule [1], [2]: the tracks at the inside and outside extremities of the disk can expect poor service relative to internal tracks.

However, relative to FIFO sequencing the above discrimination may very well be a small price to pay for the significant increase in system capacity.

The following two methods can be used to reduce the discrimination at the "extremity" tracks while retaining the system capacity of the SSTF rule (under the assumption that we have a nonzero usage of all tracks). The first rule is called the SCAN rule [1] and simply applies the SSTF criterion in one direction only: the head scans across the tracks in a given direction, servicing requests as they are encountered, until one of two events occurs:

(i) The last track in the given direction has been processed, or
(ii) there are no further requests ahead of the head position in the given direction.

In both cases, the head motion is reversed and scanning is carried out in the reverse direction (assuming a nonempty queue) until one of the above events again occurs.

The second method is a FIFO scanning rule that we shall term FSCAN [3]. At each decision point FSCAN takes the entire queue and services the requests in a scan whose direction is determined by the minimum distance the current head position is from the outermost and innermost track addresses of the requests to be serviced. Thus, the scan is preceded by a move of the disk head to the nearer of the extreme track addresses (i.e., the nearer of the least and the maximum addresses). Arrivals during a given scan are placed in a queue and not serviced until the next scan; this is the principal difference between the SCAN and FSCAN rules. Clearly, during periods of moderate to heavy loading the head will be moving back and forth across the disk with both the SCAN and FSCAN rules. After servicing several requests by the FSCAN rule the head is likely to be at a high address, assuming a left-to-right scan. Thus, assuming several requests are likely to arrive during the scan, the right-most address represented by the waiting requests will probably be the one closest to the current head position, thus causing a subsequent right-to-left scan.

Through the analysis of a simplified mathematical model we shall see that the response with SCAN is uniformly better (over all track addresses) than with FSCAN, but that SCAN still involves a certain amount of discrimination against the innermost and outermost tracks. The mathematical model is described as follows.

The set of discrete track addresses is replaced by the interval $[0, d]$; i.e., the set of real numbers in $[0, d]$. For simplicity, but without loss of generality, we shall assume $d = 1$ and that the speed at which the head can move across the interval is a. Since $d = 1$ we shall also refer to a as the time required for the disk head to move across the address space (unit interval). We assume a Poisson input of requests at rate λ and we assume that the "tracks" addressed are distributed across $[0, 1]$ according to an arbitrary but given density function $f(x)$, $0 \leq x \leq 1$. Formally, the probability that a given arrival falls in the interval $(x, x + \Delta x)$ is given by $f(x)\,\Delta x$. We denote the cumulative distribution function by $F(y) = \int_0^y f(x)\,dx$.

Note that our assumption is consistent with the simplification that only one request per track exists at any given time. Informally, since the probability of more than one arrival in a small interval Δx is of order $(\Delta x)^2$ we may regard the intervals Δx as corresponding to tracks. Clearly this approximation worsens as the actual number N of tracks becomes small.

We shall assume that after the seek operation the time required to service any request is a constant T. A more general assumption can be made here, but it is not essential to the type of comparisons that we intend to make. Clearly, we are lumping not only latency delays and transfer times in T, but also delays incurred by starting and stopping the head motion and head switching times. Finally, we assume that *the head is always scanning when it is not servicing a request*, and that *the direction of the scan is reversed only when the head reaches a boundary*[1] at 0 or 1. Our assumptions are contained in the following statements:

(i) If in one crossing of the head $n \geqq 0$ requests are served, then the crossing time is given by $nT + a$.

(ii) The pmf for the number (n) of arrivals in time t in the interval (b, b'), $0 \leqq b < b' \leqq 1$, is given by the Poisson distribution

$$g_n(b, b'|t) = \frac{\{\lambda t[F(b') - F(b)]\}^n}{n!} \exp\{-\lambda t[F(b') - F(b)]\} \tag{1}$$

with mean value

$$\bar{g}(b, b't) = \lambda t[F(b') - F(b)]. \tag{2}$$

3. Analysis of the SCAN rule. Our objectives are first, an expression for the mean time for the head to move a distance x in the limit of statistical equilibrium; and second, a measure of mean waiting or response times.

For convenience we assume that the odd numbered crossings of the disk head are left-to-right. Let $y_{2n+1}(x)$, $n \geqq 0$, denote the random variable whose value is the time taken by the head on the $(2n + 1)$st crossing to move left-to-right a distance x from position 0, servicing the requests that it encounters enroute. Similarly, for the $(2n)$th crossing $y_{2n}(x)$ denotes the random variable corresponding to a right-to-left move of distance x from position 1. We shall use the term *cycle* to designate a complete scan of the disk head from 0 to 1 and back to 0. We next develop an expression for the expected value of $y_{2n+1}(x + \Delta x)$ for a small interval $(x, x + \Delta x)$.

From our description of the SCAN rule we know that the last time prior to the $(2n + 1)$st crossing that the interval $(x, x + \Delta x)$ received service was while the head was moving right-to-left through this interval on the $(2n)$th crossing. Thus, given $y_{2n+1}(x)$, $y_{2n}(1)$ and $y_{2n}(1 - x)$ we have from (1) that the mean number of requests serviced in $(x, x + \Delta x)$ on the $(2n + 1)$st crossing is $\lambda f(x)\,\Delta x[y_{2n+1}(x) + y_{2n}(1) - y_{2n}(1 - x)]$, plus a term of order $(\Delta x)^2$. Since each such request requires T time units and since the head requires $a\,\Delta x$ time units to cross the interval, we have, neglecting terms of order $(\Delta x)^2$,

$$\begin{aligned} &E[y_{2n+1}(x + \Delta x)|y_{2n+1}(x), y_{2n}(1), y_{2n}(1 - x)] \\ &\qquad = y_{2n+1}(x) + a\,\Delta x + \lambda T f(x)\,\Delta x[y_{2n+1}(x) + y_{2n}(1) - y_{2n}(1 - x)]. \end{aligned} \tag{3}$$

Letting $E[y_i(x)] \equiv \bar{y}_i(x)$ we have on removing the conditioning in (3),

$$\bar{y}_{2n+1}(x + \Delta x) = \bar{y}_{2n+1}(x) + a\,\Delta x + \lambda T f(x)\,\Delta x[\bar{y}_{2n+1}(x) + \bar{y}_{2n}(1) - \bar{y}_{2n}(1 - x)]. \tag{4}$$

[1] This simplification of the general model appears essential to an analysis of the SCAN rule. Although the simplification is not so clearly essential to the FSCAN analysis, the authors have not yet been able to analyse the general FSCAN model.

Now consider the limit $\bar{y}_r(x) \equiv \lim_{n\to\infty} \bar{y}_{2n+1}(x)$. Under the assumptions of our problem the existence of this limit requires that $\lambda < 1/T$. In more general terms, the existence of statistical equilibrium requires that the maximum service rate $1/T$ be greater than the arrival rate λ. Thus, taking limits in (4) we have

$$\bar{y}_r(x + \Delta x) = \bar{y}_r(x) + a\,\Delta x + \lambda T f(x)\,\Delta x \bar{Z}_r(x), \tag{5}$$

where $\bar{Z}_r(x) \equiv \lim_{n\to\infty} [\bar{y}_{2n+1}(x) + \bar{y}_{2n}(1) - \bar{y}_{2n}(1 - x)]$ is the average time in equilibrium for the head to move from x to 0 and back to x.

To evaluate $\bar{Z}_r(x)$, we first observe that

$$\bar{Z}_r(x) = \bar{j}_x T + 2ax, \tag{6}$$

where $\bar{j}_x$ is the (equilibrium) mean number of requests with track addresses between 0 and x which are served per cycle. But the mean number served per cycle must equal the mean number arriving per cycle. Hence,

$$\bar{j}_x = \lambda \bar{c} \int_0^x f(t)\,dt = \lambda \bar{c} F(x), \tag{7}$$

where $\bar{c}$ is the mean cycle time. Now the mean cycle time can be written as

$$\bar{c} = \bar{m}_c T + 2a, \tag{8}$$

where $\bar{m}_c$ is the (equilibrium) mean number served in a cycle. Since the mean number served in $\bar{c}$ must be the mean number arriving in $\bar{c}$,

$$\bar{m}_c = \lambda \bar{c}. \tag{9}$$

Substituting (9) into (8) we find

$$\bar{c} = \frac{2a}{1 - \lambda T}. \tag{10}$$

Substituting (10) into (7) and then (7) into (6), we get

$$\bar{Z}_r(x) = \frac{2a\lambda T}{1 - \lambda T} F(x) + 2ax. \tag{11}$$

Substituting (11) into (5), rearranging and taking the limit $\Delta x \to 0$ yields

$$y_r'(x) = a + \frac{2a\lambda^2 T^2}{1 - \lambda T} f(x)F(x) + 2a\lambda T x f(x).$$

Integrating and using $\bar{y}_r(0) = 0$, we have

$$\bar{y}_r(x) = ax + \frac{a\lambda^2 T^2}{1 - \lambda T} F^2(x) + 2a\lambda T \int_0^x t f(t)\,dt. \tag{12}$$

We can similarly find $\bar{y}_l(x) \equiv \lim_{n\to\infty} \bar{y}_{2n}(x)$, the mean time in equilibrium for the head to move from 1 to $1 - x$. Specifically we obtain

$$\bar{y}_l(x) = ax + \frac{a\lambda^2 T^2}{1 - \lambda T}[1 - F(1 - x)]^2 + 2a\lambda T \int_0^x t f(1 - t)\,dt. \tag{13}$$

Using the results in (12) and (13) we shall now develop an expression for the mean waiting time of requests arriving during statistical equilibrium.

Suppose we observe the system during statistical equilibrium. Let $\bar{\omega}(x)$ be the mean time for the head to move from its current position to x. Thus $\bar{\omega}(x)$ is the mean (virtual) waiting time for a request which arrives to position x. To calculate $\bar{\omega}(x)$, we first condition it on the position and direction of the head at the time of arrival. Let $\bar{\omega}_r(x|v)$ be the mean time for the head to move to x if it is at v and traveling right when the request arrives.

We have

$$\bar{\omega}_r(x|v) = \begin{cases} \bar{y}_r(x) - \bar{y}_r(v), & 0 \leqq v < x \leqq 1, \\ \bar{y}_r(1) - \bar{y}_r(v) + \bar{y}_l(1 - x), & 0 \leqq x \leqq v \leqq 1. \end{cases} \tag{14}$$

To compute

$$\bar{\omega}_r(x) = \int_0^1 \bar{\omega}_r(x|v)\, dP_r(v) \tag{15}$$

we need the stationary probability distribution $P_r(v)$ that the disk head is in the interval $(0, v)$ given that it is moving right. We identify $P_r(v)$ with the expected fraction of time the head spends in the interval $(0, v)$ in statistical equilibrium. Accordingly,

$$P_r(v) = \bar{y}_r(v)/\bar{y}_r(1) \tag{16}$$

with the corresponding density function

$$p_r(v) = \bar{y}_r'(v)/\bar{y}_r(1). \tag{17}$$

Substituting into (15) the expressions in (14) and (17) and then carrying out the integration, we have

$$\bar{\omega}_r(x) = \{\bar{y}_r^2(x) - \bar{y}_r(x)\bar{y}_l(1 - x) - \bar{y}_r(1)[\bar{y}_r(x) - \bar{y}_l(1 - x)] + \tfrac{1}{2}\bar{y}_r^2(1)\}/\bar{y}_r(1). \tag{18}$$

Similarly we find

$$\bar{\omega}_l(x) = \{\bar{y}_l^2(1 - x) - \bar{y}_r(x)\bar{y}_l(1 - x) + \bar{y}_l(1)[\bar{y}_r(x) - \bar{y}_l(1 - x)] + \tfrac{1}{2}y_l^2(1)\}/\bar{y}_l(1). \tag{19}$$

Now, q_r, the probability that a random request finds the head moving to the right is the proportion of time the head spends moving to the right. Hence,

$$q_r = \frac{\bar{y}_r(1)}{\bar{y}_r(1) + \bar{y}_l(1)}. \tag{20}$$

Similarly,

$$q_l = \frac{\bar{y}_l(1)}{\bar{y}_r(1) + \bar{y}_l(1)}. \tag{21}$$

The unconditional waiting time can be expressed as

$$\bar{\omega}(x) = \bar{\omega}_r(x)q_r + \bar{\omega}_l(x)q_l$$

which becomes, after substituting the results in (18)–(21),

$$\bar{\omega}(x) = \frac{1}{\bar{y}_r(1) + \bar{y}_l(1)}\{[\bar{y}_r(x) - \bar{y}_l(1 - x)]^2 - [\bar{y}_r(x) - \bar{y}_l(1 - x)][\bar{y}_r(1) - \bar{y}_l(1)] + \tfrac{1}{2}[\bar{y}_r^2(1) + y_l^2(1)]\}.$$

Using expressions (12) and (13) to eliminate the functions $\bar{y}_r$ and $\bar{y}_l$ we obtain after tedious but routine manipulations

$$\bar{\omega}(x) = \frac{2a}{1 - \lambda T}\{[x - \tfrac{1}{2})(1 - \lambda T) + \lambda T(F(x) - \tfrac{1}{2})]^2 + \tfrac{1}{4}\}. \tag{22}$$

This is the basic result we have been seeking for the SCAN rule. Before proceeding to the analysis of the FSCAN rule we discuss the meaning of (22).

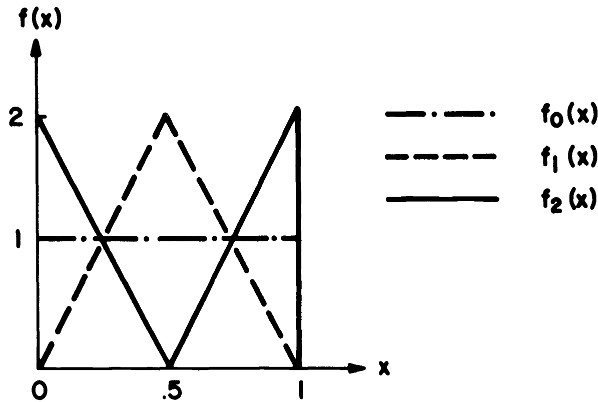
First, it will be convenient to rewrite (22) as follows:

$$\bar{\omega}(x) = \frac{a}{2(1 - \lambda T)} + \frac{2a[h(x)]^2}{1 - \lambda T}, \tag{23}$$

where

$$h(x) = (x - \tfrac{1}{2})(1 - \lambda T) + \lambda T(F(x) - \tfrac{1}{2}). \tag{24}$$

Since $0 \leqq \lambda T < 1$, we see that $h(x)$ is a convex combination (weighted average) of the two functions $x - \frac{1}{2}$ and $F(x) - \frac{1}{2}$. Both of these functions increase from



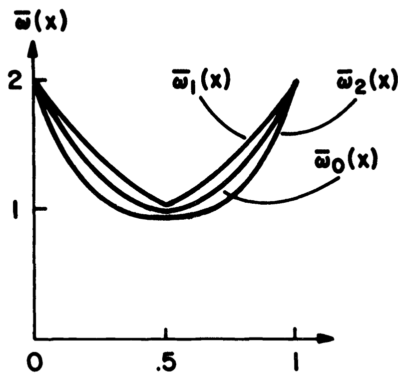


FIG. 2. *Examples for symmetric densities*

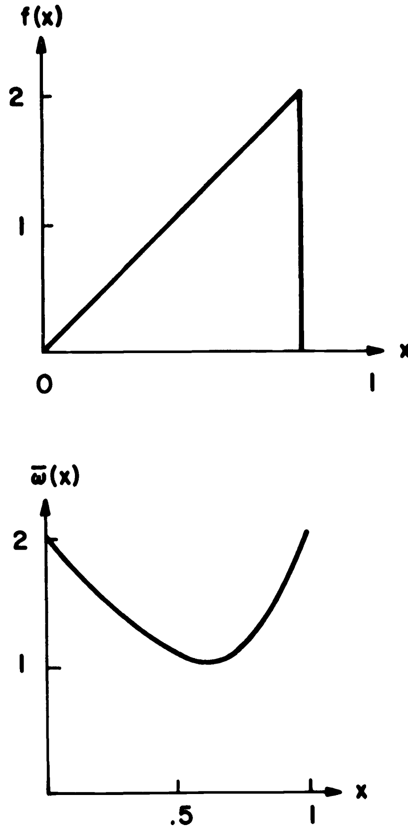


FIG. 3. *An asymmetric density*

$-\frac{1}{2}$ to $\frac{1}{2}$ as x varies from 0 to 1, and in fact $h(x)$ is strictly increasing from $-\frac{1}{2}$ to $\frac{1}{2}$ on this range. Thus $h(x)$ has a unique zero, say at x_0, in the interval (0, 1). Therefore, it is seen that for any density $f(x)$, $\bar{\omega}(0) = \bar{\omega}(1) = a/(1 - \lambda T)$; that $\bar{\omega}(x)$ is strictly decreasing for $0 \leqq x \leqq x_0$, and strictly increasing for $x_0 \leqq x \leqq 1$; and that the minimum value of $\bar{\omega}(x)$ is $\bar{\omega}(x_0) = a/[2(1 - \lambda T)]$. Further, if $f(x)$ is symmetric about $x = \frac{1}{2}$, then $x_0 = \frac{1}{2}$ and $\bar{\omega}(x)$ is also symmetric about $x = \frac{1}{2}$. (See Fig. 2, discussed below.)

It is interesting to compare the waiting time $\bar{\omega}(x)$ for any density $f(x)$ symmetric about $x = \frac{1}{2}$ with the waiting time $\bar{\omega}_0(x)$ for $f_0(x) = 1$; i.e., for a uniform distribution of track addresses. In the latter case $F_0(x) = x$, so that (23) becomes

$$\bar{\omega}_0(x) = \frac{a}{2(1 - \lambda T)} + \frac{2a(x - \frac{1}{2})^2}{1 - \lambda T}.$$

If the distribution function $F(x)$ associated with $f(x)$ satisfies $F(x) \leqq x$ for $0 \leqq x \leqq \frac{1}{2}$, i.e., requests for service tend to crowd toward the center of the disk, it is easily seen from (24) that $\bar{\omega}(x) \geqq \bar{\omega}_0(x)$, i.e., discrimination at the extremities of the disk will tend to increase. On the other hand, if $F(x) \geqq x$ for $0 \leqq x \leqq \frac{1}{2}$,

i.e., requests tend to crowd toward the extremities of the disk, we can see from (24) that $\bar{\omega}(x) \leqq \bar{\omega}_0(x)$, i.e., discrimination at the extremities of the disk will tend to decrease. These properties are shown in Fig. 2 for examples concentrating addresses in the center ($f_1(x)$), uniformly ($f_0(x)$), and at the extremities ($f_2(x)$) of the disk. (In Fig. 3 is shown $\bar{\omega}(x)$ for an asymmetric density $f(x) = 2x$.) This somewhat counter-intuitive result can be understood if we realize that when requests for service tend to crowd toward the center of the disk, the head will rarely visit the extremities, and thus an arrival there will surely have a long wait, whereas when requests for service tend to crowd near the extremities, the head will spend much time there, and an arrival near one end has a good chance of experiencing a very short wait.

Formula (24) shows quite nicely the effect of loading on $\bar{\omega}(x)$. Under heavy loading (λT near 1), prominence is given to the term $F(x) - \frac{1}{2}$, while under light loading, $F(x) - \frac{1}{2}$ has little effect, a result which is to be expected. It is also seen from (23) that $\bar{\omega}(x)$ is directly proportional to a, the time it takes the head to move across the disk without servicing any requests, another result which is to be expected.

4. Analysis of the FSCAN rule. To analyze the FSCAN system, we first use a Markov chain argument to find the (equilibrium) first two moments ($\bar{n}_c, \bar{n}_c^2$) of the number of requests serviced in a crossing. Let X_n be the number of requests serviced in the nth crossing. Let $\{q_j\}$ denote the stationary probability distribution and let

$$q_{ij} = \Pr\{X_{n+1} = j | X_n = i\}$$

denote the one-step transition probabilities. It is easily verified that the Markov chain $\{X_n\}$ is homogeneous; i.e., the q_{ij} are independent of the time parameter n. Moreover, for all $i \geqq 0$ and $j \geqq 0$, q_{ij} is simply the probability of j Poisson arrivals in $t_i = iT + a$. Thus

$$q_{ij} = \frac{(\lambda t_i)^j}{j!} e^{-\lambda t_i}. \tag{25}$$

From the equation defining the stationary distribution

$$q_j = \sum_{i=0}^{\infty} q_{ij} q_i, \qquad j = 0, 1, 2, \cdots,$$

and the generating function

$$Q(z) = \sum_{j=0}^{\infty} q_j z^j$$

we obtain

$$Q(z) = \sum_{i=0}^{\infty} q_i \sum_{j=0}^{\infty} q_{ij} z^j.$$

Substituting (25) into the previous expression and simplifying we get

$$Q(z) = \sum_{i=0} q_i \varepsilon^{-\lambda t_i (1-z)}.$$

Substituting $t_i = iT + a$ we arrive at

$$Q(z) = \varepsilon^{-\lambda a(1-z)} Q(\varepsilon^{-\lambda T(1-z)}). \tag{26}$$

From (26) we can find the moments by differentiation. Specifically, we have the first two moments

$$\bar{n}_c = \frac{\lambda a}{1 - \lambda T}, \tag{27}$$

$$\bar{n}_c^2 = \frac{(\lambda a)^2 + 2\lambda^2 a T \bar{n}_c + \bar{n}_c}{1 - (\lambda T)^2}. \tag{28}$$

Note from (27) that the mean cycle time $2\bar{n}_c/\lambda$ is the same as that given by (10); this we should expect on the basis of the same arguments given in support of the value in (10).

Now suppose we observe the system at a random point in equilibrium. As in SCAN, let $\bar{\omega}(x)$ be the mean (virtual) waiting time for a request which arrives at x. Thus, $\bar{\omega}(x)$ is the sum of the mean time for the head to complete the present crossing plus the mean time the head spends in the next crossing (both traveling and processing requests in front of x) before position x is reached.

Then

$$\bar{\omega}(x) = \sum_{n=0} p_n \bar{\omega}_n(x), \tag{29}$$

where p_n is the stationary probability that a random arrival finds the system executing a crossing that serves n requests, and where $\bar{\omega}_n(x)$ is the mean (virtual) waiting time if the random arrival finds the system executing a crossing that services n requests. Since the direction of head motion is equally probably left or right, and since the present crossing is of length $t_n = nT + a$, we have

$$\bar{\omega}_n(x) = \frac{t_n}{2} + \frac{1}{2}\left[\lambda T t_n \int_0^x f(t)\,dt + ax\right] + \frac{1}{2}\left[\lambda T t_n \int_x^1 f(t)\,dt + a(1-x)\right]$$

or

$$\bar{\omega}_n(x) = \frac{t_n}{2}(1 + \lambda T) + \frac{a}{2} \tag{30}$$

which we note is independent of x and $f(x)$. By means of the arguments used to justify (16) we can establish the following relation between p_n and q_n:

$$p_n = \frac{nT + a}{\bar{n}_c T + a} q_n. \tag{31}$$

In particular, in a large number of crossings, q_n is approximately that fraction of crossings which serves n requests. Since each such crossing requires exactly t_n time units we arrive at $t_n q_n / \sum_{n=0}^{\infty} t_n q_n$ as the fraction of time spent in crossings servicing n requests. On substituting for t_n we get (31). Substituting (30) and (31) into (29) we get

$$\bar{\omega}(x) = \frac{a}{2} + \frac{1 + \lambda T}{2(\bar{n}_c T + a)} \sum_{n=0}^{\infty} t_n^2 q_n$$

$$= \frac{a}{2} + \frac{1 + \lambda T}{2(\bar{n}_c T + a)} \sum_{n=0}^{\infty} [T^2 n^2 q_n + 2aTn q_n + a^2 q_n].$$

Using the moments given in (27) and (28) we find on substitution and simplification

$$\bar{\omega}(x) = \frac{a + \lambda T^2/2}{1 - \lambda T}. \tag{32}$$

This is the expression to compare with (24) for the SCAN system. In particular, as can be seen from (24), $\bar{\omega}(x) \leqq \bar{\omega}(1) = a/(1 - \lambda T)$ for all x and $f(x)$. Therefore, the FSCAN result always exceeds the SCAN result by at least $\lambda T^2/2(1 - \lambda T)$. Since the performance difference increases rapidly with the loading factor λT $(0 < \lambda T < 1)$, we have a strong argument for the use of the SCAN policy.

5. Final remarks. By means of an idealized mathematical model of disk seek operations we have obtained precise results for the mean response times arising with the SCAN and FSCAN servicing rules. The principal conclusions are that SCAN produces uniformly better response times than FSCAN, but that SCAN discriminates against requests for addresses at the extremities of the address space. This discrimination was found to be less for those address distributions concentrating seeks at the extreme addresses.

The most stringent assumption made in modeling head motion has been the requirement that the head can not reverse direction until it encounters a boundary. It is not easy to see what net effect this simplification has on the relative performance of SCAN and FSCAN; however, the simplification becomes less unrealistic as the loading increases or when address distributions are assumed which concentrate disk operations at the extreme addresses. In any case, the removal of the simplification makes the resulting model far less tractable, and a satisfactory approach has not yet been found.

It is worth mentioning that the NSCAN rule discussed in [3] is a generalization of the FSCAN rule that we have examined. NSCAN operates as FSCAN except that a maximum of N requests can be served in a crossing. Two specific models can be assumed: (a) there is a bounded queueing facility that can hold at most N requests, or (b) there is no bound on the number of requests waiting, but at most N can be serviced in a crossing. In either case, in the limit $N \to \infty$ we have the FSCAN rule. In this paper an analysis of NSCAN rules was not thought to be of significant interest since their performance must compare less favorably with SCAN than that of FSCAN. However, extensions of the FSCAN analysis to these rules would appear to be routine, although more elaborate.

REFERENCES

[1] E. G. Coffman, Jr., *Analysis of a drum input/output queue under scheduled operation in a paged computer system*, J. Assoc. Comput. Mach., 16 (1969), pp. 73–90. (Erratum: (1969), p. 646.

[2] P. J. Denning, *Effects of scheduling on file memory operations*, Proc. AFIPS SJCC, vol. 30, Thompson Books, Inc., Washington, D. C., 1966, pp. 9–21.

[3] H. Frank, *Analysis and optimization of disk storage devices for time-sharing systems*, J. Assoc. Comput. Mach., 16 (1969), pp. 602.

[4] T. J. Teorey and T. B. Pinkerton, *A Comparative Analysis of Disk Scheduling Policies*, Proc. 3rd Symposium on Operations Systems Principals, Digital Systems Lab., Stanford University, Stanford, Calif., 1971, pp. 114–121.

SIAM J. COMPUT.
Vol. 1, No. 4, December 1972

ON LANGUAGES ACCEPTED IN POLYNOMIAL TIME*

RONALD V. BOOK†

Abstract. The family NP (P) of languages accepted by nondeterministic (deterministic) Turing machines operating in polynomial time is distinct from many well-known families of languages defined by tape-bounded or time-bounded Turing machines. In particular, it is shown that NP (P) is not equal to the family of context-sensitive languages, the family of languages accepted by deterministic linear bounded automata, or the family of languages accepted by deterministic Turing machines operating within exponential time.

Key words. Automata-based complexity, complexity classes of formal languages, polynomial time, time-bounded machines, tape-bounded machines.

Introduction. In automata-based complexity much effort has been expended in attempting to answer questions regarding time–tape trade-offs, deterministic simulation of nondeterministic machines, etc. Recently nondeterministic Turing machines operating in polynomial time have been studied in order to classify the "relative complexity" of certain (nonautomata theoretic) problems [6], [12]. The relationship between the family NP (P) of languages accepted by nondeterministic (deterministic) Turing machines operating in polynomial time and other complexity classes of languages is of interest in order to further classify these problems.

It has been shown [6] that any language in NP is "polynomially reducible" to various languages accepted by deterministic linear bounded automata and hence by deterministic Turing machines operating in exponential time. Here we compare P and NP to many well-known families of languages defined by tape-bounded or time-bounded Turing machines and show that these families are distinct.

In § 1 we compare P and NP with certain tape-bounded classes and in § 2 with certain time-bounded classes. In § 3 we give quite different proofs of some results from § 1 and 2, and in § 4 we provide an outline of another approach to these results.

1. We begin by defining the families of languages under investigation. The functions f used to bound the amount of time or tape used in a Turing machine's computation are such that for all $x, y \geqq 0, f(x) \geqq x$ and $f(x) + f(y) \leqq f(x + y)$. Such functions are nondecreasing. Further, most functions used are "self-computable" in the sense that there is a Turing machine M_1 which upon input w runs for precisely $f(|w|)$ steps and halts, and a machine M_2 which upon input w marks precisely $f(|w|)$ consecutive tape squares and halts.[1] (See [2].)

DEFINITION. Let f be a bounding function. For a Turing acceptor M, $L(M)$ is a set of strings accepted by M.

* Received by the editors May 23, 1972.

† Center for Research in Computing Technology, Division of Engineering and Applied Physics, Harvard University, Cambridge, Massachusetts 02138. This research was supported in part by the National Aeronautics and Space Administration under Grant NGR-22-007-176 and by the National Science Foundation under Grant GJ-30409.

[1] For a string w, $|w|$ is the length of w.

(i) A multitape Turing acceptor M operates within time bound f if for each input string w accepted by M, every accepting computation of M on w has no more than $\max(|w|, f(|w|))$ steps.

(ii) Define $\mathrm{NTIME}(f) = \{L(M) | M$ is a nondeterministic multitape Turing acceptor which operates within time bound $f\}$ and $\mathrm{DTIME}(f) = \{L(M) | M$ is a deterministic multitape Turing acceptor which operates within time bound $f\}$.

(iii) A multitape Turing acceptor M operates within tape bound f if for each input string w accepted by M, every accepting computation of M on w visits no more than $\max(|w|, f(|w|))$ tape squares on any one of its storage tapes.

(iv) Define $\mathrm{NTAPE}(f) = \{L(M) | M$ is a nondeterministic Turing acceptor which operates within tape bound $f\}$ and $\mathrm{DTAPE}(f) = \{L(M) | M$ is a deterministic Turing acceptor which operates within tape bound $f\}$.

The specific families of languages which we study in this paper can be formally defined using the above notation.

DEFINITION. The family of languages accepted by nondeterministic Turing machines which operate in polynomial time is $\mathrm{NP} = \bigcup_{k=1}^{\infty} \mathrm{NTIME}(x^k)$. The family of languages accepted by deterministic Turing machines which operate in polynomial time is $\mathrm{P} = \bigcup_{k=1}^{\infty} \mathrm{DTIME}(x^k)$.

In [6] the family P is referred to as $\mathscr{L}_*$ and the family NP as $\mathscr{L}_*^+$.

There are several other families of languages to which we frequently refer.

DEFINITION. Let i be the identity function, $i(x) = x$.

(i) The family of context-sensitive languages is the family $\mathrm{CS} = \mathrm{NTAPE}(i)$.

(ii) The family of languages accepted by deterministic linear bounded automata is $\mathrm{DLBA} = \mathrm{DTAPE}(i)$.

(iii) The family of languages accepted by deterministic Turing machines which operate in polynomial storage is

$$\mathscr{T} = \bigcup_{k=1}^{\infty} \mathrm{DTAPE}(x^k).$$

(iv) The family of quasi-realtime languages is the family $\mathrm{Q} = \mathrm{NTIME}(i)$ [1].

The family DLBA is the family of languages whose characteristic functions are in $\mathscr{E}^2$ (where $\mathscr{E}^2$ is the subclass of primitive recursive functions defined by Grzegorczyk). Sometimes this family is referred to as the family of deterministic $\mathscr{E}^2$ relations or $\mathscr{E}^2_*$. Then the family CS is referred to as the family of nondeterministic $\mathscr{E}^2$ relations.

Recall that for any function f, $\mathrm{NTAPE}(f) \subseteq \mathrm{DTAPE}(f^2)$ [13]; thus,

$$\mathscr{T} = \bigcup_{k=1}^{\infty} \mathrm{NTAPE}(x^k).$$

Further, for any real numbers, $1 \leqq r < s$, $\mathrm{DTAPE}(x^r) \subsetneq \mathrm{DTAPE}(x^s)$ [14]. Hence, there is no real number t such that $\mathscr{T} = \mathrm{DTAPE}(x^t)$ or $\mathscr{T} = \mathrm{NTAPE}(x^t)$. These facts are helpful in establishing our results comparing P and NP to various families defined by tape-bounded machines.

THEOREM 1. *If there exists a real number* $r \geqq 1$ *such that* $\mathrm{DTAPE}(x^r) \subseteq P$, *then* $\mathscr{T} = \mathrm{P} = \mathrm{NP}$. *Similarly, if there exists a real number* $r \geqq 1$ *such that* $\mathrm{DTAPE}(x^r) \subseteq \mathrm{NP}$, *then* $\mathscr{T} = \mathrm{NP}$.

Proof. We give the proof for the first part, the proof for the second part being identical.

First, note that $\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathscr{T}$, since a machine can use no more tape than it does time. Second, suppose $r \geqq 1$ is such that $\mathrm{DTAPE}(x^r) \subseteq \mathrm{P}$. Without loss of generality, assume that r is rational, say $r = p/q$, where $p \geqq q \geqq 1$ are integers. To show that $\mathscr{T} \subseteq \mathrm{P}$ it is sufficient to show that $\mathrm{DTAPE}(x^s) \subseteq \mathrm{P}$ for $s = 2qr = 2p$.

Let M_1 be a deterministic Turing machine which operates within tape bound x^s. Let $L_1 = L(M_1)$. If Σ is a finite alphabet such that $L_1 \subseteq \Sigma^*$, let c be a new symbol, $c \notin \Sigma$. Let $L_2 = \{wc^m | w \in L_1, |wc^m| = |w|^s\}$. Since M_1 operates within tape bound x^s, it is clear that one can construct a deterministic linear bounded automaton M_2 from M_1 such that $L(M_2) = L_2$. Thus, $L_2 \in \mathrm{DLBA} \subseteq \mathrm{DTAPE}(x^r) \subseteq \mathrm{P}$. But if $L_2 \in \mathrm{P}$, then $L_1 \in \mathrm{P}$ since the difference (with respect to time) between recognizing L_1 and L_2 is simply the computation of a polynomial. Hence, $\mathrm{DTAPE}(x^s) \subseteq \mathrm{P}$.

Since for any real number $r \geqq 1$, $\mathrm{DTAPE}(x^r) \subseteq \mathrm{NTAPE}(x^r)$, the result of Theorem 1 will hold if $\mathrm{NTAPE}(x^r)$ is substituted for $\mathrm{DTAPE}(x^r)$.

THEOREM 2. *There is no pair* (r, s) *of real numbers,* $1 \leqq r \leqq s$, *such that* $\mathrm{DTAPE}(x^r) \subseteq \mathrm{P} \subseteq \mathrm{DTAPE}(x^s)$ *or* $\mathrm{NTAPE}(x^r) \subseteq \mathrm{P} \subseteq \mathrm{NTAPE}(x^s)$. *Similarly, there is no pair* (r, s) *of real numbers,* $1 \leqq r \leqq s$, *such that* $\mathrm{DTAPE}(x^r) \subseteq \mathrm{NP} \subseteq \mathrm{DTAPE}(x^s)$ *or* $\mathrm{NTAPE}(x^r) \subseteq \mathrm{NP} \subseteq \mathrm{NTAPE}(x^s)$.

Proof. If $\mathrm{DTAPE}(x^r) \subseteq \mathrm{P}$, then by Theorem 1, $\mathscr{T} = \mathrm{P}$. Hence, if $\mathrm{P} \subseteq \mathrm{DTAPE}(x^s)$, then $\mathscr{T} = \mathrm{DTAPE}(x^s)$. But as noted above $\mathrm{DTAPE}(x^s) \subsetneq \mathscr{T}$. Thus, either $\mathrm{DTAPE}(x^r) \nsubseteq \mathrm{P}$ or $\mathrm{P} \nsubseteq \mathrm{DTAPE}(x^s)$. The proof for NP is identical. Since $\mathrm{DTAPE}(x^r) \subseteq \mathrm{NTAPE}(x^r)$ and $\mathrm{NTAPE}(x^s) \subseteq \mathrm{DTAPE}(x^{2s})$, if $\mathrm{NTAPE}(\cdot)$ is substituted for $\mathrm{DTAPE}(\cdot)$ throughout, then the results still hold.

COROLLARY. *For any real number* $r \geqq 1$,

(i) $\mathrm{P} \neq \mathrm{DTAPE}(x^r)$,
(ii) $\mathrm{P} \neq \mathrm{NTAPE}(x^r)$,
(iii) $\mathrm{NP} \neq \mathrm{DTAPE}(x^r)$,
(iv) $\mathrm{NP} \neq \mathrm{NTAPE}(x^r)$.

In particular, $\mathrm{P} \neq \mathrm{DLBA}$, $\mathrm{P} \neq \mathrm{CS}$, $\mathrm{NP} \neq \mathrm{DLBA}$, *and* $\mathrm{NP} \neq \mathrm{CS}$.

It is not known whether $\mathrm{DLBA} \subseteq \mathrm{P}$ or $\mathrm{DLBA} \subseteq \mathrm{NP}$. From Theorem 2, we see that if for some real number s, $\mathrm{P} \subseteq \mathrm{DTAPE}(x^s)$ or $\mathrm{NP} \subseteq \mathrm{DTAPE}(x^s)$, then there is no real number $r \geqq 1$ such that $\mathrm{DTAPE}(x^r) \subseteq \mathrm{P}$ or $\mathrm{DTAPE}(x^r) \subseteq \mathrm{NP}$ (or $\mathrm{NTAPE}(x^r) \subseteq \mathrm{P}$ or $\mathrm{NTAPE}(x^r) \subseteq \mathrm{NP}$).

2. We turn to comparing P and NP with families defined by time-bounded machines.

The first result is analogous to Theorem 1.

THEOREM 3. *If there exists a real number* $p > 1$ *such that* $\mathrm{DTIME}(p^x) \subseteq \mathrm{NP}$, *then*

$$\bigcup_{k=1}^{\infty} \mathrm{DTIME}(k^x) \subsetneq \mathrm{NP} = \mathscr{T} = \bigcup_{j=1}^{\infty} \mathrm{DTIME}(2^{x^j}).$$

COROLLARY. *There is no real number* $p > 1$ *such that* $\mathrm{DTIME}(p^x) = \mathrm{NP}$. *Further,* $\bigcup_{k=1}^{\infty} \mathrm{DTIME}(k^x) \neq \mathrm{NP}$.

Note that $\bigcup_{k=1}^{\infty} \mathrm{DTIME}(k^x)$ is the family of languages accepted by deterministic Turing machines operating in exponential time. In [5] it is shown that $\bigcup_{k=1}^{\infty} \mathrm{DTIME}(k^x)$ is precisely the family of languages accepted by (deterministic or nondeterministic) auxiliary pushdown machines which operate within tape

bound $i(x) = x$. Thus, $\mathrm{CS} \subseteq \bigcup_{k=1}^{\infty} \mathrm{DTIME}(k^x)$. It is not known whether $\mathrm{NP} \subseteq \bigcup_{k=1}^{\infty} \mathrm{DTIME}(k^x)$.

To obtain Theorem 3 we establish two lemmas. In both cases the proof is similar to that of Theorem 1.

LEMMA 1. *If there exists a real number $p > 1$ such that* $\mathrm{DTIME}(p^x) \subseteq \mathrm{NP}$, *then* $\mathrm{DTIME}((p^2)^x) \subseteq \mathrm{NP}$.

Proof. Suppose such a p exists. Let $L_1 \in \mathrm{DTIME}((p^2)^x)$. Let M_1 be a deterministic Turing machine such that $L(M_1) = L_1$ and M_1 operates within time bound $(p^2)^x$. Let Σ be a finite alphabet such that $L_1 \subseteq \Sigma^*$ and let c be a new symbol, $c \notin \Sigma$.

If $L_2 = \{wc^{|w|} \mid w \in L_1\}$, then one can construct a deterministic Turing machine M_2 from M_1 such that $L(M_2) = L_2$ and such that the running time of M_2 is bounded by $(p^2)^{|w|}$. Since $|wc^{|w|}| = 2|w|$, this means that M_2 operates within time bound p^x. Hence, $L_2 = L(M_2) \in \mathrm{DTIME}(p^x)$. Since $\mathrm{DTIME}(p^x) \subseteq \mathrm{NP}$, this means that $L_2 \in \mathrm{NP}$. But if $L_2 \in \mathrm{NP}$, then $L_1 \in \mathrm{NP}$ since the difference (with respect to time) between recognizing L_1 and L_2 is simply the computation of a polynomial. Hence, $\mathrm{DTIME}((p^2)^x) \subseteq \mathrm{NP}$.

LEMMA 2. *If* $\bigcup_{k=1}^{\infty} \mathrm{DTIME}(k^x) \subseteq \mathrm{NP}$, *then* $\bigcup_{j=1}^{\infty} \mathrm{DTIME}(2^{x^j}) \subseteq \mathrm{NP}$.

Proof. If $L_1 \in \bigcup_{j=1}^{\infty} \mathrm{DTIME}(2^{x^j})$, then there is a deterministic Turing machine M_1 and a constant j such that $L(M_1) = L_1$ and M_1 operates within time bound 2^{x^j}. Let Σ be a finite alphabet such that $L_1 \subseteq \Sigma^*$ and let c be a new symbol, $c \notin \Sigma$.

If

$$L_2 = \{wc^m \mid w \in L_1, |wc^m| = |w|^j\},$$

then one can construct a deterministic Turing machine M_2 such that $L(M_2) = L_2$ and such that the running time of M_2 is bounded by $2^{|w|^j}$. Since $|wc^m| = |w|^j$, this means that M_2 operates within time bound 2^x. Hence,

$$L_2 = L(M_2) \in \mathrm{DTIME}(2^x) \subseteq \bigcup_{k=1}^{\infty} \mathrm{DTIME}(k^x).$$

Since $\bigcup_{k=1}^{\infty} \mathrm{DTIME}(k^x) \subseteq \mathrm{NP}$, this means that $L_2 \in \mathrm{NP}$. But if $L_2 \in \mathrm{NP}$, then $L_1 \in \mathrm{NP}$ since the difference between recognizing L_1 and L_2 is simply the computation of a polynomial. Hence $\bigcup_{j=1}^{\infty} \mathrm{DTIME}(2^{x^j}) \subseteq \mathrm{NP}$.

Proof of Theorem 3. Suppose there exists a real number $p > 1$ such that $\mathrm{DTIME}(p^x) \subseteq \mathrm{NP}$. By use of Lemma 1, for any integer $q > 1$, $\mathrm{DTIME}((p^q)^x) \subseteq \mathrm{NP}$. Thus,

$$\bigcup_{k=1}^{\infty} \mathrm{DTIME}(k^x) \subseteq \mathrm{NP}.$$

By Lemma 2, this means that $\bigcup_{j=1}^{\infty} \mathrm{DTIME}(2^{x^j}) \subseteq \mathrm{NP}$. As noted in § 1, $\mathrm{NP} \subseteq \mathscr{T}$. It is clear that

$$\mathscr{T} \subseteq \bigcup_{k=1}^{\infty} \bigcup_{j=1}^{\infty} \mathrm{DTIME}(k^{x^j}).$$

But

$$\bigcup_{k=1}^{\infty} \bigcup_{j=1}^{\infty} \mathrm{DTIME}(k^{x^j}) = \bigcup_{j=1}^{\infty} \mathrm{DTIME}(2^{x^j}),$$

so we have

$$\mathrm{NP} = \mathscr{T} = \bigcup_{j=1}^{\infty} \mathrm{DTIME}(2^{x^j}).$$

From [9], we have

$$\bigcup_{k=1}^{\infty} \mathrm{DTIME}(k^x) \subsetneq \bigcup_{j=1}^{\infty} \mathrm{DTIME}(2^{x^j}).$$

Our other result with respect to time is a statement of two necessary and sufficient conditions for P and NP to be the same.

THEOREM 4. *The following are equivalent:*

(a) P = NP;

(b) P *is closed under nonerasing homomorphic mappings*;

(c) $\mathrm{Q} \subseteq \mathrm{P}$.

Proof. It is clear that the family NP is closed under nonerasing homomorphic mappings so that (a) implies (b). In [1] it is shown that $L \in \mathrm{Q}$ if and only if there exist $L_2 \in \mathrm{DTIME}(i) \subset \mathrm{P}$ and a nonerasing homomorphism h such that $L = \{h(w) | w \in L_2\}$. Thus, (b) implies (c).

To see that (c) implies (a), again we use an argument similar to the proof of Theorem 1. If $L_1 \in \mathrm{NP} = \bigcup_{j=1}^{\infty} \mathrm{NTIME}(x^j)$, then there exist an integer $k \geqq 1$ and a nondeterministic Turing machine M_1 such that $L(M_1) = L_1$ and M_1 operates within time bound x^k. Let Σ be a finite alphabet such that $L_1 \subseteq \Sigma^*$ and let c be a new symbol, $c \notin \Sigma$. Let

$$L_2 = \{wc^m | w \in L_1, |wc^m| = |w|^k\}.$$

Since M_1 operates within time bound x^k, one can construct a machine M_2 from M_1 such that $L(M_2) = L_2$ and M_2 operates within time bound $i(x) = x$. Thus, $L_2 \in \mathrm{Q}$ so $\mathrm{Q} \subseteq \mathrm{P}$ implies $L_2 \in \mathrm{P}$. But if $L_2 \in \mathrm{P}$, then clearly $L_1 \in \mathrm{P}$. Hence, $\mathrm{NP} \subseteq \mathrm{P}$. Since $\mathrm{P} \subseteq \mathrm{NP}$, we have NP = P.

Those familiar with formal language theory may note that condition (b) is equivalent to the assertion that P is an abstract family of languages (AFL).

3. Now we give a totally different proof of part of the corollary to Theorem 2 and the corollary to Theorem 3.

PROPOSITION 1. *For any rational number* $r \geqq 1$, $\mathrm{NP} \neq \mathrm{DTAPE}(x^r)$, $\mathrm{NP} \neq \mathrm{NTAPE}(x^r)$, *and* $\mathrm{NP} \neq \bigcup_{k=1}^{\infty} \mathrm{DTIME}(k^{x^r})$.

Proof. By results in [2], [3], $\mathrm{DTAPE}(x^r)$, $\mathrm{NTAPE}(x^r)$, and $\bigcup_{k=1}^{\infty} \mathrm{DTIME}(k^{x^r})$ are principal AFLs.[2] We show that NP is not a principal AFL, thus obtaining the result.

In [7] it is shown that for any real numbers r, s, if $1 \leqq r < s$, then $\mathrm{NTIME}(x^r) \subsetneq \mathrm{NTIME}(x^s)$. Hence $\mathrm{NTIME}(x)$, $\mathrm{NTIME}(x^2)$, $\mathrm{NTIME}(x^3) \cdots$ forms an infinite heirarchy of families of languages. By results in [2], each $\mathrm{NTIME}(x^k)$ is a principal AFL. Thus by results in [8], $\mathrm{NP} = \bigcup_{k=1}^{\infty} \mathrm{NTIME}(x^k)$ is not a principal AFL.

[2] An abstract family of languages (AFL) is a family of languages closed under the following operations: union, concatenation, Kleene +, intersection with regular sets, nonerasing homomorphic mappings (a homomorphism $h: \Sigma^* \to \Delta^*$ is nonerasing if $h(w) = e$ implies $w = e$), and inverse homomorphic mappings. A family of languages is a principal AFL if it is the smallest AFL containing some given language [8].

COROLLARY. $NP \subsetneq \bigcup_{k=1}^{\infty} NTIME(k^x)$.

Proof. By results in [2], $\bigcup_{k=1}^{\infty} NTIME(k^x)$ is a principal AFL. As shown above, NP is not a principal AFL so $NP \neq \bigcup_{k=1}^{\infty} NTIME(k^x)$.

The family $\bigcup_{k=1}^{\infty} NTIME(k^x)$ of languages accepted by nondeterministic Turing machines operating in exponential time is the class of spectra of formulas of first order logic with equality [11].

Results in [2], [3], [8] show that NP cannot be the same as many families which have been studied in automata and formal language theory since NP is not a principal AFL.

4. The results established in §§ 1–3 were discovered by examining the application of "bounded erasing operators" to the families CS, DLBA, P, NP, etc. In particular, we have shown that NP is closed under "polynomial erasing" where neither CS nor DLBA has this property, the closure of either CS or DLBA under polynomial erasing being $\mathscr{T}$. In this section we define some of these notions and rephrase some of the results of [2], [4] in terms of these families. It is hoped that these techniques are applicable to other questions concerning complexity classes of languages, and are helpful in suggesting possible results to others.

We begin by defining the notion of a bounded erasing operator [2].

DEFINITION. If $h: \Sigma^* \to \Delta^*$ is a homomorphism, $L \subseteq \Sigma^*$, and f is a function such that for some $k > 0$ and all $w \in L$, $|w| \leqq kf(|h(w)|)$, then h is f-bounded on L. For any family $\mathscr{L}$ of languages[3] and any function f, the *image of $\mathscr{L}$ under f-bounded erasing* is $H_f[\mathscr{L}] = \{h(L) | L \in \mathscr{L}$ and h is a homomorphism which is f-bounded on $L\}$.

The first use of the bounded erasing operators is shown in the following "representation" results from [2].

PROPOSITION 2. *For any function f,*

$$NTIME(f) = H_f[Q], \quad NTAPE(f) = H_f[CS], \text{ and } DTAPE(f) = H_f[DLBA].$$

Hence, $NTIME(f) \subseteq DTAPE(f)$.

In [4] the composition of operators H_f and H_g is studied. Sufficient conditions on f, g, and $\mathscr{L}$ such that $H_g[H_f[\mathscr{L}]] = H_{f \circ g}[\mathscr{L}]$ (where $(f \circ g)(x) = f(g(x))$) are established. Applied to the families of languages and bounding functions studied here, the following results become important.

PROPOSITION 3. *For rational numbers $r, s \geqq 1$, if $f(x) = x^r$ and $g(x) = x^s$ are bounding functions, then*

(i) $H_g[H_f[Q]] = H_g[NTIME(f)] = NTIME(f \circ g)$;
(ii) $H_g[H_f[CS]] = H_g[NTAPE(f)] = NTAPE(f \circ g)$;
(iii) $H_g[H_f[DLBA]] = H_g[DTAPE(f)] = DTAPE(f \circ g)$.

A family $\mathscr{L}$ is closed under "polynomial erasing" if for any real number $r > 1$, if $f(x) = x^r$, then $H_f[\mathscr{L}] \subseteq \mathscr{L}$. As shown in [4], there is no real number $s \geqq 1$ such that $NTAPE(x^s)$ or $DTAPE(x^s)$ is closed under H_f, i.e., $\mathscr{L} \subsetneq H_f[\mathscr{L}]$. On the other hand, the proof of Theorem 1 shows that NP is closed under polynomial erasing. The proof of Theorem 3 shows that $\bigcup_{k=1}^{\infty} DTIME(k^x)$ is not closed under polynomial erasing.

---

[3] We assume that (i) if $L \in \mathscr{L}$, then there is a finite alphabet Σ such that $L \subseteq \Sigma^*$, and (ii) there exists $L \in \mathscr{L}$ such that $L \neq \emptyset$.

In [4] the results on bounded erasing are used to show the following result announced in [10]: for any pair of real numbers (r, s) such that $1 \leqq r < s$, $\mathrm{NTAPE}(x^r) \subsetneq \mathrm{NTAPE}(x^s)$.

The results in § 2 show that NP is the smallest AFL which is closed under polynomial erasing and contains P. If NP is simply the smallest AFL containing P, then one can extend results in § 2 and § 3 to show that $\mathrm{NP} \subsetneq \bigcup_{k=1}^{\infty} \mathrm{DTIME}(k^x)$.

The results stated in this section are "representation" and "translation" results. In particular, Proposition 3 is similar to the translation results of [3], [4], [7], [10], [13].

The bounded-erasing operators are "algebraic" and not "measure dependent." It would be interesting to know whether the type of "translation" results obtained here can be found in abstract complexity theory.

Acknowledgment. It is a pleasure to thank Stephen Cook for his criticism of a previous version of this paper.

REFERENCES

[1] R. Book and S. Greibach, *Quasi-realtime languages*, Math. Systems Theory, 4 (1970), pp. 97–111.

[2] R. Book, S. Greibach and B. Wegbreit, *Time- and tape-bounded Turing acceptors and AFLs*, J. Comput. System Sci., 4 (1970), pp. 606–621.

[3] R. Book, S. Greibach, O. Ibarra and B. Wegbreit, *Tape-bounded Turing acceptors and principal AFLs*, Ibid., 4 (1970), pp. 622–625.

[4] R. Book and B. Wegbreit, *A note on AFLs and bounding erasing*, Information and Control, 19 (1971), pp. 18–29.

[5] S. Cook, *Characterizations of pushdown machines in terms of time-bounded computers*, J. Assoc. Comput. Mach., 18 (1971), pp. 4–18.

[6] ———, *The complexity of theorem-proving procedures*, Proc. Third ACM Symposium on Theory of Computing, 1971, pp. 151–158.

[7] ———, *A hierarchy for nondeterministic time complexity*, Proc. Fourth ACM Symposium on Theory of Computing, 1972, pp. 187–192.

[8] S. Ginsburg and S. Greibach, *Principal AFL*, J. Comput. System Sci., 4 (1970), pp. 308–338.

[9] J. Hartmanis and R. Stearns, *On the computational complexity of algorithms*, Trans. Amer. Math. Soc., 117 (1965), pp. 285–306.

[10] O. Ibarra, *A note concerning nondeterministic tape complexities*, submitted for publication. Also, *Abstract* 71T-C22, Notices Amer. Math. Soc., 18 (1971), p. 165.

[11] N. Jones and A. Selman, *Turing machines and the spectra of first-order formulas with equality*, Proc. Fourth ACM Symposium on Theory of Computing, 1972, pp. 157–167.

[12] R. Karp, *Reducibility among combinatorial problems*, Proc. IBM Symposium on Computational Complexity, to appear.

[13] W. Savitch, *Relationships between nondeterministic and deterministic tape complexities*, J. Comput. System Sci., 4 (1970), pp. 177–192.

[14] R. Stearns, J. Hartmanis and P. Lewis, *Hierarchies of memory limited computations*, Conf. Record IEEE Sixth Annual Symposium on Switching Circuit Theory and Logical Design, 1965, pp. 179–190.

SIAM J. COMPUT.
Vol. 1, No. 4, December 1972

A NOTE ON THE INTERSECTION OF COMPLEXITY CLASSES OF FUNCTIONS*

LEONARD J. BASS†

Abstract. The classes of computable functions defined by a bound on the computation time are shown not to be closed under infinite descending intersection.

Key words. Blum theory, computational complexity theory, complexity classes, computable functions.

Of great interest recently has been an investigation of the properties of complexity classes as a way of gaining insight into the computable functions. These classes have been shown to be closed under infinite increasing union [3]. This note answers negatively the question of closure under infinite decreasing intersection of complexity classes of functions.

Let $\lambda_i \phi_i$ be a standard indexing of the partial recursive functions [5]. $\{\Phi_i\}$ is said to be a Blum measure of complexity if $\Phi_i(x)$ is defined if and only if $\phi_i(x)$ is defined and the predicate $\Phi_i(x) \leqq y$ is decidable for all i, x, y. We shall fix ϕ_i and Φ_i for the remainder of this paper. We say $t(x) \leqq s(x)$ almost everywhere (a.e.) if $t(x) \leqq s(x)$ for all but a finite number of x. We say $t(x) \leqq s(x)$ infinitely often (i.o.) if it is not the case that $s(x) < t(x)$ a.e.

$$F(t) = \{i\,;\phi_i \text{ is total and } \Phi_i(x) \leqq t(x) \text{ a.e.}\},$$
$$\mathscr{F}(t) = \{\phi_i\,;\phi_i \text{ is total and } \Phi_i(x) \leqq t(x) \text{ a.e.}\}.$$

We say (following [1]) a set $\{g_1\}$ is a measured set if $g_i(x) \leqq y$ is decidable for all i, x and y (for example $\{\Phi_i\}$). Finally [3] if g is a total recursive function we say h is g-honest if for some $\phi_k = h$ we have $\Phi_k(x) \leqq g(x, \phi_k(x))$ a.e.

An easy result of McCreight and Meyer [3] is that any measured set is g-honest for some monotone total recursive g.

We shall need a result pertaining to the Blum speed-up theorem [1] which is an easy consequence of a theorem proven by Meyer and Fischer [2].

THEOREM 1, [2]. *Let $\lambda xyh(x, y)$ be any total recursive function. Then there is a total recursive function f and a sequence $p_0, p_1, \cdots$ such that*

(i) *for all i, $p_i(x) \geqq h(x, p_{i+1}(x))$ a.e.;*

(ii) *for all i, there is a j such that $\phi_j = f$ and $\Phi_j(x) \leqq p_i(x)$ a.e.; and*

(iii) *if $\phi_j = f$, then there is an i such that $p_i(x) < \Phi_j(x)$ a.e.*

We also need another result of McCreight and Meyer [3]. This says that large output takes a long time to write.

THEOREM [3]. *There exists a total monotone recursive f' such that for all i, $\phi_i(x) \leqq f'(x, \Phi_i(x))$ a.e.*

We are now in a position to prove our result. This theorem says that the classes of functions $\mathscr{F}$ are not closed under infinite descending classes of programs. Robertson [4] has proved the same theorem for the F.

* Received by the editors August 5, 1971.

† Department of Computer Science and Experimental Statistics, University of Rhode Island, Kingston, Rhode Island 02881. This result is taken from the author's doctoral dissertation written at Purdue University under the direction of Paul Young. This work was supported by the National Science Foundation under Grants GP 6120 and GJ 27127.

THEOREM. *There exists a sequence of total functions* $p_0, p_1, \cdots$ *such that for all* $i, p_i(x) \geqq p_{i+1}(x)$ *a.e. and such that there does not exist a function* t *with* $\mathscr{F}(t) = \bigcap_i \mathscr{F}(p_i)$.

Proof. Since $\{\Phi_i\}_{i=0}^{\infty}$ is a measured set, it is g-honest for some total monotone g. Also we have $\phi_i(x) \leqq f'(x, \Phi_i(x))$ a.e. for all i. Let $h(x, y) = \max(f'(x, y), g(x, y))$. Then by condition (ii) of Theorem 1 (due to Meyer–Fischer) there is a sequence $p_0, p_1 \cdots$ and a function f such that for all i there is a j such that $\phi_j = f$ and $\Phi_j(x) \leqq p_i(x)$ a.e. Thus $f \in \bigcap_i \mathscr{F}(p_i)$.

Now suppose for the sake of contradiction that for some t we have $\mathscr{F}(t) = \bigcap_i \mathscr{F}(p_i)$. Then we must have $f \in \mathscr{F}(t)$. Thus there is a $\phi_j = f$ such that $\Phi_j(x) \leqq t(x)$ a.e. By conditions (i), (ii), and (iii) of Theorem 1 there are i, k, and u such that $\phi_k = f$ and

$$h(x, p_u(x)) < h(x, \Phi_k(x)) \leqq h(x, p_{i+1}(x)) \leqq p_i(x) < \Phi_j(x) \quad \text{a.e.}$$

Since Φ_k is a recursive function, we have $\Phi_k = \phi_n$ for some ϕ_n. Since ϕ_n is g-honest, we have

$$\Phi_n(x) \leqq g(x, \phi_n(x)) \leqq h(x, \phi_n(x)) = h(x, \Phi_k(x)) < \Phi_j(x) \leqq t(x) \quad \text{a.e.}$$

We thus have $\phi_n \in \mathscr{F}(t)$. By our choice of f' we have $f'(x, \Phi_m(x)) \geqq \phi_m(x) = \Phi_k(x)$, where ϕ_m is any way of computing Φ_k. We thus have

$$h(x, \Phi_m(x)) \geqq f(x, \Phi_m(x)) \geqq \Phi_k(x) > p_u(x) \geqq h(x, p_{u+1}(x)) \quad \text{a.e.}$$

Thus by the monotonicity of h we have $\Phi_m(x) > p_{u+1}(x)$, where ϕ_m is any way of computing ϕ_n. Thus $\phi_n \notin \bigcap_i \mathscr{F}(p_i)$. But $\phi_n \in \mathscr{F}(t) = \bigcap_i \mathscr{F}(p_i)$, a contradiction.

REFERENCES

[1] M. BLUM, *A machine independent theory of the complexity of recursive functions*, J. Assoc. Comput. Mach., 14 (1967), pp. 322–336.

[2] A. MEYER AND P. FISCHER, *Computational speed-up by effective operators*, J. Symbolic Logic, to appear.

[3] E. MCCREIGHT AND A. MEYER, *Classes of computable functions defined by bounds on computation*, Proc. ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1969, pp. 79–88.

[4] E. L. ROBERTSON, *Complexity classes of partial recursive functions*, Tech. Rep. 123, University of Wisconsin, Madison, 1971.

[5] H. ROGERS, JR., *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York, 1967.

SIAM J. COMPUT.
Vol. 1, No. 4, December 1972

ANALYSIS AND SYNTHESIS OF SORTING ALGORITHMS*

C. L. LIU†

Abstract. The problem of analyzing and synthesizing sorting algorithms is studied. That is, given a sorting algorithm we want to investigate how it works in a step-by-step manner and consequently to assert that it indeed arranges the objects according to a certain ordering relationship, and conversely, given an ordering relationship according to which a set of objects are to be arranged, we want to determine an algorithm that will yield the desired result.

Key words. Sorting algorithms, sorting networks

1. Introduction. By sorting a set of objects[1], we mean to arrange them in such a way that a certain ordering relationship between them is satisfied. An algorithm leading to a correct arrangement of these objects is called a sorting algorithm. With the possible exception of some trivial degenerate cases, all sorting algorithms can be decomposed into steps (phases). In this paper, we study the problem of analyzing and synthesizing sorting algorithms. That is, given a sorting algorithm we want to investigate how it works in a step-by-step manner and consequently to assert that it indeed arranges the objects according to a certain ordering relationship, and conversely, given an ordering relationship according to which a set of objects are to be arranged, we want to determine an algorithm (more specifically, a sequence of steps) that will yield the desired result. Some previous work on this subject is that of Batcher [1], Bose and Nelson [2], Gale and Karp [3], Knuth [4], Levy and Paull [5].

We introduce first some notation. Let A be a finite set which is a set of *locations*. Let Z denote the set of integers. Let ϕ be a function mapping A into Z. For $a \in A$, $\phi(a)$ is the *content* of location a. We call the ordered pair (A, ϕ) a *configuration*. Let π be a one-to-one function mapping A onto A which defines a *permutation of the contents* of the locations. For $a \in A$, $\pi(a)$ is the location to which the content of a, $\phi(a)$, will be moved. Let ϕ_π be a function from A into Z such that

$$\phi_\pi(a) = \phi(\pi^{-1}(a)).$$

We say that (A, ϕ_π) is a configuration induced from (A, ϕ) by π.

Let P be a partial ordering relation over A. A configuration (A, ϕ) is said to be *consistent* with P if a_1Pa_2 implies that $\phi(a_1) \leqq \phi(a_2)$ for all a_1, a_2 in A. By *sorting a configuration* (A, ϕ) with respect to P, we mean to determine a permutation π on A such that (A, ϕ_π) is consistent with P. It is not difficult to see that for a given configuration (A, ϕ) and a partial ordering relation P the permutation π and thus the configuration (A, ϕ_π) are not necessarily unique. We shall use (A, ϕ_P) to denote a configuration induced from (A, ϕ) by some π such that (A, ϕ_π) is

* Received by the editors January 11, 1972.

† Department of Computer Science, University of Illinois, Urbana, Illinois 61801. This work was supported in part by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, Office of Naval Research, under Contract N00014-70-A-0362-0001.

[1] We limit our discussion to the sorting of integers (or any set of objects over which there is a natural linear ordering).

consistent with P. We say that (A, ϕ_P) is a configuration induced from (A, ϕ) by P.

A partial ordering relation C over A is called a *shift* if (i) a_1Ca_2 and a_1Ca_3 implies that either[2] a_2Ca_3 or a_3Ca_2 and (ii) a_2Ca_1 and a_3Ca_1 implies that either[2] a_2Ca_3 or a_3Ca_2 for any a_1, a_2, a_3 in A. A maximal sequence of locations $a_{11}, a_{12}, \cdots, a_{1m}$ such that $a_{1k}Ca_{1k}, a_{1k}Ca_{1,k+1}, a_{1k}Ca_{1,k+2}, \cdots, a_{1k}Ca_{1m}$, for $k = 1, 2, \cdots, m$, is called a *chain* of the shift C. Thus, corresponding to a shift the locations in A are partitioned into chains. For a given configuration (A, ϕ) and a shift C over A, a configuration consistent with C can readily be obtained by rearranging the integers in the locations in each chain into ascending order according to C. As a matter of fact, we shall adopt the convention that for a given configuration (A, ϕ) and a shift C over A, the configuration (A, ϕ_C) is the one induced from (A, ϕ) by the permutation that permutes the contents of the locations in each chain. A sequence of shifts over A, $(C_k, C_{k-1}, \cdots, C_2, C_1)$, is said to be a *sorting algorithm* for P if for any arbitrary ϕ, the configuration[3] $(A, \phi_{C_kC_{k-1}\cdots C_2C_1})$ is consistent with P. A sorting algorithm described by a sequence of shifts is nonadaptive in the sense that each step in the algorithm is predetermined and is independent of the outcome of previous steps. (We look at each shift in the sequence as a step. It is conceivable that an adaptive procedure is followed to rearrange the contents of the locations in the chains of a shift.)

The physical significance of these abstract notions should now become obvious: Given a set of integers stored in a set of locations as described by the configuration (A, ϕ), we can rearrange the integers so that the partial ordering relation P will be satisfied by following a sorting algorithm $(C_k, C_{k-1}, \cdots, C_2, C_1)$. That is, we rearrange the integers stored in A according to C_k, we then rearrange these (rearranged) integers according to C_{k-1}, and so on. Eventually, after we rearrange the integers according to C_1, we will have the integers sorted according to P.

2. An analysis result. Let (A, ϕ) be a given configuration. Let P be a partial ordering relation and C be a shift over A. Suppose that we have sorted the integers in A so that the configuration (A, ϕ_P) is consistent with P. We want to know the partial ordering relation that the new configuration, (A, ϕ_{PC}), will be consistent with if the integers in each chain of C are rearranged according to C.

For a partial ordering relation P over A, a location a_1 is said to *precede* a location a_2 in P if a_1Pa_2. We also say that the location a_2 *is preceded* by the location a_1. Given a shift C over A, we define the *rank* of a location to be the number of locations preceding it in C. Clearly, the rank of each location is a positive integer. Moreover, in a chain of m locations, there is a location of rank 1, a location of rank 2, $\cdots$, and a location of rank m.

Let P be a partial ordering relation and C be a shift over A. We define a binary relation Q over A as follows: Let a_1 be a location in a chain A_1 and a_2 be a location in a chain A_2 of the shift C. Let the rank of a_1 be i and the rank of a_2

[2] Since C is a partial ordering relation, it is clear that this is an "exclusive or" relationship for $a_2 \neq a_3$.

[3] We write $\phi_{C_kC_{k-1}}$ to mean $(\phi_{C_k})_{C_{k-1}}$, and $\phi_{C_kC_{k-1}\cdots C_2C_1}$ to mean $((((\phi_{C_k})_{C_{k-1}})\cdots)_{C_2})_{C_1}$.

be j. Then $a_1 Q a_2$ if and only if every j locations in A_2 are preceded, in the partial ordering relation P, by i or more locations in A_1. The binary relation Q is said to be a composition of the partial ordering relation P and the shift C, which will be denoted by $P \otimes C$. As one would suspect at this point, the configuration (A, ϕ_{PC}) is consistent with the binary relation Q. We shall show this indeed is the case. We should, however, establish first that Q is a *partial ordering relation* over A.

LEMMA 1. *Let a_1 and a_2 be two locations in the same chain of C. Then $a_1 Q a_2$ if and only if $a_1 C a_2$.*

Proof. Let a_1 be a location of rank i and a_2 be a location of rank j in a chain A_1 of C.

Suppose that $a_1 C a_2$. That is, $i \leqq j$. Since every j locations in A_1 are preceded, in the partial ordering relation P, by at least j locations (themselves) in A_1, we have $a_1 Q a_2$.

Suppose that $a_1 Q a_2$. We define the *height* of a location a_u in A_1 to be 1 if there is no a_v in A_1, $a_u \neq a_v$, such that $a_v P a_u$. Because P is a partial ordering relation, and because A_1 is a finite set of locations, there is one or more locations in A_1 whose height is 1. Recursively, the height of a location a_u in A_1 is said to be n if there is a location a_v in A_1 whose height is $n - 1$ such that $a_v P a_u$ and if there is no location a_w in A_1 whose height is n or higher such that $a_w P a_u$. Because P is a partial ordering relation, the height of a location is a uniquely determined positive integer. For a given integer j, let us select j locations from A_1 by selecting locations of lowest possible heights. That is, we shall not include locations whose heights are n until locations whose heights are $n - 1$ or lower have all been included starting with locations of height 1. Clearly such a set of j locations are preceded only by j locations (themselves) in A_1. Therefore, $a_1 Q a_2$ implies that $i \leqq j$. That is, $a_1 Q a_2$ implies that $a_1 C a_2$. $\square$

According to Lemma 1, $Q \supseteq C$. Intuitively, it is clear that this is necessarily the case because the configuration (A, ϕ_{PC}) must be consistent with C.

THEOREM 1. *Q is a partial ordering relation over A.*

Proof. Let a be a location of rank i in a chain A_1. Since, in the partial ordering relation P, every i locations in A_1 are preceded by at least i locations (themselves) in A_1, aQa, and Q is reflexive.

Let a_1 and a_2 be two locations in the same chain of C. According to Lemma 1, $a_1 Q a_2$ implies that $\neg(a_2 Q a_1)$ for $a_1 \neq a_2$. Now, let a_1 be a location of rank i in a chain A_1 and a_2 be a location of rank j in a chain A_2, where $A_1 \neq A_2$. Suppose that $a_1 Q a_2$ and $a_2 Q a_1$. That is, in the partial ordering relation P every i locations in A_1 are preceded by j or more locations in A_2 and every j locations in A_2 are preceded by i or more locations in A_1. Let I_1 be a subset of i locations in A_1. Let J_1 be a subset of j locations in A_2 that precede the locations in I_1. Let I_2 be a subset of i locations in A_1 that precede the locations in J_1, and J_2 be a subset of j locations in A_2 that precede the locations in I_2 and so on. Consider the sequence of subsets of locations $I_1 J_1 I_2 J_2 \cdots I_n J_n \cdots$. Since there is only a finite number of distinct subsets of i locations in A_1 and a finite number of distinct subsets of j locations in A_2, sooner or later a subset will appear for the second time in the sequence. Let I_l be a subset that appears twice in the sequence. That is, we have $I_1 J_1 I_2 \cdots I_l J_l I_{l+1} \cdots J_{r-1} I_r J_r I_l \cdots$. Let $a_{l1}, a_{l2}, \cdots, a_{li}$ be the locations in I_l.

Location a_{l1} precedes at least one of the locations in J_r. Let a_{r1} denote one such location. That is, $a_{l1}Pa_{r1}$. Also, location a_{r1} precedes at least one of the locations in I_r which, in turn, precedes one of the locations in J_{r-1} and so on. Repeating the argument, we obtain $a_{r1}Pa_{lu}$ for some a_{lu} in I_l. We thus have

$$a_{l1}Pa_{r1}, \quad a_{r1}Pa_{lu}.$$

Similarly, we have

$$a_{l2}Pa_{r2}, \quad a_{r2}Pa_{lv},$$
$$\vdots$$
$$a_{li}Pa_{ri}, \quad a_{ri}Pa_{lw}$$

for $a_{r2}, \cdots, a_{ri}$ in J_r and $a_{lv}, \cdots, a_{lw}$ in I_l. Again, since there is only a finite number of locations in I_l and J_r, transitivity of the relation P implies that $a_{lx}Pa_{rx}$ and $a_{rx}Pa_{lx}$ for some a_{lx} in I_l and some a_{rx} in J_r. However, this is a contradiction to the assumption that P is a partial ordering relation since, clearly, $a_{lx} \neq a_{rx}$. Therefore, it is not possible that a_1Qa_2 and a_2Qa_1. We conclude that Q is antisymmetric.

Let a_1 be a location of rank i in a chain A_1, a_2 be a location of rank j in a chain A_2, and a_3 be a location of rank k in a chain A_3. Suppose that a_1Qa_2 and a_2Qa_3. Since in the partial ordering relation P every k locations in A_3 are preceded by j or more locations in A_2 which are, in turn, preceded by i or more locations in A_1, every k locations in A_3 are preceded by i or more locations in A_1. Therefore, we have a_1Qa_3, and Q is transitive. □

THEOREM 2. *For any configuration (A, ϕ_P), the configuration (A, ϕ_{PC}) is consistent with Q.*

Proof. Suppose that a_1Qa_2.

Let a_1 and a_2 be two locations in the same chain of C. According to Lemma 1, a_1Ca_2. Therefore, $\phi_{PC}(a_1) \leqq \phi_{PC}(a_2)$.

Let a_1 be a location of rank i in a chain A_1, and a_2 be a location of rank j in a chain A_2, where $A_1 \neq A_2$. Since a_2 is a location of rank j, $\phi_{PC}(a_2)$ is the jth smallest integer among the contents of the locations in A_2. That is, for a certain subset of j locations $a_{21}, a_{22}, \cdots, a_{2j}$ in A_2,

$$\phi_{PC}(a_2) = \max[\phi_P(a_{21}), \phi_P(a_{22}), \cdots, \phi_P(a_{2j})].$$

Let $a_{11}, a_{12}, \cdots, a_{1i}$ be i of the locations in A_1 that precede the locations $a_{21}, a_{22}, \cdots, a_{2j}$ in the partial ordering relation P. Thus,

$$\max[\phi_P(a_{11}), \phi_P(a_{12}), \cdots, \phi_P(a_{1i})] \leqq \max[\phi_P(a_{21}), \phi_P(a_{22}), \cdots, \phi_P(a_{2j})].$$

Since a_1 is a location of rank i, $\phi_{PC}(a_1)$ is the ith smallest integer among the contents of the locations in A_1. Thus,

$$\phi_{PC}(a_1) \leqq \max[\phi_P(a_{11}), \phi_P(a_{12}), \cdots, \phi_P(a_{1i})].$$

Therefore, we have $\phi_{PC}(a_1) \leqq \phi_{PC}(a_2)$.

We conclude that (A, ϕ_{PC}) is consistent with Q. □

THEOREM 3. *For any configuration (A, ϕ_{PC}), Q is the largest partial ordering relation that (A, ϕ_{PC}) is consistent with.*

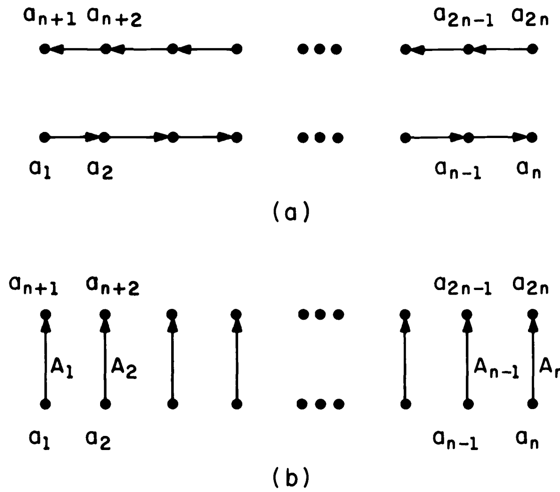


FIG. 1

Proof. Let a_1 be a location of rank i in a chain A_1, and a_2 be a location of rank j in a chain A_2, where $A_1 \neq A_2$. Suppose that $(a_1, a_2) \notin Q$. This means that there is a subset of j locations $a_{21}, a_{22}, \cdots, a_{2j}$ in A_2 that are preceded by at most $i - 1$ locations in A_1. Let $a_{11}, a_{12}, \cdots, a_{1,i-1}$ denote these locations. Let us choose ϕ_P such that

$$\phi_P(a_x) = \begin{cases} 0 & \text{if in the partial ordering relation } P, a_x \text{ precedes one or more} \\ & \text{of the locations } a_{21}, a_{22}, \cdots, a_{2j}, \\ 1 & \text{otherwise.} \end{cases}$$

Note that the configuration (A, ϕ_P) indeed is consistent with P. Since

$$\phi_P(a_{21}) = \phi_P(a_{22}) = \cdots = \phi_P(a_{2j}) = 0,$$

$$\phi_P(a_{11}) = \phi_P(a_{12}) = \cdots = \phi_P(a_{1,i-1}) = 0,$$

and moreover, the contents of all other locations in A_1 are 1, it follows that

$$\phi_{PC}(a_2) = 0,$$

$$\phi_{PC}(a_1) = 1.$$

Therefore, $\phi_{PC}(a_1) > \phi_{PC}(a_2)$. $\square$

We illustrate now the results we obtained by an example. Suppose that we are to select among $2n$ integers the n largest ones. Let $\phi(a_1), \phi(a_2), \cdots, \phi(a_n)$, $\phi(a_{n+1}), \phi(a_{n+2}), \cdots, \phi(a_{2n})$ be the $2n$ integers which have been sorted such that $\phi(a_1) \leqq \phi(a_2) \leqq \cdots \leqq \phi(a_n)$ and $\phi(a_{n+1}) \geqq \phi(a_{n+2}) \geqq \cdots \geqq \phi(a_{2n})$ as shown in Fig. 1(a).[4] Let us compare $\phi(a_i)$ with $\phi(a_{n+i})$ and interchange their positions if $\phi(a_i) > \phi(a_{n+i})$ for $i = 1, 2, \cdots, n$. The claim is: The n integers in the upper row of locations are the n largest integers we want to select. To prove the claim using the concepts we have developed, we let P be the partial ordering relation shown in Fig. 1(a) and C be the shift shown in Fig. 1(b). Let $Q = P \otimes C$. We want to

[4] We shall represent partial ordering relations by their Hasse diagrams.

show that $a_i Q a_{n+j}$ for $i = 1, 2, \cdots, n$ and $j = 1, 2, \cdots, n$. However, according to Theorem 2, such a result is obvious because every subset of two locations in the chain A_j is preceded by at least one location in the chain A_i for all A_i and A_j.

We now establish a relationship between Q and P and C.

THEOREM 4. *Let* $Q = P \otimes C$. *Then* $Q \subseteq C \circ P \circ C$.

Proof. Suppose that $a_{1i} Q a_{2j}$. If a_{1i} and a_{2j} are in the same chain of C, according to Lemma 1, $a_{1i} C a_{2j}$. Clearly, $(a_{1i}, a_{2j}) \in C \circ P \circ C$. Let a_{1i} and a_{2j} be in different chains of C. Let $a_{11}, a_{12}, \cdots, a_{1i}, \cdots, a_{1m}$ denote the locations in A_1 with the rank of a_{1r} being r for $r = 1, 2, \cdots, m$. Let $a_{21}, a_{22}, \cdots, a_{2j}, \cdots, a_{2n}$ denote the locations in A_2 with the rank of a_{2r} being r for $r = 1, 2, \cdots, n$. Consider the subset of j locations $a_{21}, a_{22}, \cdots, a_{2j}$ in A_2. Since these j locations are preceded by at least i locations in A_1, there is a location a_{1k} in A_1 and a location a_{2l} in A_2 such that $a_{1k} P a_{2l}$ with $k \geqq i$ and $l \leqq j$. That is, $a_{1i} C a_{1k}$, $a_{1k} P a_{2l}$, and $a_{2l} C a_{2j}$. It follows that $(a_{1i}, a_{2j}) \in C \circ P \circ C$. □

Let $Q = P \otimes C$. We say that C *preserves* P if $P \subseteq Q$. According to Theorem 1 we have the following theorem which is due to Gale and Karp [3].

THEOREM 5. *A shift C preserves a partial ordering relation P if and only if*:

(i) *For* $a_1 \neq a_2$, $a_1 C a_2$ *implies that* $\neg(a_2 P a_1)$.

(ii) *For a location* a_1 *of rank i in a chain* A_1 *and a location* a_2 *of rank j in a chain* A_2, *where* $A_1 \neq A_2$, $a_1 P a_2$ *implies that in the partial ordering relation P every j locations in* A_2 *are preceded by i or more locations in* A_1.

If C preserves P, then using Theorem 4, it can be shown that $Q = C \circ P \circ C$. However, the result can be sharpened to that in Theorem 6 which is due to Gale and Karp [3]. We present an alternative proof to the theorem.

THEOREM 6. *Let* $Q = P \otimes C$. *If C preserves P, then* $Q = P \circ C$.

Proof. Suppose that $a_{1i} Q a_{2j}$. If a_{1i} and a_{2j} are in the same chain of C, clearly, $(a_{1i}, a_{2j}) \in P \circ C$. Suppose that a_{1i} and a_{2j} are in two different chains A_1 and A_2. Let $a_{11}, a_{12}, \cdots, a_{1i}, \cdots, a_{1m}$ be the locations in A_1 with the rank of a_{1r} being r for $r = 1, 2, \cdots, m$. Let $a_{21}, a_{22}, \cdots, a_{2j}, \cdots, a_{2n}$ be the locations in A_2 with the rank of a_{2r} being r for $r = 1, 2, \cdots, n$. Consider the subset of j locations $a_{21}, a_{22}, \cdots, a_{2j}$ in A_2. If $(a_{1i}, a_{2u}) \in P$ for some $u \leqq j$, then $(a_{1i}, a_{2j}) \in P \circ C$. Assume that $(a_{1i}, a_{2u}) \notin P$ for all $u \leqq j$. There must be a location a_{1k} in A_1, $k > i$, such that $(a_{1k}, a_{2u}) \in P$ for some $u \leqq j$, because in the partial ordering relation P the j locations $a_{21}, a_{22}, \cdots, a_{2j}$ are preceded by at least i locations in A_1. Since $(a_{1k}, a_{2u}) \in P$ implies that $(a_{1k}, a_{2j}) \in Q$, repeating the argument, we have $(a_{1l}, a_{2v}) \in P$ for some $l > k$ and $v \leqq j$, which, in turn, implies $(a_{1l}, a_{2j}) \in Q$. Again, repeating the argument, we finally have $(a_{1m}, a_{2j}) \in Q$. However, since the subset of j locations $a_{21}, a_{22}, \cdots, a_{2j}$ are preceded by almost $m - 1$ locations in A_1 (all locations except a_{1i}), we have a contradiction. □

COROLLARY 6.1. $Q = C \circ P$.

Proof. It is not difficult to show that $Q^{-1} = P^{-1} \circ C^{-1}$ which implies $Q = C \circ P$. □

COROLLARY 6.2. *If C preserves P, C commutes with P.*

3. Analysis of sorting algorithms. The results in § 2 can now be applied to the analysis of sorting algorithms. Let $C_k, C_{k-1}, \cdots, C_2, C_1$ be a sorting algorithm. Clearly, for any initial configuration (A, ϕ), the configuration $(A, \phi_{C_k C_{k-1} \cdots C_2 C_1})$

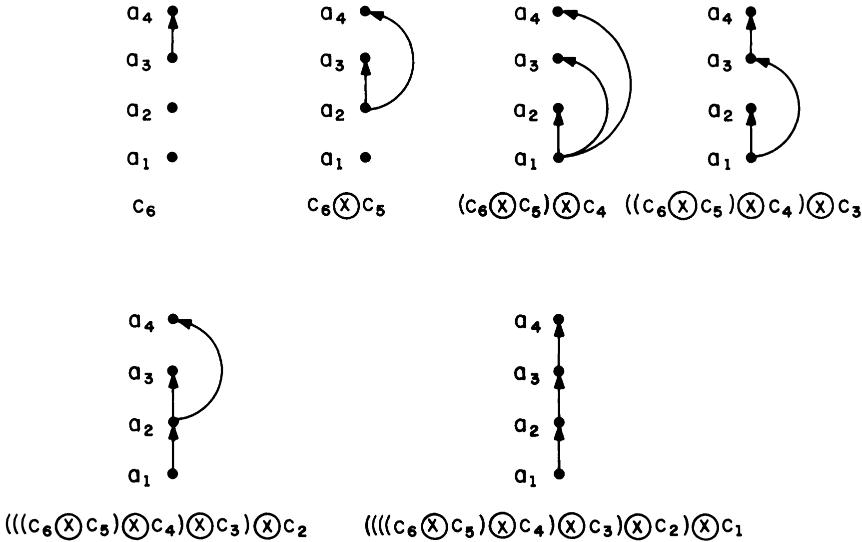


FIG. 2

is consistent with the partial ordering relation

$$(((C_k \otimes C_{k-1}) \otimes \cdots) \otimes C_2) \otimes C_1.$$

As an illustrative example, let us consider the problem of sorting four integers by the well-known procedure "bubble sort." Let $A = \{a_1, a_2, a_3, a_4\}$. Let $C_6 = (a_3, a_4)$,[5] $C_5 = (a_2, a_3)$, $C_4 = (a_1, a_2)$, $C_3 = (a_3, a_4)$, $C_2 = (a_2, a_3)$, $C_1 = (a_3, a_4)$. In Fig. 2 the partial ordering relations constructed in a step-by-step manner are shown.

We want to point out that although

$$(((C_k \otimes C_{k-1}) \otimes \cdots) \otimes C_2) \otimes C_1$$

is a partial ordering relation that the configuration $(A, \phi_{C_kC_{k-1}\cdots C_2C_1})$ is consistent with, it might not be the *largest* one. This is the main point of our discussion in this section. Let P be a partial ordering relation and C_1 and C_2 be two shifts over A. We ask the question: Under what condition will $(P \otimes C_2) \otimes C_1$ be the largest partial ordering relation that $(A, \phi_{PC_2C_1})$ is consistent with? (For a reader who begins to worry about the correctness of Theorem 3, we hasten to point out that although Theorem 3 guarantees that $(P \otimes C_2) \otimes C_1$ is the largest partial ordering relation that the configuration $(A, \phi_{(P\otimes C_2)C_1})$ is consistent with, $(P \otimes C_2) \otimes C_1$ might not be the largest partial ordering relation that the configuration $(A, \phi_{PC_2C_1})$ is consistent with.)

We establish first an alternative way of defining the partial ordering relation $P \otimes C$. Let C be a shift over A. Let τ be a one-to-one function from A onto A

[5] To simplify the notations, we write (a_3, a_4) to mean a chain in which a_3Ca_4. We also omit all trivial chains.

such that a and $\tau(a)$ are in the same chain of C for every $a \in A$. In other words, τ is a permutation of the locations in each of the chains. We shall call τ a *C-permutation* of A. Let P be a partial ordering relation over A. Let $\tau(P)$ denote a binary relation over A such that if $(a_1, a_2) \in P$, $(a_1, a_2) \notin C$, and $(a_2, a_1) \notin C$, then $(\tau(a_1), \tau(a_2)) \in \tau(P)$. A C-permutation τ is said to be *P-compatible* if $(\tau(P) \cup C)^*$ is a partial ordering relation over A.[6] Let $\mathscr{T}$ denote the set of all P-compatible C-permutations of A.

LEMMA 2. *Let τ be a C-permutation of A that is P-compatible. Then*

$$(\tau(P) \cup C)^* \supseteq P \otimes C.$$

Proof. Let $a_{11}, a_{12}, \cdots, a_{1i}, \cdots, a_{1m}$ be the locations in a chain A_1 of C with the rank of a_{1r} being r for $r = 1, 2, \cdots, m$. Let $a_{21}, a_{22}, \cdots, a_{2n}$ be the locations in a chain A_2 of C with the rank of a_{2r} being r for $r = 1, 2, \cdots, n$. Suppose that $(a_{1i}, a_{2j}) \in P \otimes C$. Consider the j locations $\tau^{-1}(a_{21}), \tau^{-1}(a_{22}), \cdots, \tau^{-1}(a_{2j})$ in A_2. Since they are preceded, in the partial ordering relation P, by i or more locations in A_1, there exists $(a_{1t}, a_{2s}) \in \tau(P)$ for $t \geqq i$, $s \leqq j$. Thus, we have

$$(a_{1i}, a_{2j}) \in (\tau(P) \cup C)^*. \qquad \square$$

THEOREM 7. *The intersection of all partial ordering relations $(\tau(P) \cup C)^*$ corresponding to all P-compatible C-permutations τ is equal to $P \otimes C$. That is,*

$$\bigcap_{\tau \in \mathscr{T}} (\tau(P) \cup C)^* = P \otimes C.$$

Proof. According to Lemma 2,

$$\bigcap_{\tau \in \mathscr{T}} (\tau(P) \cup C)^* \supseteq P \otimes C.$$

We shall show that if $(a_1, a_2) \notin P \otimes C$, there exists $\tau_0 \in \mathscr{T}$ such that $(a_1, a_2) \notin (\tau_0(P) \cup C)^*$. Suppose that a_1 is in a chain A_1 with its rank being i, and a_2 is in a chain A_2 with its rank being j. Since $(a_1, a_2) \notin P \otimes C$, there is a subset of j locations in A_2 which are preceded, in the partial ordering relation P, by at most $i - 1$ locations in A_1. Let J denote such a subset of j locations in A_2 and let I denote the set of the predecessors in A_1. Let K denote the subset of locations in A_2 which, in their partial ordering relation P, are predecessors of the locations in I and J. Let $L = J \cup K$. Let us define a binary relation P' over A:

$$P' = \{(x, y) | x \in L \text{ and } y \in A_2 - L, \text{ or } x \in I \text{ and } y \in A_1 - I\}.$$

It is not difficult to see that $(P \cup P')^*$ is a partial ordering relation over A. Let us embed $(P \cup P')^*$ in a linear ordering relation T over A.

We now define a C-permutation τ_0 such that for any two locations x and y in the same chain of C, xTy implies that $\tau_0(x) C \tau_0(y)$. We note first that τ_0 is P-compatible. Moreover, in the partial ordering relation $(\tau_0(P) \cup C)^*$ the $|L|$ lowest locations in A_2 will only be preceded by the $|I|$ lowest locations in A_1. Thus

$$(a_t, a_2) \notin \tau_0(P)$$

for all a_t in A_1 whose rank is equal to i or higher. $\square$

[6] We use R^* to denote the transitive and reflexive closure of the binary relation R.

After obtaining a characterization of the partial ordering relation $P \otimes C$ as that in Theorem 7, we are ready to answer the question raised at the beginning of this section.

A shift C is said to *rearrange* P if there exists a P-compatible C-permutation τ of A such that $P \otimes C = (\tau(P) \cup C)^*$.

LEMMA 3. *If C rearranges P, then for any configuration (A, ϕ) that is consistent with $P \otimes C$ there is an initial configuration (A, ϕ') such that (A, ϕ') is consistent with P and (A, ϕ'_C) is equal to (A, ϕ).*

Proof. Let τ be a P-compatible C-permutation such that

$$P \otimes C = (\tau(P) \cup C)^*.$$

Let $\phi'(a) = \phi(\tau^{-1}(a))$ for all $a \in A$. Clearly, (A, ϕ') is consistent with P, and (A, ϕ'_C) is equal to (A, ϕ). □

THEOREM 8. *If C_2 rearranges P, then $(P \otimes C_2) \otimes C_1$ is the largest partial ordering relation that $(A, \phi_{PC_2C_1})$ is consistent with.*

Proof. Suppose that $(a_1, a_2) \notin (P \otimes C_2) \otimes C_1$. Using the argument in the proof of Theorem 3, we can construct a configuration (A, ϕ) that is consistent with $P \otimes C_2$, with $\phi_{C_1}(a_1) > \phi_{C_1}(a_2)$. According to Lemma 3, there is an initial configuration (A, ϕ') that is consistent with P with (A, ϕ'_{C_2}) equal to (A, ϕ). Consequently, $(A, \phi'_{PC_2C_1})$ is not consistent with the partial ordering relation $(((P \otimes C_2) \otimes C_1) \cup (a_1, a_2))^*$. □

COROLLARY 8.1. *For any initial configuration (A, ϕ), if C_i rearranges $((C_k \otimes C_{k-1}) \otimes \cdots) \otimes C_{i+1}$ for $i = 2, \cdots, k-1$, then $((C_k \otimes C_{k-1}) \otimes \cdots \otimes C_2) \otimes C_1$ is the largest partial ordering relation that $(A, \phi_{C_kC_{k-1}\cdots C_2C_1})$ is consistent with.*

It seems that the converses of Lemma 3 and Theorem 8 are also true. We state the following as conjectures.

CONJECTURE 1. *If C does not rearrange P, then there exists a configuration (A, ϕ) consistent with $P \otimes C$ such that there is no configuration (A, ϕ') which is consistent with P and (A, ϕ'_C) is equal to (A, ϕ).*

CONJECTURE 2. *If C_k does not rearrange P, then there exist $C_{k-1}, C_{k-2}, \cdots, C_2, C_1$ such that $(((P \otimes C_k) \otimes C_{k-1}) \cdots \otimes C_2) \otimes C_1$ is* not *the largest partial ordering relation that the configuration $(A, \phi_{PC_kC_{k-1}\cdots C_2C_1})$ is consistent with.*

4. Construction of sorting algorithms. The results presented in § 2 also lead to a recursive procedure for designing sorting algorithms. Let R be a partial ordering relation over A according to which the integers in A are to be sorted. If we choose a shift C over A such that a_1Ca_2 implies that $\neg(a_2Ra_1)$, for $a_1 \neq a_2$, we can determine a partial ordering relation P such that $R \subseteq C \otimes P$. The decomposition step can then be repeated to decompose P until a sorting algorithm consisting of a sequence of shifts over A is determined.

Let us illustrate the idea with a simple example: Let $A = \{a_1, a_2, \cdots, a_9\}$ and R be the partial ordering relation shown in Fig. 3(a). Let C_1 be the shift shown in Fig. 3(b) where the three chains are labeled A_1, A_2, A_3. We construct P in Fig. 3(c) so that the conditions: (i) every two locations in A_2 are preceded by two or more locations in A_1, (ii) every three locations in A_3 are preceded by one or more locations in A_2, are satisfied. (Note that condition (i) implies that every two locations in A_2 are preceded by one or more locations in A_1 and every three

locations in A_2 are preceded by two or more locations in A_1.) We then decompose P by first choosing the shift C_2 in Fig. 3(d) and then determining the partial ordering relation C_3 in Fig. 3(e) such that $P \subseteq C_3 \otimes C_2$. Since C_3 is a shift, (C_3, C_2, C_1) is a sorting algorithm for R. That is, $R \subseteq (C_3 \otimes C_2) \otimes C_1$.

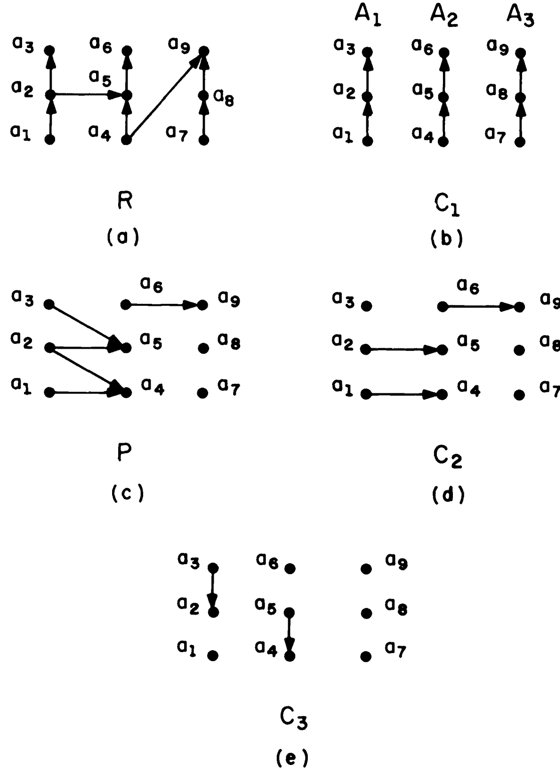


FIG. 3

The procedure suggested above does not include a systematic construction of the partial ordering relation P. Thus, in constructing P care must be exercised so that P is indeed a partial ordering relation. (Specifically, the antisymmetric law must be satisfied.) Theorem 9 below provides a systematic decomposition procedure.

Let R be a partial ordering relation over A. Let C be a shift such that $C \subseteq R$. Clearly, $(R - C)^*$ is a partial ordering relation.

THEOREM 9. *The shift C preserves the partial ordering relation $(R - C)^*$.*

Proof. To simplify the notations, let $P = (R - C)^*$. First, for $a_1 \neq a_2$, a_1Ca_2 implies that $\neg(a_2Pa_1)$. Next, let $a_{11}, a_{12}, a_{13}, \cdots, a_{1i}$ be locations in a chain A_1, where a_{1r} is of rank r for $r = 1, 2, \cdots, i$. Let $a_{2j}, a_{2,j+1}, a_{2,j+2}, \cdots, a_{2n}$ be locations in another chain A_2, where a_{2r} is of rank r for $r = j, j+1, \cdots, n$. Suppose that $a_{1i}Pa_{2j}$. Note that $a_{1u}Pa_{2v}$ for $u = 1, 2, \cdots, i$ and $v = j, j+1, \cdots, n$. That is, every one of the locations $a_{2j}, a_{2,j+1}, \cdots, a_{2n}$ is preceded by the locations

$a_{11}, a_{12}, \cdots, a_{1i}$ in P. Since every subset of j locations in A_2 must contain one or more of the locations $a_{2j}, a_{2,j+1}, \cdots, a_{2n}$, every subset of j locations in A_2 is preceded by i or more locations in A_1. Therefore, according to Theorem 5, C preserves P. □

COROLLARY 9.1. $(R - C)^* \otimes C = R$.

Proof. To simplify the notations, let $Q = (R - C)^* \otimes C$. Since $C \subseteq Q$ and $(R - C)^* \subseteq Q$, $R \subseteq Q$. Since $C \subseteq R$, $(R - C)^* \subseteq R$, $Q = C \circ (R - C)^* \subseteq R$. Therefore, $Q = R$. □

Repeated applications of Theorem 9 lead to a decomposition of the partial ordering relation R into a sequence of shifts $C_k, C_{k-1}, \cdots, C_2, C_1$ such that $(C_k, C_{k-1}, \cdots, C_2, C_1)$ is a sorting algorithm for R. To be explicit, we first choose C_1 such that $C_1 \subseteq R$, we then choose C_2 such that $C_2 \subseteq (R - C_1)^*$, and then choose C_3 such that $C_3 \subseteq ((R - C_1)^* - C_2)^*$, and so on. To assure that R will be decomposed into a *finite* sequence of shifts, we want to select the shifts such that

$$C_1 \cap (R - C_1)^* = A^0, \quad C_2 \cap ((R - C_1)^* - C_2)^* = A^0,$$
$$C_3 \cap (((R - C_1)^* - C_2)^* - C_3)^* = A^0,$$

and so on.[7] In other words, in selecting the shift C_i we should follow the rule that if $(a_1, a_2) \in C_i$ then for any location a_3 such that $(a_1, a_3) \in (((R - C_1)^* - C_2)^* \cdots - C_{i-1})^*$ and $(a_3, a_2) \in (((R - C_1)^* - C_2)^* \cdots - C_{i-1})^*$, we should have $(a_1, a_3) \in C_i$ or $(a_3, a_2) \in C_i$.

As an example, suppose we want to design a sorting algorithm for arranging six integers in ascending order. Let R be the partial ordering relation shown in Fig. 4(a), and C_1 be the shift shown in Fig. 4(b). We obtain $(R - C_1)^*$ as shown in Fig. 4(c). Let C_2 be the shift shown in Fig. 4(d). We obtain $((R - C_1)^* - C_2)^*$ as shown in Fig. 4(e). Let C_3 be the shift shown in Fig. 4(f) and C_4 be the shift shown in Fig. 4(g). Since C_3 preserves the shift C_4, and $C_4 \otimes C_3 = ((R - C_1)^* - C_2)^*$, we conclude that (C_4, C_3, C_2, C_1) is an algorithm for sorting six integers linearly.

The result can be extended to construct an algorithm for arranging $2n$ integers into ascending order. Let $A = \{a_1, a_2, \cdots, a_n, a_{n+1}, \cdots, a_{2n}\}$. Let P be a partial ordering over A such that

$$\begin{aligned} P = &\{(a_i, a_j) | 1 \leqq i \leqq n, i < j\} \\ &\cup \{(a_i, a_j) | n + 1 \leqq i \leqq 2n, i < j\} \\ &\cup \{(a_i, a_{n+i}) | 1 \leqq i \leqq n\} \\ &\cup \{(a_{n+i}, a_{i+k}) | 1 \leqq i \leqq n - k\} \cup A^0. \end{aligned}$$

Let C be a shift over A such that

$$C = \{(a_{n+i}, a_{i+2k}) | 1 \leqq i \leqq n - 2k\} \cup A^0.$$

It can readily be shown that C preserves P. Consequently, a sorting algorithm

[7] A^0 denotes the binary relation $\{(a_i, a_i) | a_i \in A\}$.

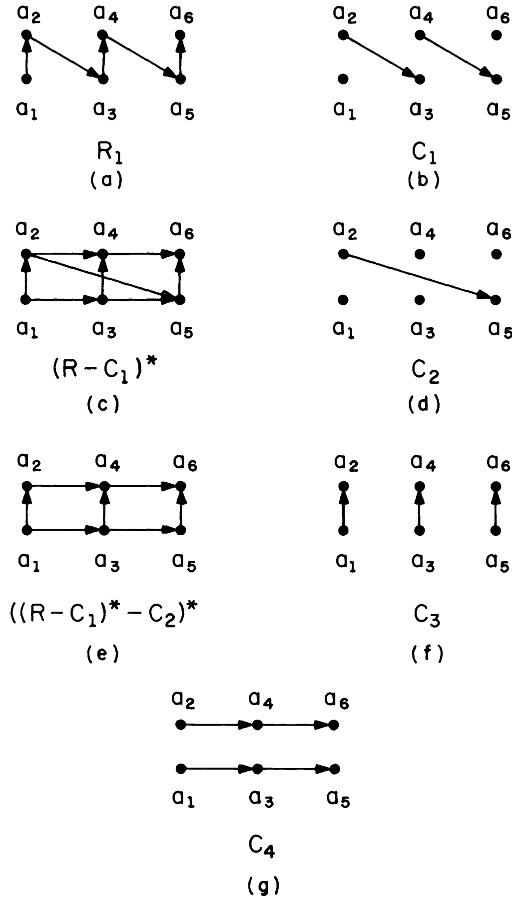


FIG. 4

for arranging $2n$ integers into ascending order is

$$
\begin{aligned}
C_k &= \{(a_i, a_j) | 1 \leqq i \leqq n, i < j\} \\
&\quad \cup \{(a_i, a_j) | n + 1 \leqq i \leqq 2n, i < j\} \cup A^0, \\
C_{k-1} &= \{(a_i, a_{n+i}) | 1 \leqq i \leqq n\} \cup A^0, \\
C_{k-2} &= \{(a_{n+i}, a_{i+t}) | 1 \leqq i \leqq n - t\} \cup A^0, \\
&\quad \text{where } t \text{ equals the largest power of 2 not exceeding } n, \\
C_{k-3} &= \{(a_{n+i}, a_{i+t/2}) | 1 \leqq i \leqq n - t\} \cup A^0, \\
&\quad \vdots \\
C_4 &= \{(a_{n+i}, a_{i+8}) | 1 \leqq i \leqq n - 8\} \cup A^0, \\
C_3 &= \{(a_{n+i}, a_{i+4}) | 1 \leqq i \leqq n - 4\} \cup A^0, \\
C_2 &= \{(a_{n+i}, a_{i+2}) | 1 \leqq i \leqq n - 2\} \cup A^0, \\
C_1 &= \{(a_{n+i}, a_{i+1}) | 1 \leqq i \leqq n - 1\} \cup A^0.
\end{aligned}
$$

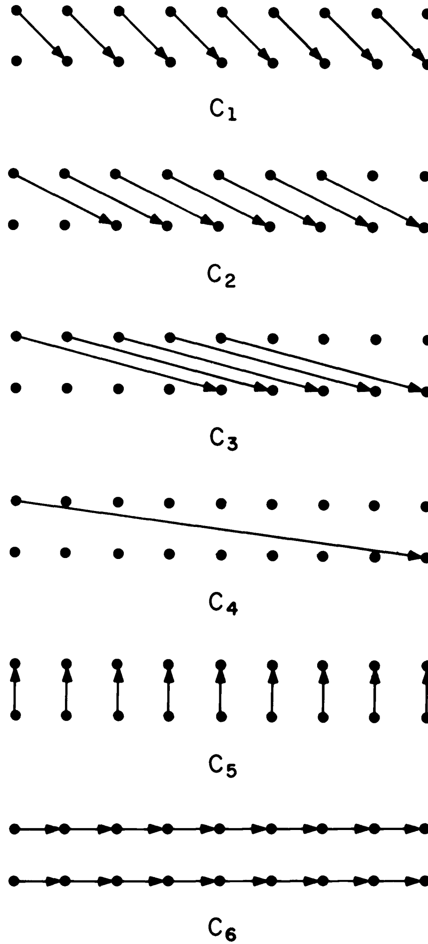


FIG. 5

For instance, Fig. 5 shows such a sorting algorithm for 18 integers. It is interesting to observe that this algorithm is indeed the odd-even merge algorithm for two sorted lists of the same length due to Batcher [1]. Such an observation is not immediately obvious. Indeed, our description of the algorithm is quite different from that of Batcher's. We leave it to the reader to compare the two descriptions, and hope that our description will add some insight to how the odd-even merge algorithm works.

Note that for any sorting algorithm $(C_k, C_{k-1}, \cdots, C_1)$ constructed according to the result in Theorem 9, C_{k-1} preserves C_k, C_{k-2} preserves $C_k \otimes C_{k-1}$, and so on. In other words, in such a sorting algorithm every shift preserves the partial ordering relation which resulted from previous steps. Theorem 10 below suggests a variation to the construction procedure so that sorting algorithms containing shifts that *do not* preserve the result of previous sorting steps might also be discovered.

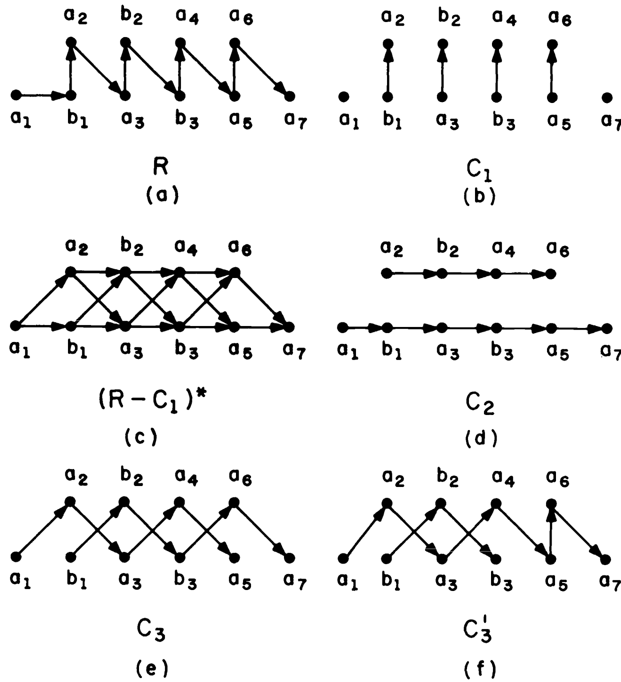


FIG. 6

Let P be a partial ordering relation and C be a shift over A. Let τ be a C-permutation of A as defined in § 3. Let us define a relation P' over A such that $a_1P'a_2$ if and only if $\tau(a_1)P\tau(a_2)$. Clearly, P' is also a partial ordering relation.

THEOREM 10. $P \otimes C = P' \otimes C$.

Proof. Every j locations in a chain A_1 are preceded by i or more locations in another chain A_2 in the partial ordering relation P' if and only if every j locations in A_1 are preceded by i or more locations in A_2 in the partial ordering relation P. □

Theorem 10 enables us to modify the construction procedure that is based on Theorem 9. Specifically, after selecting C_1 and determining $P = (R - C_1)^*$, we can construct P' as in Theorem 10 so that $P \otimes C_1 = P' \otimes C_1$.

The results in Theorem 10 can be applied to derive Batcher's odd-even merge algorithm for merging two sorted lists which are not of the same length. For the sake of simplicity, let us, instead of deriving the general result, show how to merge a sorted list of seven integers and a sorted list of three integers. Let R be the partial ordering relation shown in Fig. 6(a),[8] and let C_1 be the shift shown in Fig. 6(b). We obtain $(R - C_1)^*$ as shown in Fig. 6(c). Let C_2 be the shift shown in Fig. 6(d), and C_3 be the shift shown in Fig. 6(e). Since C_2 preserves C_3, and $C_3 \otimes C_2 = (R - C_1)^*$, (C_3, C_2, C_1) is a sorting algorithm for R. However, it can easily be seen that for the shift C_3' shown in Fig. 6(f), $C_3' \otimes C_2 = C_3 \otimes C_2 = (R - C_1)^*$. Therefore, (C_3', C_2, C_1) is also a sorting algorithm for R. Note that

[8] Note that we label the locations in such a way that the final result will become clearer.

C'_3 has two chains, one contains seven locations and the other contains three locations.

REFERENCES

[1] K. E. BATCHER, *Sorting networks and their applications*, Proc. Spring Joint Computer Conference, Atlantic City, N.J., 1968, pp. 307–314.

[2] R. C. BOSE AND R. J. NELSON, *A sorting problem*, J. Assoc. Comput. Mach., 9 (1962), pp. 282–296.

[3] D. GALE AND R. M. KARP, *A phenomenon in the theory of sorting*, IEEE Conference Record of the 1970 Eleventh Annual Symposium on Switching and Automata Theory, Santa Monica, Calif., 1970, pp. 51–59.

[4] D. E. KNUTH, *The Art of Computer Programming*, vol. III, Addison-Wesley, Reading, Mass. (in press).

[5] S. Y. LEVY AND M. C. PAULL, *An algebra with application to sorting algorithms*, Proc. Third Annual Princeton Conference on Information Sciences and Systems, Princeton, N.J., 1969, pp. 286–291.

SIAM J. COMPUT.
Vol. 1, No. 4, December 1972

A MINIMUM DISTANCE ERROR-CORRECTING PARSER FOR CONTEXT-FREE LANGUAGES*

ALFRED V. AHO† AND THOMAS G. PETERSON‡

Abstract. We assume three types of syntax errors can debase the sentences of a language generated by a context-free grammar: the replacement of a symbol by an incorrect symbol, the insertion of an extraneous symbol, or the deletion of a symbol. We present an algorithm that will parse any input string to completion finding the fewest possible number of errors. On a random access computer the algorithm requires time proportional to the cube of the length of the input.

Key words. Syntax error, error correction, parsing, context-free grammar, computational complexity

1. Introduction. What should a compiler do when it discovers an error in the source program? Surprisingly, much of the literature published to date on compiler design has not adequately answered this question. Even in the area of parser design, relatively few papers have considered parsing algorithms that recover gracefully from syntax errors [3]–[8], [10]. Many published parsing algorithms call for a parser merely to halt and report error on encountering the first syntax error.

In this paper we present a parsing algorithm for context-free grammars that will parse any input string to completion finding the fewest possible number of syntax errors. On a random access computer the algorithm takes time proportional to the cube of the length of the input. Although this is the fastest known minimum distance error-correcting parsing algorithm for context-free grammars, the time required is probably excessive for most compiler applications. Nevertheless, we feel the algorithm can serve as the yardstick against which the error-detecting and correcting capabilities of other parsing algorithms can be evaluated.

2. Syntax errors. Let L be a nonempty set of strings over some finite alphabet Σ. We assume that a string not in L is derived from some sentence in L by a sequence of error transformations. In this paper we assume the following three types of syntax errors are possible:

(i) the replacement of a symbol by another symbol,
(ii) the insertion of an extraneous symbol, and
(iii) the deletion of a symbol.

We shall describe these errors in terms of three transformations T_R, T_I and T_D, respectively, from Σ^* to the subsets of Σ^* defined as follows. For x and y in Σ^*:

(i) xby is in $T_R(xay)$ for all $a \neq b$ in Σ.
(ii) xay is in $T_I(xy)$ for all a in Σ.
(iii) xy is in $T_D(xay)$ for all a in Σ.

We write $x \vdash y$ if y is in $T_i(x)$ for some i in $\{R, I, D\}$. Note that $\vdash$ is symmetric.

If ρ is a binary relation, ρ^k will denote the composition of ρ with itself k times, ρ^0 the identity relation, and ρ^* the reflexive and transitive closure of ρ.

* Received by the editors June 6, 1972, and in revised form October 5, 1972.
† Bell Laboratories, Murray Hill, New Jersey 07974.
‡ Bell Laboratories, Whippany, New Jersey 07981.

We define a Hamming distance on strings in Σ^* by letting $d(x, y)$ be the smallest integer k for which $x \overset{k}{\vdash} y$.

Suppose L is a language over alphabet Σ. We define $E_k(L)$, the *set of strings in* Σ^* *with k errors*, as follows:

(i) $E_0(L) = L$.

(ii) $E_k(L) = \{x \text{ in } \Sigma^* |$ there exists a string w in $E_{k-1}(L)$ such that $w \vdash x$ and x is not in $E_{k-l}(L)$ for $1 \leqq l \leqq k\}$.

A string x is in $E_k(L)$ if and only if there is a sentence w in L such that w is distance k from x. Thus if $x \in E_k(L)$ and $x \vdash y$, then $y \in E_i(L)$ for some i such that $k - 1 \leqq i \leqq k + 1$.

Given a string x in $E_k(L)$ there is a sequence of intermediate strings $w_0, w_1, \cdots, w_k$ such that $w_i \in E_i(L)$ for $0 \leqq i \leqq k$ and $w_k = x$. A sequence of error transformations used to derive w_i from w_{i-1} for $1 \leqq i \leqq k$ will define a set of k errors in x.

For example, suppose $L = \{abc\}$. Then given the string $bbdc$, we can say the first b is a replacement error and the d is an insertion error because $abc \underset{T_R}{\vdash} bbc \underset{T_I}{\vdash} bbdc$.

Note that for an x in $E_k(L)$ there can be many different sequences of intermediate strings and transformations as above, so that in general the errors in x are not unique. If desired, uniqueness can be achieved by some lexicographic conventions regarding how the sequence of intermediate strings and error transformations are to be chosen.

3. Context-free grammars. A *context-free grammar* (grammar for short) is a 4-tuple $G = (N, \Sigma, P, S)$, where

(i) N and Σ are finite disjoint alphabets of *nonterminals* and *terminals*, respectively.

(ii) P is a finite set of productions of the form $A \to \alpha$, where A is in N and α is in $(N \cup \Sigma)^*$.

(iii) S is a distinguished symbol in N.

If $A \to \alpha$ is in P, we write $\beta A \gamma \underset{G}{\Rightarrow} \beta\alpha\gamma$ for all β and γ in $(N \cup \Sigma)^*$. If γ is in Σ^*, we also write $\beta A \gamma \underset{G,rm}{\Rightarrow} \beta\alpha\gamma$ to denote a right-most derivation. Similarly, if β is in Σ^*, we write $\beta A \gamma \underset{G,lm}{\Rightarrow} \beta\alpha\gamma$ to denote a left-most derivation. We shall drop the subscript G from $\Rightarrow$, $\underset{lm}{\Rightarrow}$, $\underset{rm}{\Rightarrow}$ whenever possible.

The language *generated by* G, denoted $L(G)$, is the set $\{w \text{ in } \Sigma^* | S \overset{*}{\Rightarrow} w\}$. If w is in $L(G)$, then there exists a sequence of strings in $(N \cup \Sigma)^*$, $\alpha_0, \alpha_1, \cdots, \alpha_n$, such that $\alpha_0 = S$, $\alpha_{i-1} \underset{rm}{\Rightarrow} \alpha_i$ for $1 \leqq i \leqq n$ and $\alpha_n = w$. Suppose production p_i in P is used to derive α_i from α_{i-1} for $1 \leqq i \leqq n$. The sequence of productions $\pi = p_1 p_2 \cdots p_n$ is called a *parse of w according to G* and we shall write $S \overset{\pi}{\Rightarrow} w$ to denote the right-most derivation using this sequence of productions.

For the remainder of this paper we shall make the following assumptions about a grammar $G = (N, \Sigma, P, S)$:

(i) We assume $L(G) \neq \varnothing$.

(ii) We assume G contains no useless symbols. That is, for each X in $N \cup \Sigma$ there is a derivation of the form $S \overset{*}{\Rightarrow} wXy \overset{*}{\Rightarrow} wxy$, where w, x and y are in Σ^*.

The concept of one grammar covering another grammar will be useful. Let $G_1 = (N_1, \Sigma_1, P_1, S_1)$ and $G_2 = (N_2, \Sigma_2, P_2, S_2)$ be two grammars such that $L(G_1) \subseteq L(G_2)$. We say G_2 *covers* G_1 if there is a homomorphism h from P_2 to P_1 such that if x is in $L(G_1)$ and π is a parse of w according to G_1, then there is a parse π' of w according to G_2 such that $h(\pi') = \pi$.

4. Error-correcting parser. Our problem can be stated as follows: Given a grammar $G = (N, \Sigma, P, S)$, we want an algorithm that takes as input any string x in Σ^* and produces as output a parse for some string w in $L(G)$ such that the distance between w and x is as small as possible. (If x is in $L(G)$, then clearly we will produce a parse for x.) An algorithm of this nature will be called an *error-correcting parser* for G.

We can design such an error-correcting parser for G as follows. First we add to G a set of error productions to obtain a covering grammar G' such that $L(G') = \Sigma^*$. Then we design a parser for G' that uses as few error productions as possible in parsing an input string x. The error productions used in a derivation of w according to G' will indicate the positions and types of the errors in w.

5. Error productions. The following algorithm will add error productions to the grammar $G = (N, \Sigma, P, S)$ such that the extended grammar covers G and generates Σ^*.

ALGORITHM 1. Let $G = (N, \Sigma, P, S)$ be a context-free grammar. From P construct a new set of productions P' as follows:

1. If $A \to \alpha_0 b_1 \alpha_1 b_2 \alpha_2 \cdots b_m \alpha_m$, $m \geqq 0$, is a production in P such that α_i is in N^* and b_i is in Σ, then add the production $A \to \alpha_0 E_{b_1} \alpha_1 E_{b_2} \alpha_2 \cdots E_{b_m} \alpha_m$ to P', where each E_{b_i} is a new nonterminal.

2. For all a in Σ add to P' the productions
   (a) $E_a \to a$,
   (b) $E_a \to b$ for all b in Σ, $b \neq a$,
   (c) $E_a \to Ha$,
   (d) $I \to a$,
   (e) $E_a \to e$, where e is the empty string.

   Here H and I are new nonterminals.

3. Add to P' the productions
   (a) $S' \to S$,
   (b) $S' \to SH$,
   (c) $H \to HI$,
   (d) $H \to I$.

   Here S' is a new start symbol.

Let $G = (N', \Sigma', P', S')$, where $N' = N \cup \{S', H, I\} \cup \{E_a | a \in \Sigma\}$. We shall call G' the *covering grammar* for G. □

In P' a production of the form $E_a \to b$, $E_a \to e$ or $I \to a$ is called a *terminal error production*. The production $E_a \to b$ introduces a replacement error. $E_a \to e$ introduces a deletion error. $I \to a$ introduces one insertion error. Since H can derive any nonempty string, the production $E_a \to Ha$ introduces a sequence of one or more insertion errors in front of a. To place one or more insertion errors at the end of a sentence we apply the production $S' \to SH$.

To see that G' covers G we note that by using the productions added to P' in step 1, we have $S \overset{*}{\underset{G}{\Rightarrow}} a_1 a_2 \cdots a_n$ if and only if $S \overset{*}{\underset{G'}{\Rightarrow}} E_{a_1} E_{a_2} \cdots E_{a_n}$. Using the

productions $E_a \to a$, we have $E_{a_1}E_{a_2} \cdots E_{a_n} \overset{*}{\Rightarrow} a_1 a_2 \cdots a_n$. Thus for each w in $L(G)$, there is a parse of w according to G' from which the parse of w according to G can be recovered by means of a homomorphism.

Using the productions added to P' in steps 2 and 3, E_a can derive any terminal symbol $b \neq a$, or any terminal string ending in a, or the empty string. Moreover, $S' \to SH$ is also a production and H derives any nonempty terminal string. Thus, since $L(G)$ is presumed to be nonempty, $L(G') = \Sigma^*$.

The grammar G' is ambiguous. We will parse an input string according to G' but because of the following theorem we can try to use as few terminal error productions as possible.

THEOREM 1. *x is in $E_k(L(G))$ if and only if there is a derivation of x in G' using k terminal error productions and no derivation using fewer than k terminal error productions.*

Proof. Let h be the homomorphism from P' to P such that $h(p') = p$ if p' is a production added to P' in step 1 of Algorithm 1 and $h(p') = e$ otherwise. To prove the theorem we shall prove the following statement by induction on k.

(*) $S \underset{G}{\overset{\pi}{\Rightarrow}} w$, $x \in E_k(L(G))$ and $w \overset{k}{\vdash} x$ if and only if $S' \underset{G'}{\overset{\pi'}{\Rightarrow}} x$, $h(\pi') = \pi$, π' contains k terminal error productions and there is no π'' with fewer terminal error productions such that $S' \underset{G'}{\overset{\pi''}{\Rightarrow}} x$.

Basis: If $k = 0$, π' will contain only productions from step 1 and step 2(a). Thus (*) is trivially true.

Inductive step: Suppose $S \underset{G}{\overset{\pi}{\Rightarrow}} w$, $y \in E_{k+1}(L(G))$ and $w \overset{k+1}{\vdash} y$. Then there is an x in $E_k(L(G))$ such that $w \overset{k}{\vdash} x \vdash y$. From the inductive hypothesis, there is a derivation $S' \underset{G'}{\overset{\pi'}{\Rightarrow}} x$ such that $h(\pi') = \pi$ and π' contains k terminal error productions and there is no derivation of x using fewer error productions. If $x \vdash y$ by T_R such that the ith symbol in x, say a, is replaced by an erroneous symbol, say b, then we can modify the derivation $S' \underset{G'}{\overset{\pi'}{\Rightarrow}} x$ replacing the production of the form $E_a \to a$ that is used to derive the ith terminal symbol[1] in x by the derivation $E_a \Rightarrow b$. Thus, $S' \underset{G'}{\overset{\pi''}{\Rightarrow}} y$, where $h(\pi'') = \pi$ and π'' contains $k + 1$ terminal error productions. Similar arguments can be made if $x \vdash y$ using transformation T_I or T_D.

Now suppose that there is a derivation $S' \underset{G'}{\overset{\rho}{\Rightarrow}} y$ such that $h(\rho) = \pi$ and ρ uses $l < k + 1$ terminal error productions. Clearly $l > 0$. Let the first terminal error production in ρ be $E_a \to b$. We could replace this production by $E_a \to a$ to obtain a parse ρ' such that $S' \underset{G'}{\overset{\rho'}{\Rightarrow}} x'$, $h(\rho') = \pi$ and ρ' contains $l - 1$ terminal error productions. But then by the inductive hypothesis x' would be in $E_{l-1}(L(G))$. However, this is impossible since $x' \vdash y$ and y is in $E_{k+1}(L(G))$. A similar argument prevails if the first terminal error production in ρ is $I \to a$ or $E_a \to e$.

The proof of the converse is straightforward.

6. Minimum distance parsing. We shall now describe a parsing algorithm for G' that uses as few terminal error productions as possible. This algorithm is

[1] Note that if $x \in E_k(L(G))$, $y \in E_{k+1}(L(G))$ and $x \vdash y$ by a replacement error in position i of x, then the ith symbol in x cannot be a replacement error.

essentially Earley's [2] algorithm (without look ahead) with a provision added to keep count of the number of terminal error productions used. We employ the notation used in [1] to describe Earley's algorithm.

Informally our algorithm works as follows. Let G' be the covering grammar for G and let $x = a_1 a_2 \cdots a_n$ be the input string to be parsed. We construct a sequence $\mathscr{I}_0, \mathscr{I}_1, \cdots, \mathscr{I}_n$ of lists of items. An item is an object of the form

$$[A \to \alpha \cdot \beta, i, k],$$

where

(i) $A \to \alpha\beta$ is a production in G';

(ii) $\cdot$ is a special metasymbol, indicating how much of the production is currently applicable to a parse;

(iii) i is an integer, $0 \leqq i \leqq n$, indicating the input position at which a derivation from α began.

(iv) k is a nonnegative integer indicating the number of terminal error productions used in the derivation from α.

Item $[A \to \alpha \cdot \beta, i, k]$ will be on list $\mathscr{I}_j$ if and only if for some γ in $(N \cup \Sigma)^*$,

$$S' \overset{*}{\underset{G'}{\Rightarrow}} a_1 a_2 \cdots a_i A \gamma, \tag{1}$$

$$\alpha \overset{*}{\underset{G'}{\Rightarrow}} a_{i+1} \cdots a_j \tag{2}$$

such that derivation (2) uses k terminal error productions and there is no derivation of $a_{i+j} \cdots a_j$ from α using fewer terminal error productions.

Note that x is in $L(G)$ if and only if $[S' \to S \cdot, 0, 0]$ is in $\mathscr{I}_n$. From Theorem 1, x is in $E_k(L(G))$ if and only if an item of the form $[S' \to \alpha \cdot, 0, k]$ is in $\mathscr{I}_n$ and no item of the form $[S' \to \beta \cdot, 0, l]$ is in $\mathscr{I}_n$ for $l < k$.

The sequence $\mathscr{I}_0, \mathscr{I}_1, \cdots, \mathscr{I}_n$ will be called the sequence of *parse lists* for x. The following algorithm can be used to construct the parse lists for an input string.

ALGORITHM 2.

Input: The covering grammar $G' = (N', \Sigma', P', S')$ and an input string $x = a_1 a_2 \cdots a_n$ in Σ^*.

Output: $\mathscr{I}_0, \mathscr{I}_1, \cdots, \mathscr{I}_n$, the parse lists for x.

Method: Initially, all lists are empty. Construct $\mathscr{I}_0$ as follows:

1. Add items $[S' \to \cdot S, 0, 0]$ and $[S' \to \cdot SH, 0, 0]$ to $\mathscr{I}_0$.
2. If $[A \to \alpha \cdot B\beta, 0, k]$ is in $\mathscr{I}_0$ and $B \to \gamma$ is a production in P', then add item $[B \to \cdot \gamma, 0, 0]$ to $\mathscr{I}_0$.
3. If $[A \to \alpha \cdot B\gamma, 0, k]$ and $[B \to \beta \cdot . 0, l]$ are in $\mathscr{I}_0$, then add $[A \to \alpha B \cdot \beta, 0, m]$ to $\mathscr{I}_0$, where $m = k + l + 1$ if $B \to \beta$ is a terminal error production and $m = k + l$ otherwise. Store with item $[A \to \alpha B \cdot \beta, 0, m]$ two pointers, the first to item $[A \to \alpha \cdot B\beta, 0, k]$, the second to item $[B \to \beta \cdot, 0, l]$. However, if $[A \to \alpha B \cdot \beta, 0, m']$ is already in $\mathscr{I}_0$ and $m' \leqq m$, then do not add $[A \to \alpha B \cdot \beta, 0, m]$ to $\mathscr{I}_0$. On the other hand, if $[A \to \alpha B \cdot \beta, 0, m'']$ is in $\mathscr{I}_0$ and $m'' > m$, then delete this item from $\mathscr{I}_0$.
4. Repeat steps 2 and 3 until no new items can be added to $\mathscr{I}_0$.

Suppose that $\mathscr{I}_0, \mathscr{I}_1, \cdots, \mathscr{I}_{j-1}$ have been constructed. We construct $\mathscr{I}_j$, $1 \leqq j \leqq n$, as follows:

5. For each item $[A \to \alpha \cdot a\beta, i, k]$ in $\mathscr{I}_{j-1}$ such that $a = a_j$, add $[A \to \alpha a \cdot \beta, i, k]$ to $\mathscr{I}_j$. Along with this item add a pointer to item $[A \to \alpha \cdot a\beta, i, k]$ in $\mathscr{I}_{j-1}$.

6. If $[B \to \gamma \cdot, i, k]$ is in $\mathscr{I}_j$ and $[A \to \alpha \cdot B\beta, h, l]$ is in $\mathscr{I}_i$, then add item $[A \to \alpha B \cdot \beta, h, m]$ to $\mathscr{I}_j$, where $m = k + l + 1$ if $B \to \gamma$ is a terminal error production and $m = k + l$ otherwise. Include with item $[A \to \alpha B \cdot \beta, h, m]$ two pointers, the first to item $[A \to \alpha \cdot B\beta, h, l]$ in $\mathscr{I}_i$ and the second to $[B \to \gamma \cdot, i, k]$ in $\mathscr{I}_j$.

However, if $[A \to \alpha B \cdot \beta, h, m']$ is already in $\mathscr{I}_j$ for some $m' \leqq m$, then do not add $[A \to \alpha B \cdot \beta, h, m]$ to $\mathscr{I}_j$. Likewise, if $[A \to \alpha B \cdot \beta, h, m'']$ is in $\mathscr{I}_j$ for some $m'' > m$, then delete this item from $\mathscr{I}_j$.

7. If $[A \to \alpha \cdot B\beta, i, k]$ is in $\mathscr{I}_j$ and $B \to \gamma$ is in P', then add $[B \to \cdot \gamma, j, 0]$ to $\mathscr{I}_j$.

8. Repeat steps 6 and 7 until no new items can be added to $\mathscr{I}_j$.

In this manner construct the sequence of parse lists $\mathscr{I}_0, \mathscr{I}_1, \cdots, \mathscr{I}_n$.

This algorithm is essentially Earley's algorithm with one additional field in each item to keep track of the number of terminal error productions used. We are only interested in derivations using as few terminal error productions as possible. The pointers stored with the items will be used to reconstruct a parse from the sequence of parse lists. The following lemmas describe the behavior of Algorithm 2.

Let $\mathscr{I}_0, \mathscr{I}_1, \cdots, \mathscr{I}_n$ be the parse lists for $x = a_1 a_2 \cdots a_n$ constructed by Algorithm 2.

LEMMA 1. *If* $[A \to \alpha \cdot \beta, i, k]$ *is in* $\mathscr{I}_j$, *then*

(i) $S' \overset{*}{\Rightarrow} a_1 \cdots a_i A\gamma$ *for some* γ, *and*

(ii) $\alpha \overset{*}{\Rightarrow} a_{i+1} \cdots a_j$ *using* k *terminal error productions.*

Proof. The proof is a straightforward induction on the order in which items are added to the parse lists. □

LEMMA 2. *If*

(i) $S' \underset{lm}{\overset{*}{\Rightarrow}} a_1 \cdots a_h A\beta' \underset{lm}{\Rightarrow} a_1 \cdots a_h \alpha\beta\beta'$,

(ii) $\alpha \overset{*}{\Rightarrow} a_{h+1} \cdots a_i$ *using* k *terminal error productions,*

(iii) $\beta \overset{*}{\Rightarrow} a_{i+1} \cdots a_j$ *using* l *terminal error productions, and*

(iv) *there is no derivation of the form* $\alpha\beta \overset{*}{\Rightarrow} a_{h+1} \cdots a_j$ *using fewer than* $k + l$ *terminal error productions,*

then $[A \to \alpha \cdot \beta, h, k]$ *is in* $\mathscr{I}_i$ *and* $[A \to \alpha\beta \cdot, h, k + l]$ *is in* $\mathscr{I}_j$.

Proof. The proof is an induction on the sum of the lengths of derivations (i), (ii) and (iii). □

From Lemmas 1 and 2 we can conclude that x is in $E_k(L(G))$ for some $k \geqq 0$ if and only if $\mathscr{I}_n$ contains an item of the form $[S' \to \alpha \cdot, 0, k]$ and no item of the form $[S' \to \beta \cdot, 0, l]$, where $l < k$.

From the parse lists we can extract a parse for x that uses the fewest number of terminal error productions by means of the following algorithm.

ALGORITHM 3.

Input: $\mathscr{I}_0, \mathscr{I}_1, \cdots, \mathscr{I}_n$, the parse lists for $x = a_1 \cdots a_n$.

Output: A parse π for x according to G' such that π contains as few terminal error productions as possible.

Method: In $\mathscr{I}_n$ choose an item of the form $[S' \to \alpha \cdot, 0, k]$, where k is as small as possible. Let π initially be the empty string. Then execute the routine *parse*($[S' \to \alpha \cdot, 0, k], \mathscr{I}_n$) where *parse*($[A \to \alpha \cdot \beta, i, l], \mathscr{I}_j$) is defined as follows:

1. If $\beta = e$, then let π be the previous value of π followed by production $A \to \alpha$. Otherwise, π is unchanged.

2. (a) If $\alpha = \alpha' a$, execute *parse*($[A \to \alpha' \cdot a\beta, i, l], \mathscr{I}_{j-1}$), where $[A \to \alpha' \cdot a\beta, i, l]$ is the item in $\mathscr{I}_{j-1}$ to which item $[A \to \alpha' a \cdot \beta, i, l]$ on $\mathscr{I}_j$ has a pointer. Return.

(b) If $\alpha = \alpha' B$, then execute *parse*($[B \to \gamma \cdot, h, m], \mathscr{I}_j$) followed by *parse*($[A \to \alpha' \cdot B\beta, i, k], \mathscr{I}_h$), where $[A \to \alpha' \cdot B\beta, i, k]$ on $\mathscr{I}_h$ and $[B \to \gamma \cdot, h, m]$ on $\mathscr{I}_j$ are the two items pointed to by item $[A \to \alpha \cdot \beta, i, l]$ on $\mathscr{I}_j$. Return.

(c) If $\alpha = e$, return. □

Algorithm 3 traces out a right-most derivation of x using the pointers stored with the items to guide the derivation. After executing *parse*($[S' \to \alpha \cdot, 0, k], \mathscr{I}_n$), π will be a sequence of productions in a right-most derivation of x from S' in G' using k terminal error productions. We can obtain a parse of a string w in $L(G)$ such that $w \overset{k}{\vdash} x$ by applying the homomorphism h in the proof of Theorem 1 to π.

Now let us examine the time complexity of finding a minimum distance parse for an input string $x = a_1 \cdots a_n$. We shall use the notation $g(n)$ is $O(f(n))$ to mean there is a constant c such that $g(n) \leqq cf(n)$ for all $n \geqq 1$.

LEMMA 3. *Algorithm 2 can be implemented to run in time $O(n^3)$ on a random access computer, where n is the length of the input string to be parsed.*

Proof. Steps 1–4 of Algorithm 2 compute $\mathscr{I}_0$. Since $\mathscr{I}_0$ contains a fixed number of items, these steps can be executed in constant time.

Let us now examine the amount of time required to compute $\mathscr{I}_j$. We first note that each list of items $\mathscr{I}_i$, $0 \leqq i < j$, contains at most ci items for some constant c, because for each h, $0 \leqq h \leqq i$, there is at most one item of the form $[A \to \alpha \cdot \beta, h, k]$ on $\mathscr{I}_i$. Thus, step 5 of Algorithm 2 can be executed in $O(j - 1)$ time.

We shall now show that the repeated application of steps 6 and 7 can be implemented in $O(j^2)$ time in the following manner. We can construct a directed graph D in which each node of D is labeled by an item on $\mathscr{I}_j$ and an edge is drawn from a node labeled I to a node labeled I' if item I on $\mathscr{I}_j$ can cause item I' to appear on $\mathscr{I}_j$ because of step 6 or 7. For example, if item $[B \to \gamma \cdot i, \ ]$ is on $\mathscr{I}_j$ and list $\mathscr{I}_i$ contains item $[A \to \alpha \cdot B\beta, h, k]$, then by step 6 we must add item $[A \to \alpha B \cdot \beta, h, \ ]$ to $\mathscr{I}_j$. Thus we would have a node I labeled $[B \to \gamma \cdot, i, \ ]$ in D and an edge from this node to a node I' labeled $[A \to \alpha B \cdot \beta, h, \ ]$. We would also label the edge from I to I' with k, the error count of item $[A \to \alpha \cdot B\beta, h, k]$ on $\mathscr{I}_i$. We shall use k subsequently to help determine the error count for item $[A \to \alpha B \cdot \beta, h, \ ]$. When we first construct this graph, however, we shall initially leave the error counts in all items in $\mathscr{I}_j$ empty except for items of the form $[A \to \alpha a_j \cdot \beta, i, k]$, those whose error count is zero, and those of the form $[A \to \alpha \cdot, j, k]$. Thus the graph D will contain $O(j)$ nodes and $O(j^2)$ edges, and can be constructed in time $O(j^2)$.

To determine the items that will finally be on $\mathscr{I}_j$ we must now determine the correct values for the empty error counts. We can find these values in time $O(j^2)$ as follows. First we isolate the strongly connected components of D. This can be done in time proportional to the number of edges in D. (See [9], for example.) If we treat the strongly connected components of D as single nodes, we have essentially reduced D to a directed acyclic graph D'. We can then evaluate the minimum error counts for the nodes of D by examining the nodes of D in such an order that all direct predecessors of a node N in D' are examined before N is examined.

If N represents a strongly connected component of D, then we need to traverse the edges in N only a fixed number of times to percolate the minimum error counts throughout the nodes of N. This follows from the fact that consideration of an item I cannot cause its own error count to subsequently decrease.

In this fashion we can determine the minimum error count for all items labeling the nodes of D in time proportional to the number of edges in D.

In conclusion, Algorithm 2 can be implemented in $O(n^3)$ time because each list of items can be created in $O(n^2)$ time. □

Algorithm 3 can be implemented in $O(n)$ time. Also the parse for the word in $L(G)$ can be created from the parse according to G' in $O(n)$ time. Thus we have a minimum distance error-correcting parser that operates in time $O(n^3)$.

7. Concluding remarks. We can extend this parsing method to include other types of errors. For example, certain transposition errors can be generated by error productions and we could include these productions with the other error productions.

If we do not want to generate all of Σ^* with our covering grammar, we could restrict the error productions so that they would only generate the syntax errors that are most likely to occur. Wirth [10] has considered a scheme of this nature in conjunction with precedence parsing.

In compiler applications it is desirable to use a fast parsing algorithm such as LL or LR parsing. On encountering an error we could invoke our error-correcting parser. Other methods for error recovery and correction in LR parsing are discussed in [4]–[8].

Finally, it is interesting to ask how a programming language can be designed so as to maximize the distance between correct programs.

REFERENCES

[1] A. V. AHO AND J. D. ULLMAN, *The Theory of Parsing, Translation and Compiling*, Vol. I, *Parsing*, Prentice-Hall, Englewood Cliffs, N.J., 1972.

[2] J. EARLEY, *An efficient context-free parsing algorithm*, Comm. ACM, 13 (1970), pp. 94–102.

[3] E. T. IRONS, *An error-correcting parse algorithm*, Ibid., 6 (1963), pp. 669–673.

[4] L. R. JAMES, *A syntax directed error recovery method*, Tech. Rep. CSRG-13, Computer Systems Research Group, University of Toronto, Toronto, 1972.

[5] R. P. LEINIUS, *Error detection and recovery for syntax directed compiler systems*, Doctoral thesis, University of Wisconsin, Madison, 1970.

[6] J. P. LEVY, *Automatic correction of syntax errors in programming languages*, Tech. Rep. 71–116, Dept. of Computer Science, Cornell University, Ithaca, N.Y., 1971.

[7] G. T. MCGRUTHER, *An approach to automating syntax error detection, recovery, and correction for LR(k) grammars*, M.S. thesis, Naval Postgraduate School, Monterey, Calif., 1972.

[8] T. G. PETERSON, *Syntax error detection, correction and recovery in parsers*, Doctoral thesis, Stevens Institute of Technology, Hoboken, N.J., 1972.

[9] R. TARJAN, *Depth-first search and linear graph algorithms*, this Journal, 1 (1972), pp. 146–160.

[10] N. WIRTH, *PL*360, *a programming language for the* 360 *computers*, J. Assoc. Comput. Mach., 15 (1968), pp. 37–74.

SIAM J. COMPUT.
Vol. 1, No. 4, December 1972

COMPATIBILITY AND COMPLEXITY OF REFINEMENTS OF THE RESOLUTION PRINCIPLE*

RICHARD B. KIEBURTZ† AND DAVID LUCKHAM‡

Abstract. This paper studies a number of logically complete search strategies (refinements) for improving the performance of automatic theorem-proving programs based on the resolution principle. These strategies restrict the number of deductions generated by the program at the expense of sometimes missing the shortest proof.

By considering elementary proof-preserving transformations on resolution proof trees, (i) it is shown that the conjunction of set-of-support, resolution-with-merging, and linear form deduction is again a complete refinement; (ii) bounds are obtained on the possible increase in complexity of the proof trees when the linear form and resolution-with-merging refinements are imposed.

Finally, examples are given which demonstrate the savings in time and storage when refinements are used to prove some theorems of moderate difficulty in group theory and ternary boolean algebra.

Key words. theorem-proving, resolution principle, mathematical logic, artificial intelligence.

1. Introduction. In experiments with automatic deduction programs based on the resolution principle it is necessary, in order to get proofs of interesting theorems, to restrict the resolvents which are deduced by the program. If resolution was pursued exhaustively without any restriction, the available memory space in even the largest existing implementations would usually be filled before a proof was found.

Some of the most useful methods for restricting the deductions operate by providing a condition on finite sets of clauses so that the program generates deductions from only those sets satisfying the given condition. Below we shall discuss the effect some of these conditions have on the set $R^n(S)$ of resolvents of level $l \leqq n$ deducible from the initial set S of clauses. Let $R(A, B)$ denote the set of resolvents of clauses A and B, let $P(A, B)$ be a condition on pairs of clauses, and let $\tilde{R}^n(S)$ denote the subset of $R^n(S)$ defined by

$$\tilde{R}^0(S) = S,$$

$$\tilde{R}^{n+1}(S) = \{C|(C \in R(A, B) \,\&\, A, B \in \tilde{R}^n(S) \,\&\, P(A, B)) \vee C \in \tilde{R}^n(S)\}.$$

It turns out that these methods often yield a *refinement* of the resolution principle in the sense that $\tilde{R}^n(S)$ is a proper subset of $R^n(S)$ for all n, and at the same time the completeness of the proof procedure is preserved.

Several refinements have been proposed by previous authors. First of all, Wos, Robinson and Carson (1965) introduced the set-of-support strategy whereby every clause deduced is required to have a predecessor in a subset K of an inconsistent set S of basis clauses, where $S - K$ is a consistent set. About the same time, J. A. Robinson (1965a) developed the notion of hyper-resolution from a refinement which involved computing resolvents of pairs of clauses only if one of them was false of a particularly simple model. More recently, Slagle [12] and Luckham [8]

* Received by the editors November 10, 1971, and in revised form November 1, 1972.

† Department of Computer Sciences, State University of New York at Stony Brook, Stony Brook, New York 11790.

‡ Stanford Artificial Intelligence Laboratory, Stanford University, Stanford, California 94305.

have considered various forms of deduction relative to more general models. Andrews [3] has studied resolution with merging, in which a pair of clauses is resolved only if it contains either a clause from S or else a "merge" (a merge is a clause C deduced by resolution from clauses A and B such that some literal of C has ancestors in *both* A and B). Finally, Loveland [7] and Luckham [8] have introduced the ancestry-filter form deduction (also commonly called linear format) in which the deduced clauses form a linear sequence $\{C_1, C_2, \cdots, C_n\}$ such that C_1 and C_2 are members of S, and for any $i > 2$, C_i is a resolvent of C_{i-1} and either a deduced clause $C_j, j \leqq i - 1$, *or* a member of S. The point of all of these refinements is to restrict the resolvents generated at any level n to $\tilde{R}^n(S)$. This, hopefully, delays the exponential explosion in the number of resolvents long enough to find a proof. Several experiments in proving theorems of elementary theories have been published which show refinements to be helpful and often crucial in obtaining proofs.

In operation, a theorem-proving program may run a number of such refinements in conjunction with each other and with editing strategies which eliminate some clauses when they are generated. Typical editing strategies set bounds on the length of clauses or on the depth of functional nesting in any term, or eliminate clauses which are redundant in the sense that they are subsumed by some clause already deduced. We shall say that a given set of refinements and strategies is *compatible* if the deductive system remains logically complete when those refinements and strategies are used in conjunction (i.e., only those deductions satisfying the conjunction of all the individual constraints for each refinement and strategy, are retained). In certain practical situations it is important to know if the refinements are compatible with one another or with the editing strategies. This information is often helpful to the user, for example, when the program fails to find a proof when run with a particular set of refinement and editing strategies.

The following results are known concerning compatability of these refinements. Set-of-support is compatible with resolution-with-merging [3] and with ancestry-filter form deduction [7], [8], and jointly, with merging and ancestry-filter [2], [14]. Model-relative deduction (and hyper-resolution) is not compatible with resolution-with-merging or with ancestry-filter form when the choice of model is unrestricted [8].

However, the question of compatibility is only a part of the overall question of usefulness. Although the experimental evidence so far favors the use of these refinements, it is not at all clear when their use, singly or in conjunction, improves the chances of finding a proof. There is, for example, the additional question of the effect of a refinement on the complexity of the proof. When applied to some problems, a refinement can exclude the simplest proofs from consideration, and force the program to search for longer proofs or proofs containing more complicated functional terms. The usefulness of one or more refinements may therefore be a question of the advantages gained in a "trade-off" between generating fewer resolvents at each level, and computing resolvents at deeper levels to find a proof. Sometimes more complex proofs are easier to find with a given refinement than the simpler proofs are without it. The outcome of such a trade-off is therefore not obvious, but without information on possible increase in complexity we are in no position to study it. Furthermore, in practice, the user must allow for a possible

increase in proof complexity (if he knows about it) by relaxing the relevant editing strategies.

We shall give sharp bounds on the increase in level of the proof-tree and in depth of functional nesting in terms which may occur when these refinements are employed. First, we give a method for constructing from any general resolution proof, another general resolution proof satisfying the conjunction of the refinement conditions. The complexity bounds are derivable directly from the construction. In addition the methods used here represent a step in the spirit of Andrews [3] towards developing a calculus of operations on proof-trees. This may help to provide a basic theory for certain promising applications of resolution proof procedures, for example, question-answering and proof by analogy.

Some questions concerning compatibility remain open. For what useful[1] choices of model are the model-relative refinements of [12] and [8] compatible with ancestry-filter form? Similarly, when is hyper-resolution compatible with ancestry-filter form? The conjunction of ancestry-filter form and the general subsumption editing strategy is incomplete [6]. Which strategy should be modified and how? In [6] it is proved that if the set of hypotheses S is free of subsumptions, then ancestry-filter form is compatible with the strategy of deleting any resolvent in a proof tree which is subsumed by a previous clause in that proof tree or by a member of S. It should be noted that such results concerning subsumption cannot be derived from similar results about ground clauses, since a subsumption-free ground deduction may in fact, be an instance of a general resolution deduction which is *not* subsumption-free.

2. Definitions and notation. We shall assume that the reader is familiar with resolution as a rule of inference and shall not define the basic entities clause, literal, atom, resolvent, unifier, etc. Individual clauses are denoted by upper case letters A, B, C, D, E, sometimes with superscripts and subscripts. Literals are lower case letters $p, q, \neg p, \neg q$, and by $|p|$ we shall mean the *atom* of the literal p. Substitutions are denoted by lower case Greek letters.

By a *resolution tree*, we shall mean a finite, connected, directed binary tree with arrows on the arcs directed from the end (or leaf) nodes toward the root and in which every node is labeled by a clause. The clause appearing on any node which is not a leaf of the tree is a resolvent of the clauses appearing on the two nodes immediately preceding it. The precedence relation defined by the arrows is a partial order; we follow convention in defining the precedence as a reflexive relation. We shall speak of nodes which are *predecessors* or *successors* of a given node: when we mean the predecessor or successor which is connected to a node by an arc with no intervening nodes, we use the adjective *immediate.*

In everything that follows, we shall restrict our discussion to binary resolution trees. We denote resolution trees by Tr. To designate the set of nodes which are leaves of a resolution tree we use the notation $\Lambda(\mathrm{Tr})$. Individual nodes will be denoted by lower case letters u, v, w, x, y, z, often with subscripts and primes. The clause attached to a particular node x will be denoted by $\mathrm{cl}(x)$. The set of clauses which appear on the leaves of a tree is called the *base set*, denoted Basis (Tr). We

[1] If every clause in S is false of the model, and model-relative refinements do not restrict the resolvents.

shall sometimes refer to a clause in the base set as a *leaf clause* of a resolution tree Tr. A resolution tree Tr is said to *deduce A from an axiom set S* if $A = \mathrm{cl}(\mathrm{root\ Tr})$, and for every leaf $x \in \Lambda(\mathrm{Tr})$, $\mathrm{cl}(x)$ is a clause in S.

The ordinal, or *level* of any resolution tree Tr is defined inductively as follows. If root $\mathrm{Tr} \in \Lambda(\mathrm{Tr})$ then $\ell(\mathrm{Tr}) = 0$. Otherwise, root Tr has immediate predecessors y_1 and y_2. Let Tr_1 and Tr_2 be the largest subtrees rooted on y_1 and y_2 respectively. Then $\ell(\mathrm{Tr}) = \max[\ell(\mathrm{Tr}_1), \ell(\mathrm{Tr}_2)] + 1$. The level of a clause B in a resolution tree Tr is defined as the level of the smallest subtree of Tr which deduces B from S. The cardinality of a resolution tree (i.e., the finite number of nodes of the tree) is denoted by $|\mathrm{Tr}|$.

If x and y are nodes of a resolution tree Tr, and x is a predecessor of y, and a literal $p\theta$ occurs in $\mathrm{cl}(y)$, then an occurrence of p in $\mathrm{cl}(x)$ is defined to be an *ancestor* of the occurrence of $p\theta$ in $\mathrm{cl}(y)$ in case one of the following sets of conditions holds: Either

(i) $x = y$ and θ is the empty substitution;

(ii) y has a predecessor z in Tr whose immediate predecessors are nodes x and x_1; σ is the unifier employed in the resolution of $\mathrm{cl}(x)$ and $\mathrm{cl}(x_1)$; $p\sigma \in \mathrm{cl}(z)$ is an ancestor of $p\theta$ in y; p in $\mathrm{cl}(x)$ is not eliminated by the resolution of x and x_1.

Note that this defines ancestry even in trees which contain tautologous clauses.

A resolution tree is said to be in *vine-form* if, for every node which is not a leaf, at least one of its immediate predecessors is a leaf. A resolution tree Tr which is in vine-form is said to be a *vine* relative to S if Basis (Tr) $\subseteq S$. There is one and only one node of a vine-form tree which has two immediate predecessors, both of which are leaves. One of these leaves is designated as the *top node* of the vine-form tree.

The *stem* of a vine-form tree, denoted by Stem (Tr), is the ordered set of nodes which consists of the top node and all of its successors in the tree. The ordering is that imposed by the precedence relation, and the nodes of Stem (Tr) can be indexed from 0 to $\ell(\mathrm{Tr})$.

A resolution tree Tr representing a deduction from S is said to be in *ancestry-filter form* (AFF) if it has a vine-form subtree Tr′ (not necessarily representing a deduction from S) whose root is common with the root of Tr, satisfying the following:

(i) For every node $y_i \in \mathrm{Stem}(\mathrm{Tr}')$, $i > 0$, if x is an immediate predecessor of y_i, then either $\mathrm{cl}(x)$ is a clause of S or else $\mathrm{cl}(x) = \mathrm{cl}(y_j)$ for some $y_j \in \mathrm{Stem}(\mathrm{Tr}')$, $j < i$.

(ii) If x is an immediate predecessor of $y_i \in \mathrm{Stem}(\mathrm{Tr}')$ such that $\mathrm{cl}(x) = \mathrm{cl}(y_j)$ where $y_j \in \mathrm{Stem}(\mathrm{Tr}')$, then the maximal subtree of Tr rooted at x is identical to the maximal subtree rooted at y_j.

Note that Tr′ is unique, up to isomorphism. Thus the clauses deduced on the vine-form subtree of an AFF resolution tree form a sequence in which each clause is a resolvent of the previously deduced clause with either a member of the base set S *or* an instance of a clause deduced previously in the sequence.

A clause A is said to be a *merge* in Tr if A is a resolvent of clauses B_1 and B_2 such that $B_1 = \tilde{B}_1 \dot{\cup} P$, $B_2 = \tilde{B}_2 \dot{\cup} Q$, θ is a unifier of (P, Q), $A = \tilde{B}_1\theta \cup \tilde{B}_2\theta$,

and $|A| < |\tilde{B}_1\theta| + |\tilde{B}_2\theta|$.[2] Thus if there are literals $p' \in B_1$ and $q' \in B_2$ such that $p'\theta = q'\theta$, then A is a merge, the literal $p'\theta$ is said to be a *merge literal*, and the node z is said to be a *merge node*. The *merge set* of a resolution tree, $M(\mathrm{Tr})$, is the set of clauses which are merges in Tr.

A resolution tree Tr is said to be a deduction of A from S by resolution with merging if Tr deduces A from S, and the clause of at least one of the immediate predecessors of every nonleaf is in $S \cup M(\mathrm{Tr})$.

3. The ground resolution lemma. In proving many theorems about resolution trees, it is more convenient to give proofs for ground resolution than for general resolution. This avoids the question of specifying the substitutions involved, or of keeping track of factoring of literals. However, the applicability to general resolution of a theorem proved about ground resolution must always be established.

The most important lemma in establishing the generalization of ground resolution theorems to general resolution is the so-called lifting lemma, given by J. A. Robinson [11]. One version of the lemma is as follows.

LIFTING LEMMA. *Let S be any set of clauses, and H_S be the corresponding Herbrand universe of terms. If $\tilde{S}$ is any set of ground instances of clauses in S, instantiated over H_S, and if $\widetilde{\mathrm{Tr}}(\tilde{A})$ is a ground resolution tree deducing $\tilde{A}$ from $\tilde{S}$, then there is a resolution tree* $\mathrm{Tr}(A)$ *deducing A from S by general resolution, an isomorphism $f: \widetilde{\mathrm{Tr}} \to \mathrm{Tr}$, and for every node $\tilde{x} \in \widetilde{\mathrm{Tr}}$,* $\mathrm{cl}(\tilde{x})$ *is a ground instance of* $\mathrm{cl}(f\tilde{x})$.

The completeness of a given refinement can be established by showing that it is complete for ground resolution, providing we know that the refinement "lifts"—i.e., that if the ground resolution tree $\widetilde{\mathrm{Tr}}$ satisfies the refinement condition, then the "lifted" tree, Tr, also satisfies it. For example, if $\widetilde{\mathrm{Tr}}$ is AFF, then Tr is also AFF; but the condition between clauses A and B at different nodes of $\widetilde{\mathrm{Tr}}$, "A does not subsume B", need not be true of Tr.

The use of the lifting lemma usually leads to nonconstructive completeness proofs. That is, the proof does not provide an algorithm for mapping a given general resolution tree into another one which satisfies the refinement. Without such a transformation between the general resolution trees, it is difficult to measure the way in which the refinement affects the complexity of the tree.

Thus we give a lemma which will be needed to obtain bounds on the increase in complexity of a proof tree which may occur because of the imposition of a refinement. The lemma is, in a weak sense, the converse of the lifting lemma, for while it is not true that every deduction by general resolution has an exact counterpart by ground resolution, we establish the existence of a counterpart which is related by a subsumption condition.

First, however, we must introduce some additional notation. We shall say that a clause B is a *σ-instance* of a clause A if $B = A\sigma$. Composition of substitutions is defined in the obvious way; $\theta = \lambda\tau$ if for all clauses A, $A\theta = (A\lambda)\tau$. We shall say that a substitution θ is a *special case* of a substitution τ, if there is a substitution λ for which $\theta = \tau\lambda$.

[2] Following [3], we use the notation $\dot{\cup}$ to indicate the union of sets which are disjoint.

If a substitution θ is the result of applying the unification algorithm [10] to the pair of sets of literals (P, Q), then θ is said to be a *most general unifier* (m.g.u.) of (P, Q), and has the property that any other substitution which unifies (P, Q) is a special case of θ. If every resolvent in a resolution tree is deduced by employing a m.g.u. of the pair of sets of literals which are eliminated, we say that it is a *general resolution tree.*

We shall also adopt the convention, when dealing with general resolution, that the variables occurring in each leaf clause shall be distinct from the variables in any other leaf clause. With this convention, all unifiers in any given resolution tree can be composed into a single substitution which unifies the literals at every resolution in the tree. We define a *general simultaneous unifier* (g.s.u.) recursively as follows. If Tr is a general resolution tree rooted on z, then $\tilde{\theta}$ is a g.s.u. of Tr if either:

(i) $\ell(\mathrm{Tr}) = 0$ and $\tilde{\theta} = \varepsilon$, the empty substitution, or,

(ii) $\ell(\mathrm{Tr}) > 0$;

z has immediate predecessors y_1 and y_2 and the subtrees rooted on these nodes have g.s.u. $\tilde{\theta}_1$ and $\tilde{\theta}_2$; there is a m.g.u. θ_z which unifies sets of literals from $\mathrm{cl}(y_1)$ and $\mathrm{cl}(y_2)$ in the resolution at z; and $\tilde{\theta} = (\tilde{\theta}_1 \cup \tilde{\theta}_2)\tilde{\theta}_z$.

The combination of the g.s.u. $\tilde{\theta}_1$ and $\tilde{\theta}_2$ by taking their union (as sets of substitution elements) is equivalent to the (commutative) composition $\tilde{\theta}_1\tilde{\theta}_2$ because these two substitutions contain no variables in common.

We also note a property of any substitution θ which is a m.g.u. or a g.s.u., that $\theta\theta = \theta$.

LEMMA 1. *Let* Tr *be a general resolution tree deducing A from S. Suppose $\tilde{\theta}$ is a general simultaneous unifier of* Tr, *and σ is any substitution which is a special case of $\tilde{\theta}$. Then there is a ground resolution tree* Tr′ *deducing an instance A' of A, and an order-preserving, surjective map f*: Tr → Tr′ *satisfying*

(i) $(\forall y' \in \mathrm{Tr}')(\exists y \in \mathrm{Tr})\ (y' = fy$ *and* $\mathrm{cl}(y') \subseteq \mathrm{cl}(y)\sigma)$,

(ii) $A' \subseteq A\sigma$,

(iii) *every leaf clause of* Tr′ *is σ-instance of a clause from S.*

Proof. The proof is by induction on the level $\ell(\mathrm{Tr})$.

Basis step: If $\ell(\mathrm{Tr}) = 0$, then Tr consists of a single node. Let $A' = A\sigma$, and the lemma is satisfied trivially.

Induction step: Assume the lemma holds for trees of level n or less and suppose $\ell(\mathrm{Tr}) = n + 1$. Let y_1 and y_2 be the immediate predecessors of root Tr. Let $B_1 = \mathrm{cl}(y_1)$, $B_2 = \mathrm{cl}(y_2)$ and λ_1 and λ_2 be g.s.u. of Tr_1 and Tr_2, respectively. There are sets of literals, P and Q, such that $\tilde{B}_1 \dot{\cup} P, = B_1$ $\tilde{B}_2 \dot{\cup} Q, = B_2$ and a substitution θ, $\theta \in$ m.g.u.(P, Q), for which $A = \tilde{B}_1\theta \cup \tilde{B}_2\theta$.

As a consequence of the separation of variables convention, the g.s.u. satisfy $\tilde{\theta} = (\tilde{\theta}_1 \cup \tilde{\theta}_2)\theta$. Let $\sigma = \tilde{\theta}\lambda$ for some λ. Then, since $\tilde{\theta}_1 \cup \theta_2 = \tilde{\theta}_1\tilde{\theta}_2$ (because variables are separated) we have $\sigma = \tilde{\theta}_1\tilde{\theta}_2\theta\lambda$, and find that σ is a special case of $\tilde{\theta}_1$ (and of $\tilde{\theta}_2$, by a symmetrical argument).

Apply the induction hypothesis to $\mathrm{Tr}_1\sigma$, obtaining Tr_1', deducing B_1'. If $P\sigma \not\subseteq B_1'$, then $B_1' \subseteq \tilde{B}_1\sigma \subseteq A\sigma$. To justify the second inclusion, we note that $\tilde{\theta}_1 \cup \tilde{\theta}_2$ does not replace any variables which occur in $\tilde{B}_1$, $\tilde{B}_2$ or A. Thus, $A\sigma = A(\tilde{\theta}_1 \cup \tilde{\theta}_2)\theta\lambda = A\theta\lambda = (\tilde{B}_1\theta \cup \tilde{B}_2\theta)\theta\lambda = \tilde{B}_1\theta\lambda \cup \tilde{B}_2\theta\lambda$, and also $\tilde{B}_1\sigma = \tilde{B}_1\theta\lambda$. In this case, let $A' = B_1'$ and $\mathrm{Tr}' = \mathrm{Tr}_1'$. Let $f = f_1 \cup f_1'$, where $f_1 : \mathrm{Tr}_1 \to \mathrm{Tr}_1'$ is

the map obtained from the induction hypothesis, and f'_1 has as its domain all nodes of Tr which are not in Tr_1, and maps all such nodes into root Tr'_1.

If $P\sigma \subseteq B'_1$, apply the induction hypothesis to Tr_2. If $Q\sigma \not\subseteq B'_2$, then let $\mathrm{Tr}' = \mathrm{Tr}'_2$ and choose A' and f in a similar manner to that given above.

Otherwise, $P\sigma \subseteq B'_1$ and $Q\sigma \subseteq B'_2$. Since $B'_1 \subseteq B_1$, $B'_2 \subseteq B_2$, and σ is a unifier of (P, Q), there is an immediate resolvent of B'_1 and B'_2 (requiring only the empty substitution as a unifier) which is A'. It is evident that $A' \subseteq A\sigma$.

Let Tr′ consist of the subtrees Tr'_1 and Tr'_2, whose root nodes are immediate predecessors of root Tr′, and let cl(root Tr′) = A'. Then Tr′ is a resolution tree deducing A' from σ-instances of clauses from S. Finally let $f = f_1 \cup f_2 \cup f_0$, where f_1 and f_2 are obtained from the hypothesis and f_0 maps root Tr onto root Tr′. The construction is complete.

COROLLARY 1.1. *If S is any set of clauses, and* Tr *is a resolution tree deducing* Nil, *the empty clause, from S, then there is a ground substitution* σ, *and a finite set of ground clauses S′, each of which is a* σ*-instance of a clause from S, and a ground resolution tree* Tr′ *which deduces* Nil *from S′.*

The proof follows immediately from Lemma 1 by choosing the instantiating substitution, σ, to produce a ground instance of every leaf clause in Tr.

The instantiation lemma (Lemma 1), together with the lifting lemma provide us with a chain of transformations: instantiate an arbitrary resolution tree to a ground resolution tree, transform this to a ground tree satisfying a refinement, and then lift the result back to a general resolution tree satisfying the refinement.

It is of great practical importance to know whether or not a specific refinement strategy may actually increase some measure of the complexity of a proof. Typical complexity measures used in automatic deduction include the level of a resolution tree, the maximum number of literals in a clause, and the maximum depth of function nesting in any term. The maximum depth of function nesting is a particularly crucial measure of complexity, as it restricts the size of the subset of the Herbrand universe within which a proof may be found.

We can immediately give bounds on the depth of function nesting in the instantiated tree Tr′, in terms of the maximum depth of nesting in Tr.

COROLLARY 1.2. *Let* Tr(A) *deduce A from S by general resolution. Suppose* $\ell(\mathrm{Tr}) = n$, *d is the maximum depth of function nesting in the leaf clauses of* Tr, *and d′ is the maximum depth of function nesting in any term of a unifier in* Tr. *Then there is a ground resolution tree* Tr′(A') *deducing an instance A′ of A from S′, a finite set of ground instances of the clauses of S, and the maximum depth of function nesting in the clauses of S′ is* $nd' + d$.

Proof. Assume Tr has the g.s.u. $\tilde{\theta}$ and let $\sigma = \tilde{\theta}\lambda$, where λ replaces all variables in Tr that are not replaced by $\tilde{\theta}$, by the constant a. Applying Lemma 1 to Tr, σ yields a ground tree, Tr′. We claim that Tr′ satisfies the corollary.

First we study the nesting depth in $\tilde{\theta}$. Let u be a level $k + 1$ node in Tr. Suppose nodes v and w are the immediate predecessors of u, and that kd' is an upper bound on the depth of nesting of functions in the terms of $\tilde{\theta}_v$ and $\tilde{\theta}_w$. Now, $\tilde{\theta}_u = (\tilde{\theta}_v \cup \tilde{\theta}_w)\theta_u$, where θ_u is the unifier at u. Therefore the terms in $\tilde{\theta}_u$ cannot have a depth of nesting greater than $(k + 1)d'$. Thus, by a simple induction, the depth of nesting in $\tilde{\theta}$, and hence in $\tilde{\theta}\lambda$, is bounded by nd'. This implies that the nesting in Tr′ is bounded by $nd' + d$, which completes the proof.

We shall make further use of this corollary to establish a bound on the increase of depth of functional nesting which may be required by the ancestry-filter refinement strategy.

4. Compatability of the ancestry-filter and resolution with merging refinement. In this section we sketch a procedure for constructing, from an unrestricted resolution proof tree, a new proof tree which satisfies the restrictions of the ancestry-filter and resolution-with-merging refinements. The construction could be the basis for a proof of compatibility of the refinements, although such a proof is not given here.

CONSTRUCTION. Let Tr be a ground resolution tree deducing a clause A from a basis set S. Let w be any leaf node of Tr. We wish to construct a ground resolution tree Tr′ deducing a clause A' from S', and having the properties:

1. $A' \subseteq A$;
2. Tr′ is AFF and if w' is the top node of Tr′, then $\text{cl}(w') = \text{cl}(w)$;
3. if y_i is in Stem Tr′ and a node x is resolved with y_i in Tr′, then $\text{cl}(x) \in S \cup M_i(\text{Tr}')$.

The conditions imply that Tr′ satisfies both the AFF and the resolution-with-merging refinements.

The construction is described as a recursive procedure. In order to guarantee a termination condition for the recursion, let us introduce the artifice of node-marking in the proof tree to which we shall apply the construction. Let St be the largest subtree of Tr which has top node w and which satisfies conditions 2 and 3. St cannot be empty since it contains at least the subtree consisting of w alone. Initially, let all nodes of Tr which are not also in St be marked, as indicated by check marks in Fig. 1.

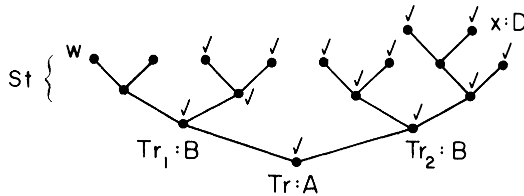


FIG 1.

Either the proof tree is in the desired form, or its root node is among those marked. Suppose the latter, and let y_1 and y_2 denote the immediate predecessors of root Tr. Suppose $\text{cl}(y_1) = B_1 = B'_1 \dot{\cup} \{\neg p\}$ and $\text{cl}(y_2) = B_2 = B'_2 \dot{\cup} \{p\}$, with $|p|$ the atom resolved upon, and $A = B'_1 \cup B'_2$. Let Tr_1 and Tr_2 be the largest subtrees rooted on y_1 and y_2 respectively, and suppose w is in Tr_1. Apply the construction recursively to Tr_1, obtaining Tr'_1 which deduces $\tilde{B}_1$, where $\tilde{B}_1 \subseteq B_1$ and conditions 2 and 3 are satisfied. Note that Tr_1 contained fewer marked nodes than did Tr, since root Tr was marked and was not in Tr_1. Thus the recursion is guaranteed to terminate.

It remains to show how to append Tr'_1 to Tr_2 in such a way that the resulting tree will either satisfy conditions 1–3 or will allow another application of the

recursive construction. Since $\tilde{B}_1 \subseteq B_1$, it is either true that $\tilde{B}_1 \subseteq B'_1$, in which case Tr'_1 satisfies conditions 1–3, or else $\tilde{B}_1 = \tilde{B}'_1 \dot{\cup} \{\neg p\}$, and we must do more work.

There will be at least one node x, a leaf of Tr_2, whose clause contains an occurrence of p which is an ancestor of p in $\mathrm{cl}(y_2)$ as indicated in Fig. 2. We shall

FIG 2.

append a copy of Tr'_1 to Tr_2 at the leaf x, in order to put the top node w of Tr'_1 at the top of the resulting deduction. Note that all nodes of Tr_2 are marked.

Let $D = D' \dot{\cup} \{p\} = \mathrm{cl}(x)$ and note that $D \in S$. Pare away from the clauses of Tr_2 the occurrence of p in D and all occurrences of p in other clauses of Tr_2 when these occurrences have p in D as their unique leaf-ancestor. No change in the structure of Tr_2 occurs, since the occurrences of p which disappear are never resolved upon in Tr_2. Call the result Tr'_2, deducing $\tilde{B}_2$ from $S \cup \{D'\}$, with $\tilde{B}_2 \subseteq B_2$.

Next, let us add onto $\mathrm{cl}(x)$ the clause $\tilde{B}'_1$ as obtained from Tr'_1. The literals of B'_1 are carried on down the proof tree, allowing elimination by resolution of any literals from $\tilde{B}'_1$ which happen to merge with a literal of Tr'_2 which is resolved upon. The resulting tree we call Tr''_2, and it deduces a clause C, where $C \subseteq \tilde{B}'_1 \cup \tilde{B}_2$, from a basis set $S \cup \{\tilde{B}'_1 \cup D'\}$.

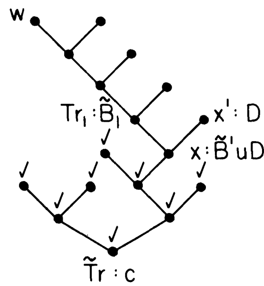


FIG 3.

Now let us modify Tr'_1 slightly at its root, by adding a new leaf x' with $\mathrm{cl}(x') = D$, to be resolved upon the atom $|p|$ with root Tr'_1, producing a new root node z. We call the extended tree $\widetilde{\mathrm{Tr}}_1$; it deduces $\tilde{B}'_1 \cup D'$ from S, and satisfies conditions 2 and 3. Replace the leaf node x of Tr''_2 by the entire subtree $\widetilde{\mathrm{Tr}}_1$, and we have $\widetilde{\mathrm{Tr}}$ (see Fig. 3), a new resolution tree deducing C from S. The added subtree with root x clearly satisfies conditions 2 and 3 so that no new marked nodes have been introduced. Thus $\widetilde{\mathrm{Tr}}$ contains fewer marked nodes than Tr. We may apply the recursive procedure to $\widetilde{\mathrm{Tr}}$ to construct $\widetilde{\mathrm{Tr}}_1$ which deduces $\tilde{C}$ and satisfies 2 and 3.

It may seem that we should now be nearly done, but this may not be the case. For, since $\tilde{C} \subseteq C \subseteq \tilde{B}'_1 \cup \tilde{B}_2 \subseteq B'_1 \cup B_2$, it may be the case that $\tilde{C}$ still contains

the literal p. (If not, then $C \subseteq A$ and we can skip the next step.) To eliminate this occurrence of p from cl(root $\widetilde{\mathrm{Tr}}_1$), we will extend $\widetilde{\mathrm{Tr}}_1$ in such a way that condition 3 remains satisfied. Recall that Tr'_1 was a tree satisfying conditions 2 and 3 and which deduces $\widetilde{B}_1$. ($\widetilde{B}_1$ contains $\neg p$, or else the construction terminated with $\mathrm{Tr}' = \mathrm{Tr}'_1$.) From this tree, prune away all proper ancestors of any node whose clause is a merge in Tr'_1. We obtain a deduction tree which deduces $\widetilde{B}_1$, from $S \cup \{M(\mathrm{Tr}'_1)\}$, and the deduction contains no merges. Call this tree Tr_3.

Since Tr_3 contains no merges, it has a unique leaf node u, with clause E, containing an instance of $\neg p$ which is an ancestor of the occurrence of $\neg p$ in cl(root Tr_3). It is not difficult to show that a merge-free vine form tree can be reordered to form a new vine-form tree having a prescribed node at the top, and that the clause deduced is a subset of the original clause. Thus we obtain Tr'_3 (as

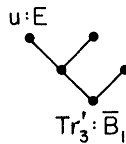


FIG 4.

shown in Fig. 4), a vine form tree having E at the top, and deducing $\bar{B}_1$ from $S \cup M(\mathrm{Tr}'_1)$ where $\bar{B}_1 \subseteq \widetilde{B}_1$.

Using the same procedure we used to join Tr'_1 to a leaf of Tr'_2, we now replace the leaf u of Tr'_3 by $\widetilde{\mathrm{Tr}}_1$, resolving upon $|p|$ at the base of $\widetilde{\mathrm{Tr}}_1$, using the clause E. We get $\widetilde{\mathrm{Tr}}'$ (shown in Fig. 5), which deduces C' from $S \cup M(\widetilde{\mathrm{Tr}}')$ and $C' \subseteq A$.

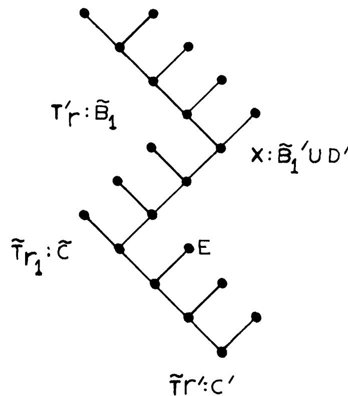


FIG 5.

$\widetilde{\mathrm{Tr}}'$ satisfies the conditions. For general proof trees we must apply Lemma 1, the above construction and finally the lifting lemma as outlined in § 3. This completes the description of the construction procedure.

As a direct consequence of the construction, we can give the following result.

LEMMA 2. *If there is a ground resolution tree* Tr *deducing A from S, and for every nonleaf node of* Tr, *at least one of the immediate predecessors contains a unit clause, then there is a ground resolution tree* Tr′, *deducing A' from S, and satisfying the conditions* 1–3, *and* Tr′ *is a vine.*

Proof. Since Tr describes a unit proof, no clause in Tr contains a merge. Thus the leaf-ancestor of each literal in Tr is unique, and it is always the case, referring to the details of the construction that $\tilde{C} \subseteq \tilde{A}$, and that the construction terminates with $\mathrm{Tr}' = \widetilde{\mathrm{Tr}}'$ and without the addition of the "tail" Tr'_3. Since this last step is the only point at which any clause from $M(\mathrm{Tr}')$, the set of merges, can be resolved with a clause on the stem of Tr′, the fact that this step is unnecessary means that the construction produces Tr′ as a vine.

5. Bounds on the complexity increase of proof trees using refinement strategies. We now give some results on the increase in complexity which may be required when the AFF or the resolution-with-merging refinements are employed.

THEOREM 1. *Let* Tr *be a resolution tree of level n deducing a clause A from a basis set S; let B be a distinguished clause of S. Then there is an* AFF *resolution tree* Tr′ *deducing A' from S, where A' subsumes A, and*:

(i) Tr′ *has top node w; either $\mathrm{cl}(w) = B$ or else $B \notin \mathrm{Basis}(\mathrm{Tr}')$*;

(ii) $\ell(\mathrm{Tr}') \leqq f_1(n)$, *where f_1 is the function defined recursively by*

$$f_1(0) = 0,$$

$$f_1(1) = 1,$$

$$f_1(n+1) = 2(f_1(n) + 1), \qquad n \geqq 1.$$

Furthermore, for each n, there is a set of clauses S and a clause A such that $A \in R^n(S)$, but no AFF *deduction of A', $A' \subseteq A$, is possible with level less than $f_1(n)$.*

Proof. To show that $f_1(n)$ is a bound we use induction on n. The basis step for $n = 0$ or $n = 1$ is obvious. Assume the bound to hold for deductions of clauses in $R^n(S)$ and suppose $A \in R^{n+1}(S)$. Then root Tr has two immediate predecessors, y_1 and y_2, and $\mathrm{cl}(y_1), \mathrm{cl}(y_2) \in R^n(S)$.

Let $B_1 = B'_1 \dot{\cup} \{p\} = \mathrm{cl}(y_1)$ and $B_2 = B'_2 \dot{\cup} \{\neg p\} = \mathrm{cl}(y_2)$. There is an AFF tree Tr'_1 deducing $\tilde{B}_1$ where $\tilde{B}_1 \subseteq B_1$, and having the prescribed top-node clause (if not, interchange y_1 with y_2 and try again). The level of Tr'_1, is bounded by $f_1(n)$, by hypothesis. If $p \notin \tilde{B}_1$, then $\tilde{B}_1$ subsumes A, and the theorem is satisfied. Assume this is not the case. Then there is an AFF deduction of $\tilde{B}_2$, where $\tilde{B}_2 \subseteq B_2$, by a tree Tr'_2 in which the clause of the top-node of Tr'_2 contains $\neg p$. By the construction of § 4 we append an extra node to Stem (Tr'_1) and an extra leaf whose clause is the same as the top node clause of Tr'_2. Call the newly formed tree Tr''_1 and note that $\ell(\mathrm{Tr}''_1) \leqq f_1(n) + 1$.

Now we delete from the clauses of Tr'_2 all literals $\neg p$ whose sole leaf-ancestor is $\neg p$ in the top node, and append the clause $\tilde{B}_1 - \{p\}$ to the top node of Tr'_2 forming Tr''_2. These operations do not increase the level. Next, we replace the top leaf of Tr''_2 by the entire tree Tr''_1, obtaining Tr″. It is seen that $\ell(\mathrm{Tr}'') \leqq 2f_1(n) + 1$.

Finally, if $p \in \mathrm{cl}(\text{root Tr}'')$, then we must resolve root Tr″ with the root of a copy or Tr''_1 to eliminate p. This adds a new node to the stem of Tr″, increasing

its level by one. The resulting tree is Tr′, and $l(\mathrm{Tr}') \leqq 2(f_1(n) + 1)$, and the bound is proved.

To prove that the bound is the best possible, consider the symmetric set of 2^n tautology-free clauses of n literals each, which is formed by taking all non-tautologous combinations of positive and negative instances of n atoms. This set is inconsistent, and has a deduction of Nil in level n by unrestricted resolution. A deduction of Nil by the AFF refinement requires a level of $f_1(n)$. This completes the proof.

We note from the example which assures us that the bound is the best possible, that a better bound cannot be found if we relax the condition of the theorem which allows us to specify the distinguished clause B as the clause of the top node. The same is not true of the level bound when resolution-with-merging is combined with AFF, however. Then, if the choice of top node is unrestricted, the bound is the same as is given by Theorem 1. However, if the choice of top node is restricted to a distinguished node (or set of support), then the best bound we can prove is defined by the function f_2:

$$f_2(0) = 0,$$

$$f_2(1) = 1,$$

$$f_2(n + 1) = 3f_2(n) + 2.$$

It is not known that $f_2(n)$ is the best possible level bound for resolution-with-merging in AFF with set-of-support, but it is known that $f_1(n)$ is not a bound for this composition of refinements. As an example, consider the set of ground clauses

$$Q, R$$

$$\neg R$$

$$\neg Q, P$$

$$\neg Q, \neg P$$

from which there is a level 2 deduction of Nil. With unrestricted choice of the top node, there is a level 3 deduction of Nil satisfying the conditions 1–3 of § 4, but if the clause of the top node is required to be $\neg R$, then the shortest such proof is of level 5, which is $f_2(2)$.

We now turn our attention to obtaining a bound on the increase in depth of functional nesting, which in practice, is more crucial than the increase in level associated with a refinement.

THEOREM 2. *If A is deduced from S by a general resolution tree* Tr, *and if no literal of the clauses appearing on* Tr *has functional nesting to a depth greater than d, d' is the maximum depth of function nesting in any term of a unifier in* Tr, *and $\ell(\mathrm{Tr}) = n$, then there is a tree* Tr′ *in* AFF (*or* AFF *with resolution with merging and set of support*) *deducing A' from S, $A' \subseteq A$, and no literal occurring in any clause appearing on* Tr′ *has depth of functional nesting greater than $nd' + d$.*

Proof. The proof follows directly from Corollary 1.2, the construction of § 4 and the lifting lemma.

Furthermore, it appears that a bound of nd' is tight. An example for $d = 1$, $d' = 1, n = 2$ is given by the following inconsistent set of clauses:

(A1) $P(fx), Q(x,fy)$.
(A2) $\neg P(fx), Q(x,fy)$,
(A3) $P(fx), Q(fy, x)$,
(A4) $\neg P(fx), Q(fy, x)$,

where f is a unary function symbol. There is obviously a level 2 deduction of Nil, and the depth of functional nesting is 1; i.e., no deeper nesting occurs than already exists in (A1)–(A4). But an AFF deduction of Nil from the same set is the following:

(1)	$P(fx), Q(x,fy)$	(A1)
(2)	$\neg P(fx), Q(x,fy)$	(A2)
(3)	$Q(x,fy)$	$R(1,2)$
(4)	$P(fx), \neg Q(fy, x)$	(A3)
(5)	$P(ffy')$	$R(3,4)$
(6)	$\neg P(fx), \neg Q(fy, x)$	(A4)
(7)	$\neg Q(fy, fy')$	$R(5,6)$
(8)	Nil	$R(3,7)$

In clause (5), the term ffy' appears, which has depth of functional nesting equal to 2. An example for $d = 1, n = 3$ is easily obtained, forming (A1′)–(A4′) from (A1)–(A4) by appending the new literal $R(x, y, fz)$, and forming (A5′)–(A8′) by appending $\neg R(fz, y, x)$ to (A1)–(A4) above. An example for any n can be formed inductively from our example for $n = 2$. If S_n is an example for n, in which the variables are $x_1, \cdots, x_n$, then form S_{n+1} as follows. Let $P(x_1, \cdots, x_n, fx_{n+1})$ be a new literal, and from it form another literal consisting of P followed by an odd permutation of the terms $(x_1, \cdots, x_n, fx_{n+1})$. Let one of the new literals be negated and the other be positive. Let S_{n+1} be the set of all clauses formed by appending one of the new literals to a clause of S_n. Obviously, S_{n+1} is inconsistent; however, we conjecture that there is no proof of the inconsistency in AFF within a nesting bound of n.

6. Examples of proofs. The increase in complexity of the AFF with merging proof trees discussed in § 5 leaves the practical value of these refinements open to doubt. The situation would seem to be a simple "trade-off" between generating fewer resolvents at any level and having to search deeper levels of the set of all resolvents for a proof.

An on-line interactive theorem-proving program has been constructed[3] which makes available to the user all of the refinements discussed above (set-of-support, model-relative deduction, ancestry-filter form, resolution-with-merging) and some others as well. (For a full description of the program see Allen and Luckham (1970).) When the program is started, the user can choose which refinements and editing strategies he wishes to apply to a particular problem, and he may alter his choice at any time during the search for a proof. This facility allows the user to take advantage of his own experience with the class of problems he is working on,

[3] At the Stanford University Artificial Intelligence Project.

Note added in proof. The experiments described here were made in 1969; the current 1973 program is twenty times faster.

and it also gives us an easy way to compare the performances of various combinations of refinements.

As a result of our own experience with a variety of problems in elementary algebra and number theory we can make the following comments. The two most effective of these refinement strategies are model-relative deduction and ancestry-filter form. For almost all problems there seems to be a choice of model (and a very simple model) such that deduction relative to that model greatly reduces the number of irrelevant deductions and is often a crucial factor in obtaining a proof. The ancestry-filter refinement has yielded some striking results. It has been very simple problems with short proofs (level 5 or 6) that show unfavorable statistics under the AFF condition. Essentially, the AFF condition imposes a strong restriction on the growth of the number of deductions generated as a function of level. In many examples this seems to be linear. For more difficult problems (where the shortest proof tree has level 9 or 10, say, or the proof depends on deriving a preliminary lemma) it usually pays to search the extra levels with the AFF refinement. Even in cases where it turns out that more deductions are retained under the AFF refinement, it often happens that far fewer deductions are *generated*, so that the deeper AFF proof tree is obtained faster than the shorter trees because much less time is spent on editing computations (e.g., see Table 1, Examples 2 and 3). Also, the refinement has appeal as a natural heuristic. Many human proofs satisfy AFF; a point at which a resolvent of nonaxiom clauses is computed corresponds intuitively to using a previously proved lemma. The conjunction of model-relative deduction and AFF, although incomplete in general, is very effective. If AFF is used without model-relative deduction, it is important to use the set-of-support strategy in conjunction with it to reduce the level 1 deductions. Resolution-with-merging does not seem to be nearly as effective as the other two refinements, and we almost always use it in conjunction with at least one other strategy.

The following three examples will serve to illustrate most of the above comments. While not entirely trivial, these examples seem to us typical of the sort of problem mathematicians would prefer to leave to the machines.

A ternary Boolean algebra (TBA), defined by A. A. Grau (1947), is a system consisting of a set S and two operations under which the system is closed, one ternary, $(x\ y\ z)$, and the other unary, x', satisfying the following axioms:

A1. $(x\, y\, (u\, v\, w)) = ((x\, y\, u)\, v\, (x\, y\, w))$

A2. $(y\, x\, x) = x,$

A3. $(x\ y\ y') = x,$

A4. $(x\ x\ y) = x,$

A5. $(y'\ y\ x) = x.$

In a recent abstract in the *Notices of the American Mathematical Society*, Chinthayamma (1969) announces without proof that Grau's axioms A1–A3 are sufficient to define a TBA, and that they are also independent, and some new sets of axioms are given.

The program has been used to establish all of the dependence results announced in [4], by employing the simple refinements we have been discussing.

Examples 1 and 2 below are typical illustrations of the sort of proof involved, and how the refinements affect the performance of the program (Table 1).

TABLE 1

(See text for notation.)

Refinement strategies	Clauses		Time	Proof level
	Generated	Retained		
Example 1				
1. M ∧ Merging ∧ AFF	2036	367	490 secs.	8
2. Support ∧ Merging ∧ AFF	2372	534	661 secs.	8
3. M ∧ Merging	4106	439	922 secs.	8
4. Support ∧ Merging	4500	544	run stopped	5 (no proof)
Example 2				
5. M ∧ Merging ∧ AFF	984	226	280 secs.	10
6. M ∧ AFF	984	226	280 secs.	10
7. M ∧ Merging	2620	432	958 secs.	8
8. Support ∧ Merging ∧ AFF	5390	546	1202 secs.	10
Example 3				
9. M ∧ Merging ∧ AFF	1558	376	572 secs.	14
10. Support ∧ Merging ∧ AFF	2569	518	run stopped	4 (no proof)
11. M ∧ Merging	12493	975	run stopped	10 (no proof)

Example 1. Derive Grau's axiom A5 from axioms A1–A4.

Using the conjunction of AFF and resolution-with-merging and model-relative deduction (relative to the model of all negative literals) the program generates the proof below in about 8 minutes:

$$
\begin{aligned}
x &= (x\ y\ y') && \text{A3} \\
&= (x\ y\ (y'\ y'\ y)) && \text{A4} \\
&= ((x\ y\ y')\ y'\ (x\ y\ y)) && \text{A1} \\
&= (x\ y'\ y) && \text{A3 and A2 (lemma)} \\
&= (x\ y'\ (y'\ y\ y)) && \text{A2} \\
&= ((x\ y'\ y'\ y\ (x\ y'\ y)) && \text{A1} \\
&= (y'\ y\ x) && \text{A2 and lemma.}
\end{aligned}
$$

In [4] Chinthayamma gives the following alternative set of axioms for TBA:

B1. $(y\ x\ x) = x$,

B2. $(y'\ x\ y) = x$,

B3. $((y\ u\ v)\ (x\ v\ u)\ z) = (y\ (v\ u\ z)\ (x\ v\ u))$.

Example 2. Derive $(y\ x\ y') = x$ (Grau's Theorem 3.5 in [5]) from Axioms B1–B3.

The proof of $(y\ x\ y') = x$ from Axioms (A) given in [5] uses the theorem $(x')' = x$, which in turn is proved using the lemma $(x\ y'\ y) = x$ (see Proof of Example 1). When all of the elementary refinements are used in conjunction, the program generates a proof in under 5 minutes (a formal language print out of the proof is given in the Appendix):

$(y\ x\ z) = (y'\ (y\ x\ z)\ y)$	B2
$= (y'\ (y\ x\ z)\ (x'\ y\ x)$	B2
$= ((y'\ x\ y)\ (x'\ y\ x)\ z)$	B3
$= (x\ y\ z)$	B2 (Lemma 1)
$\therefore\quad x = (x\ y'\ y)$	Lemma 1 and B2 (Lemma 2)
$= (x\ (y\ y'\ y')\ y)$	B1
$= ((x\ y'\ y)\ y\ y')$	B3
$= (x\ y\ y')$	Lemma 2 (Grau's (A3))
$= (y\ x\ y')$	Lemma 1.

Thus the AFF proofs found by the program are very natural. Generally the user can easily pick out the "interesting looking" deductions that play the role of lemmas, when they first appear on the on-line console (see e.g. lines 21, 17 and 3 in Example 2 of the Appendix). It is possible for the user to direct the program to pay special attention to these deductions, although this was not done here.

The final example is from elementary group theory.

Example 3.[4] Let K be a system consisting of a set S and a binary operation defined on S. If K is closed and associative and has an element e such that $e^2 = e$, and every element in S has a left inverse with respect to e and at most one right inverse with respect tȯ e, then K is a group.

The problem here is to prove that e is a left identity element. When all of the refinements are used, the program generates a level 14 proof from a set of 16 axioms (see Appendix) in $9\frac{1}{2}$ minutes. If either model-relative deduction or the AFF condition is dropped, the computation time required to find a proof becomes impractical even though a level 11 proof can be found using model-relative deduction alone. The formal proof in the Appendix is easily translated into natural notation.

Table 1 gives the results of some experiments with the three examples. M denotes deduction relative to the model consisting of all negated literals; "Merging" denotes resolution with merging; "Support" denotes the use of the negation of the theorem to be proved as the set of support. The editing bounds were fixed throughout to exclude clauses of length greater than 4, and terms with a depth of nesting of function symbols greater than 2.

Appendix. Below are the formal axioms and proofs for Examples 2 and 3; both proofs were obtained using the conjunction of all three elementary refine-

[4] This problem was suggested by Dr. L. Wos.

ments (i.e. lines 5 and 9 of Table 1). The disjunction connective is omitted from the clauses.

The proofs are printed out in reverse order. Each line is followed by numbers indicating the lines from which it was derived. Also the proofs tend to look more complicated than they are because the derivation of a lemma is printed out each time the lemma is used in the proof.

Example 2. Interpret $P(X, Y, Z, W)$ as $(x, y, z) = w$ and $c(x)$ as x'.

Axioms.

1. $P(X2\ X1\ X1\ X1)$; — axiom B1
2. $P(C(C2)\ X1\ X2\ X1)$; — axiom B2
3. $\neg P(X2\ X4\ X5\ X6) \neg P(X1\ X5\ X4\ X7)$ $\neg P(X5\ X4\ X3\ X8) \neg P(X6\ X7\ X3\ X9)\ P(X2\ X8\ X7\ X9)$; — axiom B3
4. $\neg P(X2\ X4\ X5\ X6) \neg P(X1\ X5\ X4\ X7)$ $\neg P(X5\ X4\ X3\ X8) \neg P(X2\ X8\ X7\ X9)\ P(X6\ X7\ X3\ X9)$; — axiom B3
5. $\neg P(A\ B\ C(A)\ B)$; — negation of Grau's Theorem 3.5.

Proof.

NIL 1 2

1. $P(X1,X13,C(X1(,X13)$ 3 4
2. $\neg P(A,B,C(A),B)$ AXIOM 5
3. $P(X13,X1,C(X1),X13)$ 5 6
4. $P(X1,X2,X3,X11) \neg P(X2,X1,X3,X11)$ 7 8
5. $P(X13,X2,C(X2),X13) \neg P(X10,X2,C(X2),X2)$ 9 10
6. $P(C(X2),X1,X2,X1)$ AXIOM 2
7. $P(X6,X1,X3,X11) \neg P(X1,X2,X3,X11) \neg P(C(X1),X2,X1,X6)$ 11 12
8. $P(C(X2),X1,X2,X1)$ AXIOM 2
9. $P(X1,X12,X3,X1) \neg P(X2,C(X2),X3,C(X12)) \neg P(X10,X2,C(X2),X12)$ 13 14
10. $P(X2,X1,X1,X1)$ AXIOM 1
11. $P(X6,X2,X3,X1) \neg P(X5,X4,X3,X1) \neg P(X10,X5,X4,X2) \neg P(C(X2), X4,X5,X6)$ 15 16
12. $P(C(X2),X1,X2,X1)$ AXIOM 2
13. $P(X6,X2,X3,X1) \neg P(X5,X4,X3,C(X2)) \neg P(X10,X5,X4,X2) \neg P(X1, X4,X5,X6)$ 17 18
14. $P(X1,C(X2),X2,X1)$ 19 20
15. $P(X6,X7,X3,X9) \neg P(X2,X8,X7,X9) \neg P(X5,X4,X3,X8) \neg P(X1,X5, X4,X7) \neg P(X2,X4,X5,X6)$ AXIOM 4
16. $P(C(X2),X1,X2,X1)$ AXIOM 2
17. $P(X1,C(X2),X2,X1)$ 21 22
18. $P(X6,X7,X3,X9) \neg P(X2,X8,X7,X9) \neg P(X5,X4,X3,X8) \neg P(X1,X5, X4,X7) \neg P(X2,X4,X5,X6)$ AXIOM 4
19. $P(X1,X2,X3,X11) \neg P(X2,X1,X3,X11)$ 23 24
20. $P(C(X2),X1,X2,X1)$ AXIOM 2

21. $P(X1,X2,X3,X11)$ $\neg P(X2,X1,X3,X11)$ 25 26
22. $P(C(X2),X1,X2,X1)$ AXIOM 2
23. $P(X6,X1,X3,X11)$ $\neg P(X1,X2,X3,X11)$ $\neg P(C(X1),X2,X1,X6)$ 27 28
24. $P(C(X2),X1,X2,X1)$ AXIOM 2
25. $P(X6,X1,X3,X11)$ $\neg P(X1,X2,X3,X11)$ $\neg P(C(X1),X2,X1,X6)$ 29 30
26. $P(C(X2),X1,X2,X1)$ AXIOM 2
27. $P(X6,X2,X3,X1)$ $\neg P(X5,X4,X3,X1)$ $\neg P(X10,X5,X4,X2)$ $\neg P(C(X2),$ $X4,X5,X6)$ 31 32
28. $P(C(X2),X1,X2,X1)$ AXIOM 2
29. $P(X6,X2,X3,X1)$ $\neg P(X5,X4,X3,X1)$ $\neg P(X10,X5,X4,X2)$ $\neg P(C(X2),$ $X4,X5,X6)$ 33 34
30. $P(C(X2),X1,X2,X1)$ AXIOM 2
31. $P(X6,X7,X3,X9)$ $\neg P(X2,X8,X7,X9)$ $\neg P(X5,X4,X3,X8)$ $\neg P(X1,X5,$ $X4,X7)$ $\neg P(X2,X4,X5,X6)$ AXIOM 4
32. $P(C(X2),X1,X2,X1)$ AXIOM 2
33. $P(X6, X7, X3, X9)$ $\neg P(X2,X8,X7,X9)$ $\neg P(X5,X4,X3,X8)$ $\neg P(X1,X5,$ $X4,X7)$ $\neg P(X2,X4,X5,X6)$ AXIOM 4
34. $P(C(X2),X1,X2,X1)$ AXIOM 2

QED

Example 3. Interpet $P(x\ y\ z)$ as the predicate defined in terms of binary operation by $(x\ y) = Z$, $G(x)$ as the left inverse of x with respect to E, $R(x, y)$ as $x = y$, and $F(x, y)$ as the binary operation.

Axioms.

1. $P(X1\ X2\ F\ (X1\ X2))$;
2. $P(E\ E\ E)$;
3. $P(G(X1)\ X1\ E)$;
4. $\neg P(X1\ X2\ X3)$ $\neg P(X2\ X4\ X5)$ $\neg P(X1\ X5\ X6)$. $P(X3\ X4\ X6)$;
5. $\neg P(X1\ X2\ X3)$ $\neg P(X2\ X4\ X5)$ $\neg P(X3\ X4\ X6)$ $P(X1\ X5\ X6)$;
6. $\neg P(X1\ X2\ E)$ $\neg P(X1\ X3\ E)$ $R(X2\ X3)$;
7. $R(X1\ X1)$;
8 $\neg P(X1\ X2\ X3)$ $\neg P(X1\ X2\ X4)$ $R(X3\ X4)$;
9. $\neg R(X1\ X2)$ $\neg R(X2\ X3)$ $R(X1\ X3)$;
10. $\neg R(X1\ X2)$ $\neg P(X1\ X3\ X4)$ $P(X2\ X3\ X4)$;
11. $\neg R(X1\ X2)$ $\neg P(X3\ X1\ X4)$ $P(X3\ X2\ X4)$;
12. $\neg R(X1\ X2)$ $\neg P(X3\ X4\ X1)$ $P(X3\ X4\ X2)$;
13. $\neg R(X1\ X2)$ $R(F(X1\ X3)\ F(X2\ X3))$;
14. $\neg R(X1\ X2)$ $R(F(X3\ X1)\ F(X3\ X2))$;
15. $\neg R(X1\ X2)$ $R(G(X1)\ G(X2))$;
16. $\neg P(E\ A\ A)$;

Proof.

NIL 1 2

1. $P(E,X2,X2)$ 3 4
2. $\neg P(E,A,A)$ AXIOM 16
3. $P(X3,X4,X1)$ $\neg P(X3,X4,F(E,X1))$ 5 6
4. $P(X1,X2,F(X1,X2))$ AXIOM 1
5. $R(F(E,X1),X1)$ 7 8
6. $P(X3,X4,X2)$ $\neg R(X1,X2)$ $\neg P(X3,X4,X1)$ AXIOM 12
7. $R(F(E,X1),X3)$ $\neg P(G(X1),X3,E)$ 9 10
8. $P(G(X1),X1,E)$ AXIOM 3
9. $P(G(X1),F(E,X1),E)$ 11 12
10. $R(X2,X3)$ $\neg P(X1,X3,E)$ $\neg P(X1,X2,E)$ AXIOM 6
11. $P(X1,F(E,X2),X6)$ $\neg P(X1,X2,X6)$ 13 14
12. $P(G(X1),X1,E)$ AXIOM 3
13. $P(X1,E,X1)$ 15 16
14. $P(X7,F(X1,X2),X6)$ $\neg P(X3,X2,X6)$ $\neg P(X7,X1,X3)$ 17 18
15. $P(X3,X4,X1)$ $\neg P(X3,X4,F(X1,\mathrm{E}))$ 19 20
16. $P(X1,X2,F(X1,X2))$ AXIOM 1
17. $P(X1,X5,X6)$ $\neg P(X3,X4,X6)$ $\neg P(X2,X4,X5)$ $\neg P(X1,X2,X3)$ AXIOM 5
18. $P(X1,X2,F(X1,X2))$ AXIOM 1
19. $R(F(X1,E),X1)$ 21 22
20. $P(X3,X4,X2)$ $\neg R(X1,X2)$ $\neg P(X3,X4,X1)$ AXIOM 12
21. $R(F(X1,E),X3)$ $\neg P(G(X1),X3,E)$ 23 24
22. $P(G(X1),X1,E)$ AXIOM 3
23. $P(G(X1),F(X1,E),E)$ 25 26
24. $R(X2,X3)$ $\neg P(X1,X3,E)$ $\neg P(X1,X2,E)$ AXIOM 6
25. $P(X7,F(X1,E),E)$ $\neg P(X7,X1,E)$ 27 28
26. $P(G(X1),X1,E)$ AXIOM 3
27. $P(X7,F(X1,X2),X6)$ $\neg P(X3,X2,X6)$ $\neg P(X7,X1,X3)$ 29 30
28. $P(E,E,E)$ AXIOM 2
29. $P(X1,X5,X6)$ $\neg P(X3,X4,X6)$ $\neg P(X2,X4,X5)$ $\neg P(X1,X2,X3)$ AXIOM 5
30. $P(X1,X2,F(X1,X2))$ AXIOM 1

QED

REFERENCES

[1] JOHN ALLEN AND DAVID LUCKHAM (1970), *An interactive theorem-proving program*, Proc. Fifth International Machine Intelligence Workshop, Edinburgh University Press, Edinburgh.

[2] ROBERT ANDERSON AND W. W. BLEDSOE (1970), *A linear format for resolution with merging and a new technique for establishing completeness*, J. Assoc. Comput. Mach., 17, pp. 525–534.

[3] PETER B. ANDREWS (1968), *Resolution with merging*, Ibid., 15, pp. 367–381.

[4] CHINTHAYAMMA (1969), *Sets of independent axioms for a ternary Boolean algebra*, Notices Amer. Math. Soc., 16, p. 654.

[5] A. A. GRAU (1947), *Ternary Boolean algebra*, Bull. Amer. Math. Soc., 53, pp. 567–572.

[6] R. B. KIEBURTZ AND DAVID LUCKHAM (1970), *Compatible strategies for the resolution principle*, Stanford Artificial Intelligence Memo., Stanford Univ., Stanford, Calif.

[7] DONALD W. LOVELAND (1969), *A linear format for resolution*, IRIA Symposium on Automatic Demonstration (Versailles, 1968), proceeding to be published by Springer-Verlag.

[8] DAVID LUCKHAM (1969), *Refinement theorems in resolution theory*, Ibid.; also as Stanford Artificial Intelligence Memo AI-81, Computer Science Dept., Stanford University, Stanford, Calif.

[9] J. A. ROBINSON (1965a), *Automatic deduction with hyper-resolution*, Internat. J. Comput. Math., 1, pp. 227–234.

[10] ——— (1965b), *A machine-oriented logic based on the resolution principle*, J. Assoc. Comput. Mach., 12, pp. 23–41.

[11] ——— (1967), *A review of automatic theorem proving*, Proc. Symp. Applied Mathematics, vol. XIX, Amer. Math Soc., Providence.

[12] J. R. SLAGLE (1967), *Automatic theorem-proving with renamable and semantic resolution*, J. Assoc. Comput. Mach., 14, pp. 687–697.

[13] L. WOS, G. ROBINSON, AND D. CARSON (1965), *Efficiency and completeness of the set of support strategy in theorem proving*, Ibid., 12, pp. 536–541.

[14] R. A. YATES, B. RAPHAEL AND T. P. HART (1970), *Resolution graphs*, Artificial Intelligence, 1, pp. 257–290.

SIAM J. COMPUT.
Vol. 1, No. 4, December 1972

REAL-TIME STRICT DETERMINISTIC LANGUAGES*

MICHAEL A. HARRISON† AND IVAN M. HAVEL‡

Abstract. The family of strict deterministic languages has been studied for its theoretical properties and applications to parsing. In particular, these languages have been shown to be precisely the prefix-free deterministic languages. Deterministic pushdown automata are called (quasi-)real time if they have no (only a bounded number of consecutive) null moves. It is shown that for strict deterministic languages, the quasi-real-time and real-time constraints are equivalent (except for $\{\Lambda\}$). A grammatical characterization of these languages is also given. For quasi-real-time strict deterministic languages, an easy and elegant decision method is given for testing regularity. For all known methods of accepting deterministic languages, it is shown that the families of real-time languages are a proper subset of the full families. A relation is established among these sets, the simple deterministic languages, and some hierarchies.

Introduction. Characterizations of languages by automata are of particular importance for applications such as parsing. When one has an important family such as the deterministic context-free languages [8], [13], [14], [15], [17] one wishes to examine acceptance quite closely and ask if placing restrictions of one kind or another on the device affects the family of languages accepted. Such questions generally lead to interesting results because if restrictions lead to a proper subfamily we can examine the implications of the restrictions. On the other hand, if no loss of generality results from the restriction, we may have found a condition which either simplifies description or proofs. There are many possible ways[1] to restrict the action of an automaton. It is possible to restrict memory capacity, computational ability, the time of computation, or the organization of memory.

In the present work, our interest is the continuing investigation of strict deterministic languages [14], [15]. It has been argued in (and by) [14], [15] that these languages provide an interesting family and an important technique for studying the full family of deterministic context-free languages.

In [14], the effect of limiting memory size in the family of accepting automata was investigated. This led to an infinite hierarchy of strict deterministic languages. Our goal in the present paper is to examine the effect of limiting computation time. This will be done by limiting the number of consecutive Λ-moves. This restriction is an essential one in that it changes the family of acceptable languages. We shall be able to characterize the new families grammatically.

The present paper consists of this Introduction and three other sections. In § 1 some of the basic definitions concerning strict deterministic grammars are

* Received by the editors June 30, 1972, and in revised form September 22, 1972. This research was supported by the National Science Foundation under Grant GJ-474.

† Department of Computer Science, University of California, Berkeley, California 94720.

‡ Department of Computer Science, University of California, Berkeley, California. Now at ÚTIA, Vyšehradská 49, Praha 2, Czechoslovakia.

[1] Many restrictions have been studied in the literature in both the deterministic and nondeterministic cases (cf. [5], [9], [11], [12], [14], [18], [19], [20]).

recalled. We also give some constructions which were used in [14] and which are needed in our proofs.

In § 2, (quasi-)real-time computation is introduced and it is shown that for strict deterministic languages, quasi-real-time is equivalent to real-time (except for $\{\Lambda\}$). A grammatical characterization of such languages is presented. An application of these ideas is given to testing if such a set is regular. There is a very simple algorithm in this case as opposed to the general case [22]. Using these ideas, it is shown that in all types of deterministic acceptance, the family of real-time languages is a proper subset of the full family of deterministic languages.

In § 3, the simple deterministic languages of [18] are connected with strict deterministic languages through a hierarchy established in [14] and through the results derived here.

In the remainder of this section, we introduce our notational conventions. Consult [14] for any notation not defined here. Let X be a set. A *partition of* X is a collection $\pi = \{X_1, X_2, \cdots\}$ of nonempty mutually disjoint subsets $X_i \subseteq X$ such that $X = \bigcup_i X_i$. Subsets X_i are called *blocks* of partition π.

There is a natural correspondence between partitions and equivalence relations on X: if π corresponds to an equivalence $\equiv$, then $x \equiv y$ if and only if x and y are in the same block of π. In such a case we write either $\equiv (\mathrm{mod}\ \pi)$ for $\equiv$, or $X/\equiv$ for π.

Let $\pi_1 = X/\equiv_1$ and $\pi_2 = X/\equiv_2$. We define

$$\pi_1 \leqq \pi_2 \quad \text{if and only if} \quad \equiv_1 \subseteq \equiv_2.$$

The set of all partitions on X together with the operations meet and join forms a *lattice of partitions of* X.

Functions are defined as functional relations together with their domains and ranges using the notation $f: X \to Y$ or $X \xrightarrow{f} Y$. We shall often deal with *partial functions* corresponding to single-valued relations. We use the notation $f: X \to_p Y$ or $X \xrightarrow{f}_p Y$ for partial functions.

An *alphabet* is any finite nonempty sets of objects, called *letters*. We assume that all alphabets are considered as subsets of a fixed infinite set of letters, say Ω (we will not usually mention this set explicitly).

In our constructions we shall often need special auxiliary alphabets whose letters correspond to certain given objects. To make these constructions uniform we formally define for any given finite nonempty set X a special new alphabet Alph (X) by means of a fixed injection $X \to \Omega : x \mapsto \bar{x}$, i.e.,

$$\mathrm{Alph}\,(X) = \{\bar{x} | x \in X\}.$$

By convention we always assume that any alphabet of this form is disjoint from other alphabets in the particular context if they were introduced independently. When X is itself an alphabet, the Alph operator produces a new copy of X.

Let $u, v \in A^*$ be two strings. Then u is a *prefix* of v if and only if $v = uw$ for some $w \in A^*$; when $w \neq \Lambda$, u is a *proper prefix* of v. We denote by lg (w) the *length* of w, i.e., the total number of occurrences of letters in w, in particular, $\lg(\Lambda) = 0$. For any $n \geqq 0$,

$$^{(n)}w \text{ is the prefix of } w \text{ with length } \min(\lg(w), n).$$

A language $L \subseteq A^*$ is said to be *prefix-free* if and only if

$$w \in L \quad \text{and} \quad wu \in L \quad \text{implies } u = \Lambda.$$

Next we review some basic concepts concerning formal grammars.

DEFINITION. A *context-free grammar* (hereafter a *grammar*) G is a 4-tuple

$$G = \langle V, \Sigma, P, S \rangle, \tag{1}$$

where V and Σ are two alphabets, $\Sigma \subseteq V$ (letters in Σ and in $N = V - \Sigma$ are called *terminals* and *nonterminals* respectively), $S \in N$ and P is a finite relation, $P \subseteq N \times V^*$ (the set of *productions*).

By convention[2] we write "$A \to \alpha$ is in P," or sometimes only "$A \to \alpha$," instead of "$(A, \alpha) \in P$." We also write "$A \to \alpha_1|\alpha_2|\cdots|\alpha_n$" instead of "$A \to \alpha_1$ and $A \to \alpha_2$ and $\cdots$ and $A \to \alpha_n$" (here "|" is a metasymbol not in V). Where the reference to G is important we write $\to_G$ instead of $\to$.

DEFINITION. Let G be a grammar of the form (1). We define a relation $\Rightarrow \subseteq V^* \times V^*$ as follows. For any $\alpha, \beta \in V^*, \alpha \Rightarrow \beta$ if and only if $\alpha = \alpha_1 A \alpha_2$, $\beta = \alpha_1 \beta_1 \alpha_2$ and $A \to \beta_1$ is in P for some $A \in N$ and $\alpha_1, \alpha_2, \beta_1 \in V^*$. In particular, if $\alpha_1 \in \Sigma^*$ or $\alpha_2 \in \Sigma^*$, we write $\alpha \Rightarrow_L \beta$ or $\alpha \Rightarrow_R \beta$ respectively. Any $\alpha \in V^*$ is called a (*canonical*) *sentential form* if and only if $S \Rightarrow^* \alpha$ ($S \Rightarrow_R^* \alpha$).

The *language generated by* G is the language

$$L(G) = \{w \in \Sigma^* | S \Rightarrow^* w\}. \tag{2}$$

Two grammars are called *equivalent* if and only if they generate the same language.

Note that in (2) relation $\Rightarrow^*$ can be replaced by $\Rightarrow_R^*$ or by $\Rightarrow_L^*$.

1. Background of strict deterministic languages. In [14], [15] the family of strict deterministic languages was introduced and was used to obtain a number of important results. Our current results will require [14] and [15] as prerequisites. To aid the reader of this paper, we restate some of the basic definitions. In some cases, we reproduce important constructions which were previously used since some of the current proofs require them. The present discussion is abbreviated and [14] and [15] contain considerably more detail.

We begin with the key definition.

DEFINITION 1.1. Let $G = \langle V, \Sigma, P, S \rangle$ be a grammar and let π be a partition of the set V of terminal and nonterminal letters of G. Such a partition π is called *strict* if and only if

(a) $\Sigma \in \pi$, and
(b) for any $A, A' \in N$ and $\alpha, \beta, \beta' \in V^*$, if $A \to \alpha\beta$, $A' \to \alpha\beta'$ and $A \equiv A' \pmod{\pi}$, then either
 (i) both $\beta, \beta' \neq \Lambda$ and[3] ${}^{(1)}\beta \equiv {}^{(1)}\beta' \pmod{\pi}$ or
 (ii) $\beta = \beta' = \Lambda$ and $A = A'$.

---

[2] We adopt certain preferences in usage of symbols. When talking about grammars we use symbols $A, B, C, \cdots$ for elements of V or N; $a, b, c, \cdots$ for elements of Σ or $\Sigma_\Lambda = \Sigma \cup \{\Lambda\}$; $\alpha, \beta, \gamma, \cdots$ for elements of V^*; and $u, v, w, \cdots$ for elements of Σ^*.

Equation numbers are used when reference will be made to that line.

[3] Recall that ${}^{(1)}\beta$ is the first symbol of β (cf. the Introduction).

In most cases, the partition π will be clear from the context and we shall write simply $A \equiv B$ instead of $A \equiv B \pmod{\pi}$, and $[A]$ instead of $[A]_\pi = \{A' \in V | A' \equiv A \pmod{\pi}\}$.

DEFINITION 1.2. Any grammar $G = \langle V, \Sigma, P, S \rangle$ is called *strict deterministic* if and only if there exists a strict partition π of V. A language L is called a *strict deterministic language* if and only if $L = L(G)$ for some strict deterministic grammar G.

Now we give an example which will also be utilized later.

Example 1.1. Let G_1 be a grammar with the productions

$$S \to aA | aB$$

$$A \to aAa | bC$$

$$B \to aB | bD$$

$$C \to bC | a$$

$$D \to bDc | c$$

The blocks of a strict partition are Σ, $\{S\}$, $\{A, B\}$, $\{C, D\}$. The language is $L(G_1) = \{a^n b^k a^n, a^k b^n c^n | k, n \geqq 1\}$.

A given strict deterministic grammar has a set of strict partitions associated with it. Using the meet operation (cf. [14]) we see that for any strict deterministic grammar there exists a unique *minimal strict partition* π_0. In general, there is no dual concept of a maximal strict partition.

DEFINITION 1.3. For any strict partition $\pi = \{V_i\}$ in a given grammar, define

$$\|\pi\| = \max_{V_i \in \pi - \{\Sigma\}} |V_i|.$$

Note that if $\pi_1 \leqq \pi_2$ in the standard lattice ordering of partitions (cf. the Introduction), then $\|\pi_1\| \leqq \|\pi_2\|$. Thus if G is strict deterministic and if π_0 is the unique minimal strict partition on G, then $\|\pi_0\| \leqq \|\pi\|$ for any other strict partition of G.

DEFINITION 1.4. Let G be a strict deterministic grammar. We define the *degree of G* as the number

$$\deg(G) = \|\pi_0\|,$$

where π_0 is the minimal strict partition for G. For any strict deterministic language L, define its degree as follows:

$$\deg(L) = \min\{\deg(G) | G \text{ is strict deterministic and } L(G) = L\}.$$

For later applications we now present a convenient concept used in testing if a grammar is strict deterministic.

DEFINITION 1.5. Let $\alpha, \beta \in V^*$ and let A, B be two letters in V such that $A \neq B$ and we have $\alpha = \gamma A \alpha_1$ and $\beta = \gamma B \beta_1$ for some $\gamma, \alpha_1, \beta_1 \in V^*$. Then the pair (A, B) is called the *distinguishing pair of* α and β. A distinguishing pair (A, B) is said to be *terminal* if and only if $A, B \in \Sigma$.

From Definition 1.5 we can make the following observation.

FACT. *Any two strings* $\alpha, \beta \in V^*$ *have a distinguishing pair if and only if* α *is not a prefix of* β *and* β *is not a prefix if* α. *If they have a distinguishing pair, then it is unique.*

Next we recall some basic definitions concerning deterministic pushdown automata.

DEFINITION 1.6. A *deterministic pushdown automaton* (abbreviated DPDA) is a 7-tuple

$$M = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F \rangle, \tag{1.1}$$

where Q is a finite nonempty set, Σ and Γ are two alphabets, $q_0 \in Q$, $Z_0 \in \Gamma$, $F \subseteq Q$ and δ is a partial function[4]

$$\delta : Q \times \Sigma_\Lambda \times \Gamma \to_p Q \times \Gamma^* \tag{1.2}$$

with the property that for any $q \in Q$ and $Z \in \Gamma$,

$$\delta(q, \Lambda, Z) \neq \emptyset \quad \text{implies} \quad \delta(q, a, Z) = \emptyset \quad \text{for all } a \in \Sigma. \tag{1.3}$$

Certain strings over Γ are interpreted as contents of the *pushdown store*; in this interpretation we assume that the bottom of the store is on the left and the top on the right. If some $q, q' \in Q$, $a \in \Sigma_\Lambda$, $Z \in \Gamma$ and $\gamma \in \Gamma^*$ satisfy

$$\delta(q, a, Z) = (q', \gamma), \tag{1.4}$$

then formula (1.4) is called a *move* (of DPDA M); in particular, if $a = \Lambda$, (1.4) is called a Λ-*move*.

DEFINITION 1.7. Let M be a DPDA of the form (1.1) and let $\mathcal{Q} = Q \times \Sigma^* \times \Gamma^*$. The *yield relation of* M, $\vdash_M \subseteq \mathcal{Q} \times \mathcal{Q}$ (or $\vdash$ when M is understood), is defined as follows. For any $q, q' \in Q$, $a \in \Sigma_\Lambda$, $w \in \Sigma^*$, $\alpha, \beta \in \Gamma^*$ and $Z \in \Gamma$,

$$(q, aw, \alpha Z) \vdash (q', w, \alpha\beta) \quad \text{if and only if} \quad \delta(q, a, Z) = (q', \beta). \tag{1.5}$$

Set $\mathcal{Q}$ in this definition is the *configuration space* (of M) and its elements are *configurations*. The configuration (q_0, w, Z_0) for some $w \in \Sigma^*$ is called the *initial configuration* (*for* w). An instance of the yield relation on the left-hand side of (1.5) is called a *transition* (from the first configuration to the second) *corresponding to the move* on the right-hand side of (1.5). Any sequence of configurations $c_0, \cdots, c_n, \cdots \in \mathcal{Q}$ (possibly infinite) such that $c_0 \vdash \cdots \vdash c_n \vdash \cdots$ and where c_0 is an initial configuration, is called a *computation* (of M).

We now endow a DPDA with an ability to define, or *accept*, certain languages over its input alphabet.

DEFINITION 1.8. Let M be a DPDA of the form (1.1). For a given $K \subseteq \Gamma^*$ define the language $T(M, K) \subseteq \Sigma^*$ as follows:

$$T(M, K) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash^* (q, \Lambda, \alpha) \text{ for some } q \in F \text{ and } \alpha \in K\}. \tag{1.6}$$

In particular, let

$$T_0(M) = T(M, \Gamma^*),$$
$$T_1(M) = T(M, \Gamma),$$
$$T_2(M) = T(M, \{\Lambda\}).$$

The customary notation for $T_0(M)$ is $T(M)$ and for $T_2(M)$ in the case when $F = Q$,

[4] See the Introduction for our conventions about partial functions. Also recall that $\Sigma_\Lambda = \Sigma \cup \{\Lambda\}$.

is Null (M) or $N(M)$ (cf. [17][5]). Acceptance by $T_1(M)$ is equivalent to the acceptance by reinitializing the pushdown store (cf., e.g., [11]) or by empty store if moves on empty store are allowed (cf., e.g., [12]). It is immediate from (1.6) that for $i = 1, 2$, $T_i(M) \subseteq T_0(M)$. In the nondeterministic case these three types of acceptance lead to the same family of languages (the context-free languages). This is not, however, the case for DPDA.

DEFINITION 1.9. We define the following three families of languages for $i = 0, 1, 2$:

$$\Delta_i = \{T_i(M) | M \text{ is a DPDA}\}.$$

The languages in Δ_2 were previously studied in [14], [15]. They are identical to the family of languages generated by strict deterministic grammars [14, Thm. 3.5]. Note that Δ_2 does not even contain all regular events but only the prefix-free ones.

In the remainder of this paper, we shall need to go back and forth between grammars and automata. First we recall the construction [14] which carries a DPDA with one final state into a grammar.

DEFINITION 1.10. Let $\hat{M}$ be a DPDA of the form

$$\hat{M} = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, \{q_f\}\rangle \tag{1.7}$$

(i.e., $\hat{M}$ has a single final state). We define the *canonical grammar* $G_{\hat{M}}$ *of* $\hat{M}$ as follows. First define

$$G'_{\hat{M}} = \langle V, \Sigma, P, S\rangle, \tag{1.8}$$

where $V = \text{Alph}(Q \times \Gamma \times Q) \cup \Sigma$, $S = \overline{(q_0, Z_0, q_f)}$ and P is defined as follows.[6] For any $a \in \Sigma_\Lambda$, $Z, Z_1, \cdots, Z_k \in \Gamma$, $q, p, q_1, \cdots, q_k \in Q$ and $k \geqq 1$,

$$\overline{qZq_k} \to a\,\overline{pZ_1q_1}\,\overline{q_1Z_2q_2} \cdots \overline{q_{k-1}Z_kq_k} \text{ is in } P \quad \text{if and only if} \quad \delta(q, a, Z) = (p, Z_k \cdots Z_2Z_1); \tag{1.9}$$

and

$$\overline{qZp} \to a \text{ is in } P \quad \text{if and only if} \quad \delta(q, a, Z) = (p, \Lambda). \tag{1.10}$$

(No other productions are in P.) Assume that $G_{\hat{M}}$ is the reduced form of $G'_{\hat{M}}$. The canonical grammar is the principal construction which leads from pushdown automata to grammars. The same construction is used in standard proofs of equivalence of PDA and context-free grammars (cf., e.g., [17]).

When one has a DPDA with a single final state and $G_{\hat{M}}$ is defined as in Definition 1.10, then we define an equivalence relation $\equiv$ on V such that

$$A \equiv B \quad \text{if and only if} \quad A, B \in \Sigma \quad \text{or} \quad A = \overline{(q, Z, q')}, \quad B = \overline{(q, Z, q'')} \quad \text{for some } q, q', q'' \in Q, \quad Z \in \Gamma. \tag{1.11}$$

Then if we take $\pi = V/\equiv$, it is shown in [14] that $G_{\hat{M}}$ is strict deterministic under π. Moreover, it is not hard to see that if Algorithm 1 of [14] is applied to $G_{\hat{M}}$ that

[5] It is not hard to show that for any DPDA M there exists a DPDA M' such that $T_2(M) = \text{Null}(M')$.

[6] We use the simplified notation $\overline{qZq'}$ instead of $\overline{(q, Z, q')}$ for elements of $\text{Alph}(Q \times \Gamma \times Q)$.

the minimal strict partition will be computed. Moreover, this partition is precisely π (cf. [14]). Thus we may state the following fact.

PROPOSITION 1.1. *The minimal partition on* $G_{\hat{M}}$ *is precisely* π *of equation* (1.11).

We shall also need to construct a canonical DPDA from a grammar as was done in [14]. For convenience that construction is also reproduced here.

Let $G = \langle V, \Sigma, P, S\rangle$ be a grammar with the strict partition

$$\pi = \{\Sigma, V_0, V_1, \cdots, V_m\}, \tag{1.12}$$

where $m \geqq 0$ and $V_0 = [S]$. We use special indexed symbols A_{ij} for the nonterminals of G so that for all $i, 0 \leqq i \leqq m$, we have

$$V_i = \{A_{i0}, A_{i1}, \cdots, A_{in_i}\}, \tag{1.13}$$

where $n_i = |V_i|$. (Note that $\max_i n_i = \|\pi\|$.) Moreover, let $A_{00} = S$.

DEFINITION 1.11. Let $G = \langle V, \Sigma, P, S\rangle$ be a grammar with strict partition π for which we use the notation from (1.12) and (1.13). We define the *canonical* DPDA M_G *for* G as follows:

$$M_G = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, \{q_0\}\rangle, \tag{1.14}$$

where

$$Q = \{q_j | 0 \leqq j < \|\pi\|\},$$
$$\Gamma = \Gamma_1 \cup \Gamma_2,$$
$$\Gamma_1 = \{\overline{V_i, \alpha} | A \to \alpha\beta \text{ for some } A \in V_i \text{ and } \alpha, \beta \in V^*\},$$
$$\Gamma_2 = \{\overline{V_i, \alpha, V_j} | A \to \alpha B\beta \text{ for some } A \in V_i,\ B \in V_j \text{ and } \alpha, \beta \in V^*\},$$
$$Z_0 = \overline{[S], \Lambda} = \overline{V_0, \Lambda} \in \Gamma_1,$$

and δ is defined by means of four types of moves as follows: For any V_i, $V_k \in \pi - \{\Sigma\}$, $\alpha \in V^*$, $a \in \Sigma$ and $q_j \in Q$, we have:

Type 1. $\delta(q_0, \Lambda, \overline{V_i, \alpha}) = (q_0, \overline{V_i, \alpha, V_k}\overline{V_k, \Lambda})$ if $A \to \alpha B\beta$ is in P for some $A \in V_i$, $B \in V_k$ and $\beta \in V^*$.

Type 2. $\delta(q_0, a, \overline{V_i, \alpha}) = (q_0, \overline{V_i, \alpha a})$ if $A \to \alpha a\beta$ is in P for some $A \in V_i$ and $\beta \in V^*$.

Type 3. $\delta(q_0, \Lambda, \overline{V_i, \alpha}) = (q_j, \Lambda)$ if $A_{ij} \to \alpha$ is in P.

Type 4. $\delta(q_j, \Lambda, \overline{V_k, \alpha, V_i}) = (q_0, \overline{V_k, \alpha A_{ij}})$.

Otherwise δ is not defined. Moves of Types 1, 2 and 3 are called *detection moves*; the move of Type 4 is called the *reduction move.*

2. Real-time strict deterministic languages. Null rules allow a device to "pause" in its processing of an input and do a certain amount of computation without examining the input. It is of interest to know whether or not this ability affects the computational power of a device. Similar questions have been studied in [5], [12], [13]. We shall comment further on related work after giving some formal definitions.

DEFINITION 2.1. Let M be a DPDA of the form $M = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F\rangle$. M is said to be *quasi-real-time* if and only if there is an integer $t \geqq 0$ (called a

time constant of M) such that for any $q, q' \in Q$ and $\gamma, \gamma' \in \Gamma^*$,

$$(2.1) \qquad (q, \Lambda, \gamma) \vdash^k (q', \Lambda, \gamma') \quad \text{implies} \quad k \leqq t.$$

In particular, M is said to be *real-time* if and only if it has a zero time constant, i.e., if and only if $\delta(q, \Lambda, Z) = \emptyset$ for all $q \in Q$ and $Z \in \Gamma$.

A language L is called Δ_i-*(quasi-)real-time* if and only if $L = T_i(M)$ for some (quasi-)real-time DPDA $M, i = 0, 1, 2$.

The term "quasi-real-time" was used in the nondeterministic case in [2], [12]. The concept of real-time computation is widely used; in connection with nondeterministic PDA it was used in [13] where it was shown that real-time PDA are as powerful as general nondeterministic PDA. The result also follows easily from the use of Greibach normal form [10]. In the deterministic case, the nonequivalence of real-time (or quasi-real-time) DPDA and arbitrary DPDA is shown in [20].

The term "real-time (context-free) language" as a synonym to our term "Δ_1-real-time language"[7] appears in [12].

It turns out that the family of Δ_i-quasi-real-time languages coincides with the family of Δ_i-real-time for $i = 0, 1$ and, with the exception of $\{\Lambda\}$, for $i = 2$. Only the last case is discussed now, but the other cases will be treated in Theorem 2.4.

THEOREM 2.1. *A language L is Δ_2-quasi-real-time if and only if it is Δ_2-real-time or $L = \{\Lambda\}$.*

Proof. The "if" direction is a direct consequence of the definition and the fact that $\{\Lambda\} = T_2(M)$ for any DPDA M with a single move $\delta(q_0, \Lambda, Z_0) = (q_f, \Lambda)$ for some $q_f \in F$. This DPDA is quasi-real-time ($t = 1$).

For the "only if" direction, let $M = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F\rangle$ be a DPDA and assume M is quasi-real-time with time constant t. One can define an equivalent real-time DPDA M' in such a way that (i) the pushdown letters of M' are codes for n-tuples ($n = 2t + 1$) of pushdown letters of M and (ii) one move of M' has the same effect as a sequence of at most n moves of M. M' must simulate the writing and erasing moves of M in its finite state control and must transfer information to and from its pushdown storage in blocks of size n.

We shall omit the formal construction and proof, merely noting the required technique is similar to that used in [16]. A similar result is stated without proof in [2].

We now give a grammatical characterization of Δ_2-real-time languages.

DEFINITION 2.2. Let G be a strict deterministic grammar of the form

$$(2.2) \qquad G = \langle V, \Sigma, P, S\rangle$$

with minimal strict partition π. G is called a *real-time strict deterministic* (or simply *real-time*) *grammar* if and only if it is Λ-free and the following condition is satisfied for all $A, A', B, B' \in N$ and $\alpha, \beta \in V^*$.

$$(2.3) \qquad \text{If } A \to \alpha B \text{ and } A' \to \alpha B'\beta, \text{ then } A \equiv A' \pmod{\pi} \quad \text{implies } \beta = \Lambda.$$

Note that if G is a real-time strict deterministic grammar, then for any

[7] Δ_1-real-time languages form an abstract family of deterministic languages in the sense of [3].

$A, A' \in N$, if $A \to \alpha B$ is in P, $A' \to \alpha B'\beta$ is in P, and $A \equiv A'$, then $\beta = \Lambda$ and either $A = A'$ and $B = B'$ (so that there is but one rule involved) or $B \equiv B'$.

It is interesting to note that the requirement that π be minimal is necessary in Definition 2.2. Consider the following example:

$$S \to aAS|b$$

$$A \to aB$$

$$B \to aAB|aCA|b$$

$$C \to aD$$

$$D \to aAD|aCC$$

and let $\pi_0 = \{\Sigma, \{S\}, \{A, C\}, \{B, D\}\}$ and $\pi = \{\Sigma, \{S\}, \{A, B, C, D\}\}$. G is a real-time grammar. If the minimality condition were dropped from Definition 2.2, G would not be real-time with respect to π but would be with respect to π_0. (We wish to thank the referee for suggesting this example.)

It is clear from the definition that the reduced form of any real-time grammar is also real-time.

THEOREM 2.2. *A language is a Δ_2-real-time language if and only if it is generated by some reduced real-time grammar.*

Proof (*only if*). We shall show that the canonical grammar $G_{\hat{M}}$ from Definition 1.10 is real-time when M is a real-time DPDA. First note that since acceptance is by final state and empty pushdown, one can restrict attention to DPDA with one final state and there is no loss of time.[8] By Lemma 3.2 of [14], $G_{\hat{M}}$ is strict deterministic under the partition π defined in equation (1.11). By the Proposition of that section, π is minimal. Now let us recall that $G_{\hat{M}}$ has productions of two types as defined by (1.9) and (1.10).

By the real-time property of $\hat{M}$ in both cases we have $a \neq \Lambda$. Thus $G_{\hat{M}}$ is Λ-free and, moreover, in Greibach normal form.[9] To prove (1.9) let $A \to \alpha B$, $A' \to \alpha B'\beta$ and assume $A = \overline{(q, Z, q')} \equiv \overline{(q, Z, q'')} = A'$. $G_{\hat{M}}$ is in Greibach normal form, hence $\alpha \neq \Lambda$ and ${}^{(1)}\alpha \in \Sigma$. By the determinism of $\hat{M}$ and Definition 1.10 there is a unique $\gamma \in \Gamma^*$ (as well as $q' \in Q$) such that $\delta(q, {}^{(1)}\alpha, Z) = (q', \gamma)$. Consequently, $\lg(\alpha B) = \lg(\gamma) + 1 = \lg(\alpha B'\beta)$. Hence $\beta = \Lambda$.

(*If*). Let $L = L(G)$ for some reduced real-time grammar G. Our strategy will be to modify the canonical DPDA M_G (cf. Definition 1.12) in such a way that the resulting equivalent DPDA M'_G will be quasi-real-time. This is sufficient for a proof that L is a Δ_2-real-time language since the Λ-free property of G implies $L \neq \{\Lambda\}$. To make it easier to follow the proof, we shall first examine the properties of M_G related to the occurrence of Λ-moves. We shall use the same notation as in Definition 1.11. We have defined the δ-function of M by means of four types of moves: Types 1, 3 and 4 are Λ-moves, and Type 2 is always a non-Λ-move. Accordingly, we have distinguished four types of yield relations (cf. Definition

[8] More formally, the DPDA $\hat{M}$ with one final state constructed in the proof of Lemma 3.1 of [14] is real-time if and only if M is real-time; this follows directly from the construction.

[9] A grammar $G = \langle V, \Sigma, P, S\rangle$ is said to be in *Greibach normal form* if and only if $P \subseteq (N \times \Sigma V^*) \cup \{(S, \Lambda)\}$, i.e., if and only if any production of G has the form $A \to a\alpha$ ($A \in N$, $a \in \Sigma$ and $\alpha \in V^*$) or the form $S \to \Lambda$.

1.11):

$(q, aw, \gamma Z) \vdash_i (q', w, \gamma\alpha)$ if and only if

$\delta(q, a, Z) = (q', \alpha)$ is a move of Type i, $i = 1, 2, 3, 4$.

Let c, c' be two configurations of M_G such that

$$c(\vdash_1 \cup \vdash_3 \cup \vdash_4)^m c' \tag{2.4}$$

for some $m \geqq 0$, where c is the result of a Type 2 move. We are interested in finding a bound on the number m in (2.4)—we shall see that such a bound exists but after suitable modification of M_G. First we notice that (2.4) can be rewritten in the following form:

$$c\left[\prod_{i=1}^{n} (\vdash_1^{k_{1i}} \vdash_3 \vdash_4) \vdash_1^{k_2}\right] c', \tag{2.5}$$

where $k_{1i}, k_2, n \geqq 0, k_2 + \sum_{i=1}^{n} (k_{1i} + 2) = m$, and $\prod$ denotes composition of relations.

CLAIM 1. *If G is a reduced real-time grammar of the form (2.2) and c, c' any two configurations of M_G satisfying a relation of the form (2.5), then $k_{1i} = 0$ for all i and $k_2 \leqq |N|$.*

Proof of the claim. By definition, a Type 1 move can be followed by a Type 3 move if and only if there is a Λ-production in the grammar. Since G is Λ-free, $k_{1i} = 0$ for all i. Also, k_2 consecutive moves of Type 1 can occur if and only if $A \Rightarrow_G^{k_2 - 1} B\alpha$ for some $A, B \in N$ and $\alpha \in V^*$, but since there are only $|N|$ nonterminals and $A \Rightarrow^+ A\alpha$ is not possible in a reduced strict deterministic grammar [14, Thm. 2.3], $k_2 \leqq |N|$ (in fact, $k_2 \leqq \|\pi\|$). The claim is proved.

As a consequence of Claim 1 only the number n in (2.5) can be unbounded. Therefore we restrict our concern to the case $c(\vdash_3 \vdash_4)^n c'$, or more conveniently,

$$c(\vdash_4 \vdash_3)^n c',$$

where c, c' are two configurations of M_G and $n \geqq 0$. Directly from the definition of moves of Types 3 and 4 we have the following claim.

CLAIM 2. *For any $q_j, q_l \in Q - \{q_0\}$; $V_k, V_i \in \pi$ and $\alpha \in V^*$,*

$$(q_j, \Lambda, \overline{V_k, \alpha, V_i}) \dashv_4 \dashv_3 (q_l, \Lambda, \Lambda) \quad \textit{if and only if} \tag{2.6}$$

$A_{kl} \to \alpha A_{ij}$ *is a production*[10] *in G.*

Let us call any letter $\overline{V_k, \alpha, V_i} \in \Gamma$ a *saturated letter* if and only if $A \to \alpha B$ for some $A \in V_k$ and $B \in V_i$. Denote by Γ_s the set of all saturated letters (we have $\Gamma_s \subseteq \Gamma_2$).

CLAIM 3. If $Z_1 = \overline{V_k, \alpha, V_i}$ *is saturated and* $Z_2 = \overline{V_k, \alpha\beta, V_j} \in \Gamma$, *then* $\beta = \Lambda$ *and* $j = i$.

Proof of the claim. This claim is a consequence of property (2.3) of real-time grammars: since Z_1 is saturated we have $A \to \alpha B$ for some $A \in V_k$ and $B \in V_i$; and since $Z_2 \in \Gamma_2$ we have $A' \to \alpha\beta C\theta$ for some $A' \in V_k$, $C \in V_j$, and $\theta \in V^*$. Then by the strict determinism of G, $\beta C\theta = B'\beta'$ for some $B' \in N$, $\beta' \in V^*$. Bjt $\beta' = \Lambda$ by (2.3)

[10] Recall the convention $V_i = \{A_{i0}, \cdots, A_{in_i}\}$ for $V_i \in \pi$ (cf. (1.13)).

and hence $C = B'$, $\beta = \theta\Lambda$ and $V_i = [B'] = [C] = V_j$ by the strict determinism of G. The claim is proved.

By Claims 2 and 3 any saturated letter on the top of the store is always erased (never rewritten) and this is realized by a transition of the form $\vdash_4\vdash_3$. Now, for any saturated letter $Z = \overline{V_k, \alpha, V_i} \in \Gamma_s$ we can uniquely specify a partial function

$$f_Z : Q \to_p Q \tag{2.7}$$

such that $f_Z(q_j) = q_l$ if and only if $q_j, q_l \neq q_0$ and one or the other side of equivalence (2.6) holds. We can extend (2.7) to

$$f_\eta : Q \to_p Q \tag{2.8}$$

for $\eta \in \Gamma_s^+$ using (2.7) as a basis and defining inductively $f_{\eta Z} = f_\eta f_Z$, i.e., for any $q \in Q$, $f_{\eta Z}(q) = f_\eta(f_Z(q))$ provided $f_Z(q)$ is defined. Note that there are only a finite number of distinct functions f_η. In fact, $|\{f_\eta | \eta \in \Gamma_s^+\}| \leqq (|Q| + 1)^{|Q|}$.

CLAIM 4. *For any* $q, q' \in Q$; $\gamma, \gamma' \in \Gamma^*$ *and* $n \geqq 1$,

$$(q, \Lambda, \gamma)(\vdash_3\vdash_4)^n(q', \Lambda, \gamma') \quad \textit{if and only if} \quad \gamma = \gamma'\eta$$

for some $\eta \in \Gamma_s^n$ *and* $q' = f_\eta(q)$.

This claim follows from Claims 2 and 3 by a straightforward induction on n.

Thus the only effect of $(\vdash_3\vdash_4)^n$ is erasing a saturated string of length n from the store and changing the state of control. Claim 4 gives the basis for the elimination of an unbounded sequence of Λ-moves: instead of a saturated string η of length $\geqq 1$, on the store of the modified DPDA M'_G appears a single letter $\bar{f}_\eta$ representing the function f_η.

For completeness we describe formally the construction of M'_G. For this let $M_G = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, \{q_0\}\rangle$ be the canonical DPDA from Definition 1.11. Define M'_G as follows:

$$M'_G = \langle Q', \Sigma, \Gamma', \delta', q_0, Z_0, \{q_0\}\rangle, \tag{2.9}$$

where $Q' = Q \cup \{q_Z | Z \in \Gamma_s\}$ (the new states are added for technical reasons only), $\Gamma' = \Gamma \cup \text{Alph}\,\{\bar{f}_\eta | \eta \in \Gamma_s^+\}$ and δ' is defined as follows. If $A \to \alpha B$ for some $A \in V_i$ and $B \in V_k$, then define

(i) $\delta'(q_0, \Lambda, \overline{V_i}, \alpha) = (q_Z, \Lambda)$, where $Z = \overline{V_i, \alpha, V_k} \in \Gamma_s$;

(ii) $\delta'(q_Z, \Lambda, \bar{f}_\eta) = (q_0, \bar{f}_{\eta Z}\overline{V_k}, \Lambda)$ for all $\bar{f}_\eta \in \Gamma' - \Gamma$, $Z = \overline{V_i, \alpha, V_k}$; and

(iii) $\delta'(q_Z, \Lambda, \gamma) = (q_0, \gamma\bar{f}_Z\overline{V_k}, \Lambda)$, where $Z = \overline{V_i, \alpha, V_k}$, and for all $\gamma \in \Gamma$.

Moreover, for any $q_j \in Q - \{q_0\}$ and $\bar{f}_\eta \in \Gamma' - \Gamma$, define

(iv) $\delta'(q_j, \Lambda, \bar{f}_\eta) = (f_\eta(q), \Lambda)$.

In all other cases define

(v) $\delta'(q, a, Z) = \delta(q, a, Z)$.

Moves (i), (ii) and (iii) are new special cases of Type 1 moves of M_G, move (iv) is a special case of Type 4 moves. We observe that a saturated letter never appears in the store and—even if some Λ-moves were added—M'_G is quasi-real-time. It is easy to see that $T_2(M'_G) = T_2(M_G)$: Claim 4 suffices for that. Q.E.D.

It turns out that the Δ_2-real-time languages have a very convenient property which makes the problem of deciding whether or not they are regular trivial. Let us first mention that while the regularity of context-free languages is undecidable [17], the same problem for deterministic languages has been shown to be decidable

by Stearns [22]. However, Stearns' decision procedure is unsuitable for practical purposes and is only of theoretical interest. (It is shown that if for a given DPDA M the language accepted by M is regular, then the number of states of a corresponding finite automaton is bounded by an expression of order $k^{n^{n^n}}$, where k is the number of pushdown letters and n the number of states of M. Since the equivalence of a given deterministic language and a given regular set is decidable [8], the decision of regularity can be based on testing all finite automata of that size or less.) The property expressed by Theorem 2.3 gives a simple tool for recognizing regular languages among the Δ_2-real-time languages.

DEFINITION 2.3. A grammar $G = \langle V, \Sigma, P, S \rangle$ is *self-embedding* if and only if there is a nonterminal $A \in N$ such that $A \Rightarrow^* \alpha A \beta$ for some $\alpha, \beta \in V^+$.

THEOREM 2.3. *Let G be any reduced real-time grammar. Then L(G) is regular if and only if G is not self-embedding.*

Remark. There is a well-known result [4] that $L(G)$ is regular for a non-self-embedding grammar G. However, if a given grammar is self-embedding, we cannot say anything about its regularity, in general. Thus the important part of Theorem 2.3 is its "only if" direction. (For an analogous result for so called simple deterministic languages, which are discussed in the next section, cf. [18].)

To prove the theorem we need the following lemma which will have other applications as well.

LEMMA 2.1. *Let G be a reduced real-time grammar of the form* (2.2). *Assume*

$$S \Rightarrow_L^* uA\alpha \tag{2.10}$$

and

$$A \Rightarrow_L^* vA\beta \tag{2.11}$$

for some $A \in N, u \in \Sigma^*, \alpha \in V^*, v \in \Sigma^+$ *and* $\beta \in V^+$.

Then for any $n \geqq 0$ *and* $w \in \Sigma^*$,

$$S \Rightarrow^* uv^n w \quad \textit{implies} \quad \lg(w) \geqq n. \tag{2.12}$$

Proof. Let $k, m \geqq 0$ be such that $S \Rightarrow_L^k uA\alpha$ and $A \Rightarrow_L^m vA\beta$ as in (2.10) and (2.11) respectively. As a consequence of a multiple application of Lemma 2.2 of [14] any derivation of the form (2.12) can be rewritten in the form of a left-most derivation

$$s \Rightarrow_L^k uA_0\alpha_0 \Rightarrow_L^m uvA_1\beta_1\alpha_0 \Rightarrow_L^m uv^2A_2\beta_2\beta_1\alpha_0 \Rightarrow_L^m \cdots \Rightarrow_L^m uv^nA_n\beta_n \cdots \beta_1\alpha_0 \Rightarrow^* uv^n w, \tag{2.13}$$

where $A_i \in N$ and $\alpha_0, \beta_i \in V^*$ for $i = 1, \cdots, n$ and $A_0 \equiv A_1 \equiv \cdots \equiv A_n \equiv A$.

CLAIM 5. $\beta_i \neq \Lambda$ *for each* i, $1 \leqq i \leqq n$.

Proof of the claim. First note that the derivation $A \Rightarrow_L^m vA\beta$ can be written for some $B, C \in N$ in the form

$$A \Rightarrow_L^{m_1} v_1 B \Rightarrow_L v_1v_2C\beta' \Rightarrow_L^{m_2} v_1v_2v_3A\beta''\beta' = vA\beta, \tag{2.14}$$

where $\beta' \neq \Lambda$ and $m_1 + m_2 + 1 = m$. Now assume for the sake of contradiction that $\beta_i = \Lambda$ for some i, $1 \leqq i \leqq n$. Then we can extract from (2.13):

$$A_{i-1} \Rightarrow_L^m vA_i\beta_i = vA_i \tag{2.15}$$

and since $A \equiv A_{i-1}$ and using Lemma 2.2 of [14] again we obtain from (2.14) and (2.15) for some $B', C' \in N$,

$$A_{i-1} \Rightarrow_L^{m_1} v_1 B' \Rightarrow_L v_1 v_2 C' \Rightarrow_L^{m_2} v_1 v_2 v_3 A_i, \tag{2.16}$$

where $B' \equiv B$. But then we have $B' \to v_2 C'$ from (2.16) and $B \to v_2 C \beta'$ from (2.14), and by the property (2.3) of real-time grammars, $\beta' = \Lambda$. Hence the contradiction and the claim is proved.

Let us return to (2.13). Since G is Λ-free, $\beta_i \Rightarrow^* w_i \in \Sigma^+$ for $1 \leqq i \leqq n$ and $\lg(w) \geqq \sum_{i=1}^{n} \lg(w_i) \geqq n$. Q.E.D.

Proof of Theorem 2.3. Appealing to the remark after the theorem we restrict ourselves to the "only if" direction. We shall use the following known result from finite automata theory.

FACT. *Let* $L \subseteq \Sigma^*$ *be a regular language. Then there exists a number* $n \geqq 1$ *such that any string* $x \in L$, *where* $\lg(x) \geqq n$, *can be written in the form* $x = y_1 z y_2$, *where* $y_1, y_2 \in \Sigma^*, z \in \Sigma^+, \lg(zy_2) \leqq n$ *and for all* $k \geqq 0, y_1 z^k y_2 \in L$.

The proof of this fact can be found in [1].

Let G be a reduced real-time grammar which generates a regular language $L = L(G)$. Assume, for the sake of contradiction, that G is self-embedding, i.e., that $A \Rightarrow^* \alpha A \beta$ for some $A \in N$ and $\alpha, \beta \in V^+$. Since G is reduced and Λ-free we have

$$S \Rightarrow_L^* uA\alpha',$$

$$A \Rightarrow_L^* vA\beta'$$

for some $u \in \Sigma^*$, $\alpha' \in V^*$, $v \in \Sigma^+$, and $\beta' \in V^+$. By Lemma 2.1 for arbitrary $n \geqq 0$ and $w \in \Sigma^*$,

$$uv^n w \in L \quad \text{implies} \quad \lg(w) \geqq n. \tag{2.17}$$

In particular, let us take n from the Fact above and w the shortest string such that $uv^n w \in L$. This definition of w is meaningful since at least one string $w' \in \Sigma^*$ exists with the property that $uv^n w' \in L$: for instance, $w' = w_1 w_2^n w_3$, where $w_1, w_2, w_3 \in \Sigma^*$ are such that $A \Rightarrow^* w_1$, $\alpha' \Rightarrow^* w_2$, and $\beta' \Rightarrow^* w_3$ (note that G is reduced).

Now by (17), $\lg(w) \geqq n$ and if we take $x = uv^n w \in L$ in the above Fact we can find $y_1, y_2 \in \Sigma^*$ and $z \in \Sigma^+$ such that $x = y_1 z y_2$ or, since $\lg(zy_2) \leqq n \leqq \lg(w)$, we can write $x = uv^n y_1' z y_2$ for some $y_1' \in \Sigma^*$. Then the case $k = 0$ in the Fact yields $\lg(y_1' y_2) < \lg(y_1' z y_2) = \lg(w)$, which contradicts the assumption that w is the shortest string such that $uv^n w \in L$. Therefore G is not self-embedding. Q.E.D.

Thus, given a real-time DPDA M, the decision whether $T_2(M)$ is regular or not involves only a construction of the canonical grammar $G_{\hat{M}}$ (which has been shown to be real-time) and examination of the self-embedding property of $G_{\hat{M}}$.

It is of some interest to know whether Theorem 2.3 is best possible or whether it could be extended to the full family of strict deterministic languages. To show the result to be best possible, one would want to find a reduced self-embedding strict deterministic grammar generating a regular set. The grammar

$$S \to aSA \mid b$$

$$A \to \Lambda$$

is such a grammar but the self-embedding is a "trick" because of the Λ-rules. The following is a reduced Λ-free self-embedding strict deterministic grammar G:

$$S \to A|BaC|Bb$$

$$A \to aA|aBb$$

$$B \to aBa|b$$

$$C \to aC|b$$

But $L(G) = a^*ba^*b$ is regular. Thus Theorem 2.3 cannot be extended to the strict deterministic grammars.

Our last objective in this section is to prove that the real-time restriction for deterministic DPDA is essential. It is known [13] that this is not the case for general PDA.

First we show how Lemma 2.1 and the results of [14] can be used to establish that a particular language has no real-time grammar.

THEOREM 2.4. *Let* $\Sigma = \{a, b, c\}$. *There is no real-time grammar generating the language* $L = \{a^n b^k a^n, a^k b^n c^n | k, n \geqq 1\}$ (*cf. Example* 1.1).

Proof. Assume for the sake of contradiction that G is a real-time grammar of the form (2.2) generating L, and assume, without loss of generality, that G is reduced. We shall use the iteration theorem for Δ_2. Let $p = p(L)$ be the integer satisfying the iteration theorem for Δ_2 [15, Thm. 2.2] and consider the string $a^p b^p c^p \in L$. Choose a set of positions $K = \{2p + 1, \cdots, 3p\}$. Let $\phi = (v_1, \cdots, v_5)$ be the factorization of $a^p b^p c^p$ obtained by Theorem 2.2 of [15]. To satisfy conditions 3 and 2′ of Theorem 2.2 of [15], the components of ϕ must have the following form: $v_1 = a^p b^i$, $v_2 = b^k$, $v_3 = b^j c^r$, $v_4 = c^k$, and $v_5 = c^s$, where $i, j, s \geqq 0$, $k, r \geqq 1$ and $i + j = r + s = p - k$. Now following exactly the lines of the proof of Theorem 2.2 of [15] we arrive at the following derivation (cf. [15, § II, (7)]):

$$S \Rightarrow^* v_1 A v_5 \Rightarrow^+ v_1 v_2 A v_4 v_4 \Rightarrow^+ v_1 \cdots v_5 = a^p b^p c^p.$$

Thus we can write

$$S \Rightarrow_L^* v_1 A \alpha,$$

$$A \Rightarrow_L^* v_2 A \beta$$

for some $\alpha, \beta \in V^*$. Since $v_4 \neq \Lambda$ also $\beta \neq \Lambda$. Now the assumptions of Lemma 2.1 are satisfied, and thus for any $n \geqq 0$ and $w \in \Sigma^*$,

$$v_1 v_2^n w \in L \quad \text{implies} \quad \lg(w) \geqq n. \tag{2.18}$$

Let us take $n > p$ and $w = a^p$ and consider the string $v_1 v_2^n w = a^p b^i b^{kn} a^p \in L$. But here $\lg(w) = \lg(a^p) = p < n$, in contradiction to (2.18). Therefore G is not real-time. Q.E.D.

Now we show proper containment of the real-time Δ_i languages in Δ_i for $i = 0, 1, 2$.

THEOREM 2.5. *The family of* Δ_i*-real-time languages is properly contained in the family* Δ_i *for* $i = 0, 1, 2$.

Proof. Consider the following language L:

$$L = \{a^{i_1} b a^{i_2} b \cdots a^{i_{r-1}} b a^{i_r} c^s a^{i_{r-s+1}} | r \geqq 1, 1 \leqq i_j \text{ for } 1 \leqq j \leqq r, 1 \leqq s \leqq r\}.$$

It is clear that $L \in \Delta_2 \subseteq \Delta_1 \subseteq \Delta_0$. But it has been shown in [19] that L cannot be accepted by any real-time, on-line multitape Turing machine. Thus L is not a real-time Δ_i language for $i = 0, 1, 2$. Q.E.D.

It is also possible to prove this theorem using the language defined in Lemma 2.2 and our techniques. The present proof is somewhat simpler and we wish to thank the referee for suggesting it.

3. Simple deterministic languages. In [14] a nontrivial hierarchy of strict deterministic languages was established. This hierarchy was based on the degree of a language (cf. § 4) which was related to the number of states in a DPDA in [14]. In this section, we shall restrict our attention to the "simplest" class of strict deterministic languages in the heirarchy of [14], i.e., to the strict deterministic languages of degree 1.

First we show a convenient characterization of strict deterministic grammars of degree 1. This characterization makes it easier to recognize these grammars.

THEOREM 3.1. *Let G be any grammar. Then G is strict deterministic of degree* 1 *if and only if* $A \to_G \alpha|\beta$ *implies that either* $\alpha = \beta$ *or* α, β *have a terminal distinguishing pair.*[11]

Proof (*only if*). Let G be strict deterministic, $\deg(G) = 1$ and let $A \to_G \alpha|\beta$, $\alpha \neq \beta$. If α, β have no distinguishing pair, one of them is a (proper) prefix of the other, which contradicts the strict determinism of G. Let (C, D) be the distinguishing pair of α, β. Then $C \equiv D$. By definition of distinguishing pairs, $C \neq D$. If $C, D \in N$, we have a contradiction of $\deg(G) = 1$. Hence $C, D \in \Sigma$.

(*If*). Assume G has the above property. Recall Algorithm 1 of [14] which tests a given context-free grammar for the property of being strict deterministic. Now let us apply that algorithm to G. Starting in Step 1 with partition π consisting of Σ and otherwise only of singletons, the algorithm never reaches Step 6 (Step 5 is always followed by Step 3 since $\Sigma \in \pi$) and thus π is not altered. The algorithm halts in Step 8 (Step 4 is never followed by Step 7 since $A_i \equiv A_j$ implies $A_i = A_j$, which implies $\alpha_i \neq \alpha_j$ by the assumption of the indices in the algorithm). Therefore π is strict and $\|\pi\| = \|\pi_0\| = 1$. Q.E.D.

Next, consider grammars in Greibach normal form[9] with the property that for all $A \in N$, $a \in \Sigma$ and $\alpha, \beta \in V^*$,

$$A \to_G a\alpha|a\beta \quad \text{implies} \quad \alpha = \beta.$$

Following [18] we call such grammars *s-grammars*. Directly from Theorem 3.1 we see that s-grammars are strict deterministic of degree 1. It has been shown that s-grammars generate exactly the class of languages defined as follows.

DEFINITION 3.1. A context-free language L is called *simple deterministic*[12] if and only if $L = T_2(M)$ for some real-time DPDA M with $|Q| = 1$.

We shall show that the requirement that the DPDA be real-time is unessential.

THEOREM 3.2. $L \in \Delta_2$ *and* $\deg(L) = 1$ *if and only if either L is simpler deterministic or* $L = \{\Lambda\}$.

Proof. The "if" direction is immediate from Definition 3.1, Theorem 4.1 of [14] and the previously mentioned fact that the language $\{\Lambda\}$ is of degree 1.

[11] Recall Definition 1.5.

[12] The reader should not confuse this term with "simple $LR(k)$ grammars" defined in [6].

(Alternatively, we could use our Theorem 3.1 and the result in [18] which was previously discussed.)

For the "only if" direction, let G be a strict deterministic grammar of degree 1 and let $L(G) \neq \{\Lambda\}$. We can assume, without loss of generality, that G is Λ-free (cf. the Fact following Definition 4.1 of [14]). We shall show that G satisfies (2.3) from Definition 2.2. But this is immediate from the assumption that $\deg(G) = 1$, i.e., $\|\pi_0\| = 1$, since then $A \equiv A' \pmod{\pi_0}$ implies $A = A$, and $B \equiv B' \pmod{\pi_0}$ (which follows from the strictness of π_0) implies $B = B$. Now $\beta = \Lambda$ by the strictness of π_0. Thus we have (2.3) and G is real-time. The result then follows from Theorem 2.2 and Definition 3.1. Q.E.D.

COROLLARY. *For the case of degree* 1, *the family of* Δ_2*-quasi-real-time languages coincides with the family* Δ_2.

Proof. This result is a direct consequence of Theorems 3.2 and 2.1. Q.E.D.

Theorem 3.2 is an interesting result because it shows that the simple deterministic languages, which have several interesting properties (cf. [18], [17]), can be placed at the beginning of a natural hierarchy of strict deterministic languages.[13]

One of the most important mathematical properties of simple deterministic languages is that they have a rather elegant procedure for deciding their equivalence (cf. [18]).[14] It turns out that the idea of their procedure can be extended to a more general case when one of the grammars can be a general strict deterministic grammar. Since the presentation and rigorous proof of such a procedure would require a complex development, we present the result as a proposition without proof.

PROPOSITION. *It is decidable whether two deterministic languages, one of which is simple deterministic, are equivalent.*

In this proposition, languages are supposed to be presented in the form of DPDA or strict deterministic grammars (for their end-marked version; note that the equivalence problem for Δ_0 is logically the same as for Δ_2).

The above proposition may be an approach to a new attack on the equivalence problem for deterministic languages.

REFERENCES

[1] M. A. ARBIB, *Theories of Abstract Automata*, Prentice-Hall, Englewood Cliffs, N.J., 1969.

[2] R. V. BOOK AND S. A. GREIBACH, *Quasi-realtime languages*, Math. Systems Theory, 4 (1970), pp. 97–111.

[3] W. J. CHANDLER, *Abstract families of deterministic languages*, Proc. ACM Symposium on Theory of Computing, Marina Del Rey, 1969, pp. 21–30. Also (complete version) SDC Tech. Rep. TM-738/052/00, Santa Monica, Calif., 1969.

[4] N. CHOMSKY, *On certain formal properties of grammars*, Information and Control, 2 (1959), pp. 137–167.

[5] S. N. COLE, *Deterministic pushdown store machines and real-time computation*, J. Assoc. Comput. Mach., 18 (1971), pp. 306–328.

[6] F. L. DEREMER, *Simple LR(k) grammars*, Comm. ACM, 14 (1971), pp. 453–460.

[7] A. FUSOAKA, *A note on a decomposition theorem for simple deterministic languages*, Information and Control, 19 (1971), pp. 272–274.

[13] They already occupy such a place in the hierarchy of $LL(k)$ languages: they are precisely the Λ-free $LL(1)$ languages of [21].

[14] A more general but less elegant procedure for deciding the equivalence of $LL(k)$ languages is described in [21].

[8] S. GINSBURG AND S. A. GREIBACH, *Deterministic context-free languages*, Ibid., (1966), pp. 602–648.

[9] S. GINSBURG AND E. SPANIER, *Finite-turn pushdown automata*, SIAM J. Control, 4 (1966), pp. 423–434.

[10] S. A. GREIBACH, *A new normal form theorem for context free grammars*, J. Assoc. Comput. Mach., 12 (1965), pp. 42–52.

[11] ———, *An infinite hierarchy of context-free languages*, Ibid., 16 (1969), pp. 91–106.

[12] ———, *Characteristic and ultrarealtime languages*, Information and Control, 18 (1971), pp. 65–98.

[13] L. H. HAINES, *Generation and recognition of formal languages*, Ph.D. thesis, M.I.T., Cambridge, Mass., 1965.

[14] M. A. HARRISON AND I. M. HAVEL, *Strict deterministic grammars*, J. Comput. System Sci., 7 (1973).

[15] ———, *On the parsing of deterministic languages*, submitted for publication.

[16] J. HARTMANIS AND R. E. STEARNS, *On the computational complexity of algorithms*, Trans. Amer. Math. Soc., 117 (1965), pp. 285–306.

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

[18] A. J. KORENJAK AND J. E. HOPCROFT, *Simple deterministic languages*, Conference Record of 1966 IEEE 7th Annual Symposium on Switching and Automata Theory, Berkeley, Calif., 1966.

[19] A. L. ROSENBERG, *Real-time definable languages*, J. Assoc. Comput. Mach., 14 (1967), pp. 645–662.

[20] ———, *On the independence of real-time definability and certain structural properties of context-free languages*, Ibid., 15 (1968), pp. 672–679.

[21] D. J. ROSENKRANTZ AND R. E. STEARNS, *Properties of deterministic top-down grammars*, Information and Control, 17 (1970), pp. 226–255.

[22] R. E. STEARNS, *A regularity test for pushdown machines*, Ibid., 11 (1967), pp. 323–340.

SIAM J. COMPUT.
Vol. 1, No. 4, December 1972

INFIX TO PREFIX TRANSLATION: THE INSUFFICIENCY OF A PUSHDOWN STACK*

EDWARD M. REINGOLD†

Abstract. The permutations of the input string achievable by an algorithm which uses a single pushdown stack and M random access storage locations are characterized, and the characterization is used to show that no such algorithm can translate arithmetic expressions from infix to prefix.

Key words. Permutations, pushdown stack, Polish prefix.

There is a well-known algorithm which reads infix arithmetic expressions from left to right, one character at a time up to an end-marker, and, using a pushdown stack, produces from left to right, one character at a time, the suffix form of the expressions; see, for example, [5, § 1.2]. The essence of this algorithm is that it simply shuffles the characters about between the input string, the pushdown stack, and the output string. There is, however, no known corresponding algorithm for translating infix into prefix form, and the motivation for this note is to show that such an algorithm does not exist. In fact, we go much further and completely characterize the extent to which such algorithms can rearrange their input strings. The result on infix to prefix translation follows as a corollary.

The model of algorithms which we will allow is a variant of a one-way, deterministic, finite state pushdown transducer whose finite input, output, and stack alphabets coincide. Rather than defining this formally as a 7-tuple and basing our proofs on the various transition functions, output functions, etc., we will give more informal (though no less correct) proofs and refer the unsatisfied reader to Ginsburg and Rose [2, § 3] for machinery needed to formalize the definitions and the arguments.

Informally, then, this model of algorithms can perform only the following kinds of steps: (a) It can read the input string one character at a time from left to right until it reaches an end-marker. (b) The character read from the input may be put directly into the output, which is also produced one character at a time from left to right, or it may be put on the top of a pushdown stack. (c) At any time, the only element accessible on the stack is the top element which can, if desired, be popped off the stack allowing access to the second element in the stack. The element popped off the stack can be thrown away, or it can be put into the output string. Once a symbol has been put in the output string it is forever after inaccessible and immutable. Since the actions the algorithm takes can only rearrange the input string or delete characters from it, we call this model a *pushdown permuter*, or p.d.p. for short. A p.d.p. is nothing more than a control for a "switchyard" arrangement between the input string, the stack, and the output string.

* Received by the editors June 28, 1972, and in revised form September 25, 1972.

† Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801. This work was completed while the author was spending six weeks at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York. It was also supported in part by the National Science Foundation under Grant GJ-31222.

We are interested in which permutations $s_{p_1}s_{p_2}\cdots s_{p_n}$ of the input string $s_1s_2\cdots s_n$ can be produced as the output string. To simplify the notation we consider $1\,2\cdots n$ to be the input string and $p_1p_2\cdots p_n$ to be the output string. Knuth [4, § 2.2.1, exercise 5] has shown that if the algorithm can have access to nothing except the top stack symbol, then $p_1p_2\cdots p_n$ can be produced if and only if there is no subsequence[1] $p_ip_jp_k$ of $p_1p_2\cdots p_n$ for which $p_i > p_k > p_j$. For some extensions of this result to networks of stacks and queues, instead of only a single stack, see Tarjan [6] and Even and Itai [1].

The main theorem of this note generalizes the one stated in the previous paragraph to the case in which some fixed number of symbols can be stored in a random access memory, in addition to the stack. This assumption is reasonable when dealing with computer algorithms (our motivation) and it corresponds to the fact that in a computer program we can have, in addition to a stack, any fixed number of storage locations. Since the number of locations must be finite, it is clear that arbitrarily large amounts of information cannot be retained. We assume that there is a memory of M random access cells, each of which can hold at most one symbol, and we assume further that *no* other symbols besides these M can be stored outside of the pushdown stack.

THEOREM. *A p.d.p. with M memory cells can permute the input string* $1\,2\cdots n$ *to* $p_1p_2\cdots p_n$ *if and only if there is no subsequence* $xy_1\cdots y_{M+1}z_1\cdots z_{M+1}$ *of* $p_1p_2\cdots p_n$ *such that for all i and j,* $x > z_i > y_j$.

Before proving this theorem, let us see one of its consequences. It turns out that the class of permutations required to translate infix expressions to their prefix equivalents contains some of these unachievable permutations. More precisely, we have the following corollary.

COROLLARY. *There is no p.d.p. which reads infix expressions over the alphabet* $\{a, b, +, *, (,)\}$ *and translates them into prefix form.*

Proof. Suppose such a p.d.p. does exist, and let M be the number of random access cells which it has. Consider the input string

$$(\cdots(((\underset{1}{a}\ \underset{2}{+}\ \underset{3}{a})\ \underset{4}{*}\ \underset{5}{a}\ \underset{6}{+}\ \underset{7}{a}\cdots)\ \underset{4n}{*}\ \underset{4n+1}{a}\ \underset{4n+2}{+}\ \underset{4n+3}{b})\ \underset{4n+4}{*}\ \underset{4n+5}{b}\cdots)\ \underset{8n}{*}\ \underset{8n+1}{b}\ \underset{8n+2}{+}\ \underset{8n+3}{b}$$

Translating this expression into prefix form requires permuting it to $+*+*\cdots+*+aa\cdots abb\cdots b$. Any such permutation must have as a subsequence

$$8n+2 \qquad \pi_1(1, 3, 5, \cdots, 4n-1) \qquad \pi_2(4n+3, 4n+5, \cdots, 8n+1)$$

(we are ignoring the parentheses), where π_1 and π_2 are permutations of $2n$ symbols. By the theorem then, at least $2n$ memory cells are needed for the translation, and $2n$ can be made larger than any given M by considering a sufficiently long expression of the type illustrated. ∎

A stronger version of the result in this corollary was first observed by Lewis and Stearns [4] who gave an outline of a proof.

Proof of Theorem. For a permutation $p_1p_2\cdots p_n$ of $1\,2\cdots n$ consider the p.d.p. which behaves as follows: at each input symbol i the p.d.p. places i into a

[1] We define $p_{i_1}p_{i_2}\cdots p_{i_k}$ to be a subsequence of $p_1p_2\cdots p_n$ provided that $1 \leqq i_1 < \cdots < i_k \leqq n$.

vacant memory cell, if one exists, otherwise it examines i and the symbols in the M memory cells, and, of those $M + 1$ symbols, it puts the one which appears right-most in $p_1 p_2 \cdots p_n$ onto the stack. If at any point $p_1 p_2 \cdots p_k$ has been put into the output and p_{k+1} is in one of the memory cells, is at the top of the stack, or is the next symbol in the input string, then p_{k+1} is put into the output and if a memory cell is vacated, it is filled with the top symbol in the pushdown stack, which is popped up. If there are no more input symbols, then, since we kept the left-most occurring symbols in the memory cells, we put them into the output in the appropriate order, filling the vacated cells with symbols taken from the top of the pushdown stack. This process continues until all of the symbols have been put into the output, or, perhaps, until the p.d.p. gets stuck with all memory cells filled and the symbol which must be put next into the output inaccessible on the stack. We must verify that if $p_1 p_2 \cdots p_n$ has the stated property, then the p.d.p. does not get stuck; clearly, if it does not get stuck, it will produce $p_1 p_2 \cdots p_n$ from $1\ 2 \cdots n$.

Suppose at some point the p.d.p. gets stuck with all memory cells filled and the symbol u, which must be put into the output next, inaccessible below the top of the stack; we will show that this implies the existence of a subsequence which cannot exist by hypothesis. Consider the p.d.p. at the time the symbol u is put into the stack for the last time, never to come out again, and call the contents of the M memory cells at that time $y_1, y_2, \cdots, y_M$. Since the y_i stay in the memory cells while u is put into the stack, they must precede u in $p_1 p_2 \cdots p_n$, and since the p.d.p. does not get stuck until trying to put u into the output, the y_i are all in the output at the time the p.d.p. does get stuck. Let x be the largest symbol in the output preceding all of the y_i at the time the p.d.p. gets stuck. Clearly $x > y_i$ and $x > u$ because at the time u goes into the stack for the last time the p.d.p. is "waiting" for some symbol still in the input while u and the y_i have already been read. Consider the contents of the stack at the time x is put into the output: above u the stack must have at least $M + 1$ symbols $z_1, \cdots, z_{M+1}$ which follow u in $p_1 p_2 \cdots p_n$ but which were read from the input before x and after u and the y_i; otherwise u would be accessible in the memory or at the top of the stack after the y_i have been put into the output, contradicting the fact that the p.d.p. got stuck when u was to be put into the output. Letting $y_{M+1} = u$, we have a subsequence $xy_1 \cdots y_{M+1} z_1 \cdots z_{M+1}$ satisfying the proper ordering conditions on the x, y_i, z_j, contradicting the hypothesis that no such subsequence existed. Thus the p.d.p. could not have gotten stuck.

To prove the converse, suppose that a subsequence $xy_1 \cdots y_{M+1} z_1 \cdots z_{M+1}$ with the stated order properties does exist. Then since $x > y_i$ and $x > z_j$, all of the $2M + 2$ y_i and z_j must have been read and stored *before* we read x and put it into the output. Now since there are $M + 1$ y_i, they cannot be stored entirely within the M memory cells, and since the y_i precede the z_j in the input, there *must* be at least one of the y_i beneath some of the z_j on the stack, say y_r, at the time x is put into the output. Since there are $M + 1$ z_j, they cannot be stored entirely within the M memory cells, and thus y_r is inaccessible until some of the z_j have been put into the output, contradicting the fact that y_r must precede all of the z_j in the permutation. Thus a p.d.p. cannot obtain $p_1 p_2 \cdots p_n$ from $1\ 2 \cdots n$ if it has fewer than M memory cells. ∎

Acknowledgments. The author is grateful to Raymond E. Miller and the referees for their comments, and to Donald E. Knuth for suggesting the present wording of the theorem.

REFERENCES

[1] S. EVEN AND A. ITAI, *Queues, stacks and graphs*, Theory of Machines and Computations, Z. Kohavi and A. Paz, eds., Academic Press, New York, pp. 71–86.

[2] S. GINSBURG AND G. F. ROSE, *Preservation of languages by transducers*, Information and Control, 9 (1966), pp. 153–176.

[3] D. E. KNUTH, *The Art of Computer Programming*, vol. I, *Fundamental Algorithms*, Addison-Wesley, Reading, Mass., 1968.

[4] P. M. LEWIS AND R. E. STEARNS, *Syntax-directed transduction*, J. Assoc. Comput. Mach., 15 (1968), pp. 465–488.

[5] J. NIEVERGELT, J. C. FARRAR AND E. M. REINGOLD, *Computer Approaches to Mathematical Problems*, Prentice-Hall, Englewood Cliffs, N.J., in press.

[6] R. TARJAN, *Sorting using networks of queues and stacks*, J. Assoc. Comput. Mach., 19 (1972), pp. 341–346.