SIAM J. COMPUT.
Vol. 6, No. 1, March 1977

OPTIMAL POLYPHASE SORTING*

DEREK A. ZAVE†

Abstract. A read-forward polyphase merge algorithm is described which performs the polyphase merge starting from an arbitrary string distribution. This algorithm minimizes the volume of information moved. Since this volume is easily computed, it is possible to construct dispersion algorithms which anticipate the merge algorithm. Two such dispersion techniques are described. The first algorithm requires that the number of strings to be dispersed be known in advance; this algorithm is optimal. The second algorithm makes no such requirement, but is not always optimal. In addition, performance estimates are derived and both algorithms are shown to be asymptotically optimal.

Key words. Sorting, tape sorting, merge sorting, polyphase sorting, tape merging, optimal merging, optimal polyphase dispersion, blind dispersion, polyphase dispersion, Fibonacci numbers, generalized Fibonacci numbers, Zeckendorf theorem, generalized Zeckendorf theorem

1. Introduction. This paper presents a mathematical analysis of the structure of the polyphase sort with special emphasis on those properties which are related to the performance of the sort. This analysis will enable us to construct a polyphase sorting algorithm with optimal performance characteristics. We will also construct a near-optimal polyphase sort which is more suitable for applications. Finally, we will investigate the asymptotic performance of both of these algorithms.

Although the polyphase sort has been in use for over a decade, comparatively little work has been done in the direction of optimizing its performance. In an early unpublished paper [7], Sackman and Singer developed methods for predicting the performance of the polyphase merge and showed empirically that in certain cases the performance of the usual method of implementing the polyphase sort could be greatly improved. Independently, Shell [8] developed similar techniques and used them along with some empirical observations to construct an optimal polyphase sorting algorithm. D. E. Knuth [5] has also investigated the optimal polyphase sort and several of his results have been incorporated into this paper.

2. The polyphase merge. We will begin with a brief discussion of the polyphase merge which will serve primarily to introduce our terminology. Further details, as well as information on internal sorting and string merging, which we will not discuss, may be found in the books of Flores [2] and Knuth [5].

Let us suppose that we are given a collection of records containing various kinds of information and let us further suppose that some linear ordering has been defined on this collection. To *sort* the records is to arrange them into a sequence which is increasing with respect to the ordering relation. One method of accomplishing this is by means of merging. First, the collection of records is partitioned into a number of small groups of records which are each sorted to form *strings* of

* Received by the editors April 24, 1974, and in revised form January 15, 1976.

† Computer Science Department, Stanford University, Stanford, California 94305. This research was supported in part by National Science Foundation Grant MCS 72-03752 A03, by the Office of Naval Research under Contract NR 044-402, and by the IBM Corporation.

records. Second, the sorted strings are merged to form larger sorted strings, and so on, until a single sorted string containing all of the records is formed.

In practice, merge sorts are employed when there are more records to be sorted than may be accommodated by a computer's main storage. Groups of records are sorted into strings using the available main storage. The strings are then "*dispersed*" to some secondary storage medium such as mass storage or magnetic tape. The string merging operations are performed as transfers of information from one part of secondary storage to another.

The polyphase sort is a merge sort which is characterized by the manner in which the dispersed strings are merged. Let us suppose that there are $T \geqq 3$ tape units which are numbered from zero to $t = T-1$. We define the *distribution numbers* S_i^n for $i = 1, \cdots, t$ and $n \geqq 1$ by

$$
\begin{aligned}
&S_i^1 = 1 && \text{for } 1 \leqq i \leqq t,\\
&S_1^n = S_t^{n-1} && \text{for } n > 1, \text{ and} \qquad (2.1)\\
&S_i^n = S_{i-1}^{n-1} + S_t^{n-1} && \text{for } n > 1 \quad \text{and} \quad 2 \leqq i \leqq t.
\end{aligned}
$$

From this definition it is easily shown that for $n > 1$, we have

$$S_1^n < S_2^n \leqq \cdots \leqq S_t^n. \tag{2.2}$$

Suppose for some $n \geqq 1$ that $S_1^n + \cdots + S_t^n$ strings have been dispersed to the tapes in the following fashion:

tape:	0	1	2	$\cdots$	t
strings:	0	S_1^n	S_2^n	$\cdots$	S_t^n.

We will call this configuration the *perfect stage n distribution* and the sum

$$S^n = S_1^n + \cdots + S_t^n \tag{2.3}$$

will be called the *stage n perfect number.*

Example 2.1. The following table provides some values of the distribution numbers and perfect numbers when $T = 5$ $(t = 4)$:

n	S_1^n	S_2^n	S_3^n	S_4^n	S^n
1	1	1	1	1	4
2	1	2	2	2	7
3	2	3	4	4	13
4	4	6	7	8	25
5	8	12	14	15	49
6	15	23	27	29	94
7	29	44	52	56	181
8	56	85	100	108	349
9	108	164	193	208	673
10	208	316	372	401	1297

Suppose that we start with the perfect stage n distribution. If we merge together one string from each of the tapes $1, \cdots, t$, then we will obtain a single string which may be written to unit zero. If $n = 1$, then this operation will merge all of the strings since each tape contains exactly one string. If $n > 1$, then, in view of (2.2), we may perform this operation S_1^n times after which we will arrive at the distribution

$$\begin{array}{llllll} \text{tape:} & 0 & 1 & 2 & \cdots & t \\ \text{strings:} & S_1^n & 0 & S_2^n - S_1^n & \cdots & S_t^n - S_1^n \end{array}$$

From the formulas (2.1) we see that this distribution is the same as

$$\begin{array}{llllll} \text{tape:} & 0 & 1 & 2 & \cdots & t \\ \text{strings:} & S_t^{n-1} & 0 & S_1^{n-1} & \cdots & S_{t-1}^{n-1} \end{array}$$

so that if we renumber the tapes $t, 0, 1, 2, \cdots, t-1$, then we obtain the perfect stage $n-1$ distribution.

By repeating this process, we obtain the perfect distributions for stages $n-2$, $n-3$, and so on, until we arrive at the perfect distribution for stage one. A single merge will then produce the final sorted string. This method of merging a perfect number of strings is called the *polyphase merge.*

In practice, the distribution routine rarely produces a perfect number of strings. In order to use the polyphase merge in this case it is necessary to include a number of "*dummy*" (empty) strings in order to fill out the total number of strings to a perfect number. There are therefore two choices which have to be made before using the polyphase merge to sort x strings. First we must choose a starting stage number n; any n for which $x \leqq S^n$ is eligible. Second, we must decide how the $S^n - x$ dummy strings are to be distributed among the x strings. Although many methods have been proposed for distributing the dummy strings, most authors recommend starting with the smallest possible stage number n; we will refer to these approaches collectively as the *standard polyphase sort.*

Since the speed of a merge is usually limited by the transfer rate of the tape units and the speed of the merge algorithm, we see that the time required to perform the polyphase merge is approximately proportional to the total volume of information that moves through the merge. In order to make this idea precise, we assume that the dispersion routine produces strings of approximately the same size; this size will be our unit of information, the *unit string.* The size of a string formed by merging several strings is the sum of the sizes of the input strings and the size of a dummy string is zero. We say that a string is *moved* when that string or any string formed from it by a sequence of one or more merges becomes one of the inputs for a merge. The *volume* of information moved by the polyphase merge is then equal to the sum of the products of the size of each string of the starting distribution and the number of times that string is moved. In this paper we will show how this volume may be minimized.

3. The movement numbers. In general, the polyphase merge does not move all of the S^n strings of the stage n perfect distribution the same number of times. It is for this reason that the polyphase sort is much more difficult to analyze than other merge sorting algorithms. However, much useful information is supplied by the set of *movement numbers* $M_i^n(j)$ which are defined for $n \geqq 1$, $1 \leqq i \leqq t$, and all integers j by the relations

$$(3.1)\quad \begin{aligned} &M_i^1(1)=1 && \text{for } 1 \leqq i \leqq t,\\ &M_i^1(j)=0 && \text{for } j \neq 1 \quad \text{and} \quad 1 \leqq i \leqq t,\\ &M_i^n(j)=M_t^{n-1}(j-1) && \text{for } n>1, \text{and}\\ &M_i^n(j)=M_{i-1}^{n-1}(j)+M_t^{n-1}(j-1) && \text{for } n>1 \quad \text{and} \quad 2 \leqq i \leqq t. \end{aligned}$$

We claim that $M_i^n(j)$ is precisely the number of strings on tape unit i of the stage n perfect distribution which will be moved exactly j times by the polyphase merge. For this to make any sense it is necessary that $M_i^n(j)$ be nonzero only if $1 \leqq j \leqq n$ and that $S_i^n = M_i^n(1)+\cdots+M_i^n(n)$.

We will prove these assertions together by induction on n. When $n=1$, everything is obvious since each of the tapes $1, \cdots, t$ of the perfect distribution contains exactly one string which will be moved by the polyphase merge exactly once. Now suppose that $n>1$ and that everything has,been proved for stage $n-1$. For $M_i^n(j)$ to be nonzero we must have, by (3.1), $M_t^{n-1}(j-1) \neq 0$ or $i \geqq 2$ and $M_{i-1}^{n-1}(j) \neq 0$. These inequalities imply that $1 \leqq j-1 \leqq n-1$ or $1 \leqq j \leqq n-1$ which both imply that $1 \leqq j \leqq n$. We may show that $S_i^n = M_i^n(1)+\cdots+M_i^n(n)$ by summing the last two formulas of (3.1) over j and by applying the corresponding equality for stage $n-1$ and the last two formulas of (2.1). We recall that the stage n polyphase merge is performed by merging S_1^n strings from each of the tapes and then by applying the stage $n-1$ polyphase merge. A string on unit one which will be moved j times will become part of a string on the output tape which will be moved $j-1$ times. Since every string on the output tape contains exactly one string from unit one and since the output tape becomes unit t for the stage $n-1$ merge, we see that unit one must contain exactly $M_t^{n-1}(j-1)$ strings that will be moved exactly j times. If $2 \leqq i \leqq t$, then a j movement string on unit i will either be moved to the output tape or will remain on the tape. From similar considerations, we see that unit i must contain exactly $M_t^{n-1}(j-1)+M_{i-1}^{n-1}(j)$ strings which will be moved exactly j times. This completes the proof.

Example 3.1. Table 3.1 lists some of the movement numbers in the case $t=4$.

In this paper we will make use of a few sets of numbers which are defined using the movement numbers $M_i^n(j)$. We list the definitions:

$$(3.2)\quad \begin{aligned} M^n(j)&=M_1^n(j)+\cdots+M_t^n(j),\\ S_i^n(j)&=M_i^n(1)+\cdots+M_i^n(j),\\ S^n(j)&=M^n(1)+\cdots+M^n(j)=S_1^n(j)+\cdots+S_i^n(j),\\ G_i^n(j)&=S_i^n(1)+\cdots+S_i^n(j),\\ G^n(j)&=S^n(1)+\cdots+S^n(j)=G_1^n(j)+\cdots+G_t^n(j). \end{aligned}$$

In addition, we have already defined

$$\begin{aligned} S_i^n &= S_i^n(n),\\ S^n &= S^n(n)=S_1^n+\cdots+S_t^n. \end{aligned}$$

TABLE 3.1

Movement numbers for $t = 4$

	j	$M_1^n(j)$	$M_2^n(j)$	$M_3^n(j)$	$M_4^n(j)$
$n = 1$	1	1	1	1	1
$n = 2$	1	0	1	1	1
	2	1	1	1	1
$n = 3$	1	0	0	1	1
	2	1	2	2	2
	3	1	1	1	1
$n = 4$	1	0	0	0	1
	2	1	2	3	3
	3	2	3	3	3
	4	1	1	1	1
$n = 5$	1	0	0	0	0
	2	1	2	3	4
	3	3	5	6	6
	4	3	4	4	4
	5	1	1	1	1

In a number of the form $A_i^n(j)$ the superscript n is the associated stage number, the subscript i is the number of a tape unit, and j is some number of movements. $A^n(j)$ is formed from $A_i^n(j)$ by summing over $i = 1, \cdots, t$ and A_i^n is formed from $A_i^n(j)$ by setting $j = n$. In a similar fashion we may form A^n from A_i^n or $A^n(j)$.

Except for the numbers $G_i^n(j)$ and $G^n(j)$, which are used in connection with the volume function, the various sets of numbers which we have defined express some simple properties of the perfect stage n distribution:

$M_i^n(j)$ The number of strings on unit i which will be moved exactly j times.
$M^n(j)$ The number of strings which will be moved exactly j times.
$S_i^n(j)$ The number of strings on unit i which will be moved at most j times.
$S^n(j)$ The number of strings which will be moved at most j times.
S_i^n The number of strings on unit i.
S^n The total numbers of strings.

A set of numbers $A^n(j)$ is said to be a *t-array* if the following relation is satisfied for all integers n and j:

$$A^n(j) = A^{n-1}(j-1) + \cdots + A^{n-t}(j-1). \tag{3.3}$$

We will call a sum of this form a *t-sum*. When a t-array is represented as a table of numbers, then we will let j index the rows and n index the columns. It is clear that the t-array $A^n(j)$ is completely determined by its values on the vertical strip $1 - t \leqq n \leqq 0$ (or any other strip of width t). We will call this strip the *initialization region.*

Most of the sets of numbers which we have defined can be expressed as t-arrays. The t-array approach exposes many of the interesting properties of these numbers which are obscured by the original definitions. Since all of these numbers are defined in terms of the movement numbers, we will begin by showing that the movement numbers may be defined as t-arrays.

For each $i = 1, \cdots, t$ we define the t-array $A_i^n(j)$ by specifying that $A_i^{i-t}(0) = 1$ is the only nonzero element of the initialization region for $A_i^n(j)$. We will show that for all $n \geqq 1$, $1 \leqq i \leqq t$ and all j that $A_i^n(j) = M_i^n(j)$. It is clear that the only nonzero values in the columns $n = -t$ are $A_t^{-t}(-1) = 1$ and $A_i^{-t}(0) = -1$ for $1 \leqq i \leqq t$. If we let δ_b^a denote the Kronecker delta, then for $-t \leqq n \leqq 0$ we have $A_t^n(j) = \delta_0^n \delta_0^j + \delta_{-t}^n \delta_{-1}^j$ and $A_i^n(j) = \delta_{i-t}^n \delta_0^j - \delta_{-t}^n \delta_0^j$ for $1 \leqq i < t$. Therefore, for $1 - t \leqq n \leqq 0$, we have

$$A_t^{n-1}(j-1) = \delta_0^{n-1} \delta_1^j + \delta_{1-t}^n \delta_0^j = \delta_{1-t}^n \delta_0^j = A_1^n(j)$$

and for $2 \leqq i \leqq t$,

$$\begin{aligned} A_{i-1}^{n-1}(j) + A_t^{n-1}(j-1) &= \delta_{i-t-1}^{n-1} \delta_0^j - \delta_{-t}^{n-1} \delta_0^j + \delta_0^{n-1} \delta_0^{j-1} + \delta_{-t}^{n-1} \delta_{-1}^{j-1} \\ &= \delta_{i-t}^n \delta_0^j = A_i^n(j). \end{aligned}$$

These relations correspond to the last two formulas of (3.1) and since they hold for n and j in the initialization region, they can be extended to all values of n and j by a simple induction argument using the recurrence relation (3.3). Since the only nonzero values in the columns $n = 1$ are $A_i^1(1) = 1$, we see that the numbers $A_i^n(j)$ also satisfy the first two relations of (3.1). We therefore conclude that $M_i^n(j) = A_i^n(j)$ for all $n \geqq 1$.

Below we list the various t-arrays in which we will be interested and specify the nonzero values in their respective initialization regions:

$M_i^n(j)$	$M_i^{i-t}(0) = 1$	
$M^n(j)$	$M^n(0) = 1$	for $1 - t \leqq n \leqq 0$,
$S_i^n(j)$	$S_i^{i-t}(j) = 1$	for $j \geqq 0$,
$S^n(j)$	$S^n(j) = 1$	for $1 - t \leqq n \leqq 0$ and $j \geqq 0$,
$G_i^n(j)$	$G_i^{i-t}(j) = j + 1$	for $j \geqq 0$,
$G^n(j)$	$G^n(j) = j + 1$	for $1 - t \leqq n \leqq 0$ and $j \geqq 0$.

It is not difficult to show that these t-arrays satisfy the definitions given in (3.2).

Example 3.2. Table 3.2 shows a portion of the t-array $S_i^n(j)$ when $i = 2$ and $t = 4$. In this case, the only nonzero elements of the initialization region are $S_2^2(j) = 1$ for $j \geqq 0$.

TABLE 3.2

$S_2^n(j)$ for $t = 4$

j	$n = -3$	-2	-1	0	1	2	3	4	5	6	7
0	0	1	0	0	0	0	0	0	0	0	0
1	0	1	0	0	1	1	0	0	0	0	0
2	0	1	0	0	1	2	2	2	2	1	0
3	0	1	0	0	1	2	3	5	7	8	7
4	0	1	0	0	1	2	3	6	11	17	23
5	0	1	0	0	1	2	3	6	12	22	37
6	0	1	0	0	1	2	3	6	12	23	43
7	0	1	0	0	1	2	3	6	12	23	44

4. Optimal merging. In this section we will examine some of the properties of the polyphase merge when it is implemented using read-forward tape units. (*Read-forward* tape units can be thought of as queues in which strings are written at the end of the tape and are read from the beginning.) Of particular importance is the close relationship with generalized Fibonacci numbers. These results will be used to construct an optimal polyphase merge algorithm which has a number of desirable characteristics.

From (2.1) it is easily shown that

$$S_t^n = S_t^{n-1} + \cdots + S_t^1 + 1 \quad \text{for } 2 \leqq n \leqq t, \text{ and}$$

$$S_t^n = S_t^{n-1} + \cdots + S_t^{n-t} \quad \text{for } n > t.$$

If we define $F_n = 0$ for $n < 0$, $F_0 = 1$, and $F_n = S_t^n$ for $n \geqq 1$, then, from the above relations, we have

$$F_n = F_{n-1} + \cdots + F_{n-t} \tag{4.1}$$

for $n \geqq 1$. Because of the similarity of (4.1) to the defining recurrence relation for the Fibonacci numbers, we will call these numbers F_n the *t-Fibonacci numbers*.

The t-Fibonacci numbers play a central role in the problem of analyzing the motion of the strings for the read-forward polyphase merge. Indeed, suppose that the strings have been dispersed according to the perfect stage n distribution and that the string positions on each tape are numbered from zero starting at the front of the tape. If we perform the polyphase merge starting with stage n, then the number of times m that a string in position p on one of the tapes will be moved is computed by the following algorithm:

```
ALGORITHM 4.1. Simulate string motion.
Step 1. Let $m = 1$, $k = n - 1$, and $q = p$.
Step 2. If $k = 0$, then terminate.
Step 3. If $q < F_k$, then go to Step 5.
Step 4. Let $q = q - F_k$ and go to Step 6.
Step 5. Let $m = m + 1$.
Step 6. Let $k = k - 1$ and go to Step 2.
```

This algorithm simply follows the motion of the string as the polyphase merge is performed. In particular, $k+1$ is the stage number of the polyphase merge being performed. If $q < F_k = S_t^k = S_1^{k+1}$, then the string will be moved (and m incremented), but its position on the output tape will be the same as its position on the input tape. If $q \geqq F_k$, then the string will not be moved but its position will be changed to $q - F_k$ since F_k strings will have been removed from the tape. Since we are simulating the polyphase merge, we always have $q < F_{k+1} = S_t^{k+1}$ (this may also be shown by induction) so that $q = 0$ when the algorithm terminates.

Let us define the sequence $s_1, s_2, \cdots, s_{n-1}$ as follows: we let $s_j = 1$ if, when performing Algorithm 4.1, we perform Step 4 with $k = j$; otherwise, we let $s_j = 0$. Obviously, the number of times that the string in position p is moved is $n - s_1 - s_2 - \cdots - s_{n-1}$. From the mechanics of the algorithm and the fact that it terminates with $q = 0$, we find that

$$p = \sum_{j=1}^{n-1} s_j F_j.$$

Since a string cannot remain on a tape for t consecutive merges, we see that the sequence $s_1, \cdots, s_{n-1}$ cannot contain more than $t-1$ consecutive ones.

We have shown that p may be represented as a sum of distinct t-Fibonacci numbers in such a way that at most $t-1$ consecutive t-Fibonacci numbers appear in the sum. We will now study some properties of this type of representation.

We define a *t-sequence* to be a sequence $s_1, s_2, \cdots$ of zeros and ones with the properties that only finitely many ones appear and that no t consecutive ones appear. It will sometimes be convenient to assume that $s_m = 0$ for $m \leqq 0$. The *length* $L(s)$ of a t-sequence s is defined to be the largest m for which $s_m = 1$ or zero if $s_m = 0$ for all m. If s and s' are t-sequences, then we say that $s < s'$ if for some m we have $s_m < s'_m$ (i.e., $s_m = 0$ and $s'_m = 1$) and $s_n = s'_n$ for all $n > m$. It is clear that this defines a linear ordering of the set of all t-sequences.

A t-sequence s *represents* a number $F(s)$ in the sense that

$$F(s) = \sum_{n \geqq 1} s_n F_n.$$

We have the following theorem concerning such representations:

THEOREM 4.1. *For each $p \geqq 0$, there exists a unique t-sequence $R(p)$ for which $p = F(R(p))$. Furthermore, if $p < p'$, then $R(p) < R(p')$.*

First we require some lemmas:

LEMMA 4.1. *If s is a t-sequence for which $L(s) < n$, then $F(s) < F_n$.*

Proof. If $L(s) = 0$, then $F(s) = 0 < F_n$ for all $n > 0$. Now suppose that s is a t-sequence of length $m > 0$ and that the result has been proved for all t-sequences of length less than m. Clearly there must be a $k \geqq 0$ with $m - t + 1 \leqq k < m$ for which $s_k = 0$. We form the t-sequence s' by letting $s'_j = s_j$ for $j < k$ and $s'_j = 0$ for $j \geqq k$. If $k = 0$, then $F(s') = 0 < F_k$. If $k > 0$, then $L(s') < k < m$ so that by our induction hypothesis we have $F(s') < F_k$. Consequently, if $m < n$, then we have

$$\begin{aligned} F(s) &= F(s') + \sum_{j>k} s_j F_j < F_k + F_{k+1} + \cdots + F_m \\ &\leqq F_{m-t+1} + \cdots + F_m = F_{m+1} \leqq F_n. \end{aligned} \qquad \square$$

LEMMA 4.2. *If s and s' are t-sequences for which $s < s'$, then $F(s) < F(s')$.*

Proof. Let m be the largest integer for which $s_m < s'_m$. We then have $s_m = 0$ and $s_n = s'_n$ for $n > m$. From Lemma 4.1 it follows that

$$\begin{aligned} F(s) &= \sum_{k \geqq 1} s_k F_k = \sum_{k=1}^{m-1} s_k F_k + \sum_{k>m} s_k F_k \\ &< F_m + \sum_{k>m} s_k F_k \leqq \sum_{k \geqq 1} s'_k F_k = F(s'). \end{aligned} \qquad \square$$

LEMMA 4.3. *There are precisely F_n t-sequences for which $L(s) < n$.*

Proof. We will use induction on n. Clearly the result is true when $n = 1$. If $n > 1$, then we may partition the set of all t-sequences s for which $L(s) < n$ into t classes as follows: for each k with $1 \leqq k \leqq t$, we define the kth class to be the set of all such t-sequences s which have the property that $s_j = 1$ for $n - k < j < n$ (this condition is vacuous when $k = 1$) and $s_{n-k} = 0$. Assuming that the lemma has been proved for all $n' < n$, we will show that for each k that the kth class contains F_{n-k} elements. If $n-k<0$, then we must have $s_0 = 1$ for any s in the kth class and therefore the kth class contains $F_{n-k} = 0$ elements. If $n-k \geqq 0$, then for any t-sequence s in the kth class, we may construct a t-sequence s' by letting $s'_j = s_j$ for $j < n-k$ and $s'_j = 0$ for $j \geqq n-k$. It is easily seen that this construction defines a bijection between the kth class and the set of all t-sequences s' for which $L(s') < n-k$. Since the latter set contains F_{n-k} elements, so does the kth class. Summing over k, we find that there are exactly $F_{n-1} + \cdots + F_{n-t} = F_n$ t-sequences s for which $L(s) < n$. □

Proof of Theorem 4.1. It is clear that the numbers F_n are unbounded. Therefore, if $p > 0$ is given, then we can find an n for which $p < F_n$. By Lemma 4.3, there are F_n t-sequences of length less than n which by Lemma 4.1 are mapped by F into the nonnegative integers less than F_n. By Lemma 4.2, this mapping is injective and therefore, by pigeonholing, is surjective. Consequently, we can find a t-sequence $R(p)$ for which $p = F(R(p))$. Uniqueness and the strict monotony of the mapping R both follow from Lemma 4.2. □

Remarks. Theorem 4.1 is an extension of a well-known theorem of Zeckendorf which concerns the representation of integers by sums of Fibonacci numbers. The extension given here is due to Knuth ([5, Exercise 5.4.2-10]) although our proof is somewhat different. Lynch [6] has generalized this result and has shown how generalized Fibonacci numbers may be used to control dispersion and merging in the standard polyphase sort. There is another extension of Zeckendorf's theorem which contains the others as special cases. Let $r(n)$ be a positive integer-valued function of $n \geqq 1$ which has the property that $r(n) \geqq 2$ for infinitely many values of n. We define the *r-Fibonacci numbers* f_n by $f_n = 0$ for $n < 0$, $f_0 = 1$, and $f_n = f_{n-1} + \cdots + f_{n-r(n)}$ for $n \geqq 1$. Every positive integer is uniquely represented by a sum of r-Fibonacci numbers f_n with distinct subscripts $n \geqq 1$ which has the property that if $f_{m-1}, \cdots, f_{m-r(m)}$ all appear in the sum, then so does f_m. A proof may be constructed along the lines of our proof of Theorem 4.1 although some care is required when $r(n) = 1$. When $r(n) = n$ for all $n \geqq 1$ then the above result implies the existence and uniqueness of representations in the binary number system.

Let $D(p)$ be the number of ones in the t-sequence $R(p)$. In the discussion following Algorithm 4.1 we showed that if a string appears in position p on some tape of the perfect stage n distribution, then the polyphase merge will move the string exactly $n - D(p)$ times. Therefore, it is of some interest to determine those values of p for which $D(p)$ takes a given value.

Let j be a nonnegative integer. We define $E(j)$ to be the smallest nonnegative integer p for which $D(p) = j$. The following theorem and the corollary provide methods of computing $E(j)$:

THEOREM 4.2. $E(0) = 0$. *If* $j > 0$, *then* $E(j) = E(j-1) + F_{j+k}$ *where* $k = \lfloor (j-1)/(t-1) \rfloor$.

Proof. We will prove the theorem together with the fact that $L(R(E(j))) = j+k$ for $j > 0$ by induction on j. Clearly $E(0) = 0$. Now suppose that $j > 0$ and define $s = R(E(j))$, $m = L(s)$, and $p = E(j) - F_m$. Clearly $D(p) = j-1$ so that $p \geqq E(j-1)$. If we let $k = \lfloor (j-1)/(t-1) \rfloor$, then we must have $m \geqq j+k$ for otherwise s would contain t consecutive ones or would have less than j ones. It follows that $E(j) \geqq E(j-1)+F_{j+k}$ and to prove equality, it is sufficient to show that $D(E(j-1)+F_{j+k})=j$. We assume that everything has been proved for $j'<j$. If $k=0$, then we clearly have

$$E(j-1)=F_1+\cdots+F_{j-1}$$

(the sum being zero when $j=1$) and since $j<t$ we have $D(E(j-1)+F_{j+k})=j$. We also observe that $L(s)=j=j+k$. If $k>0$, then let $j'=k(t-1)+1$. Clearly $j' \leqq j$ and we have $k=\lfloor (n-1)/(t-1) \rfloor$ for $j' \leqq n \leqq j$. From our induction hypothesis we obtain

$$E(j-1)+F_{j+k}=E(j'-1)+F_{j'+k}+\cdots+F_{j+k}.$$

However, if we let $k'=\lfloor (j'-2)/(t-1) \rfloor$, then $L(R(E(j'-1)))=j'+k'-1=j'+k-2$. Since $j-j'<t-1$, it follows that the t-sequence $s'=R(E(j'-1))$ remains a t-sequence if we let $s'_n=1$ for $j'+k \leqq n \leqq j+k$. It follows at once that $D(E(j-1)+F_{j+k})=j$ and that $L(s)=j+k$. This completes the proof. □

COROLLARY 4.1. *For $j>0$ and k defined as before we have*

$$E(j)=\sum_{m=kt}^{j+k} F_m-1,$$

the sum having at most t terms.

Proof. The proof is by induction on j. When $j=1$ we have $k=0$ so the above expression is $F_0+F_1-1=1=E(1)$. Now suppose that the corollary has been proved for all $j'<j$, in particular, for $j'=k(t-1)$. Since $\lfloor (n-1)/(t-1) \rfloor = k$ for $j'<n \leqq j$ we have from the theorem,

$$E(j)=E(j')+F_{j'+k+1}+\cdots+F_{j+k}.$$

Applying the corollary with j' and $k'=\lfloor (j'-1)/(t-1) \rfloor = k-1$, we obtain

$$\begin{aligned} E(j') &= \sum_{m=k't}^{j'+k'} F_m-1=\sum_{m=kt-t}^{kt-1} F_m-1 \\ &= F_{kt}-1. \end{aligned}$$

Since $j'+k+1=kt+1$, it follows that

$$E(j)=F_{kt}+\cdots+F_{j+k}-1.$$

Finally, we observe that $j+k-kt=1+(j-1)-k(t-1)<1+(t-1)=t$ so the sum contains at most t terms. □

If $j>1$, then there are infinitely many positive integers p for which $D(p)=j$. We have just shown how to find the smallest such p, so now we will show how to find the others. We will do this by constructing an algorithm which computes, given $p>0$, the smallest $p'>p$ for which $D(p')=D(p)$.

Let $s=R(p)$ and $s'=R(p')$. We already know that $s<s'$ if and only if we can find an m for which $s_m=0$, $s'_m=1$, and $s'_k=s_k$ for $k>m$. Consequently, to find the smallest $p'>p$ for which $D(p')=D(p)$, we must first find a suitable value of m. Clearly the smaller the value of m that is chosen, the smaller the value of p'. There are three conditions that m must satisfy: First there is the condition $s_m=0$ which was given above. Second, we must have $s_k=1$ for some $k<m$ for otherwise we

would have $D(p')>D(p)$. Third, we cannot have $s_{m+1}=\cdots=s_{m+t-1}=1$ for otherwise any sequence s' with $s'_m=1$ and $s'_k=s_k$ for $k>m$ will not be a t-sequence.

Therefore, let us choose m to be the smallest integer for which $s_m=0$, $s_{m-1}=1$, and $s_{m+1}+\cdots+s_{m+t-1}<t-1$. This choice can always be made since $m=L(s)+1$ satisfies the requirements. If we define p' by

$$p'=E(s_1+\cdots+s_{m-2})+F_m+\sum_{k>m} s_kF_k,$$

then it is easily verified that $p'>p$, that $D(p')=D(p)$ and that it is the smallest integer to have these properties.

In order to use the formula above, it is necessary to know the representation $R(p)$ of p. The following algorithm computes p' by combining the conversion of p to $R(p)$ (using a technique similar to Algorithm 4.1) and the search for m. The algorithm is easily implemented on digital computers since it is fully arithmetic and does not involve t-sequences.

ALGORITHM 4.2. Find the smallest $p'>p$ for which $D(p')=D(p)$.

Step 1. Let $q=p$ and $k=0$ and choose some m for which $p<F_m$.

Step 2. If $F_m\leqq q$, then go to Step 4.

Step 3. Let $m=m-1$. If $m=0$, then go to Step 10; otherwise go to Step 2.

Step 4. Let $q'=q$, $m'=m$, and $k'=k$.

Step 5. If $m<t$, then go to Step 7.

Step 6. If $q<F_{m+1}-F_{m-t+1}$, then go to Step 7; otherwise, let $q=q-(F_{m+1}-F_{m-t+1})$, $m=m-t$, and $k=k+t-1$ and go to Step 8.

Step 7. Let $q=q-F_m$, $m=m-1$, and $k=k+1$.

Step 8. If $m=0$, then go to Step 10.

Step 9. If $F_m\leqq q$, then go to Step 5; otherwise, go to Step 3.

Step 10. Terminate with $p'=p-q'+F_{m'+1}+E(k-k'-1)$.

To understand this algorithm, let $s=R(p)$. If $F_m\leqq q$ in Step 2, then $s_m=1$ and the values of q, m, and k are saved. The check that $q\geqq F_{m+1}-F_{m-t+1}=F_m+\cdots+F_{m-t+2}$ determines whether or not $s_m=\cdots=s_{m-t+2}=1$ and $s_{m-t+1}=0$. Steps 6 and 7 decrement m in such a way as to bypass ineligible values of m, that is, those for which $s_{m+1}=1$ or $s_{m+1}=0$ and $s_{m+2}=\cdots=s_{m+t}=1$. The variable k contains the number of nonzero values of s_m which have been encountered. At completion, the last values of q, m, and k saved by Step 4 enable us to compute p'.

Example 4.1. First we list some values of F_n and $E(n)$ for the case $t=4$:

n	F_n	$E(n)$	n	F_n	$E(n)$
1	1	1	9	188	1339
2	2	3	10	361	3921
3	4	7	11	693	8897
4	8	22	12	1340	18488
5	15	51	13	2582	54126
6	29	97	14	4976	122820
7	46	285	15	9591	255232
8	98	646	16	18489	747209

If we let $p = 3913$ and let $s = R(p)$, then it is easily shown that

$$s = \{0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, \cdots\},$$

so the representation of p' has the form

$$s' = \{1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, \cdots\}$$

and it follows that $p' = 3917$.

We are now in a position to examine the problem of optimizing the polyphase merge for an arbitrary initial distribution. Suppose that the dispersion routine writes $x_1, \cdots, x_t$ strings to units $1, \cdots, t$, respectively, and that the choice is made to perform the polyphase merge starting with stage n. The only requirement on n is that $x_i \leqq S_i^n$ for each i. If this requirement is met, then it is only necessary to include $S_i^n - x_i$ dummy strings on each tape i in order to obtain the perfect stage n distribution. We have already observed that the number of times that a string is moved depends upon its tape position. Therefore, the manner of placement of the dummy strings has a direct influence on the volume of information moved.

It is quite obvious how to arrange the dispersed strings and the dummy strings so as to minimize the volume of information moved. On each unit i, we place $M_i^n(1)$ of the dispersed strings in the $M_i^n(1)$ string positions which will be moved once, $M_i^n(2)$ strings into the positions which will be moved twice, and so on, until we exhaust the x_i dispersed strings; we then place dummy strings in the remaining $S_i^n - x_i$ string positions. In this way we insure that the dummy strings are in the positions which will be moved the most.

One practical difficulty with the above approach is the problem of placing the dummy strings if the dispersed strings are already on the tapes. With read-forward tape units it is not permissible to write randomly on a tape. For this reason, we will transform the above approach into a practical algorithm in which dummy strings do not explicitly appear.

If $S_i^n(j-1) < x_i < S_i^n(j)$, then, with the above scheme, there will be some j movement string positions which contain dispersed strings and others which contain dummy strings. We have not said how they are to be arranged. We propose placing all of the j movement dispersed strings in front of all of the j movement dummy strings on each tape unit. This method has the important property that the pattern is preserved as the polyphase merge is performed. It is not difficult to see that any time during the operation of the merge, any k movement strings of nonzero length will be in front of any k movement dummy strings on the same tape.

Another important consequence of this choice is that we are able to calculate the positions of the j movement dispersed strings. Since these positions p have the property that $j = n - D(p)$, see that the first of these positions is $E(n-j)$ and that the remaining positions are calculated by repeated application of Algorithm 4.2. Since the pattern is preserved, the same observation holds throughout the polyphase merge.

The algorithm which we will present is controlled by the two arrays $C[i, j]$ and $P[j] (0 \leqq i \leqq t, 1 \leqq j \leqq n)$. $C[i, j]$ will contain the number of strings on tape i which will be moved j times and $P[j]$ contains the next j movement position on the input tapes. It is also convenient to have arrays for the numbers F_m and $E(m)$, but we will not mention these explicitly.

The inputs to the algorithm are the numbers $x_1, \cdots, x_t$ of dispersed strings on tape units $1, \cdots, t$ and the starting stage number n of the polyphase merge to be performed. (The next three sections of this paper are devoted to the proper choice of these numbers.) In order to facilitate implementation, we will explicitly mention the tape rewind operations required.

ALGORITHM 4.3. Optimal read-forward polyphase merge.

Step 1. [Initialization.] Let $C[i,j]=M_i^n(j)$ for $1 \leqq j \leqq n$ and $1 \leqq i \leqq t$. Let $C[0,j]=0$ for $1 \leqq j \leqq n$. Let $m=n$ and $u=0$. Rewind all of the tapes.

Step 2. [Initialize C.] For each $i=1, \cdots, t$ find the smallest j for which $x_i \leqq C[i,1]+\cdots+C[i,j]$; let $C[i,j]=x_i-C[i,1]\cdot\cdots-C[i,j-1]$ and let $C[i,k]=0$ for $j<k \leqq n$.

Step 3. [Test for termination.] If $m>0$, then go to Step 4. Otherwise, the sort is finished. Rewind all of the tapes. The sorted records are on tape u'.

Step 4. [Initialize for stage m.] For $j=1, \cdots, m$ let $P[j]=E(m-j)$ if $C[i,j]>0$ fr some i; otherwise, let $P[j]=F_{m-1}$.

Step 5. [Test for the end of a merge.] Find the value of j which minimizes $P[j]$ $(1 \leqq j \leqq m)$. If $P[j] \geqq F_{m-1}$, then go to Step 9.

Step 6. [Merge some strings.] Merge one string from each unit $i \neq u$ for which $C[i,j]>0$ and write the resulting string to unit u.

Step 7. [Update C.] If $m>1$, then increment $C[u,j-1]$ by one. For each $i \neq u$ for which $C[i,j]>0$, decrement $C[i,j]$ by one. If each of these decrements results in a value of zero, then let $P[j]=F_{m-1}$ and go to Step 5.

Step 8. [Update Q.] Using Algorithm 4.2, find the smallest $p>P[j]$ for which $D(p)=D(P[j])$. Let $P[j]=p$ and go to Step 5.

Step 9. [End of a merge.] Let $m=m-1$, $u'=u$, and $u=u+1 \bmod T$. Rewind tapes u and u' and go to Step 3.

In view of the discussion, this algorithm is reasonably straightforward. However, we will comment on a few points. The computations required in Step 1 can be performed without any additional storage by careful use of the recurrence relations (3.1). Our use of F_{m-1} in Steps 4, 5, and 7 is accounted for by the fact that $F_{m-1}=S_t^{m-1}=S_1^m$ which is the number of strings produced by the stage m merge; consequently F_{m-1} is the first position which will not be used for this merge.

Although the computations required by the algorithm are formidable, they do not really require much time. The bulk of the computation is performed in Steps 5, 7, and 8 which are performed once for each string that is output. Since a unit string will represent a large fraction of the storage utilized by the sort, it is clear the time required will be insignificant when compared with the time required for merging.

The storage requirements are not much larger than for other polyphase merge algorithms. The only extra storage which is not required by other algorithms is the storage for the arrays C and P and, possibly, the arrays containing the numbers $E(m)$ and F_m for a suitable range of m. We remark that the additional storage required for these arrays when merging 100000 strings, using ten tapes and the dispersion algorithm we will describe, should be less than four hundred locations.

Remarks. Shell [8] has described an optimum polyphase sort which is somewhat different from ours. He describes a method of generating the $D(0)$, $D(1)$, $D(2), \cdots$ directly and uses an array based on this sequence to control the placement of the strings and the assumed placement of the dummy strings. Unfortunately, this array becomes prohibitively large for large applications. An account of Shell's work also appears in [5, § 5.4.2].

5. The volume function. Let us suppose that we have $x \leqq S^n$ unit strings which we wish to merge with the stage n polyphase merge. Obviously, in order to minimize the volume, we should place the unit strings into the positions which will be moved the least and the dummy strings into the positions which will be moved the most. Thus, if $S^n(j) \leqq x \leqq S^n(j+1)$, then unit strings should be placed in all of the $S^n(j)$ positions which will be moved j or fewer times and in $x - S^n(j)$ of the $j+1$ movement positions. When this is done, the volume of information which will be moved by the merge is found to be

$$\sum_{k=1}^{j} kM^n(k) + (j+1)(x - S^n(j)).$$

We will call the value of this expression the *volume function* and denote it by $V^n(x)$. The expression may be simplified by observing that

$$\begin{aligned}(j+1)S^n(j) - \sum_{k=1}^{j} kM^n(k) &= \sum_{k=1}^{j} (j-k+1)M^n(k)\\ &= \sum_{k=1}^{j}\sum_{i=k}^{j} M^n(k) = \sum_{i=1}^{j}\sum_{k=1}^{i} M^n(k)\\ &= \sum_{i=1}^{j} S^n(i) = G^n(j).\end{aligned}$$

We may now write

$$V^n(x) = (j+1)x - G^n(j), \tag{5.1}$$

where $S^n(j) \leqq x \leqq S^n(j+1)$.

In § 4, we looked at the similar problem of optimizing the stage n polyphase merge when it is known that tapes $1, \cdots, t$ contain $x_1, \cdots, x_t$ dispersed strings, respectively. By similar reasoning, the volume of information moved in this case is

$$V_1^n(x_1) + \cdots + V_t^n(x_t),$$

where each $V_i^n(x_i)$ represents the contribution of tape i to the volume. This contribution is given by

$$V_i^n(x_i) = (j_i+1)x_i - G_i^n(j_i), \tag{5.2}$$

where j_i is chosen to satisfy $S_i^n(j_i) \leqq x_i \leqq S_i^n(j_i+1)$.

Obviously we must have

$$V^n(x_1 + \cdots + x_t) \leqq V_1^n(x_1) + \cdots + V_t^n(x_t).$$

We are interested in those distributions $x_1, \cdots, x_t$ for which we have equality. Such a distribution is said to be *optimal for stage n*.

THEOREM 5.1. *A distribution $x_1, \cdots, x_t$ is optimal for stage n if and only if we can find a j such that $S_i^n(j) \leqq x_i \leqq S_i^n(j+1)$ for each i.*

Proof. If the condition is satisfied, then optimality for stage n follows at once from formulas (5.1) and (5.2) and the fact that $G^n(j) = G_1^n(j) + \cdots + G_t^n(j)$.

Conversely, suppose that $x_1, \cdots, x_t$ does not satisfy the condition. We can then find a j and two indices a and b such that $x_a < S_a^n(j)$ and $x_b > S_b^n(j)$. If we define the distribution $x'_1, \cdots, x'_t$ by $x'_a = x_a + 1$, $x'_b = x_b - 1$, and $x'_i = x_i$ for $i \neq a, b$, then it is clear that

$$V_a^n(x'_a) - V_a^n(x_a) \leqq j \quad \text{and}$$

$$V_b^n(x_b) - V_b^n(x'_b) \leqq j + 1.$$

It follows that

$$V_1^n(x'_1) + \cdots + V_t^n(x'_t) < V_1^n(x_1) + \cdots + V_t^n(x_t)$$

and therefore $x_1, \cdots, x_t$ cannot be optimal for stage n. □

Example 5.1. We let $t = 4$ as in our other examples and $x = 500$. From the table in Example 2.1, we see that the smallest value of n for which $x \leqq S^n$ is 9. Let us evaluate $V^n(x)$ for this value of n. Since $S^n(5) = 338 < x < 534 = S^n(6)$, we may apply formula (5.1) with $j = 5$ to obtain

$$V^n(x) = (j+1)x - G^n(j) = 6 \cdot 500 - 478 = 2522.$$

This volume is the best possible volume obtainable with the stage 9 merge no matter how the strings are dispersed. If we let $n = 10$, then a similar calculation shows that $V^n(x) = 2448$ which illustrates how the choice of a larger stage number than the minimum may improve the performance of the polyphase sort. We will discuss this subject in § 6.

We will conclude this section with two theorems concerning the volume function which will be required later.

THEOREM 5.2. *If $x \leqq S^n$, then $V^{n+1}(x) - V^n(x) \leqq x$.*

Proof. We may assume that $x > 0$. Let j and k be the unique integers for which

$$S^n(j) < x \leqq S^n(j+1) \quad \text{and} \quad S^{n+1}(k) < x \leqq S^{n+1}(k+1).$$

From the recurrence relation for t-arrays, we see that $S^n(j+1) \leqq S^{n+1}(j+2)$, so that

$$S^{n+1}(k) < x \leqq S^n(j+1) \leqq S^{n+1}(j+2)$$

which implies that $k \leqq j + 1$. From the recurrence relation, we also have $G^n(k-1) \leqq G^{n+1}(k)$. We may now write

$$\begin{aligned} V^{n+1}(x) - V^n(x) &= (k+1)x - G^{n+1}(k) - (j+1)x + G^n(j) \\ &= (k-j)x + G^n(j) - G^{n+1}(k) \\ &\leqq (k-j)x + G^n(j) - G^n(k-1) \\ &\leqq (k-j)x + (j-k+1)S^n(j) \\ &\leqq (k-j)x + (j-k+1)x = x. \end{aligned}$$

□

THEOREM 5.3. *Suppose that* $0 \leqq x_1 \leqq \cdots \leqq x_t$ *and that* $x_i \leqq S_i^n$ *for each i. If* $x_1', \cdots, x_t'$ *is a permutation of* $x_1, \cdots, x_t$ *which has the property that* $x_i' \leqq S_i^n$ *for each i, then we have*

$$V_1^n(x_1) + \cdots + V_t^n(x_t) \leqq V_1^n(x_1') + \cdots + V_t^n(x_t').$$

Proof. First we will prove the result for a simple interchange. Suppose that $1 \leqq a < b \leqq t$ and that $x_a \leqq S_b^n$, $x_b \leqq S_a^n$, and $0 \leqq x_a < x_b$. If $x_a \leqq y < x_b$, then let j and j' be the unique integers for which

$$S_a^n(j) \leqq y < S_a^n(j+1) \quad \text{and} \quad S_b^n(j') \leqq y < S_b^n(j'+1).$$

Since $S_a^n(k) \leqq S_b^n(k)$ for all k, it is clear that $j \geqq j'$ and therefore

$$V_b^n(y+1) - V_b^n(y) = j' + 1 \leqq j + 1 = V_a^n(y+1) - V_a^n(y).$$

By summing over y, we obtain

$$V_b^n(x_b) - V_b^n(x_a) \leqq V_a^n(x_b) - V_a^n(x_a)$$

which may be rewritten as

$$V_a^n(x_a) + V_b^n(x_b) \leqq V_a^n(x_b) + V_b^n(x_a).$$

The general result is proved by permuting the numbers $x_1', \cdots, x_t'$ into $x_1, \cdots, x_t$ by a series of interchanges which successively place the proper values into positions $1, \cdots, t$ and by applying the above result at each step. It is clear that we only change the numbers y_a and y_b in positions $a < b$ when $y_b < y_a$. Also, since $y_a \leqq S_a^n \leqq S_b^n$, we never place a number which exceeds S_i^n into any position i. □

6. Optimal dispersion. In much of the literature on polyphase sorting, it is assumed that the best starting stage number when merging x strings is the smallest n for which $x \leqq S^n$. This method generally gives nice looking results when the usual polyphase merge algorithms are used. However, when an algorithm such as Algorithm 4.3 or the optimum polyphase sort of Shell [8] is employed, it is found that better results may be obtained by choosing large values of n. In this section we will investigate the problem of finding the value of n which minimizes $V^n(x)$.

A good starting point is the following lemma on t-arrays.

LEMMA 6.1. *Let A denote one of the t-arrays M, S, or G. Let j and d be positive integers and let* $n(j, d)$ *denote the smallest integer* $n \geqq 1$ *for which* $A^n(j) > A^{n+d}(j)$. *Then the following are true*:

(a) *If* $n' \geqq n(j, d)$, *then* $A^{n'}(j) \geqq A^{n'+d}(j)$.

(b) *If* $j' > j$, *then* $n(j', d) \geqq n(j, d)$.

Proof. It is easily verified that

$$A^{1-t}(0) = \cdots = A^0(0) > 0 = A^1(0) = A^2(0) = \cdots \tag{6.1}$$

and that for $j \geqq 1$,

$$0 \leqq A^{1-t}(j) = \cdots = A^0(j) \leqq A^1(j). \tag{6.2}$$

It is clear that $n(j, d)$ always exists since $A^n(j)$ is zero for n sufficiently large. From (6.1) it follows that

$$A^1(1) > A^2(1) \geqq A^3(1) \geqq A^4(1) \geqq \cdots$$

so that $n(1, 1) = 1$ and (a) is true for $n(1, 1)$.

We will now show that if (a) is true for $n(j, 1)$, then it is true for $n(j, d)$ for $j > 1$ and for $n(j+1, 1)$. Let $d > 1$ be given and let $m \geqq 1 - t$ be the smallest such integer for which $A^m(j) > A^{m+d}(j)$. It is clear that $m + d > n(j, 1)$. We will show that $A^n(j) \geqq A^{n+d}(j)$ for $n > m$. This is certainly true if $n \geqq n(j, 1)$. Also, if $m \leqq n < n(j, 1)$, then we have

$$A^n(j) \geqq A^m(j) > A^{m+d}(j) \geqq A^{n+d}(j).$$

Since $n(j, d) \geqq m$, we see that (a) is true for $n(j, d)$. From the recurrence relation for t-arrays, we have

$$A^{n+1}(j+1) - A^n(j+1) = A^n(j) - A^{n-t}(j).$$

Consequently, if we let $d = t$ in the above argument, we see that we may choose $n(j+1, 1) = m + t$ and that (a) is true for this choice. The validitv of (a) now follows by induction.

To prove (b), let $j \geqq 1$ and let $n = n(j+1, d)$. From the recurrence relation for t-arrays, we have

$$0 > A^{n+d}(j+1) - A^n(j+1) = \sum_{k=1}^{t} (A^{n+d-k}(j) - A^{n-k}(j))$$

so that $A^{n-k}(j) > A^{n+a-k}(j)$ for some k with $1 \leqq k \leqq t$. If $n - k \geqq 1$, then $n - k \geqq n(j, d)$ so that $n > n(j, d)$. If $n - k \leqq 0$, then we must have $n + d - k > n(j, 1)$ so that

$$A^1(j) \geqq A^{n-k}(j) > A^{n+d-k}(j) \geqq A^{1+d}(j)$$

and therefore $n(j, d) = 1 \leqq n$. We have thus shown that $n(j+1, d) \geqq n(j, d)$ and (b) follows. □

The lemma is particularly useful in the following form:

COROLLARY 6.1. *Let A denote one of the t-arrays M, S, or G. Then the following are true*:

(a) *If $A^n(j) < A^{n'}(j)$ for some $1 \leqq n < n'$ and $i \leqq 1$, then $A^n(j') \leqq A^{n'}(j')$ for all $j' \geqq j$.*

(b) *If $A^n(j) > A^{n'}(j)$ for some $1 \leqq n < n'$ and $j \geqq 1$, then $A^n(j') \geqq A^{n'}(j')$ for all j' with $1 \leqq j' \leqq j$.*

Proof. To prove (a) let $d = n' - n$. Certainly $n < n(j, d)$ so it follows that $n < n(j', d)$ for all $j' \geqq j$, and the result follows from the definition of $n(j', d)$. This also proves (b) since (b) is the contrapositive of (a). □

THEOREM 6.1. *If $n < n'$ and $V^n(x) > V^{n'}(x)$ for some $x \leqq S^n$, then there exists a $j < n$ for which $G^n(j) < G^n(j)$. Furthermore, if $x < y \leqq S^n$, then $V^n(y) > V^{n'}(y)$.*

Proof. Clearly $x > 0$. Let j and k be the unique integers for which

$$S^n(j) < x \leqq S^n(j+1) \quad \text{and} \quad S^{n'}(k) < x \leqq S^{n'}(k+1).$$

We observe that $j<n$. By assumption

$$(j+1)x-G^n(j)=V^n(x)>V^{n'}(x)=(k+1)x-G^{n'}(k)$$

which reduces to

$$G^n(j)<G^{n'}(k)+(j-k)x.$$

In order to prove that $G^n(j)<G^{n'}(j)$ we will show that $(j-k)x\leqq G^n(j)-G^{n'}(k)$. If $j=k$, then there is nothing to prove. If $j>k$, then we have

$$(j-k)x\leqq\sum_{i=k+1}^{j} S^{n'}(i)=G^{n'}(j)-G^{n'}(k).$$

Similarly, if $j<k$, then

$$(j-k)x=-(k-j)x\leqq-\sum_{i=j+1}^{k} S^{n'}(i)=G^{n'}(j)-G^{n'}(k).$$

Now suppose that there is a smallest y with $x<y\leqq S^n$ for which $V^n(y)\leqq V^{n'}(y)$. Let j' and k' be the unique integers for which

$$S^n(j')<y\leqq S^n(j'+1)\quad\text{and}\quad S^{n'}(k')<y\leqq S^{n'}(k'+1).$$

Since $V^n(y-1)>V^{n'}(y-1)$, we find that

$$j'+1=V^n(y)-V^n(y-1)<V^{n'}(y)-V^{n'}(y-1)=k'+1$$

from which it follows that $j'<k'$. On the other hand, since $G^n(j)<G^n(j)$, we can find an $m\leqq j$ for which $S^n(m)<S^{n'}(m)$. By (a) of Corollary 6.1, we see that $S^n(m')\leqq S^{n'}(m')$ for all $m'\geqq m$. Since $j'+1>j\geqq m$, it follows that

$$y\leqq S^n(j'+1)\leqq S^{n'}(j'+1)\leqq S^{n'}(k')<y$$

which is impossible. This completes the proof. □

COROLLARY 6.2. *Let $N(x)$ be the smallest integer n which minimizes $V^n(x)$. Then $N(x)$ is an increasing function of x.*

Proof. Suppose that $N(x)>N(x+1)$ for some x and let $a=N(x)$ and $b=N(x+1)$. Since $b<a$, we must have $V^a(x)<V^b(x)$. Also, since $x+1\leqq S^b\leqq S^a$, it follows from Theorem 6.1 that $V^a(x+1)<V^b(x+1)$ which implies that $N(x+1)\neq b$. □

Remarks. Most of these results were first proved by Knuth [5, Exercise 5.4.2-14], however, our proof of Theorem 6.1 is somewhat different. Shell [8] has observed Corollary 6.2 empirically.

In the remainder of this section, we will solve the problem of determining the range of values of x for which $N(x)$ takes a given value. We will begin by examining some of the more subtle properties of the numbers $G^n(j)$.

LEMMA 6.2. *For each $t\geqq 2$, there exists a number n_t with the property that $G^n(j)<G^{n+1}(j)$ for some j with $1\leqq j<n$, if and only if $n\geqq n_t$. In particular $n_2=8$, $n_3=5$, $n_4=4$, and $n_t=3$ for $t\geqq 5$.*

Proof. If $G^n(j)<G^{n+1}(j)$ for some j with $1\leqq j<n$, then we can find a $j'\leqq j$ for which $S^n(j')<S^{n+1}(j')$. By (a) of Corollary 6.1 we find that $S^n(k)\leqq S^{n+1}(k)$ for $k\geqq j\geqq j'$ and consequently $G^n(n-1)<G^{n+1}(n-1)$. It follows at once that such a

j exists if and only if $G^n(n-1)<G^{n+1}(n-1)$. Furthermore, if this inequality holds for n, it holds for $n+1$ since, by (a) of Lemma 6.1, we have $G^{n-k}(n-1)\leqq G^{n-k+1}(n-1)$ for $k=1,\cdots,t$ and it follows from the recurrence relation for t-arrays that

$$G^{n+2}(n)-G^{n+1}(n)=G^{n+1}(n-1)-G^{n-t+1}(n-1)>0.$$

The following table will serve to verify the values given for n_t:

t	$G^{n_t-1}(n_t-2)$	$G^{n_t}(n_t-2)$	$G^{n_t}(n_t-1)$	$G^{n_t+1}(n_t-1)$
2	58	56	109	114
3	20	20	48	56
4	11	11	32	40
$\geqq 5$	$t-1$	$t-2$	$4t-5$	$5t-9$

LEMMA 6.3. *For each $n\geqq n_t$, let j_n denote the smallest integer j for which $G^n(j)<G^{n+1}(j)$. We then have*

$$j_n\leqq j_{n+1}\leqq j_n+1\leqq j_{n+t}.$$

Proof. First we will show that $j_n\leqq j_{n+1}$. Assume that for some $k<j_n$ we have $G^{n+1}(k)<G^{n+2}(k)$. We may write

$$\begin{aligned}G^{n+1}(k)-G^n(k)&=G^n(k-1)-G^{n-t}(k-1)\\&=(G^{n+1}(k-1)-G^{n-t+1}(k-1))\\&\quad+(G^n(k-1)-G^{n+1}(k-1))\\&\quad+(G^{n-t+1}(k-j)-G^{n-t}(k-1)).\end{aligned}$$

The first parenthesized term is equal to $G^{n+2}(k)-G^{n+1}(k)$ and is therefore positive. The second term is nonnegative since $k<j_n$. Since $G^{n+1}(k)<G^{n+2}(k)$ it follows from the recurrence relation for t-arrays that $G^{n+1-m}(k-1)<G^{n+2-m}(k-1)$ for some m with $1\leqq m\leqq t$. From (a) of Lemma 6.1 it follows that $G^{n-t}(k-1)\leqq G^{n-t+1}(k-1)$ so the last parenthesized term is nonnegative. We have therefore shown that $G^n(k)<G^{n+1}(k)$ which contradicts the minimality of j_n.

Since $G^n(j_n)<G^{n+1}(j_n)$, we may show as in the proof of Lemma 6.2 that $G^{n+1}(j_n+1)<G^{n+2}(j_n+1)$ and therefore $j_{n+1}\leqq j_n+1$. Finally, since $G^{n+t}(j_{n+t})<G^{n+t+1}(j_{n+t})$, it follows that $G^{n+t-k}(j_{n+t}-1)<G^{n+t+1-k}(j_{n+t}-1)$ for some k with $1\leqq k\leqq t$. Consequently, $j_n\leqq j_{n+t-k}\leqq j_{n+t}-1$. This completes the proof. □

LEMMA 6.4. *Define the numbers N_t by $N_2=19$, $N_3=6$, and $N_t=n_t$ for $t\geqq 4$. If $n\geqq N_t$ and $j\geqq 0$, then*

$$2G^n(j)\leqq G^n(j+1)+G^{n+1}(j-1).$$

Proof. We will show that the above inequality holds for all but finitely many values of $n\geqq 1$ and $j\geqq 0$. The condition on n is sufficient to exclude these exceptions. We define the t-array D by

$$D^n(j)=G^n(j+1)+G^{n+1}(j-1)-2G^n(j).$$

It is not difficult to verify that the nonzero elements of the initialization region for

D are

$$D^0(j) = (t-1)j - t \quad \text{for } j \geqq 1 \quad \text{and}$$
$$D^n(-1) = 1 \qquad \text{for } 1-t \leqq n \leqq 0.$$

We observe that $D^0(1) = -1$ is the only negative element for the initialization region. Tables 6.1(a), 6.1(b), and 6.1(c) each display a portion of the t-array D for

TABLE 6.1(a)
Proof of Lemma 6.4 ($t \geqq 4$)

$n =$	0	1	2	3
$j = 0$	0	t		
1	-1	0	t	
2	$t-2$	-1	-1	$t-1$

TABLE 6.1(b)
Proof of Lemma 6.4 ($t = 3$)

$n =$	0	1	2	3	4	5	6
$j = 0$	0	3	2				
1	-1	0	3	5			
2	1	-1	-1	2	8		
3	3	1	0	-1	0	9	
4				4	0	-1	8

$t \geqq 4$, $t = 3$, and $t = 2$, respectively. By inspecting these tables, it is clear that there are no negative values of $D^n(j)$ with $n \geqq 0$ other than those displayed. Since the negative entries only appear in the columns for which $n < N_t$, if follows that $D^n(j) \geqq 0$ when $n \geqq N_t$. □

THEOREM 6.2. *If $n \geqq N_t$ and if we define*

$$c_n = G^n(j_n) - G^{n+1}(j_n - 1),$$

then the following are true:

(a) $S^n(j_n) \leqq c_n \leqq S^n(j_n + 1)$,

(b) $S^{n+1}(j_n - 1) \leqq c_n < S^{n+1}(j_n)$,

(c) $V^n(c_n) = V^{n+1}(c_n)$,

(d) $V^n(c_n + 1) > V^{n+1}(c_n + 1)$ *if* $c_n < S^n$,

(e) $c_n < c_{n+1}$.

Proof. From the definition of j_n we know that $G^n(j_n - 1) \geqq G^{n+1}(j_n - 1)$. We therefore have

$$S^n(j_n) = G^n(j_n) - G^n(j_n - 1) \leqq G^n(j_n) - G^{n+1}(j_n - 1)$$
$$= c_n < G^{n+1}(j_n) - G^{n+1}(j_n - 1) = S^{n+1}(j_n).$$

From Lemma 6.4,

$$c_n = G^n(j_n) - G^{n+1}(j_n - 1) \leqq G^n(j_n + 1) - G^n(j_n) = S^n(j_n + 1).$$

TABLE 6.1(c)

Proof of Lemma 6.4 ($t = 2$

$n =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$j = 0$	0	2	1	0	0	0														
1	−1	0	2	3	1	0	0													
2	0	−1	−1	2	5	4	1	0												
3	1	0	−1	−2	1	7	9	5	1											
4	2	1	1	−1	−3	−1	8	16	14	6										
5	3	2	3	2	0	−4	−4	7	24	30	20									
6		3	5	5	5	2	−4	−8	3	31	54	50								
7				8	10	10	7	−2	−12	−5	34	85	104							
8						18	20	17	5	−14	−17	29	119	189						
9								38	37	22	−9	−31	12	148	308					
10										75	59	13	−40	−19	160	456				
11												134	72	−27	−59	141	616			
12														206	45	−86	82	757		
13																251	−41	−4	839	
14																		210	−45	835

Also from Lemma 6.4,

$$S^{n+1}(j_n-1)=G^{n+1}(j_n-1)-G^{n+1}(j_n-2)\leqq G^n(j_n)-G^{n+1}(j_n-1)=c_n.$$

This completes the proof of (a) and (b).

From (a) and (b) we have

$$\begin{aligned} V^n(c_n)&=(j_n+1)c_n-G^n(j_n)\\ &=j_nG^n(j_n)-(j_n+1)G^{n+1}(j_n-1)\\ &=j_nc_n-G^{n+1}(j_n-1)=V^{n+1}(c_n)\end{aligned}$$

which is (c). To prove (d) we first observe that from (a) and (b) we have $V^{n+1}(c_n+1)-V^{n+1}(c_n)=j_n$ and $V^n(c_n+1)-V^n(c_n)\geqq j_n+1$ if $c_n<S^n$. From (c) it follows that

$$V^n(c_n+1)-V^{n+1}(c_n+1)\geqq 1+V^n(c_n)-V^{n+1}(c_n)=1.$$

By Lemma 6.3 we have $j_n\leqq j_{n+1}$ so by (a) and (b),

$$c_n<S^{n+1}(j_n)\leqq S^{n+1}(j_{n+1})\leqq c_{n+1}$$

which is (e). This completes the proof. □

COROLLARY 6.3. *$c_n\to\infty$ as $n\to\infty$.*

Proof. This follows from (e) and the fact that c_n is an integer. □

For each $t\geqq 2$, we define the sequence $L_1, L_2, \cdots$ as follows: If $t\geqq 3$, then we let $L_n=S^n$ for $n<N_t$ and $L_n=c_n$ for $n\geqq N_t$. If $t=2$, then we let $L_n=S^n$ for $n\leqq 15$, $L_{16}=2573$, $L_{17}=3954$, $L_{18}=6527$, and $L_n=c_n$ for $n\geqq N_2=19$.

THEOREM 6.3. *The sequence $L_1, L_2, \cdots$ is strictly increasing and has the property that $V^{n+1}(x)\geqq V^n(x)$ if and only if $x\leqq L_n$.*

Proof. First we will show that the sequence is strictly increasing. We already know that $S^n<S^{n+1}$ for all $n\geqq 1$ and that $c_n<c_{n+1}$ for all $n\geqq N_t$. These observations leave us with only a few special cases to consider.

When $t\geqq 3$, we must show that when $n=N_t-1$, we have $S^n=L_n<L_{n+1}=c_{n+1}$. When $t\geqq 5$ we may show from the appropriate t-arrays that $S^2=2t-1$ and $c_3=3t-2$ so that $L_n<L_{n+1}$ since $N_t=3$. When $t=4$, we have $N_t=4$ and $L_3=S^3=13<22=c_4=L_4$. For $t=3$, we have $N_t=6$ and $L_5=S^5=31<32=c_6=L_6$. For the remaining special case $t=2$, we have $L_{15}=S^{15}=1597<2573=L_{16}$, $L_{16}<L_{17}<L_{18}$, and $L_{18}=6527<10488=c_{19}=L_{19}$.

To prove the second part of the theorem, it is sufficient, in view of Theorem 6.1, to show that $V^{n+1}(L_n)\geqq V^n(L_n)$ for all $n\geqq 1$ and that $V^{n+1}(L_n+1)<V^n(L_n+1)$ whenever $L_n<S^n$. If $n<n_t$, then $G^n(j)\geqq G^{n+1}(j)$ for all j with $0\leqq j<n$, so by Theorem 6.1, we have $V^{n+1}(L_n)\geqq V^n(L_n)$. We also note that $L_n=S^n$ for $n<n_t$. When $n\geqq N_t$, then everything follows from Theorem 6.2. Since $n_t=N_t$ for $t\geqq 4$, this proves the result for $t\geqq 4$. To extend the result to the case $t=3$, we observe that in this case we have $L_5=S^5$ and $V^5(L_5)=107<108=V^6(L_5)$.

When $t=2$, there are a number of special cases to consider. First we note that $L_n=S^n$ for $8\leqq n\leqq 15$. By direct computation, we may verify that

$$V^8(L_8)=331<343=V^9(L_8),$$

$$V^9(L_9)=600<614=V^{10}(L_9),$$

$$\begin{aligned}
V^{10}(L_{10}) &= 1075 < 1092 = V^{11}(L_{10}),\\
V^{11}(L_{11}) &= 1908 < 1935 = V^{12}(L_{11}),\\
V^{12}(L_{12}) &= 3360 < 3396 = V^{13}(L_{12}),\\
V^{13}(L_{13}) &= 5878 < 5901 = V^{14}(L_{13}),\\
V^{14}(L_{14}) &= 10225 < 10240 = V^{15}(L_{14}),\\
V^{15}(L_{15}) &= 17700 < 17726 = V^{16}(L_{15}),\\
V^{16}(L_{16}) &= 30342 < 30343 = V^{17}(L_{16}),\\
V^{17}(L_{17}) &= 48950 = V^{18}(L_{17}),\\
V^{18}(L_{18}) &= 85819 < 85820 = V^{19}(L_{18}).
\end{aligned}$$

We also have

$$\begin{aligned}
V^{16}(L_{16}+1) &= 30357 > 30356 = V^{17}(L_{16}+1),\\
V^{17}(L_{17}+1) &= 48965 > 48963 = V^{18}(L_{17}+1),\\
V^{18}(L_{18}+1) &= 85835 > 85834 = V^{19}(L_{18}+1),
\end{aligned}$$

which completes the proof of the theorem. □

Two consequences of this theorem are easily proved.

COROLLARY 6.4. *$N(x)$ is the smallest integer n for which $x \leqq L_n$.*

COROLLARY 6.5. *If $V^n(x) \leqq V^{n+1}(x)$, then $V^{n'}(x) \leqq V^{n'+1}(x)$ for all $n' \geqq n$.*

Remark. Corollary 6.5 answers in the affirmative a conjecture of Knuth [5, Exercise 5.4.2-15].

Table 6.2 provides the values of L_n for $t = 2, \cdots, 7$ and $n = 1, \cdots, 19$. Since such a table is easily prepared, we are able to provide a very simple dispersion algorithm.

TABLE 6.2

L_n for $2 \leqq t \leqq 7$ and $1 \leqq n \leqq 19$

n	$t=2$	$t=3$	$t=4$	$t=5$	$t=6$	$t=7$
1	2	3	4	5	6	7
2	3	5	7	9	11	13
3	5	9	13	13	16	19
4	8	17	22	28	19	23
5	13	31	34	42	52	26
6	21	54	75	60	72	87
7	34	95	108	153	97	114
8	55	172	243	215	282	147
9	89	279	358	268	385	167
10	144	534	455	778	480	639
11	233	819	1196	1033	554	791
12	377	1634	1562	1248	1995	921
13	610	2400	4033	3909	2485	1016
14	987	4958	5378	4969	2900	4396
15	1597	7028	6455	5840	10577	5250
16	2573	14952	18560	19408	13096	5978
17	3954	20582	22875	23917	15335	6498
18	6527	44898	64188	27556	17028	30163
19	10488	60297	88058	95802	69843	35027

ALGORITHM 6.1. Optimal polyphase sort for x strings.

Step 1. Find the smallest n for which $x \leqq L_n$.

Step 2. Choose a j for which $S^n(j) \leqq x \leqq S^n(j+1)$.

Step 3. Find integers $x_1, \cdots, x_t$ for which $x = x_1 + \cdots + x_t$ and $S_i^n(j) \leqq x_i \leqq S_i^n(j+1)$ for $i = 1, \cdots, t$.

Step 4. For each $i = 1, \cdots, t$ write x_i strings to tape i.

Step 5. Use Algorithm 4.3 to perform the polyphase merge on the distribution $x_1, \cdots, x_t$ starting at stage n.

Remarks. Since Steps 2 and 3 of the above algorithm and Steps 1 and 2 of Algorithm 4.3 both require tables of the numbers $M_i^n(j)$, some of the operations of these steps can be combined. The above algorithm should be compared with Shell's optimum dispersion algorithm [8] which is directed by a table of numbers closely related to the numbers L_n. The functions $V_i^n(x)$ share many of the properties of the function $V^n(x)$ and most of the results of this section can be carried over to these functions. Unfortunately, the analogues of the numbers j_n are in general different for each i; otherwise the next section would not have to have been written.

7. Blind dispersion.

In practice, it is very difficult to predict the number of strings that a dispersion routine will provide. However, Algorithm 6.1 requires that this number be known before the strings are written to the tapes. This brings us to the problem of *blind dispersion*, that is, dispersion without knowing the number of strings in advance.

We begin by observing that no solution to the blind dispersion problem will in general be optimal. Indeed, solutions which require rearranging the contents of the tapes will require additional string motion which will result in a solution which is at best optimal. Therefore, let us consider a solution in which the strings stay on the tapes once they are written. Let us suppose that $t = 2$ and that we have dispersed $S^{10} = 144$ strings optimally. Since $N(144) = 10$ and $V^{10}(144) = 1075 < 1088 = V^{11}(144)$, it is clear that the only optimal distribution is for stage 10 when there are $S_1^{10} = 55$ strings on tape one and $S_2^{10} = 89$ strings on tape two. Let us see what happens when we add another string. Since $N(145) = 11$ and $V^{11}(145) = 1100 < 1143 = V^{12}(145)$, the best distribution of 145 strings is one which is optimal for stage 11. However, since $S_1^{10} > 52 = S_1^{11}(8) + 1$ and $S_2^{10} < 96 = S_2^{11}(8) - 1$, we see that there is no way of arriving at a distribution which is optimal for stage 11 by adding one string to our original distribution. This pathology was first discovered by D. E. Knuth.

It is not difficult to see that any blind dispersion technique which rearranges the contents of the tapes can be transformed into an equivalent (or perhaps better) method in which the rearranging is performed after all of the strings have been dispersed. The effectiveness of such a technique depends on how close the distribution, prior to rearranging, is to an optimal distribution. We remark that one kind of rearrangement which incurs no extra cost is that of renumbering the tape units. Theorem 5.3 shows that a monotone distribution provides the best renumbering possible. However, since the distributions which we will consider will be monotone or can be made monotone, we will have no use for this technique.

In the remainder of this section, we will construct a nearly optimal blind dispersion technique which requires no tape rearrangement. This dispersion technique can be used by itself or in conjunction with some rearrangement algorithm.

Suppose that $n \geqq N_t$. We define $m(n)$ to be the largest integer m for which $j_m = j_n$. From Lemma 6.3 we see that $m(n) < n+t$ and that $j_m = j_n$ for $n \leqq m \leqq m(n)$. For $i = 1, \cdots, t$ we define

$$B_i^n = \min \{S_i^m(j_n) | n \leqq m \leqq m(n)+1\}$$

and

$$B^n = B_1^n + \cdots + B_t^n.$$

THEOREM 7.1. *For* $n \geqq N_t$ *we have*

(a) $B_i^n \leqq B_i^{n+1}$ *for* $1 \leqq i \leqq t$;

(b) $B_1^n \leqq B_2^n \leqq \cdots \leqq B_t^n$;

(c) $S_i^n(j_n - 1) \leqq B_i^n \leqq S_i^n(j_n)$ *for* $1 \leqq i \leqq t$;

(d) $S_i^{n+1}(j_n - 1) \leqq B_i^n \leqq S_i^{n+1}(j_n)$ *for* $1 \leqq i \leqq t$;

(e) $B^n \leqq c_n < B^{n+t}$.

Remark. Statements (c) and (d) imply that the distribution $B_1^n, \cdots, B_t^n$ is optimal for both stage n and stage $n+1$.

Before we prove the theorem we require a lemma:

LEMMA 7.1. *If* $n \geqq N_t$, *then for* $i = 1, \cdots, t$ *we have*

$$S_i^{n+1}(j_n - 1) \leqq S_i^n(j_n).$$

Proof. We will begin by showing that for $n \geqq 1$ we have

$$S_i^n(j) - S_i^{n+1}(j-1) \geqq G^n(j-1) - G^{n+1}(j-1) \tag{7.1}$$

with only finitely many exceptions. We define the t-arrays A_i for $i = 1, \cdots, t$ and D by

$$A_i^n(j) = S_i^n(j) - S_i^{n+1}(j-1),$$
$$D^n(j) = G^n(j-1) - G^{n+1}(j-1).$$

It is not difficult to verify that the nonzero elements of the initialization regions for the t-arrays $A_1, \cdots, A_t$ are

$$A_i^{i-t}(j) = 1 \qquad \text{for } 1 \leqq i < t \text{ and } j \geqq 0,$$
$$A_i^{i-t-1}(j) = -1 \qquad \text{for } 1 < i \leqq t \text{ and } j \geqq 1,$$
$$A_i^0(j) = -1 \qquad \text{for } 1 \leqq i < t \text{ and } j \geqq 2, \text{ and}$$
$$A_t^0(0) = A_t^0(1) = 1.$$

Also, the nonzero elements of the initialization region for the t-array D are

$$D^0(j) = t - (t-1)j \quad \text{for } j \geqq 1.$$

Tables 7.1(a) to 7.1(g) each display portions of the t-arrays $A_i - D$ for various ranges of t and i. By inspection, we see that the only negative entries outside of the

TABLE 7.1(a)
Proof of Lemma 7.1 ($t \geqq 4, i = t$)

$n =$	-1	0	1	2	3
$j = 0$	0	1			
1	-1	0	1	1	
2	-1	$t-2$	-1	0	1

TABLE 7.1(b)
Proof of Lemma 7.1 ($t \geqq 3, i = 1$)

$n =$	$1-t$	$\cdots$	0	1
$j = 0$	1	$\cdots$	0	
1	1	$\cdots$	-1	1

TABLE 7.1(c)
Proof of Lemma 7.1 ($t \geqq 4, 1 < i < t$)

$n =$	$i-t-1$	$i-t$		$\cdots$	0	1	2	3
$j = 0$	0	1	0	$\cdots$	0			
1	-1	1	0	$\cdots$	-1	1	1	
2	-1	1	0	$\cdots$	$t-3$	-1	$\geqq 0$	$\geqq 1$

TABLE 7.1(d)
Proof of Lemma 7.1 ($t = 3, i = 2$)

$n =$	-2	-1	0	1	2
$j = 0$	0	1	0		
1	-1	1	-1	1	
2	-1	1	0	-1	1

TABLE 7.1(e)
Proof of Lemma 7.1 ($t = 3, i = 3$)

$n =$	-1	0	1	2	3	4	5	6
$j = 0$	0	1						
1	-1	0	1	1	1			
2	-1	1	-1	0	2	3		
3	-1	3	0	-1	0	1	5	
4	-1	5	2	2	2	-1	0	6

TABLE 7.1(f)

Proof of Lemma 7.1 ($t = 2, i = 1$)

$n =$	−1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$j = 0$	1	0	0	0	0	0	0														
1	1	−1	1	0	0	0	0	0													
2	1	−1	0	0	1	0	0	0	0												
3	1	0	0	−1	0	1	1	0	0	0											
4	1	1	1	0	−1	−1	1	2	1	0	0										
5	1	2	2	2	1	−1	−2	0	3	3	1	0									
6			3	4	4	3	0	−3	−2	3	6	4	1								
7					7	8	7	3	−3	−5	1	9	10	5							
8							15	15	10	0	−8	−4	10	19	15						
9									30	25	10	−8	−12	6	29	34					
10											55	35	2	−20	−6	35	63				
11													90	37	−18	−26	29	98			
12															127	19	−44	3	127		
13																	146	−25	−41	130	
14																			121	−66	89

TABLE 7.1(g)

Proof of Lemma 7.1 ($t = 2, i = 2$)

$n =$	−1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$j = 0$	0	1	0	0	0	0	0															
1	−1	0	1	1	0	0	0	0														
2	−1	0	−1	1	2	1	0	0	0													
3	−1	1	−1	−1	0	3	3	1	0	0												
4	−1	2	0	0	−2	−1	3	6	4	1	0											
5	−1	3	1	2	0	−2	−3	2	9	10	5	1										
6		4	2	4	3	2	−2	−5	−1	11	19	15	6									
7				6	6	7	5	0	−7	−6	10	30	34	21								
8						12	13	12	5	−7	−13	4	40	64	55							
9								25	25	17	−2	−20	−9	44	104	119						
10										50	42	15	−22	−29	35	148	223					
11												92	57	−7	−51	6	183	371				
12														149	50	−58	−45	189	556			
13																199	−8	−103	144	745		
14																		191	−111	41	899	
15																				80	−70	930

initialization region are those displayed. Except in the case $t=i=2$, we see that (7.1) holds for all $n \geqq N_t$ and $i=1,\cdots,t$. From the definition of j_n, we see that $G^n(j_n-1)\geqq G^{n+1}(j_n-1)$ and therefore by (7.1) we have $S_i^n(j_n)\geqq S_i^{n+1}(j_n-1)$. In the exceptional case we have $n=N_t=19$ and $j_{19}=15$ so we may verify directly that $S_2^{19}(15)=6050>5270=S_2^{20}(14)$. This completes the proof. □

Proof of Theorem 7.1. If $j_n=j_{n+1}$, then $m(n)=m(n+1)$ so that (a) is obvious. If this is not the case, then by Lemma 6.3, we must have $j_{n+1}=j_n+1$ so that $S_i^{n+1}(j_n)\geqq S_i^{n+1}(j_{n+1})$. Also, since $m(n+1)\leqq n+t$, we see that $S_i^{n+1}(j_n)\leqq S_i^k(j_{n+1})$ for $n+2\leqq k\leqq m(n+1)+1$; this follows from the fact that $S_i^{n+1}(j_n)$ is a term of the t-sum which computes $S_i^k(j_{n-1})$. We have therefore shown that $B_i^n\leqq S_i^{n+1}(j_n)\leqq B_i^{n+1}$, which is (a). Statement (b) follows at once from the fact that $S_1^n(j)\leqq\cdots\leqq S_t^n(j)$ for all $n\geqq 1$ and $j\geqq 1$.

To prove (c) and (d) we first observe that the definition of B_i^n implies that $B_i^n\leqq S_i^n(j_n)$ and $B_i^n\leqq S_i^{n+1}(j_n)$. It is also clear that $S_i^n(j_n-1)\leqq S_i^n(j_n)$ and $S_i^{n+1}(j_n-1)\leqq S_i^{n+1}(j_n)$. From Lemma 7.1, we have $S_i^{n+1}(j_n-1)\leqq S_i^n(j_n)$. Finally by reasoning similar to that used in the above paragraph, we have $S_i^n(j_n-1)\leqq S_i^k(j_n)$ for $n+1\leqq k\leqq m(n+1)$ and $S_i^{n+1}(j_n-1)\leqq S_i^k(j_n)$ for $n+2\leqq k\leqq m(n)+1$ if $m(n)>n$. From these inequalities, it follows at once that $S_i^n(j_n-1)\leqq B_i^n$ and that $S_i^{n+1}(j_n-1)\leqq B_i^n$ which completes the proof of (c) and (d).

By (c) we have $B^n\leqq S^n(j_n)\leqq c_n$. If we let $n'=m(n)+1$, then it is clear that $j_{n'}=j_n+1$, $j_{n'-1}=j_n$, and $n'\leqq n+t$. Therefore, by (a) and (c) we have

$$c_n\leqq c_{n'-1}<S^{n'}(j_{n'-1})=S^{n'}(j_{n'}-1)\leqq B^{n'}\leqq B^{n+t}$$

which establishes (e) and completes the proof of the theorem. □

From the theorem, two important properties of the distributions $B_1^n,\cdots,B_t^n$ are apparent. First, we may arrive at the distribution $B_1^{n+1},\cdots,B_t^{n+1}$ by simply adding strings to the distribution $B_1^n,\cdots,B_t^n$. Second, if we are dispersing for stage n and we reach the distribution $B_1^n,\cdots,B_t^n$, then we may begin dispersing for stage $n+1$ since the distribution is optimal for both stages. Clearly we can base a blind dispersion algorithm on these properties of the numbers B_i^n. However, since we will be making several refinements, it is of value to examine the general structure of such an algorithm.

We define a *quota scheme* for polyphase dispersion to be a family of nonnegative integers $Q^n, Q_1^n,\cdots,Q_t^n$, $n=1,2,\cdots$, which have the following properties for $n\geqq 1$ and $1\leqq i\leqq t$:

$$Q_i^n\leqq S_i^n,\quad Q_i^n\leqq Q_i^{n+1},\quad Q^n\leqq Q^{n+1},$$
$$Q^n\leqq Q_1^n+\cdots+Q_t^n\quad\text{and}\quad Q^n\to\infty\quad\text{as } n\to\infty.$$

Following is the dispersion algorithm which is directed by the quota scheme. The counters $x_1,\cdots,x_t$ contain the numbers of strings which have been written to tapes $1,\cdots,t$. Upon completion, the values of $x_1,\cdots,x_t$ and n are the parameters for initializing Algorithm 4.3.

ALGORITHM 7.1. Quota-directed polyphase dispersion.

Step 1. Let $n=1$ and $x_i=y_i=0$ for $i=1,\cdots,t$.

Step 2. If there are no more strings to disperse, then terminate the algorithm.

Step 3. If $x_1+\cdots+x_t=Q^n$, then let $n=n+1$ and $y_1=\cdots=y_t=0$ and repeat this step.

Step 4. Choose some i for which $x_i<y_i$. If this choice cannot be made, then go to Step 6.

Step 5. Write a string to tape unit i, let $x_i=x_i+1$, and go to Step 2.

Step 6. Find the smallest j for which $x_i<S_i^n(j)$ for some i. Let $y_i=\min(Q_i^n, S_i^n(j))$ for $i=1,\cdots,t$. Go to Step 4.

Informally, this algorithm disperses for stage n keeping $x_i\leqq Q_i^n$ for each i until $x_1+\cdots+x_t=Q^n$ and then begins dispersing for stage $n+1$. When the algorithm is dispersing for stage n, the strings are written in such a way as to minimize the growth of $V_1^n(x_1)+\cdots+V_t^n(x_t)$.

Since the choice of i made in Step 4 is arbitrary, the distribution $x_1,\cdots,x_t$ may be uncertain when the algorithm switches from stage n to stage $n+1$. For this reason, the first value of j chosen for stage $n+1$ by Step 6 may vary thereby causing the volume of the sort to vary. This uncertainty disappears if $Q^n=Q_1^n+\cdots+Q_t^n$ or if it is known that when we switch from stage n to stage $n+1$, then the distribution is optimal for stage $n+1$. Indeed, in the first case the distribution is completely known and in the second case we know that j is the smallest integer for which $x_1+\cdots+x_t<S^{n+1}(j)$. The quota scheme which we will consider has one or the other of these properties for each n.

When the quota scheme has these properties, then Algorithm 7.1 may be transformed into a simpler table-directed algorithm. The tables have the entries $n^k, q^k, q_1^k,\cdots,q_t^k$ for $k\geqq 1$ and are constructed as follows: We initialize the counter k to zero and perform Algorithm 7.1 with an unlimited supply of strings; after each time that Step 6 is performed, we increment k by one and let $n^k=n$, $q^k=Q^n$, and $q_i^k=y_i$ for each i. The simplified algorithm follows:

ALGORITHM 7.2. Simplified quota-directed polyphase dispersion.

Step 1. Let $k=1$ and $x_1=\cdots=x_t=0$.

Step 2. If there are no more strings to disperse, then terminate the algorithm.

Step 3. If $x_1+\cdots+x_t=q^k$, then let $k=k+1$.

Step 4. Choose some i for which $x_i<q_i^k$. If this choice cannot be made, then let $k=k+1$ and go to Step 3.

Step 5. Write a string to unit i, let $x_i=x_i+1$, and go to Step 2.

At termination, the parameters for the polyphase merge algorithm are n^k and $x_1,\cdots,x_t$. Since the required tables may be prepared in advance, this algorithm provides a very compact method of dispersing for the polyphase sort. For most applications, the maximum value of k should never exceed forty.

We will now present the rules for constructing the quota scheme for the blind polyphase dispersion algorithm.

1. If $n\geqq N_t$ and if $B_i^n<S_i^n(j_n)$ for some i, then we let

$$Q^n=B^n \quad\text{and}\quad Q_i^n=B_i^n \quad \text{for } i=1,\cdots,t.$$

2. If $n\geqq N_t$ and if $B_i^n=S_i^n(j_n)$ for each i, then we let

$$Q_i^n=\min(S_i^n(j_n+1), S_i^{n+1}(j_n), B_i^{n+1})$$

for $i=1,\cdots,t$ and we let

$$Q^n=\min(c_n, Q_1^n+\cdots+Q_t^n).$$

3. If $t \geqq 3$ and $1 \leqq n < N_t$, then we let

$$Q^n = S^n \quad \text{and} \quad Q_i^n = S_i^n \quad \text{for } i = 1, \cdots, t.$$

4. If $t = 2$ and $n < N_t = 19$, then we let

$$Q^n = S^n \quad \text{and} \quad Q_i^n = S_i^n \quad \text{for } n \leqq 15 \text{ and } i = 1, \cdots, t$$

and, in addition, we let

$$Q^{16} = L_{16} = 2573, \quad Q^{17} = 3845, \quad Q^{18} = L_{18} = 6527,$$
$$Q_1^{16} = S_1^{16}(15) = 986, \quad Q_2^{16} = S_2^{16}(15) = 1596,$$
$$Q_1^{17} = S_1^{18}(13) = 1383, \quad Q_2^{17} = S_2^{17}(14) = 2462,$$
$$Q_1^{18} = S_1^{18}(16) = 2567, \quad Q_2^{18} = S_2^{18}(16) = 4163.$$

To show that these rules define a quota scheme, we will begin by showing that for $n \geqq N_t$, we have

$$B^n \leqq Q^n \leqq B^{n+1} \quad \text{and} \quad B_i^n \leqq Q_i^n \leqq B_i^{n+1} \quad \text{for } i = 1, \cdots, t.$$

These relations are obvious when rule 1 is applied. If rule 2 is applied instead, we have, from Theorems 6.2 and 7.1,

$$B_i^n = S_j^n(j_n) \leqq Q_j^n \leqq B_i^{n+1} \quad \text{for } j = 1, \cdots, t \quad \text{and}$$
$$B^n = S^n(j_n) \leqq Q^n \leqq Q_1^n + \cdots + Q_t^n \leqq B^{n+1}.$$

For $t \geqq 3$, we must show that when $n = N_t - 1$, we have $S^n \leqq Q^{n+1}$ and $S_i^n \leqq Q_i^{n+1}$ for $i = 1, \cdots, t$. Clearly, it is sufficient to show that $S_i^n \leqq B_i^n$ for each i. For $t = 3$, we have $N_t = 6$ and

$$S_1^5 = 7 < 12 = B_1^6, \quad S_2^5 = 11 < 19 = B_2^6, \quad S_3^5 = 13 < 27 = B_3^6.$$

Similarly, for $t = 4$ we have $N_t = 4$ and

$$S_1^3 = 2 < 3 = B_1^4, \quad S_2^3 = 3 < 5 = Q_2^4,$$
$$S_3^3 = 4 < 6 = B_3^4, \quad S_4^3 = 4 < 7 = Q_4^4.$$

For $t \geqq 5$, we have $N_t = 3$ and using the t-array representation, it may be shown that $G^n(2) = 2t + (n-1)(t-1-n/2)$ for $1 \leqq n \leqq t$ from which it follows that $G^3(2) < \cdots < G^{t-1}(2) = G^t(2)$ so that $m(3) = t - 2$ since $j_3 = 2$. Using t-arrays, we may also show that $S_i^2(2) \leqq \cdots \leqq S_i^t(2)$ for each i so that $S_i^2(2) \leqq S_i^3(2) = B_i^3$ for each i. The proof that rule 4 also contributes to a quota scheme is straightforward once we observe that when $t = 2$ we have

$$S_1^{15} = 610, \quad S_2^{15} = 987, \quad B_1^{19} = 3588, \quad B_2^{19} = 6050.$$

We have already seen how the numbers B^n and $B_1^n, \cdots, B_t^n$ can be used to describe blind polyphase dispersion so rule 1 requires no explanation. Rule 2 represents a refinement in which Q^n is pushed to the largest value not exceeding c_n for which we can switch from stage n to stage $n+1$ with a distribution which is optimal for both stages. Rule 2 will be used for each n for which $j_{n+1} = j_n + 1$. Rules 3 and 4 simply fill out the quota scheme for small values of n. The special assignments in rule 4 were chosen subjectively to insure reasonably good performance.

If we are dispersing using the quota scheme just described, it is clear that when the number of strings x is large, then we will switch stages with a distribution which is optimal for both stages. Consequently, the volume of the sort will be $V^{N'(x)}(x)$ for some integer $N'(x)$ when x is sufficiently large. Since $Q^n \leqq c_n$ for $n \geqq N_t$, it is clear that $N'(x) \geqq N(x)$. On the other hand, since $c_n < B^{n+t} \leqq Q^{n+t}$, we see that $N'(x) < N(x) + t$.

The blind polyphase sort which we have described is almost as good as the optimal sort of Algorithm 6.1; when the number of strings is in the range of the size of most applications (say, less than a thousand), the two sorts are almost always equivalent. When the number of strings is large, it can be shown that the two algorithms are equivalent infinitely often. Indeed, this happens for $S^n(j_n)$ strings every time that $j_{n+1} = j_n + 1$. In the next section we will show that the two algorithms are also asymptotically equivalent.

Example 7.1. Table 7.2 displays a portion of the simplified quota scheme for the case $t = 4$.

TABLE 7.2
Simplified quota scheme for $t = 4$

k	n^k	q^k	q_1^k	q_2^k	q_3^k	q_4^k
1	1	4	1	1	1	1
2	2	7	1	2	2	2
3	3	9	1	2	3	3
4	3	13	2	3	4	4
5	4	21	3	5	6	7
6	4	22	4	6	7	8
7	5	30	4	7	9	10
8	5	34	4	8	11	13
9	6	36	4	8	11	13
10	6	71	10	17	21	23
11	6	75	13	22	26	28
12	7	100	13	23	30	34
13	7	108	14	27	37	44
14	8	122	14	27	37	44
15	8	241	34	57	71	79
16	8	243	44	77	92	100
17	9	338	44	78	101	115
18	9	358	50	94	128	151
19	10	423	50	94	128	151
20	10	455	50	100	144	178
21	11	472	50	100	144	178
22	11	1156	151	266	345	394

8. Asymptotic performance. In this section, we will study the performance of the algorithms which we have described when the number of strings is large. There are two volumes which we are interested in estimating. First there is the volume of the optimal polyphase sort of § 6,

$$V(x) = V^{N(x)}(x),$$

and, second, there is the volume of the blind polyphase sort, when x is large, this is

$$V'(x) = V^{N'(x)}(x).$$

We will show that when x is large, both of these volumes are asymptotically equal to

$$x \log_t x + \tfrac{1}{2}x \log_t \log_t x + O(x).$$

The reader who is not familiar with asymptotic methods may find [1] or the first chapter of [4] to be helpful.

Our starting point is an interesting connection between the movement numbers and the theory of probability. Let $y_1, y_2, \cdots$ be independent random variables which each take on the values $1, 2, \cdots, t$ with equal probability t^{-1}. Simple calculations will show that each y_i has an expectation $\mu = (t+1)/2$ and a variance $\sigma^2 = (t^2-1)/12$. For positive integers m and k we define

$$p(m, k) = \text{prob}\,(y_1 + \cdots + y_k = m). \tag{8.1}$$

LEMMA 8.1. *For $n \geqq 1$ and $j \geqq 1$, we have $M_t^n(j) = t^j p(n, j)$.*

Proof. Let $q(z) = z + z^2 + \cdots + z^t$ for the real variable z. We will begin by showing that

$$\sum_{n \geqq 1} M_t^n(j) z^n = q(z)^j. \tag{8.2}$$

Since the only nonzero values of $M_t^n(1)$ are $M_t^n(1) = 1$ when $1 \leqq n \leqq t$, we see that (8.2) is true when $j = 1$. Furthermore, given (8.2) and the fact that $M_t^{n-k}(j) = 0$ when $n \leqq k$, we may write

$$\begin{aligned}\sum_{n \geqq 1} M_t^n(j+1) z^n &= \sum_{n \geqq 1} \sum_{k=1}^{t} M_t^{n-k}(j) z^n \\ &= \sum_{k=1}^{t} z^k \sum_{n \geqq 1} M_t^{n-k}(j) z^{n-k} \\ &= \sum_{k=1}^{t} z^k q(z)^j = q(z)^{j+1}\end{aligned}$$

so that (8.2) follows by induction. From the form of $q(z)$, we see that the coefficient of z^n in $q(z)^j$ is precisely the number of ways that n may be written as the ordered sum of j integers, not necessarily distinct, chosen from the set $\{1, 2, \cdots, t\}$. Since this number is precisely $t^j p(n, j)$, the proof is complete. □

Techniques for estimating probabilities of the form (8.1) are well known. For our purposes, the best such approximation follows from a theorem of C. G. Esseen which is given on page 241 of [3]:

$$p(m, k) = \frac{e^{-s^2/2}}{\sigma\sqrt{2\pi}}\left(\frac{1}{k^{1/2}} + \frac{Q_1(s)}{k} + \frac{Q_2(s)}{k^{3/2}} + \frac{Q_3(s)}{k^2} + \frac{Q_4(s)}{k^{5/2}}\right) + o\left(\frac{1}{k^{5/2}}\right),$$

where we have written $s = (m - \mu k)/\sigma\sqrt{k}$ and where $Q_1(s)$, $Q_2(s)$, $Q_3(s)$, $Q_4(s)$ are polynomials in which the coefficients depend only on the moments of y_i which, in turn, depend only on t. It turns out that all of the centralized moments of y_i of odd order are zero. This leads to some simplifications; in particular $Q_1(s) = Q_3(s) = 0$ and $Q_2(s)$ takes on the simplified form $c(s^4 - 6s^2 + 3)$ in which c depends only on t. Using the estimates

$$\begin{aligned} e^{-s^2/2} &= 1 - \tfrac{1}{2}s^2 + O(s^4), \\ Q_2(s) &= 3c + O(s^2 + s^4), \\ e^{-s^2/2}Q_4(s) &= O(1), \end{aligned}$$

we obtain the approximation

$$p(m,k) = \frac{1}{\sigma\sqrt{2\pi}}\left(\frac{1}{k^{1/2}}(1 - s^2/2) + \frac{3c}{k^{3/2}}\right) + O\left(\frac{1}{k^{5/2}} + \frac{s^4}{k^{1/2}} + \frac{s^8}{k^{3/2}}\right)$$

which may be written in the form

$$p(m,k) = \frac{1}{\sigma\sqrt{2\pi}}\left(\frac{1}{k^{1/2}} + \frac{1}{k^{3/2}}\left(3c - \frac{(m-\mu k)^2}{2\sigma^2}\right)\right) + O\left(\frac{1 + (m-\mu k)^8}{k^{5/2}}\right).$$

To simplify subsequent calculations, we will use the symbol $\mathscr{P}_n(z)$ to represent generically an nth degree polynomial in z in which the coefficient of z^n is positive and in which all of the coefficients are functions only of t. Two distinct appearances of the symbol in the text need not represent the same polynomial. With this convention, we now have

$$p(m,k) = \frac{1}{\sigma\sqrt{2\pi k}} - \frac{1}{k^{3/2}}\mathscr{P}_2(m - \mu k) + O\left(\frac{1 + (m-\mu k)^8}{k^{5/2}}\right). \tag{8.3}$$

Lemma 8.2. *We have*

$$\sqrt{j}t^{-j}M^n(j) = \frac{\mu}{\sigma\sqrt{2\pi}} - \frac{1}{j}\mathscr{P}_2(n - \mu j) + O\left(\frac{1 + (n-\mu j)^8}{j^2}\right). \tag{8.4}$$

Proof. From the last two formulas of (3.1), we have

$$M_i^n(j) = M_t^{n-1}(j-1) + \cdots + M_t^{n-i}(j-1)$$

so that

$$M^n(j) = tM_t^{n-1}(j-1) + (t-1)M_t^{n-2}(j-1) + \cdots + M_t^{n-t}(j-1).$$

Therefore, by Lemma 8.1, (8.3), and the facts that

$$\begin{aligned} \frac{1}{(j-1)^{1/2}} &= \frac{1}{j^{1/2}} + \frac{1}{2j^{3/2}} + O\left(\frac{1}{j^{5/2}}\right), \\ \frac{1}{(j-1)^{3/2}} &= \frac{1}{j^{3/2}} + O\left(\frac{1}{j^{5/2}}\right), \end{aligned}$$

we have

$$
\begin{aligned}
t^{-j}M^n(j) &= \sum_{i=1}^{t} (t-i+1)t^{-j}M_t^{n-i}(j-1)\\
&= t^{-1}\sum_{i=1}^{t} (t-i+1)p(n-i,j-1)\\
&= \frac{t^{-1}}{\sigma\sqrt{2\pi j}}\sum_{i=1}^{t} (t-i+1) - \sum_{i=1}^{t} \frac{(t-i+1)}{j^{3/2}}\mathcal{P}_2(n-\mu j-i+\mu)\\
&\qquad + O\left(\frac{1+(n-\mu j)^8}{j^{5/2}}\right)
\end{aligned}
$$

which is easily reduced to the form (8.4). □

LEMMA 8.3. *We have*

$$
\sqrt{j}t^{-j}G^n(j) = \frac{\mu}{\sigma\sqrt{2\pi}}\frac{t^2}{(t-1)^2} - \frac{\mathcal{P}_2(n-\mu j)}{j} + O\left(\frac{1+(n-\mu j)^8}{j^2}\right). \tag{8.5}
$$

Proof. From our definitions we have

$$
\begin{aligned}
G^n(j) &= \sum_{k=1}^{j} S^n(j) = \sum_{k=1}^{j}\sum_{i=1}^{k} M^n(i)\\
&= \sum_{k=1}^{j} (j-k+1)M^n(k) = \sum_{k=0}^{j-1} (k+1)M^n(j-k).
\end{aligned}
$$

Using this formula and Lemma 8.2, we obtain

$$
\begin{aligned}
t^{-j}G^n(j) &= \sum_{k=0}^{j-1} t^{-j}(k+1)M^n(j-k)\\
&= \sum_{k=0}^{j-1} t^{-k}(k+1)\left(\frac{\mu}{\sigma\sqrt{2\pi}}\frac{1}{(j-k)^{1/2}} - \frac{\mathcal{P}_2(n-\mu j+\mu k)}{(j-k)^{3/2}}\right)\\
&\quad + O\left(\sum_{k=0}^{j-1} t^{-k}(k+1)\frac{1+(n-\mu j+\mu k)^8}{(j-k)^{5/2}}\right).
\end{aligned}
$$

In order to simplify the above approximation, we need to be able to estimate sums of the form

$$
S(a,b) = \sum_{k=0}^{j-1} \frac{k^a}{(j-k)^b} t^{-k}
$$

for $a=0,1,\cdots,9$ and $b=1/2,\ 3/2,\ 5/2$. Let us write $m=\lfloor\sqrt{j}\rfloor-1$. If j is sufficiently large, then we will have $k^a<(3/2)^k$ for all $k>m$ and $a=0,1,\cdots,11$. From the binomial expansion, it is clear that for $k\leqq m$, we have

$$
\frac{1}{(j-k)^b} = \frac{1}{j^b} + \frac{bk}{j^{b+1}} + O\left(\frac{k^2}{j^{b+2}}\right).
$$

It is also clear that

$$\sum_{k=0}^{m} k^a t^{-k} = s(a) + O((3/2t)^m),$$

where

$$s(a) = \sum_{k=0}^{\infty} k^a t^{-k}.$$

We therefore have

$$\begin{aligned} s(a,b) &= \sum_{k=0}^{m} \left(\frac{k^a}{j^b} + \frac{bk^{a+1}}{j^{b+1}} \right) t^{-k} + \sum_{k=m+1}^{j-1} \frac{k^a}{(j-k)^b} t^{-k} \\ &\quad + O\left(\sum_{k=0}^{m} \frac{k^{a+2}}{j^{b+2}} t^{-k} \right) \\ &= \frac{s(a)}{j^b} + \frac{bs(a+1)}{j^{b+1}} + O\left((3/2t)^m + \frac{1}{j^{b+1}} \right) \end{aligned}$$

since the second sum is $O((3/2t)^m)$ and the last sum is $O(1/j^{b+2})$. Since $(3/2t)^m$ is $O(1/j^{b+2})$ for each b, we may replace the above error estimate by $O(1/j^{b+2})$. The conclusion of this lemma now follows from the facts that

$$s(0) = t/(t-1) \quad \text{and} \quad s(1) = t/(t-1)^2. \qquad \square$$

COROLLARY 8.1. *If $n - \mu j = O(1)$, then*

$$\sqrt{j}\, t^{-j} G^n(j) = \frac{\mu}{\sigma\sqrt{2\pi}} \frac{t^2}{(t-1)^2} + O\left(\frac{1}{j}\right).$$

Proof. This is a simple consequence of the lemma. □

COROLLARY 8.2. *Let j_n be defined as in* § 6; *we then have $n - \mu j_n = O(1)$.*

Proof. We recall that j_n is the smallest integer j for which $G^n(j) < G^{n+1}(j)$. From the estimate (8.5), we obtain

$$\sqrt{j}\, t^{-j} (G^{n+1}(j) - G^n(j)) = \frac{\mathcal{P}_1(n-\mu j)}{j} + O\left(\frac{1 + (n-\mu j)^8}{j^2} \right).$$

With $\mathcal{P}_1$ understood to be the $\mathcal{P}_1$ appearing above, let a be the real number which satisfies $\mathcal{P}_1(n - \mu a) = 0$. If $j \geqq \lceil a \rceil + 1$, then clearly $\mathcal{P}_1(u - \mu j)$ is less than some negative quantity which is independent of n and if $j \leqq \lfloor a \rfloor - 1$, then $\mathcal{P}_1(n - \mu j)$ is larger than some positive quantity which is independent of n. For j in the range $\lfloor a \rfloor - 1 \leqq j \leqq \lceil a \rceil + 1$, we have $n - \mu j = O(1)$ and the error estimate above becomes $O(j^{-2})$ so that the first term of the estimate dominates. It follows that j_n lies within this range for large n and the proof is complete. □

THEOREM 8.1. $V(x) = x \log_t x + \frac{1}{2} x \log_t \log_t x + O(x)$.

Proof. Let $c_n = G^n(j_n) - G^{n+1}(j_n - 1)$ be defined as in § 6. From Corollaries 8.1 and 8.2 it is easily shown that

$$\sqrt{j_n}\, t^{-j_n} c_n = \frac{\mu}{\sigma\sqrt{2\pi}} \frac{t}{t-1} + O(j_n^{-1}). \tag{8.6}$$

If x is sufficiently large, then for $n=N(x)$ we have $c_{n-1}<x\leqq c_n$. Let j be the unique integer for which $S^n(j)<x\leqq S^n(j+1)$. From Theorem 6.2, it is clear that $j_{n-1}-1\leqq j\leqq j_n$. Using the formula

$$V(x)=(j+1)x-G^n(j)$$

we may write

$$V(x)-x\log_t x=x(j+1-\log_t x)-G^n(j). \tag{8.7}$$

From (8.6) we have

$$\log_t c_n=j_n-\tfrac{1}{2}\log_t j_n+O(1)$$

and therefore, since $0\leqq j_n-j_{n-1}\leqq 1$,

$$\begin{aligned} j+1-\log_t x &< j+1-\log_t c_{n-1}\\ &=j+1-j_n+\tfrac{1}{2}\log_t j_{n-1}+O(1)\\ &\leqq j_n+1-j_n+\tfrac{1}{2}\log_t j_n+O(1)\\ &=\tfrac{1}{2}\log_t j_n+O(1). \end{aligned}$$

Similarly,

$$j+1-\log_t x\geqq j+1-\log_t c_n=\tfrac{1}{2}\log_t j_n+O(1)$$

and we have shown that

$$j+1-\log_t x=\tfrac{1}{2}\log_t j_n+O(1). \tag{8.8}$$

Comparing Corollary 8.1 and (8.6) we see that

$$G^n(j)=O(c_{n-1})=O(x).$$

From (8.8) and the fact that $j_n-2\leqq j\leqq j_n$, we have

$$j_n^{-1}\log_t x=1+O(j_n^{-1}\log_t j_n)$$

so that

$$\log_t j_n-\log_t\log_t x=O(j_n^{-1}\log_t j_n)=O(1).$$

Putting everything together, (8.7) becomes

$$\begin{aligned} V(x)-x\log_t x&=\tfrac{1}{2}x\log_t j_n+O(x)\\ &=\tfrac{1}{2}x\log_t\log_t x+O(x) \end{aligned}$$

and the proof is complete. □

COROLLARY 8.3. $V'(x)=x\log_t x+\tfrac{1}{2}x\log_t\log_t x+O(x)$.

Proof. In § 7 we showed that $N(x)\leqq N'(x)<N(x)+t$. From Theorem 5.3, it follows that

$$0\leqq V'(x)-V(x)=V^{N'(x)}(x)-V^{N(x)}(x)\leqq(N'(x)-N(x))x\leqq(t-1)x$$

so that $V'(x)-V(x)=O(x)$ and the result follows from the theorem. □

It is well known (see § 5.4.4 of [5]) that the best possible volume for a merge sort which performs p-way merges is $x\log_p x+O(x)$. For this reason, a tape sort with T tape units has an optimum value of $x \log_{T-1} x + O(x)$ since such a sort can perform at most $T-1$-way merges. A tape sort with T tape units which has a volume asymptotic to $x \log_{T-1} x$ is said to be *asymptotically optimal.* Theorem 8.1 and Corollary 8.3 imply that both the optimal polyphase sort and the blind polyphase sort are asymptotically optimal.

Remarks. The optimal polyphase sort appears to be the first known example of an asymptotically optimal read-forward tape sort. Other examples will appear in [9]. Several asymptotically optimal read-backward sorts are known (see, for example, § 5.4.4 of [5]) but these sorts have volumes of the form $x \log_{T-1} x + O(x)$ which is smaller than the volume we have derived for the optimal polyphase sort. One wonders if the volume $x \log_t x + \frac{1}{2}x \log_t \log_t x + O(x)$ can be improved upon for read-forward sorts or whether it represents some theoretical minimum. A simplified self-contained analysis of the optimal polyphase sort, which is probably suitable for students, appears in [10].

9. Concluding remarks. Two questions concerning the optimal polyphase sort remain open for investigation. First there is the problem of estimating the amount of time the algorithm spends waiting for tapes to rewind and second there is the problem of optimizing the read-backward polyphase sort.

The rewind time is significant since both the blind and the optimal polyphase sorts perform large numbers of tape rewind operations. Of course we may suppose that the total amount of rewinding corresponds to the volume of information moved. However, the polyphase merge rewinds two tapes simultaneously so it is conceivable that a highly unbalanced situation may arise in which one of the two tapes being rewound would be considerably longer than the other. This might cause the total rewind wait time to vary from the volume of the merge to twice that volume.

In the read-backward polyphase sort, the tape units act as stacks so the direction in which a string is written is reversed when the string is moved. Therefore, strings which will be moved an odd number of times must be written in the opposite direction from strings which will be written an even number of times. For this reason, strings are no longer interchangeable so the dispersion routine must concern itself with the details of placing the dummy strings.

In this paper, we have limited the discussion to the traditional polyphase merge in which the appointment of the output tapes is cyclic. The polyphase merge, however, is just a special case of the class of single-output read-forward merge algorithms. Some information about these techniques can be found in the exercises for §§ 5.4.2 and 5.4.4 of [5]. It is known for example that in certain special cases, the optimal polyphase sort can be beaten by other methods of merging. In [9] it is shown that a large class of single-output read-forward merge algorithms also give rise to asymptotically optimal sorting algorithms.

Acknowledgments. This paper would have been a pale shadow of itself had I not encountered a preprint of the first half of [5]. Professor Knuth also read an early draft of the paper and made several valuable suggestions. This research was conducted while I was employed at Sperry UNIVAC in Roseville, Minnesota, and began as a study of possible sorting procedures for the DMS 1100 Schema Description (DDL) translator which I was then implementing. I am indebted to

Mr. E. H. Moulton for many valuable conversations and to Mr. A. G. Reiter, Dr. H. C. Gyllstrom, and M. D. Thompson, for allowing me to pursue this subject when I should have been doing something else.

REFERENCES*

[1] N. G. De Bruijn, *Asymptotic Methods in Analysis*, North-Holland, Amsterdam, 1970.

[2] I. Flores, *Computer Sorting*, Prentice-Hall, Englewood Cliffs, N.J., 1969.

[3] B. V. Gnedenko and A. N. Kolmogorov, *Limit Distributions for Sums of Independent Random Variables*, Addison-Wesley, Reading, Mass., 1954.

[4] D. E. Knuth, *Fundamental Algorithms*, The Art of Computer Programming 1, Addison-Wesley, Reading, Mass., 1968.

[5] ———, *Sorting and Searching*, The Art of Computer Programming 3, Addison-Wesley, Reading, Mass., 1973.

[6] W. C. Lynch, *The t-Fibonacci numbers and polyphase sorting*, Fibonacci Quart., 8 (1970), pp. 6–22.

[7] B. S. Sackman and T. Singer, *A vector model for merge sort analysis, Part I, Polyphase merge sort*, paper presented at the ACM Sort Symposium, Princeton, N.J., November, 1962.

[8] D. L. Shell, *Optimizing the polyphase sort*, Comm. ACM, 14 (1971), pp. 713–719; Corrigendum, Comm. ACM, 15 (1972), p. 28.

[9] D. A. Zave, *A note on merge sort analysis*, in preparation.

[10] ———, *A simplified analysis of the optimal polyphase sort*, IEEE Trans. Software Engineering, to appear.

* The references consist of only those books and papers which are referenced in the text. An extensive bibliography on computer sorting has been prepared by R. L. Rivest and D. E. Knuth which appears in *Computing Reviews*, 13 (1972), pp. 283–289.

SIAM J. COMPUT.
Vol. 6, No. 1, March 1977

SINGLE MACHINE JOB SEQUENCING WITH PRECEDENCE CONSTRAINTS*

DONALD L. ADOLPHSON†

Abstract. We consider the problem of sequencing N jobs on a single machine when each job has a known processing time and a known deferral rate and a general precedence relationship exists among the jobs. The problem is to find the minimum cost sequence which is consistent with the precedence relationship. This problem has been solved in certain special cases obtained by restricting the form of the precedence graph. The results of this paper constitute a proper generalization of all previously solved special cases. These results are used to derive an alternate efficient approach to the case in which the precedence graph is a rooted tree, as well as an algorithm for attacking the general problem having an arbitrary acyclic precedence graph. The general algorithm has an $O(n^3)$ time bound and though it only gives partial solutions in some instances, the class of problems that it completely solves is a proper superset of the corresponding classes for all previous polynomial time algorithms.

Key words. job scheduling, precedence relations, graphs, networks, trees, analysis of algorithms

Introduction. We consider the problem of scheduling a set of jobs $N = \{1, 2, \cdots, n\}$ to be processed on a single machine. We assume that each job j has a known processing time t_j and a known deferral rate v_j. For a given permutation α of N, the finish time of job j, denoted $F_j(\alpha)$, is the processing time t_j plus the processing times of all jobs k which precede j in the ordering associated with the permutation α. The total cost of job j relative to the permutation α is $v_j \cdot F_j(\alpha)$. The objective is to find a permutation α of N, such that the total cost, given by $C(\alpha) = \sum_{j=1}^{n} v_j \cdot F_j(\alpha)$ is minimized.

If the n jobs are independent, then the optimal ordering may be determined by computing the ratio v_j/t_j for each job and then ordering the jobs from largest to smallest ratio. This result is due originally to Smith [8]. In this paper, we consider the above problem with the additional stipulation that a precedence relationship exists among the jobs. The precedence relationship may be represented by a directed acyclic graph G in which a directed path from node i to node j indicates that job i must precede job j in any feasible ordering. The problem now is to find the feasible ordering with the lowest cost. The special case in which G is a collection of rooted trees has been solved satisfactorily by Horn [5], by Adolphson and Hu [1], and in a slightly different context by Garey [3]. The results of [1] have been extended by Knuth [6] to the case where G is a series-parallel graph. Various other special cases have been treated by Sidney [7].

In this paper we study the general problem with an arbitrary acyclic precedence graph. In § 1 we prove a theorem which is a generalization of the main results of [1], [3], [5] and [6]. In § 2 we reexamine the rooted tree case and show how the main theorem leads to an $O(n \log n)$ algorithm. The approach here is entirely different than that of [1] in which another $O(n \log n)$ approach is given.

* Received by the editors January 3, 1975, and in final revised form February 20, 1976.

† Department of Finance, Business Economics and Quantiative Methods, University of Washington, Seattle, Washington 98105.

The algorithm given here has the advantages that it may require less storage and it is more easily generalized to an algorithm for the arbitrary acyclic graph case. The general algorithm is not guaranteed to give a complete solution but the class of problems that it does solve is a proper superset of the corresponding classes for all previous polynomial time algorithms. This extension is given in § 3 along with a verification of an upper bound of $O(n^3)$ on the time complexity of the problem. Section 4 provides a comparison of the results of this paper with results of other works on the same problem.

1. Basic results. In this section we establish the basic results which allow us to transform the precedence graph, G, without changing the optimal sequence. In order to do this we first need to introduce some terminology to be used throughout this paper.

We say that i precedes j, written $i \to j$, whenever there is a directed path from node i to node j in G. If $i \to j$ we also say that j succeeds i. The set of all nodes which precede node j is denoted by $P(j)$ and the set of all nodes which succeed node j is denoted $S(j)$. These definitions can be extended to subsets of nodes as follows: $P(J) = \bigcup_{j\in J} P(j)$ and $S(J) = \bigcup_{j\in J} S(j)$. Two nodes, i and j are said to be unrelated, written $i \sim j$, if $i \neq j$, $i \notin P(j)$ and $j \notin P(i)$. This definition can also be extended to sets of nodes as follows: $I \sim J$ if $I \cap J = \emptyset$, $I \cap P(J) = \emptyset$ and $J \cap P(I) = \emptyset$. If $i \to j$ and there is no node k such that $i \to k$ and $k \to j$, then i immediately precedes j and j immediately succeeds i. The sets of immediate predecessors and immediate successors of node j are given by $\bar{P}(j)$ and $\bar{S}(j)$, respectively. When it is convenient we will assume that the graph G has been augmented by a dummy initial node, 0, which precedes all other nodes and/or by a dummy terminal node, $n+1$, which succeeds all other nodes of G. We will assume that $v_0 = -\infty$, $v_{n+1} = \infty$, $t_0 = t_{n+1} = 1$. The notation for v_j, t_j and r_j can also be extended to subsets of nodes in the following manner: $v_J = \sum_{j\in J} v_j$, $t_J = \sum_{j\in J} t_j$, and $r_J = v_J/t_J$.

The following lemma establishes necessary but not sufficient conditions for a sequence α to be optimal.

LEMMA 1.1. *Let α be an optimal sequence with subsequence I immediately followed by subsequence J. If $I \sim J$, then $r_I \geqq r_J$. If $I \sim J$ and $r_I = r_J$, then the sequence α' obtained by interchanging I and J is also optimal.*

Proof. The importance of the assumption that $I \sim J$ is that the sequence α' obtained by interchanging I and J is also feasible. Since α is an optimal sequence, $C(\alpha) \leqq C(\alpha')$. In other words, in changing from α to α' the total increase in cost, $v_I \cdot t_J$, must be as large as the total decrease in cost, $v_J \cdot t_I$. Dividing both sides by $t_I \cdot t_J$ yields $r_I \geqq r_J$. If $r_I = r_J$, then the total increase equals the total decrease so that $C(\alpha') = C(\alpha)$. Q.E.D.

The following theorem provides justification for two important transformations of the precedence graph. This theorem constitutes a generalization of the main results of [1], [3], [5] and [6]. Intuitively the theorem tells us that if r_j is relatively large, then we want to schedule job j as soon as possible.

DEFINITION. A node j is *maximal* if

$$r_j = \max\{r_k \colon k \in S(\bar{P}(j)) \cup \bar{P}(j),\ k \notin S(j)\}. \tag{1.1}$$

THEOREM 1.2. *If j is maximal, then there exists an optimal sequence containing the subsequence (i, j) for some $i \in \bar{P}(j)$.*

Proof. Let α be an optimal sequence and let k be the node immediately preceding j in α. If $k \in \bar{P}(j)$, nothing remains to be proven. If $k \in S(\bar{P}(j))$, then since $k \notin S(j)$ we know that $k \sim j$. Hence from Lemma 1.1

$$r_k \geqq r_j. \tag{1.2}$$

Together (1.1) and (1.2) imply $r_k = r_j$ so that from Lemma 1.1 we can interchange k and j to obtain another optimal sequence in which node j appears earlier in the sequence.

If $k \notin S(\bar{P}(j)) \cup \bar{P}(j)$, then let K be the maximal subsequence immediately preceding j in α such that

$$K \cap [S(\bar{P}(j)) \cup \bar{P}(j)] = \varnothing. \tag{1.3}$$

Then if node i immediately precedes K in α, we know that $i \in S(\bar{P}(j)) \cup \bar{P}(j)$. Therefore $\{i\} \sim K$ and from Lemma 1.1

$$r_i \geqq r_K. \tag{1.4}$$

Also, from (1.3) we see that $K \sim \{j\}$ so that

$$r_K \geqq r_j. \tag{1.5}$$

Together (1.1), (1.4) and (1.5) imply $r_K = r_j$ so that by Lemma 1.1, we can interchange K and $\{j\}$ to obtain an optimal sequence in which j appears earlier in the sequence. After a finite number of interchanges, we must eventually find an optimal sequence satisfying the conditions of the theorem since j is always moved closer to the beginning. Q.E.D.

If node j is maximal and $\bar{P}(j) = \{i\}$, then the theorem tells us that there is an optimal sequence in which j immediately follows i. If this is true then we may think of jobs i and j as a single job with processing time $t_i + t_j$ and deferral rate $v_i + v_j$. This transformation changes the total cost of the optimal sequence but since it adds the fixed cost $v_i \cdot t_j$ to all sequences, the optimal sequence does not change. This transformation is summarized below:

T1. If node j is maximal and $\bar{P}(j) = \{i\}$, then we may condense node j into node i.

There is another transformation justified by Theorem 1.2 which may not be as obvious. Suppose node j is maximal but $|\bar{P}(j)| > 1$. Then we do not know which node i in $\bar{P}(j)$ immediately precedes j in an optimal sequence. Suppose however that all immediate predecessors of j have the same set of successors. Then Theorem 1.2 tells us that there must be an optimal ordering in which j precedes all other successors of $\bar{P}(j)$. An example of this is shown in Fig. 1.1. The original graph is given in Fig. 1.1(a) and an equivalent transformed graph is shown in Fig. 1.1(b). The transformation may be summarized as follows:

T2. If node j is maximal, $|\bar{P}(j)| > 1$, and $S(i) = S(\bar{P}(j))$ for all $i \in \bar{P}(j)$, then let j precede all other nodes of $S(\bar{P}(j))$ in the precedence graph.

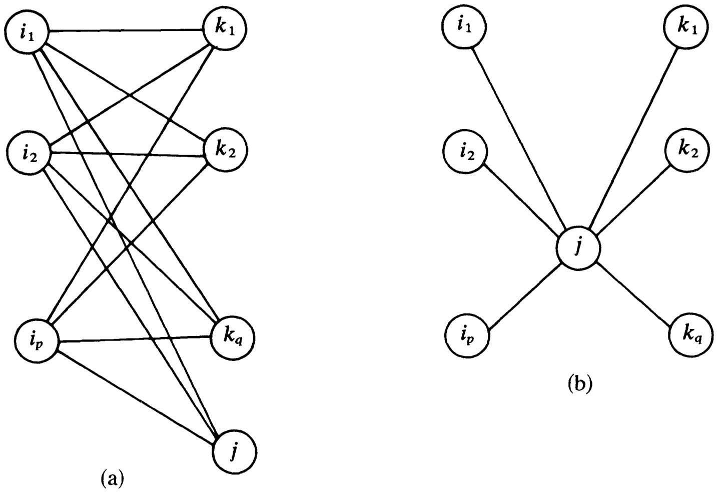


FIG. 1.1

By taking advantage of the natural symmetry of the single machine sequencing problem, we can think of sequencing jobs from last to first instead of from first to last. In this case, jobs with small ratios will tend to be scheduled as late as possible. The analogue of Theorem 1.2 is stated below without proof since the proof is entirely analogous to the proof of Theorem 1.2.

DEFINITION. A node j is *minimal* if

$$(1.1') \qquad r_j = \min\{r_k : k \in P(\bar{S}(j)) \cup \bar{S}(j), k \notin P(j)\}.$$

THEOREM 1.2′. *If node j is minimal, then there exists an optimal sequence containing the subsequence (j, i) for some $i \in \bar{S}(j)$.*

The transformations analogous to T1 and T2 are summarized below:

T1′. If node j is minimal and $\bar{S}(j) = \{i\}$ then we may condense node j into node i.

T2′. If node j is minimal, $|\bar{S}(j)| > 1$, and $P(i) = P(\bar{S}(j))$ for all $i \in \bar{S}(j)$, then let j succeed all other nodes of $P(\bar{S}(j))$ in the precedence graph.

We conclude this section with three examples using the four transformations developed in this section.

Example 1. Consider the graph shown in Fig. 1.2 with values v_j, t_j given by each node. All arcs are directed from left to right. We will use the notation

$$\hat{S}(j) = [S(\bar{P}(j)) \cup \bar{P}(j)] - S(j)$$

throughout these examples.

Initially we may apply T1 at nodes 2 and 5 and apply T1′ at node 6. As a result of these transformations, node 5 is condensed into the dummy initial node 0; node 6 is condensed into the dummy terminal node 8; and node 2 is condensed into node 1 so that the ratio for node 1 becomes $r_1 = (2+25)/(1+1) = 27/2$. At this point T1 may be applied at nodes 3, 4 and 7. Finally T1 may be applied at node 1. From this sequence of T1 and T1′ transformations we see that the optimal sequence is 5-4-7-1-2-3-6.

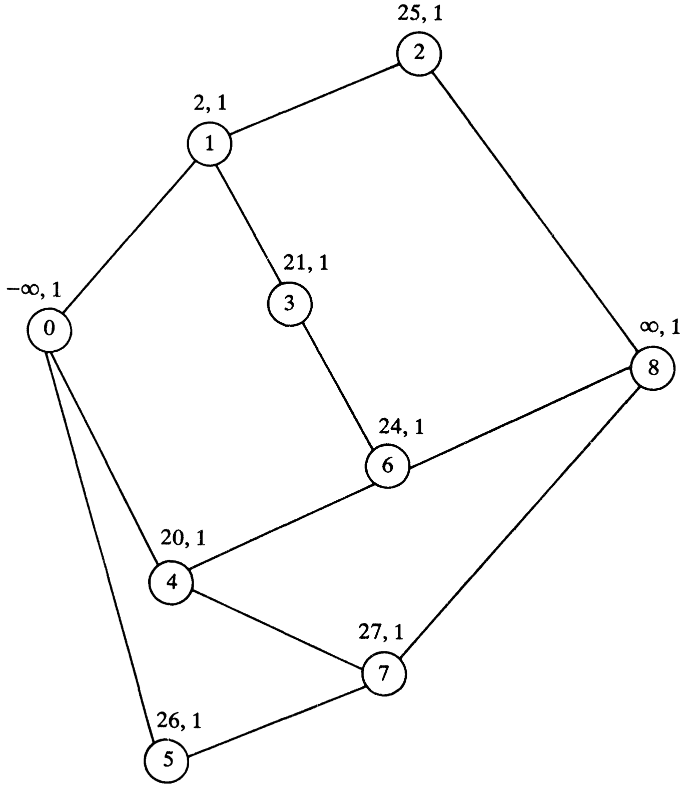


FIG. 1.2

Example 2. The previous example made no use of T2 or T2′. This example shows the need for these transformations. Consider the graph of Figure 1.3(a). It can be verified that neither T1 nor T1′ can be applied to this graph. However $r_5 = \max\{r_k : k \in \hat{S}_5\}$ and $S(1) = S(2) = \{4, 5, 6\}$ so that T2 may be applied. The resulting graph is shown in Figure 1.3(b). An optimal sequence 3-2-1-5-4 is then found by applying T1′ at node 1, and T1 at nodes 5, 4, 2 and 3.

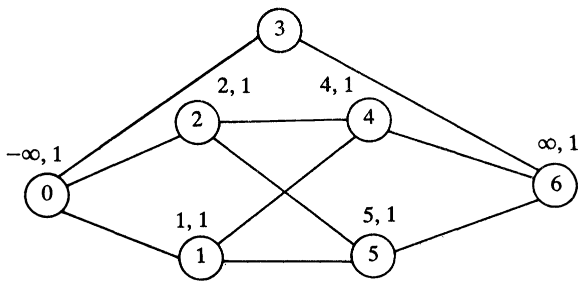


FIG. 1.3(a)

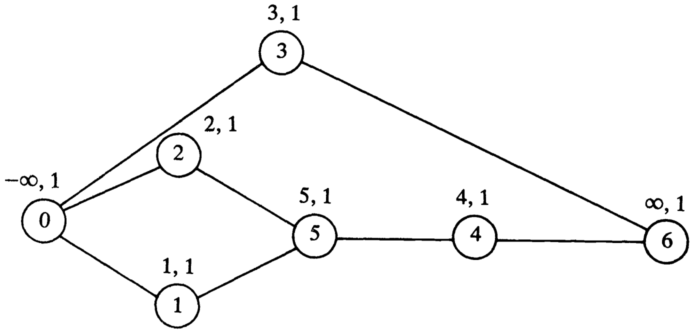


FIG. 1.3(b)

One purpose of T2 is to increase the number of nodes i satisfying $|\bar{S}(i)| = 1$. For example, T1′ may not be applied at node 1 in Fig. 1.3(a) since $|\bar{S}(1)| > 1$, but it can be applied at node 1 in Fig. 1.3(b).

Example 3. This example shows that in some cases none of the transformations T1, T2, T1′ or T2′ apply. The problem shown in Fig. 1.4 is due to Horn [5].

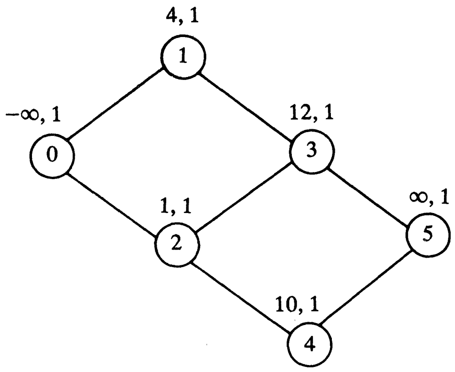


FIG. 1.4

Node 3 satisfies the conditions of Theorem 1.1 but since $|\bar{P}(3)| > 1$, T1 cannot be applied. In addition, T2 cannot be applied since $S(1) \neq S(2)$. Similar remarks apply to node 2 and transformations T1′ and T2′.

2. The rooted tree case. In this section we examine efficient methods of implementing the job sequencing algorithm when the graph G is a rooted tree or a collection of rooted trees. We discuss in some detail an implementation which can be extended in a natural way to the case of an arbitrary acyclic graph.

For our purpose, we define a rooted tree to be a graph such that for each node $j \in N$, $|\bar{P}(j)| = 1$. The single immediate predecessor of node j is called the father of j. An immediate successor of j is called a son of j. Any successor is called a descendant and any predecessor is called an ancestor. The only transformation needed for this case is a specialized version of T1 which is given below.

T3. Condense node j into node i whenever
(i) i is the father of j,
(ii) $r_j \geqq r_i$,
(iii) $r_j = \max\{r_k : K \text{ is a descendant of } i\}$.

First, we note that the algorithm must be capable of sorting n ratios so that a lower bound on the computation is $O(n \log n)$. There are two approaches to the problem, each of which achieves this lower bound.

The first approach is to scan the nodes of the tree from the leaves to the root. In other words, we never examine a node until we have examined all of its sons. An $O(n \log n)$ implementation using leftist trees to represent priority queues has been given by Adolphson and Hu [1]. The approach of working from the ends of the graph is also implicit in the work of Garey [3].

An alternate approach is to scan the nodes from largest ratio to smallest ratio. This approach may require less storage if appropriate data structures are employed and it can more easily be extended to testing for transformations T1, T2, T1′, T2′ on an arbitrary graph.

Using this approach it is helpful to think of the condensing of nodes in terms of set union operations. In this context the sets correspond to the chains of nodes from the original graph which have been condensed into a single node in the current graph. The algorithm is initiated with $n+1$ singleton sets corresponding to nodes $0, 1, \cdots, n$. After n set unions, the algorithm terminates with a single set consisting of all nodes $0, 1, \cdots, n$.

In the description of the algorithm which follows, the chain of nodes beginning with node i will be denoted by B_i. The last node in B_i will be denoted by E_i. F_i gives the father of node i, initially; at termination, F_i gives the immediate predecessor of job i in the optimal sequence discovered by the algorithm. F_0 is a pointer to the last job in the sequence.

The algorithm to be presented may be justified as follows. At each step we find the node in the current graph having the largest ratio. The conditions of T3 must hold for this node so we proceed to condense it into its father in the current graph. In order to find the father in the current graph we need to find the first node in the chain of nodes containing the father in the original graph. The large node is then condensed into the current father and the process is repeated. The input to the algorithm is v_i, t_i and F_i.

The rooted tree algorithm.

Step 0. $v_0 \leftarrow -\infty$, $t_0 \leftarrow 1$, $(r_i \leftarrow v_i/t_i,\ i = 0, \cdots, n)$
$(E_i \leftarrow i$ and $B_i = \{i\},\ i = 0, 1, \cdots, n)$

Step 1. Let j be the remaining node with the largest ratio. If $j = 0$, go to Step 4. Otherwise, delete j from the list, $k \leftarrow F_j$, and FIND i such that $k \in B_i$.

Step 2. $v_i \leftarrow v_j + v_j$

$t_i \leftarrow t_i + t_j$

$r_i \leftarrow v_i/t_i$

$F_j \leftarrow E_i$

$E_i \leftarrow E_j$.

Step 3. $B_i \leftarrow B_i \cup B_j$;
Go to Step 1.
Step 4. $F_0 \leftarrow E_0$.

Before analyzing the algorithm, we present a numerical example. Consider the tree with v_j, t_j given by each node in Fig. 2.1. This example is due to Horn [5].

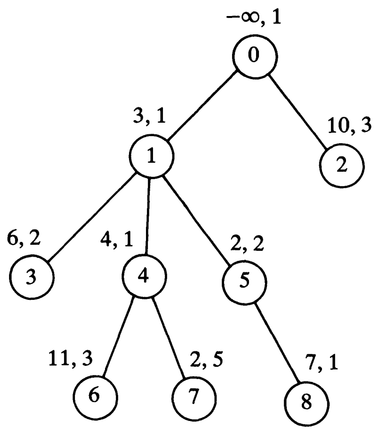


FIG. 2.1

We summarize the computations in the following manner.

Iteration 1

$$\begin{array}{l} j=8 \\ k=5 \\ i=5 \end{array} \Rightarrow \begin{cases} r_5 \leftarrow 9/3 \\ F_8 \leftarrow 5 \\ E_5 \leftarrow 8 \\ B_5 \leftarrow \{5, 8\} \end{cases}$$

Iteration 2

$$\begin{array}{l} j=4 \\ k=1 \\ i=1 \end{array} \Rightarrow \begin{cases} r_1 \leftarrow 7/2 \\ F_4 \leftarrow 1 \\ E_1 \leftarrow 4 \\ B_1 \leftarrow \{1, 4\} \end{cases}$$

Iteration 3

$$\begin{array}{l} j=6 \\ k=4 \\ i=1 \end{array} \Rightarrow \begin{cases} r_1 \leftarrow 18/5 \\ F_6 \leftarrow 4 \\ E_1 \leftarrow 6 \\ B_1 \leftarrow \{1, 4, 6\} \end{cases}$$

Iteration 4

$$\begin{array}{l} j=1 \\ k=0 \\ i=0 \end{array} \Rightarrow \begin{cases} r_0 \leftarrow -\infty \\ F_1 \leftarrow 0 \\ E_0 \leftarrow 6 \\ B_0 \leftarrow \{0, 1, 4, 6\} \end{cases}$$

Iteration 5
$$\begin{array}{l} j=2\\ k=0\Rightarrow\\ i=0\end{array}\begin{cases} r_0\leftarrow -\infty\\ F_2\leftarrow 6\\ E_0\leftarrow 2\\ B_0\leftarrow\{0,1,4,6,2\}\end{cases}$$

Iteration 6
$$\begin{array}{l} j=5\\ k=1\Rightarrow\\ i=0\end{array}\begin{cases} r_0\leftarrow -\infty\\ F_5\leftarrow 2\\ E_0\leftarrow 8\\ B_0\leftarrow\{0,1,4,6,2,5,8\}\end{cases}$$

Iteration 7
$$\begin{array}{l} j=3\\ k=1\Rightarrow\\ i=0\end{array}\begin{cases} r_0\leftarrow -\infty\\ F_3\leftarrow 8\\ E_0\leftarrow 3\\ B_0\leftarrow\{0,1,4,6,2,5,8,3\}\end{cases}$$

Iteration 8
$$\begin{array}{l} j=7\\ k=4\Rightarrow\\ i=0\end{array}\begin{cases} r_0\leftarrow -\infty\\ F_7\leftarrow 3\\ E_0\leftarrow 7\\ B_0\leftarrow\{0,1,4,6,2,5,8,3,7\}\end{cases}$$

Step 4. $i=0$
$F_0\leftarrow 7$.

The optimal sequence 0-1-4-6-2-5-8-3-7 is then determined by walking through the list F and reversing the order.

It is not difficult to show that the algorithm is $O(n \log n)$. The total time spent on Step 2 is clearly $O(n)$. Step 1 involves deleting an element from a priority queue as well as accounting for the change in value of r_i at the preceding iteration. Both of these operations require at most $O(\log n)$ steps if a heap is used to represent the priority queue. Since Step 1 is executed n times, the total amount of time spent on priority queue operations is $O(n \log n)$. Finally we examine the total time involved in set processing. Step 1 requires n FIND operations and Step 4 requires an additional trivial FIND. Step 3 involves n UNION operations. Thus the set processing involves n UNION operations with $n+1$ intermixed FIND operations. This is precisely the problem studied by Hopcroft and Ullman [4], and more recently analyzed by Tarjan [9]. It is shown by Tarjan that the total time required is $O(n \log n)$ or better using appropriate versions of the set-union algorithm. Hence the total time required is $O(n \log n)$.

3. The general algorithm. In this section we extend the ideas of the preceding section to develop an efficient method of testing for and applying the reduction transformations. There are some important differences which should be noted between the algorithm for the general graph as opposed to the algorithm for the rooted tree. First of all, the algorithm of this section is not guaranteed to find the optimal sequence. The algorithm terminates when either the optimal sequence is found or when none of the transformations can be applied. The example shown in Fig. 1.4 is an example of the latter. A second difference is that we need to scan the nodes of the graph in both directions. That is, we scan the nodes from largest to smallest looking for applications of T1 and T2 and then we scan the nodes from smallest to largest looking for applications of T1′ and T2′. A third difference is that not all transformations can be detected in a single scan of the nodes so that we need to continue scanning the nodes until all nodes have been condensed into the dummy nodes or until we are able to scan the nodes in both directions without applying any transformations.

Before describing the details of the algorithm and verifying an $O(n^3)$ bound on the time complexity, we shall give an outline of the algorithm.

The general algorithm.

Step 0. Compute ratios $r_j = v_j/t_j$ and initialize working matrices.

Step 1. Scan the nodes from largest to smallest ratio checking for T1 and T2. Apply the appropriate transformation whenever possible.

Step 2. Scan the nodes from smallest to largest ratio, checking for T1′ and T2′. Apply the appropriate transformation whenever possible.

Step 3. If all nodes have been condensed, then output the optimal sequence and stop.

If no transformations are applied at Steps 1 and 2, output information about condensed nodes and stop.

Otherwise, return to Step 1.

The algorithm described here uses four $n \times n$ working matrices. Given an adjacency matrix for the precedence graph G we can generate matrices $A = [a_{ij}]$ and $\bar{A} = [\bar{a}_{ij}]$ where

$$a_{ij} = \begin{cases} 1 & i \in P(j), \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad \bar{a}_{ij} = \begin{cases} 1 & i \in \bar{P}(j), \\ 0 & \text{otherwise.} \end{cases}$$

Both matrices can be constructed in no more than $O(n^3)$ time. (See [2].)

The algorithm will use a matrix $B = [b_{ij}]$ to keep track of subset relationships among sets of successors. The elements of the matrix are defined as

$$b_{ij} = \begin{cases} |S(j) - S(i)| & \text{if } S(i) \subseteq S(j), \\ -1 & \text{otherwise.} \end{cases}$$

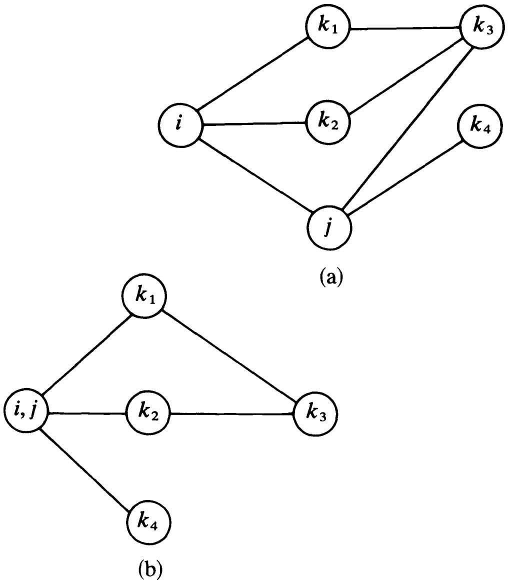
A matrix $B' = [b'_{ij}]$ is defined in an analogous manner to keep track of subset relationships between sets of predecessors. The matrices B and B' will be used in checking for transformations T2 and T2′ respectively. Each element b_{ij} can be determined from a single pass through row i and row j of A so that the total time required to construct B and B' is $O(n^3)$.

Since both T1 and T2 operate on maximal nodes we need an efficient method of finding maximal nodes. One way to accomplish this is to scan the nodes from largest to smallest. Every time a node k is scanned but not condensed we increment a counter, $C(i)$, for all $i \in P(k) \cup \{k\}$. If j is the largest node not yet scanned, then the current values of $C(i)$ represent the number of nodes in $S(i) \cup \{i\}$ having a ratio strictly greater than r_j. If $C(i) > C(j)$ for some node $i \in \bar{P}(j)$, then there must be a node $k \in S(i) \cup \{i\} - S(j) - \{j\}$ with $r_k > r_j$; hence j is not maximal. On the other hand if $C(i) = C(j)$ for all $i \in \bar{P}(j)$, then every large node succeeding i also succeeds j so that j is a maximal node. Some problems may arise if several nodes have the same ratio, but these problems can be avoided if we wait until all nodes with a given ratio have been scanned before we increment the counters.

Using the above procedure, the time required to scan a node and determine if it is maximal and perform the associated increments is $O(n)$. If the current node j is maximal, we can determine if $|\bar{P}(j)| = 1$ in $O(n)$ time. If $|\bar{P}(j)| > 1$, then we can check for transformation T2 by scanning column j of $\bar{A}$ and row k of B for some $k \in \bar{P}(j)$. If T2 is to be applied then b_{ik} must be zero (indicating $S(i) = S(k)$) whenever $\bar{a}_{ij} = 1$ (indicating $i \in \bar{P}(j)$). This check also requires $O(n)$ time. Thus the total time spent on checking for T1 and T2 is $O(n)$ for each node scanned. The same result clearly holds also for T1′ and T2′.

In carrying out the transformations, we assume that the matrices remain $n \times n$ matrices and a list of rows and columns to be ignored will be maintained. This means that each time a matrix entry is checked we must check to see if it should be ignored. This multiplies the work only by a constant factor.

A T1 transformation requires no change of the A matrix but some care is required to update the $\bar{A}$ matrix. Consider the partial graph shown in Fig. 3.1.



(a)

(b)

FIG. 3.1

Note that both k_3 and k_4 are in $\bar{S}(j)$ in Fig. 3.1(a). However, $k_3 \notin \bar{S}(i)$ and $k_4 \in \bar{S}(i)$ in Fig. 3.1(b). The reason for this is that k_1 (or k_2) is a successor of i and a predecessor of k_3. In general we need to find the set of nodes K defined by

$$K = S(\bar{S}(i) - \{j\}).$$

K can be determined in $O(n^2)$ time by making a single pass through each row of A corresponding to a node in $\bar{S}(i) - \{j\}$. We may now update row i of $\bar{A}$ by the following

$$\bar{a}_{ik} \leftarrow \begin{cases} 1 & \text{if } \bar{a}_{ik} = 1 \quad \text{or} \\ & \text{if } \bar{a}_{jk} = 1 \quad \text{and } k \notin K, \\ & \\ 0 & \text{otherwise.} \end{cases}$$

The matrix B may be updated after a T1 transformation as follows. If T1 is applied at node j, node j is removed from all successor sets containing node j. For an arbitrary node i, the cardinality of $S(i)$ is now determined by

$$|S(i)| \leftarrow |S(i)| - a_{ij}.$$

Consider an element b_{ik} of B. If $b_{ik} \geqq 0$, then the new b_{ik} is given by

$$\begin{aligned} b_{ik} = |S(k)| - |S(i)| \leftarrow & |S(k)| - a_{kj} - (|S(i)| - a_{ij}) \\ & = |S(k)| - |S(i)| - a_{kj} + a_{ij} \\ & = b_{ik} - a_{kj} + a_{ij}. \end{aligned}$$

On the other hand, if $b_{ik} = -1$, then this will change only if $S(i) - S(k) = \{j\}$ in which case the new b_{ik} will be zero. The above condition occurs only if $b_{ki} = 1$, $a_{ij} = 1$ and $a_{kj} = 0$. In either case we see that the time required to update an element of B is bounded by a constant so that the amount of time required to update the entire matrix is bounded by $O(n^2)$. An analogous procedure may be used for B' which also requires $O(n^2)$ time.

We see from the above discussion that the time required to update the working matrices whenever T1 is applied is bounded by $O(n^2)$. A similar argument clearly holds for T1′. Since each T1 or T1′ transformation reduces the number of nodes, the number of the T1 and T1′ transformations is $O(n)$. Therefore the total amount of time spent updating the working matrices after T1 and T1′ transformations is bounded by $O(n^3)$.

When a T2 transform is made, the following three sets are affected:

1. $I = \bar{P}(j)$,
2. $\{j\}$,
3. $K = \bar{S}(I) - \{j\}$.

Each of these three sets can be constructed in $O(n^2)$ time or less. The changes which take place are:

1. $\bar{S}(i) = \{j\}$ for all $i \in I$,
2. $\bar{P}(k) = \{j\}$ for all $k \in K$,
3. $P(k) \leftarrow P(k) \cup \{j\}$ for all $k \in K \cup S(K)$,
4. $S(j) \leftarrow S(j) \cup K \cup S(K)$.

The first two changes require changing a single row and column of $\bar{A}$. The third and fourth involve changing a single row and column of A. The third change also requires changing row j of B' since $P(j) \subseteq P(k)$ for all $k \in K \cup S(K)$. The fourth change also requires changing column j of B since $S(k) \subseteq S(j)$ for all $k \in K \cup S(K)$. Since each element of B or B' can be constructed from A in $O(n)$ time, and $O(n)$ elements are changed, we see that the total updating time in a T2 transformation is $O(n^2)$.

In order to complete the analysis we need to establish a bound on the number of T2 transformations which can occur. Let D be the set of nodes having more than one immediate successor. First we note that a T1 or T1′ transformation can never add a node to D. Next we note that we only apply T2 if $\bar{P}(j) \subseteq D$ and $|\bar{P}(j)| \geqq 2$. After the T2 transformation, the whole set $\bar{P}(j)$ containing at least two elements is removed from D. The only node which could be added to D after the transformation is j. In any case the cardinality of D will strictly decrease so that the number of T2 transformations is bounded by $O(n)$. A similar argument holds for T2′ transformations. We can therefore conclude that since no more than $O(n^2)$ time is spent on any T2 or T2′ transformation, the total time spent on T2 and T2′ transformations is $O(n^3)$.

The number of iterations of the main loop of the program is bounded by the number of transformations which has been shown to be $O(n)$. The number of nodes scanned in any iteration is bounded by $2n = O(n)$. It was shown earlier that the amount of time spent checking a single node for a T1 or T2 transformation is $O(n)$. The total amount of time spent adjusting priority queues was shown in the previous section to be $O(\log n)$ per node if appropriate data structures are used. Therefore the total amount of time spent scanning the nodes and maintaining priority queues is:

$$O(n^2)[O(n)+O(\log n)] = O(n^2) \cdot O(n) = O(n^3).$$

The analysis of the set processing changes very little from the analysis of the previous section. Therefore we can conclude that the total time for the algorithm is bounded by $O(n^3)$.

The algorithm given here is not necessarily the most efficient. We have verified an upper bound of $O(n^3)$ but it may be possible to reduce this somewhat through the use of appropriate data structures. The limiting factor may well be the time required to generate the transitive closure and/or the transitive reduction of G. The time complexity of these processes is shown to be equivalent to the time complexity of Boolean matrix multiplication and the same bound may very well apply to the algorithm here. The best bound known for those problems is $O(n^{\log_2 7})$. (See [2].) These questions are left here as open questions to the reader.

4. Comparison with other results. In this section we shall investigate and partially characterize the class of sequencing problems which can be solved by the algorithm of the preceding section. The results will be compared with other work on this sequencing problem.

An instance of the sequencing problem discussed in this paper is given by a triple (G, v, t) where G is an acyclic graph, v and t are real functions over the nodes of G. We will let P_1 be the class of problems solved by the algorithm of [1] or [5]; we will let P_2 be the class of problems solved by the algorithm of [6]; we will let P_3 be the class of problems solved by the algorithm of [3]; and we will let P_4 be the class of problems solved by the algorithm of the preceding section.

The subset relationships among these four sets is illustrated graphically in Fig. 4.1 where a directed path indicates a subset relationship. We see from this figure that $P_1 \subseteq P_2 \subseteq P_4$ and $P_1 \subseteq P_3 \subseteq P_4$ but $P_2 \not\subseteq P_3$ and $P_3 \not\subseteq P_2$. The relationships are elucidated in the discussion which follows.

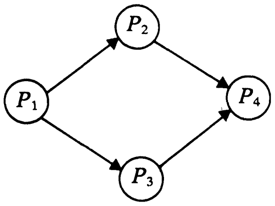


FIG. 4.1

The algorithms of [1] and [5] are designed to operate on rooted trees and it can be shown that $(G, v, t) \in P_1$ if and only if G is a collection of rooted trees. The algorithm of [6] is designed to work on series-parallel graphs and it can be shown that $(G, v, t) \in P_2$ if and only if G is a series-parallel graph. Since any collection of rooted trees can be converted to an equivalent series-parallel graph by adding dummy initial and terminal nodes, $P_1 \subseteq P_2$. The algorithm of [3] is the only previous work which deals directly with an arbitrary acyclic graph. It is shown in [3] that $(G, v, t) \in P_3$ whenever G is a collection of rooted trees so that $P_1 \subseteq P_3$.

We see from the above that P_2 and P_3 both include P_1. The triple shown in Fig. 4.2 is an example of a series-parallel graph which cannot be solved by the algorithm of [3]; hence $P_2 \not\subseteq P_3$.

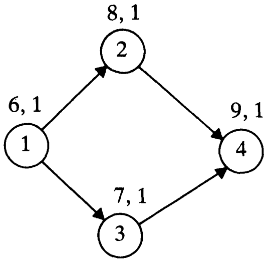


FIG. 4.2

If we change v_4 to 13 in Fig. 1.4, then we have an instance of the sequencing problem which can be solved by the algorithm of [3] even though the graph is not series parallel; hence $P_3 \not\subseteq P_2$.

As was mentioned earlier, Theorem 1.2 and its analogue, Theorem 1.2′, constitute a generalization of the main results of [1], [3], [5] and [6]. Hence P_4 includes all of the other sets. Furthermore, the example of Fig. 1.2 cannot be solved by any of the previous methods so that P_4 properly includes all other classes.

Another way to compare the results is to examine the class of graphs such that the sequencing problem is solved for any functions v and t. This class is known to be collections of rooted trees for P_1 and series parallel graphs for P_2. This class is not known for P_3 although it is shown that the class includes collections of rooted trees but does not include all series-parallel graphs. It has been shown that this class of graphs for P_4 includes all series-parallel graphs. Furthermore, the graph of Fig. 1.3, which is not series-parallel, may be solved for any v and t functions so that the class of graphs, which are always solved by the algorithm of this paper, properly includes the class of all series-parallel graphs. The problem of characterizing completely these graphs is left here as an open problem.

In summary we have proven a result (Theorem 1.2) which constitutes a proper generalization of all previous results on the single machine sequencing problem with precedence constraints. We have used these results to give an efficient algorithm for the rooted tree case and an $O(n^3)$ algorithm for the general case which determines the optimal sequence for a broad class of problems but not for the whole class of sequencing problems considered here. It is the opinion of this author that the general problem is polynomial complete, but this question will also be left as an open question for the reader.

REFERENCES

[1] D. Adolphson and T. C. Hu, *Optimal linear ordering*, SIAM J. Appl. Math., 25 (1973), pp. 403–423.

[2] A. V. Aho, M. R. Garey and J. D. Ullman, *The transitive reduction of a directed graph*, this Journal, 1 (1972), pp. 131–137.

[3] M. R. Garey, *Optimal task sequencing with precedence constraints*, Discrete Math., 4 (1973), pp. 37–56.

[4] J. Hopcroft and J. D. Ullman, *Set merging algorithms*, this Journal, 2 (1973), pp. 294–303.

[5] W. A. Horn, *Single-machine job sequencing with treelike precedence ordering and linear delay penalties*, SIAM J. Appl. Math., 23(2) (1972), pp. 189–202.

[6] D. E. Knuth, Private communication.

[7] J. B. Sidney, *Decomposition algorithms for single-machine sequencing with precedence relations and deferral costs*, J. Operations Res. Soc. Amer., 23 (1975), pp. 283–298.

[8] W. W. Smith, *Various optimizers for single stage production*, Naval Res. Logist. Quart., 3 (1956), pp. 59–66.

[9] R. E. Tarjan, *Efficiency of a good but not linear set union algorithm*, J. Assoc. Comput. Mach., 22 (1973), pp. 215–225.

SIAM J. COMPUT.
Vol. 6, No. 1, March 1977

SMALLEST AUGMENTATIONS TO BICONNECT A GRAPH*

ARNIE ROSENTHAL and ANITA GOLDNER†

Abstract. We provide an $O(|V|+|E|)$ algorithm which, given a graph G, finds a smallest set of edges which, when added to G, produces a graph with no cutpoints.

Key words. graph augmentation, biconnected graph, block-cutpoint graph, graphical connectivity, linear time algorithms

1. Introduction. A graph which cannot be separated into two or more disjoint subgraphs by the removal of any single point and the edges incident to it is said to be *biconnected*. Any graph can be biconnected by the addition of appropriate lines. A set $\mathscr{A}$ of edges such that $G \cup \mathscr{A}$ is biconnected is called a *biconnection* of G. We present here a fast algorithm to find a smallest biconnection of an arbitrary graph. This problem is one of a class of graphical augmentation problems investigated by Eswaran and Tarjan [2].

The speed of an algorithm is described by its *time complexity*. Let x be some variable whose value is related to the "size" of the problem, e.g., the number of vertices of the graph, in an algorithm that concerns graphs, and let $f(x)$ be a function of x. The time complexity of an algorithm is $O(f(x))$ (read, "order $f(x)$") if there exists a constant k such that the algorithm can be performed in $kf(x)$ steps or less. Note that if a finite sequence of $O(f(x))$ algorithms were run, the total time would still be $O(f(x))$.

Given a graph with vertex set V and edge set E, we show our algorithm to be $O(|V|+|E|)$.

DEFINITIONS. A *graph* $G=(V,E)$ is a finite set of *vertices* (*points*), V, and a finite set of *edges* (*lines*) E. The edges are unordered pairs (u, v) of distinct vertices. Two vertices u, v which comprise an edge are said to be *adjacent*; this is denoted $u \underset{G}{\sim} v$, or simply $u \sim v$. The *degree* of a vertex is the number of vertices adjacent to it. This is written $d_G(v)$, or just $d(v)$. A *path* in G from v_1 to v_n is a sequence of edges $(v_1, v_2), \cdots, (v_{n-1}, v_n)$, such that the vertices $v_1, \cdots, v_n$ are distinct. A *cycle* is a path $(v_1, v_2), \cdots, (v_{n-1}, v_n)$ together with the edge (v_n, v_1). G is *connected* if and only if every pair of vertices of G is joined by a path in G. G is a *tree* if G is connected and acyclic. The maximal connected subgraphs of G are its *components*. A *forest* is a graph the components of which are all trees. A degree-1 vertex of a forest is called a *leaf*; all vertices of degree 2 or more are *interior*; vertices of degrees 0 and 1 are *exterior*. A vertex v of a connected graph G whose removal disconnects G is a *cutpoint*. If G has no cutpoints, then G is said to be *biconnected*. A maximal biconnected subgraph of G is a *block*.

Central to the algorithm is the concept of the *block-cutpoint graph* of a graph G, denoted bc(G). Each block and each cutpoint of G is represented by a vertex of bc(G). We call the vertices of bc(G) which represent blocks its

* Received by the editors August 7, 1975, and in revised form March 2, 1976.

† Department of Computer and Communications Sciences, University of Michigan, Ann Arbor, Michigan 48104. This work was supported in part by Grant AFOSR 71-2076.

b-*vertices*, and those representing cutpoints its c-*vertices*. (If v is a point of bc(G), $\bar{v}$ will sometimes be used to denote the corresponding block or cutpoint of G.) Two vertices u, v of bc(G) are adjacent if and only if $\bar{u}$ is a cutpoint contained in the block $\bar{v}$ or vice versa. It can easily be shown [3, p. 37] that bc(G) is always a forest; it will be known as the bc-tree of G when G is connected.

We shall biconnect a graph G indirectly, by biconnecting bc(G), and adding "corresponding" edges to G. Let $x = (j, k)$ where j and k are exterior b-vertices of T. Then $\bar{x}$ will denote an arbitrary edge added to G which corresponds to x in the sense that $\bar{x} = (u, v)$, where u and v are noncutpoints belonging to $\bar{j}, \bar{k}$ respectively. (A block corresponding to an exterior vertex must contain a noncutpoint. Thus we have defined a bar notation for vertices, and one with a somewhat different interpretation for edges).

An *endblock* of a graph is a block which contains exactly one cutpoint. We note that a block of a graph G is an endblock if and only if the corresponding b-vertex of bc(G) is a leaf of bc(G).

Notation.

V	The set of vertices of a graph G.
E	The set of edges of a graph G.
bi(G)	The smallest number of lines needed to biconnect G.
$G+x$	The graph $(V, E \cup \{x\})$, where $G = (V, E)$.
$G-v$	The maximal subgraph of G not containing the vertex v.
GUA	The graph $(V, E \cup A)$, where $G = (V, E)$ and A is set of edges.
B_G	A biconnected graph containing G as a subgraph.
L_T	The number of leaves of T, a tree.
$q(G)$	The number of blocks of G, excluding G itself.
C_T	The set of c-vertices of the bc-tree, T.
$u \underset{G}{\sim} v$	Vertices u and v joined by an edge of G.
$d_G(v)$	The degree of vertex v in G.
bc(G)	The block-cutpoint graph of G.

In [2], Eswaran and Tarjan prove a lower bound on the number of edges in a minimal biconnection of a graph G and that the bound can always be attained [2, Thms. 6, 7]. Their results may be stated in the following way.

Let G be a graph with n components, and let $T = \text{bc}(G)$. Let x denote the smallest integer not less than x. Then $\text{bi}(G) = \max_{v \in C_T} [d_T(v) - 1 + (n-1), q(G) + \lceil L_T/2 \rceil]$.

In the following, we provide our own proof that this bound on the size of a biconnection can be attained, and we construct a linear-time algorithm based on the method of our proof. We first state two lemmas. Their proofs are straightforward, and hence omitted.

2. Theorems and proofs.

LEMMA 1. *Let G be a graph with n components, and let S be a set of edges such that $G \cup S$ is connected. Then S has a subset S' of $n-1$ edges such that $G \cup S'$ is connected. Furthermore no set of fewer than $n-1$ edges will connect G.*

Let $q(G)$ be the number of components of G (not including G itself) which are already blocks.

LEMMA 2. *Let $T = \mathrm{bc}(G)$, G not connected. Let $G' = G + x$, where x is an edge joining two of G's components. Let $T' = \mathrm{bc}(G')$. Then $q(G') + \lceil L_{T'}/2 \rceil \geqq q(G) + \lceil L_T/2 \rceil - 1$.* (The proof distinguishes three cases, depending on the number, 0, 1, or 2, of biconnected components to which x is incident. We note that equality obtains whenever neither end of x is a cutpoint).

We proceed by (Theorem 1) verifying the result of Eswaran and Tarjan for connected graphs satisfying the condition

$$(*) \qquad d_T(v) - 1 \leqq \left\lceil \frac{L_T}{2} \right\rceil \quad \text{for all } v \in C_T.$$

Next (Theorem 2), we verify it for all connected graphs, and finally (Theorem 3), for all possible graphs.

We will need to consider two types of connected graphs. In the first, the degree of all c-vertices in T is bounded by $\lceil L_T/2 \rceil + 1$. These graphs can be biconnected by adding edges which join appropriate pairs of endblocks, according to the Path Algorithm of Theorem 1. (See Fig. 1.) In the second type, some cutpoint has a large enough degree in T to violate $(*)$. The graph is biconnected (as is explained more precisely in Theorem 2) by first linking the "surplus" blocks of v^* (e.g., e_1, e_2, e_3 in Fig. 2) so that v^* no longer violates $(*)$, and then proceeding as in the first case (adding e_4 and e_5 in the example).

In disconnected graphs, we first connect the graph by adding edges to exterior vertices of the bc-tree. Then the scheme above is executed.

We prove two lemmas about trees.

LEMMA 3. *In any tree T with vertex set V, $L_T = 2 + \sum_{v \in V} \max[0, d_T(v) - 2]$.*

Proof. Let T have n vertices, $n \geqq 2$. Then T has $n-1$ edges. Therefore, $\sum_{v \in V} d_T(v) = 2(n-1)$. If v is a leaf node, $\max[0, d_T(v) - 2] = 0 = d_T(v) - 2 + 1$. If v is not a leaf, $\max[0, d_T(v) - 2] = d_T(v) - 2$. Hence, $\sum \max[0, d_T(v) - 2] = \sum d_T(v) - 2n + L_T = 2(n-1) - 2n + L_T = L_T - 2$, giving us the desired result.

LEMMA 4. *Suppose $L_T > 2$. Then T can contain at most two vertices z_1, z_2 satisfying*

$$(**) \qquad d_T(v) - 1 \geqq \left\lceil \frac{L_T}{2} \right\rceil.$$

If it contains two, then no other vertex has degree greater than 2, and the relation is satisfied with equality.

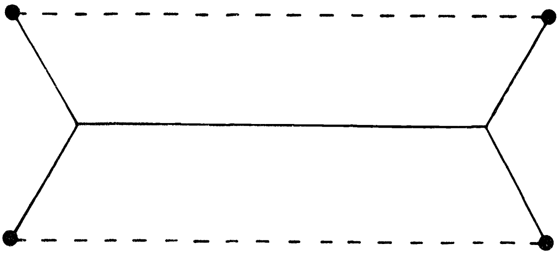


FIG 1. *Biconnection of a graph by the method of Theorem* 1

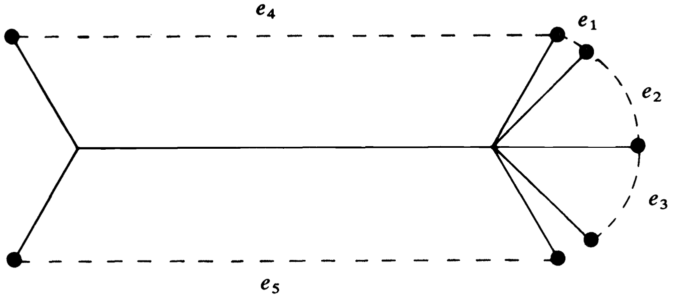


FIG. 2. *Biconnection of a graph by the method of Theorem* 2. (*After the addition of* e_1, e_2, e_3, *the altered* bc-*tree is the graph of Fig.* 1, *before biconnection.*)

Proof. Suppose z_1, z_2 satisfy $(**)$. Then, by Lemma 3,

$$(d(z_1)-1)+(d(z_2)-1)\geqq 2\left\lceil\frac{L_T}{2}\right\rceil \geqq L_T=2+\sum_v \max[0, d(v)-2]$$

$$=2+(d(z_1)-2)+(d(z_2)-2)+\sum_{v\neq z_1,z_2}\max[0, d(v)-2].$$

Using the first and last of the above expressions, we get, after simplifying,

$$0\geqq \sum_{v\neq z_1,z_2}\max[0, d(v)-2].$$

Therefore, $d(v)\leqq 2$ for all v but z_1, z_2.

In view of Lemma 4, in any tree it is possible to find a path P which includes any vertices that satisfy condition $(**)$ and any given vertex b^*, and two leaves.

THEOREM 1. *Let G be a connected graph such that*

$$(*) \qquad d_T(v)-1\leqq\left\lceil\frac{L_T}{2}\right\rceil \quad \textit{for all } v\in C_T.$$

Then G can be connected by the addition of $\lceil L_T/2\rceil$ *lines.*

Proof. The proof is by induction on $\lceil L_T/2\rceil$. If $\lceil L_T/2\rceil$ equals zero, the graph is already a block, so the theorem holds. Suppose the theorem holds for all G such that $\lceil L_T/2\rceil\leqq k$, some $k\geqq 0$. Suppose that for some G, $\lceil L_T/2\rceil = k+1$. We show that by the addition of one line via the Path Algorithm below, we can obtain a graph G' such that $L_{T'}=L_T-2$ and $(*)$ holds so that G' can be biconnected using the induction hypothesis. (The proof of Theorem 1 will be continued.)

PATH ALGORITHM. Let T be given. (See Fig. 3.)

Step 1. Find a path $P=(a_1, a_2), \cdots, (a_{m-1}, a_m)$, (a_m, b^*), (b^*, a_{m+1}), $\cdots, (a_{p-1}, a_p)$ such that

A1. a_1 and a_p are leaves of T.

A2. Any c-vertex satisfying $(*)$ with equality is on P.

A3. b^* is a given b-vertex of T.

(Lemma 4 assures us that such a path may be found in any tree.)

Step 2. Form T' from T as follows: Merge all b-vertices and degree-2 c-vertices on P into a single b-vertex, given the name b^*. Let b^* be adjacent to exactly those c-vertices which are adjacent to at least one vertex on P and to at least one vertex not on P. Note that for each c-vertex v of P which is not merged, $d_{T'}(v) = d_T(v) - 1$. For c-vertices *not* on P, $d_{T'(v)} = d_T(v)$.

What modification of G corresponds to performing the Path Algorithm on T?

PROPOSITION 1.1. *T' is the block-cutpoint tree of $G' = G + \bar{x}$, where $x = (a_1, a_p)$ and $\bar{x}$ is a corresponding edge of G'.*

Proof. First notice that the blocks of G' are indeed "B^*" (the union of $\bar{x}$ and of all blocks of G corresponding to b-vertices on P), and those blocks corresponding to b-vertices not on P. Its cutpoints are all those cutpoints of G which are adjacent to at least one block not represented on P. So there is a one-to-one correspondence betweeen points of T' and the set of blocks and cutpoints of G'. (The block B^* corresponds to b^*).

By the construction, the vertex b^* of T' is adjacent to exactly those c-vertices corresponding to cutpoints of G' contained in B^*. Each of those c-vertices is in turn adjacent in T' only to b^* and to the b-vertices corresponding to its other blocks in G'. Furthermore, if v is a c-vertex of T not on P, then $v \underset{T'}{\sim} w$ if and only if $v \underset{T}{\sim} w$, which holds if and only if $\bar{v}$ is in $\bar{w}$ in G, and hence in G'.

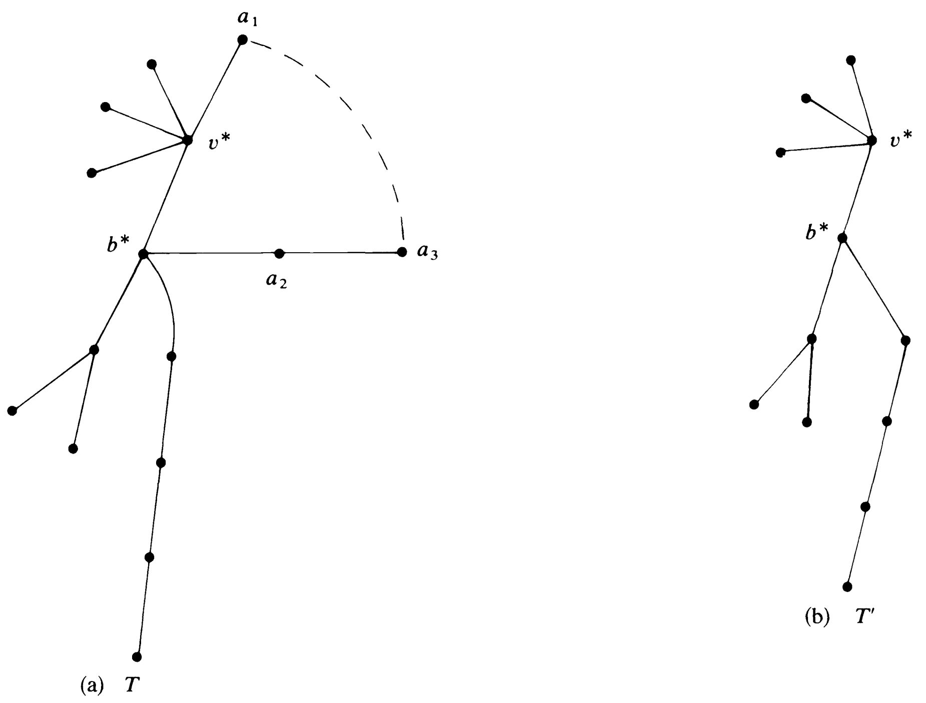


FIG. 3. (a) *A single application of the Path Algorithm*; (b) *the derived* bc-*tree*

Since each adjacent pair of points in T' corresponds to a block-cutpoint relationship in G', T' is indeed bc(G').

PROPOSITION 1.2. $L_{T'}=L_T-2$, *i.e.*, $\lceil L_{T'}/2\rceil=\lceil L_T/2\rceil-1$.

Proof. The proof is obvious.

PROPOSITION 1.3. *If* $(*)$ *holds in* T *then it holds in* T'.

Proof. Suppose for some $z\in T'$, $d_{T'}(z)-1>\lceil L_{T'}/2\rceil$. Then by Proposition 1.2, $d_{T'}(z)-1>\lceil L_T/2\rceil-1$. Changing the relation to "greater than or equal" we obtain (a) $d_{T'}(z)-1\geqq\lceil L_T/2\rceil$. By assumption, $(*)$ holds for z in T, i.e., (b) $d_T(z)-1\leqq\lceil L_T/2\rceil=\lceil L_{T'}/2\rceil+1$. Putting (a) and (b) together, we get $d_T(z)-1\leqq d_{T'}(z)-1$, so $d_T(z)\leqq d_{T'}(z)$. Since $d_T(z)\geqq d_{T'}(z)$ if z is a c-vertex, we must have $d_T(z)=d_{T'}(z)=\lceil L_{T'}/2\rceil+2=\lceil L_T/2\rceil+1$. Thus z satisfies $(*)$ with equality in T. But by A2 of the Path Algorithm, z is on P, so $d_{T'}(z)=d_T(z)-1$, a contradiction.

Proof of Theorem 1 (*continued*). Since, by hypothesis, G' can be biconnected by the addition of $\lceil L_{T'}/2\rceil$ lines, the original graph, G, can be biconnected by the addition of $\lceil L_T/2\rceil$ lines, and the induction hypothesis holds for all k. Q.E.D.

We may now go on to consider arbitrary connected graphs.

THEOREM 2. *A connected graph* G *with* bc-*tree* T *can be biconnected using* $\max_{v\in C_T}[d_T(v)-1,\ \lceil L_T/2\rceil]$ *edges.*

Proof. The idea is to reduce G to a graph which satisfies $(*)$ and can be biconnected by the Path Algorithm. (The proof of Theorem 2 will be continued.)

PROPOSITION 2.1. Let $\beta(G)\equiv\max_v[d_T(v)-1,\ \lceil L_T/2\rceil]$. *Let* $\delta\equiv\max_v[0,d_T(v)-1-\lceil L_T/2\rceil]=\beta(G)-\lceil L_T/2\rceil$. *By adding* 2δ *edges we can obtain a graph* G' *which satisfies the condition* $(*)$ $d_{T'}(v)-1\leqq\lceil L_{T'}/2\rceil$, *for all* c-*vertices*, v.

Proof of proposition. If $\delta=0$, the result is immediate. Suppose $\delta>0$. Let v^* be a maximal degree c-vertex, and let γ be the number of components of $T-v^*$ which contain only one degree-1 vertex of T. (Call such components v^*-*chains*). We show that $\gamma\geqq 2\delta+2$. There are $d(v^*)-\gamma$ components which each contain at least 2 leaves, so $L_T\geqq\gamma+2(d(v^*)-\gamma)$. Thus $\gamma\geqq 2d(v^*)-L_T\geqq 2(d(v^*)-1-\lceil L_T/2\rceil)+2=2\delta+2$.

PROPOSITION 2.2. *By collapsing* $2\delta+1$ v^*-*chains* $c_1,\cdots,c_{2\delta+1}$ *of* T *into a single degree*-1 *vertex* b^*, *adjacent to* v^*, *we get the* bc-*tree* T' *of* $G'=G\cup\{e_1,\cdots,e_{2\delta}\}$, *where each* e_i *is an edge incident to endblocks of the components of* $G-\bar{v}^*$ *corresponding to* c_i *and* c_{i+1}.

Proof. b^* will correspond to the block of G' composed of the $2\delta+1$ components of $G-\bar{v}^*$ joined by the e_i's. One must verify that this is indeed a block, and that the changes made to G and T to get G' and T' match up appropriately. We omit the details. Note that the tree T' has $L_{T'}=L_T-2\delta$ and $d_{T'}(v^*)=d_T(v^*)-2\delta$.

PROPOSITION 2.3. *In the new tree*, T', v^* *does not violate* (*).

Proof. $d_T(v^*)-1=\lceil L_T/2\rceil+\delta$ from the definition of δ, and the assumption $\delta>0$. Therefore, $d_T(v^*)-1-2\delta=[L_T/2\rceil-\delta=\lceil(L_T-2\delta)/2\rceil$. Therefore, $d_{T'}(v^*)-1=\lceil L_{T'}/2\rceil$.

PROPOSITION 2.4. $d_{T'}(v')\geqq d_{T'}(v)$ *for all other* v. *Thus, no vertex* v *violates* $(*)$ *in* T'.

Proof. Suppose for some v', $d_{T'}(v') - 1 > \lceil L_{T'}/2 \rceil$, in violation of ($*$). From Lemma 3,

$$\begin{aligned} L_T &\geqq 2 + d_T(v') - 2 + d_T(v^*) - 2 \\ &= (d_T(v') - 1) + (d_T(v^*) - 1) \\ &> \left\lceil \frac{L_{T'}}{2} \right\rceil + \left(\delta + \left\lceil \frac{L_T}{2} \right\rceil \right) \\ &= \left(\left\lceil \frac{L_T}{2} \right\rceil - \delta \right) + \left(\delta + \left\lceil \frac{L_T}{2} \right\rceil \right) = 2 \left\lceil \frac{L_T}{2} \right\rceil . \end{aligned}$$

But $2\lceil L_T/2 \rceil \geqq L_T$, so we have a contradiction.

Proof of Theorem 2 (*continued*). We have now realized our original intention, of reducing G (via its bc-tree) to a graph which can be biconnected using Theorem 1. Thus we can biconnect G by adding $2\delta + \lceil L_{T'}/2 \rceil = 2\delta + \lceil L_T/2 \rceil - \delta = \delta + \lceil L_T/2 \rceil = \beta(G)$ edges, as desired, proving the theorem.

To finish the job, we consider an arbitrary, possibly disconnected graph, G. (Versions of both Theorem 2 and the following theorem appear in [2]).

THEOREM 3. *Let G be a graph with n connected components,* $n \geqq 1$. *Then*

$$\operatorname{bi}(G) = \max_{v \in C_T} \left[d_T(v) - 1, q(G) + \left\lceil \frac{L_T}{2} \right\rceil - (n-1) \right] + (n-1).$$

Proof. We can connect G by the addition of $n-1$ lines (cf. Lemma 1), which we may choose to be incident at noncutpoints of blocks corresponding to exterior vertices of T. Then if G' is the graph so obtained, and $T' = \operatorname{bc}(G')$,

(i) $\max_{v \in C_{T'}} [d_{T'}(v) - 1] = \max_{v \in C_T} [d_T(v) - 1]$, and

(ii) $\lceil L_{T'}/2 \rceil = q(G) + \lceil L_T/2 \rceil - (n-1)$, by Lemma 2 applied $n-1$ times.

Theorem 2 says G' can be connected by

$$\max_{v \in C_{T'}} \left[d_{T'}(v) = 1, \left\lceil \frac{L_{T'}}{2} \right\rceil \right] = \max_{v \in C_T} \left[d_T(v) - 1, q(G) + \left\lceil \frac{L_T}{2} \right\rceil - (n-1) \right]$$

edges. G may thus be biconnected by $\max_{v \in C_T} [d_T(v) - 1,\ q(G) + \lceil L_T/2 \rceil - (n-1)] + (n-1)$ edges. Since this is exactly the lower bound obtained in [2], it is indeed $\operatorname{bi}(G)$.

3. The biconnection algorithm.

Let A denote the augmenting set.

```
1. $A := \varnothing$.
2. Find the connected components $\{G_j = (V_j, E_j)\}$.
3. Find the set $\{T_j\}$ of bc-trees of the $G_j$'s.
4. For $j = 2$ until # (components)
       begin
           $T_1 := T_1 \cup T_j + x$, where $T_1 \cap T_j = \varnothing$ and $x$ is an edge joining
           exterior points of $T_1$ and $T_j$.
           $G_1 := G_1 \cup G_j + \bar{x}$, where $\bar{x}$ is an edge corresponding to $x$
           $A := A \cup \{\bar{x}\}$
       end
```

(G_1 is now a connected graph such that $G_1 \supset G$. T_1 is its bc-tree.)

```
5. Find all vertex degrees in $T_1$. Let $v^*$ be a highest degree c-vertex in $T_1$.
   Find $L_{T_1}$.
6. If $d_{T_1}(v^*) > \lceil L_{T_1}/2 \rceil + 1$ then
        begin
            $\delta := d_{T_1}(v^*) - (\lceil L_{T_1}/2 \rceil + 1)$.
            Find a set of $2\delta + 1$ $v^*$-chains of $T_1$.
            (Proposition 2.1 asserts that such $v^*$-chains exist.)
            Let $x_1, \cdots, x_{2\delta}$ be any $2\delta$ edges which will interconnect all
            these leaves.
            $G := G_1 \cup \{\bar{x}_1, \cdots, \bar{x}_{2\delta}\}$.
            $A := A \cup \{\bar{x}_1, \cdots, \bar{x}_{2\delta}\}$.
            Replace the block thus created in $T_1$ by a single edge from $v^*$
            to a leaf.
        end.
Let $T$ denote the tree resulting from steps 1 through 6.
   7.1. Choose $b^*$ to be a b-vertex.
   7.2. While $T$ is not a single b-vertex begin
        7.21 Find $v^*$, a highest degree c-vertex in $T$.
        7.22 If $d_T(v^*) = \lceil L_T/2 \rceil + 1$, find the other c-vertex of maximal
             degree, if any, and call it $w^*$.
        7.23 Find a path $P = (a_0, a_1), (a_1, a_2), \cdots, (a_{p-1}, a_p)$ in $T$, going
             through $v^*$, $b^*$, and $w^*$ (if it exists), where $a_0, a_p$ are leaves.
             (Such a path exists, by Lemma 4).
        7.24 $G := G + (\bar{a}_0, \bar{a}_p)$.
        7.25 Update $T$ so it is again the bc-tree of $G$. (See Path Algorithm,
             Step 2.)
        7.26 $A := A \cup \{\bar{a}_0, \bar{a}_p\}$.
             end
```

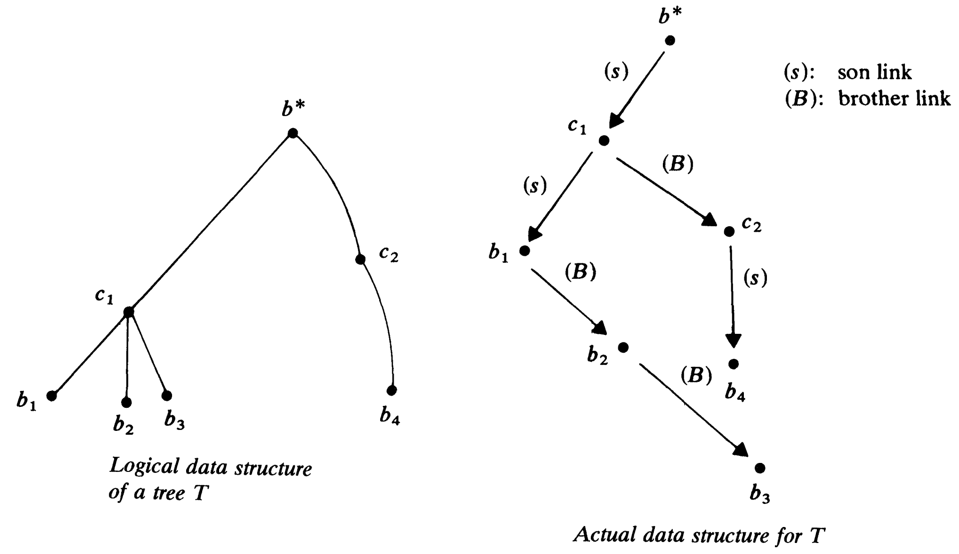
4. Data structure considerations. We now describe an implementation that will run in time $O(|V|+E|)$. We direct our attention to step 7 since linear-time algorithms for the other steps are already established.

T is represented as a tree rooted at b^*. The left-son, right-brother form shown in [4, p. 333] is one appropriate data structure. (See Fig. 4). Along those lines, we assume for each node the following pointers:

(a) father
(b) first_son
(c) brother
(d) last_son.

Note that we can use father pointers to traverse the tree between any vertex and the root. Each vertex effectively has a list of sons, obtained by using the first-son and brother pointers. We will find it convenient to have for each vertex a pointer (last_son) to its last son, for the purpose of concatenating the lists of sons of two merged vertices.

We need to distinguish between b- and c-vertices. We create a logical value, block(v), which is true if and only if v is a b-vertex.



Logical data structure of a tree T

Actual data structure for T

FIG. 4

We will also find it handy to have an array #active_sons(v), that holds the number, as of the most recent update, of sons of v. An array of stacks stack(i), $i = 1, \cdots, |V|$, contains stacks of vertices believed to be of degree i.

Finally, we include a logical array, denoted flag(v), which is true if and only if v is b^* or has been merged with it. The existence of this variable lets us dispense with changing father pointers of sons of merged vertices. A vertex has b^* as its current father if and only if flag(father(v)) is true.

To form P, then, we need only:

1. Follow father pointers from v^* until we reach a vertex flagged as part of b^*, flagging appropriate vertices along the way and making the others sons of b^*.

2. Follow son pointers from v^* until a leaf is reached.

3. Follow son pointers from b^* (avoiding the son we visited traveling up from v^* in step 1 till a second leaf is reached. (If two vertices of degree $L/2$ exist, v^* will be set equal to the one which occurs lower in the tree.)

When a vertex w is flagged, we do not scan through the list of sons of father(w) to delete w, since that could require repeated long searches and increase the complexity of the algorithm. Instead, we delete it if and when it is visited during the search for an unflagged son later on.

For the purpose of precisely analyzing the complexity of the algorithm, we now present a set of procedures to accomplish step 7 of the Biconnection Algorithm.

(Assume the existence of a procedure, lower(a, b), the values of whose arguments are vertices of a tree, T. Lower(a, b) simply starts at vertex b and scans upwards in the tree using father pointers. If a is found before the root is reached, the values of variables a and b are exchanged. Assume also the existence of a procedure highest_degree_c-vertices(v^*, w^*), which finds the highest degree c-vertices as in Lemma 4, setting w^* to null if there are not two c-vertices satisfying the condition: $d_T(v)-1=\lceil L_T/2\rceil$.)

```
procedure step_7:
    integer procedure another_son($v$);
    comment returns $u$, the first unflagged brother of $v$, and resets pointers to
    skip over flagged sons of $v$;
        $w$ := first_son($v$);
        $u$ := brother($w$);
        while $u \neq$ null and flag($u$) = true begin                    (1)
            $u$ := brother($u$);
            brother($w$) := $u$;
        end;
        $u$
procedure visit($v$);
    adjust #active_sons of $v$ and if necessary, of $b^*$, to reflect the
    shrinking of the path—in particular, for $v$ a degree-2 c-vertex on $P$, let
    #active_sons($v$) = 0;
    if block($v$) or #active_sons($v$) = 0 then begin
        flag($v$) := true;
        add sons of $v$ to list of $b^*$'s sons;
        comment note that when a first_son is visited, the another_son proce-
        dure is called to delete flagged nodes at the beginning of the brother
        chain. This is used later to prove the linearity of the algorithm;
        if $v$ = first_son (father($v$)) then begin
            first_son(father($v$)) := another_son(father($v$));
            adjust last_son pointer if necessary;
        end;
    else begin
        comment come here if and only if $v$ is a c-vertex that will not be
        absorbed into $b^*$;
        if $b^* \neq$ father($v$) then make $v$ a son of $b^*$;
    end;
procedure upward($v$);
    while flag($v$) = false begin                                      (3)
        visit($v$);
        $v$forbid := $v$;
        $v$ := father($v$);
    end;
procedure downward($r$, leaf);
    leaf := $r$;
    $v$ := first_son($r$);
    if $v^*$ = $v$forbid then $v$ := another_son($r$);
    while $v \neq$ null begin                                          (2)
        visit($v$);
        leaf := $v$;
        $v$ := first_son($v$);
    end;
    **comment** the following **while** loop performs the algorithm of step 7, by
    calling the above procedures;
while $\#$active_sons($b^*$) $\neq 0$ begin
    highest_degree_c-vertices($v^*$, $w^*$);
    if $w^* \neq$ null then lower ($v^*$, $w^*$);
    begin
        upward($v^*$);
        downward($v^*$, leaf1);
        downward($b^*$, leaf2);
    end;
end;
```

Analysis. We give the time order of complexity of each step in the Biconnection Algorithm.

(i) The connected components can be found in linear time [3].

(ii) The bc-tree, T, can be found in $O(|V|+|E|)$ time, using the algorithm of Tarjan [1].

(iii) The vertex degrees of T are found by counting the edges of each vertex, which can be done in $O(|E|)$ time.

(iv) Total time spent finding highest degree c-vertices is $O(|V|)$. Initially we perform a radix sort (Knuth [4]) to obtain lists LIST(i) of vertices which have exactly i active sons. At each iteration it takes constant time to remove a vertex v from the highest numbered nonempty list, say LIST(j). If $\#$ active_sons(v) $= j$ then we may immediately output v. The test fails only if a son of v has been deleted since v was placed in LIST(j). (This can be done only $|V|$ times.) In this case v is placed in LIST ($\#$active_sons(v)).

(vi) If there is a vertex v^* whose degree exceeds $\lceil L_T/2 \rceil + 1$, it can be removed by adding $2\delta = 2(d(v^*) - (\lceil L_T/2 \rceil + 1))$ edges, which is at most $O(|V|)$.

(vii) We note that each repetition of step 7.2 decreases the number of edges of T by at least 1. So the total number of repetitions is at most $|E|$. If the program had no loops, this would establish $O(|E|)$ as its order. Four loops must however be considered. Three of these are indicated by marginal numbers in the program.

(1) In another_son. The body of the **while** loop will be executed at most $|V|$ times during the call to step_7. Another_son is called by visit, to delete the first_son of a node, and by downward(b^*, leaf2). Each execution of the **while** loop will cause a particular node to be bypassed during all subsequent executions, so each node will be associated with at most one execution.

(2) In downward. Each execution of this **while** statement corresponds to the traversal of the edge joining some vertex v and first_son(v). Each downward execution causes the deletion of all edges from c-vertices to b-vertices along its path; hence for every two edges traversed, at least one disappears. Thus downward will visit at most $2|E|$ nodes, during the execution of step_7.

(3) In upward. The same observations apply as for downward. The **while** loop is executed a total of at most $2|E|$ times, considering all executions of upward.

The fourth loop occurs in the procedure lower(a, b), which is not given explicitly here. However, since we are merely scanning upward in the tree from vertex b, the time order of complexity is the same as that of procedure upward.

Thus step_7 is a sequence of steps, all of which are $O(|V|)$ or $O(|E|)$, and the algorithm is $O(|V|+|E|)$, as claimed.

A natural generalization of the biconnection problem is to attach a weight to each possible edge $(v, w) \in E$ which could be part of an augmentation, and to seek a minimum weight biconnection. For the trivial case of all vertices adjacent to a single v^*, finding a minimum weight biconnection is equivalent to finding a minimum spanning tree for $V-v^*$.

The general weight problem appears intractable. Karp and Cook have shown the existence of a large class of unsolved problems (called NP-complete) such that a polynomial-time algorithm for one could be converted into polynomial-time algorithms for all. (See Aho [1] for a discussion of this class and its other implications.) Eswaran and Tarjan [2] have shown that finding a minimum weight biconnection of an arbitrary graph is NP-complete. Spira [5] has shown that even if the domain is restricted to connected graphs of degree 3, the problem remains NP-complete.

REFERENCES

[1] A. Aho, J. Hopcroft and J. Ullman, *Analysis and Design of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

[2] K. Eswaran and R. Tarjan, *Augmentation problems*, this Journal, 5 (1976), pp. 653–665.

[3] F. Harary, *Graph Theory*, Addison-Wesley, Reading, Mass., 1969.

[4] D. Knuth, *The Art of Computer Programming*, vol. 1, Addison-Wesley, Reading, Mass., 1969.

[5] P. Spira, Personal communication.

[6] R. Tarjan, *Depth first search and linear graph algorithms*, this Journal, 1 (1972), pp. 146–159.

SIAM J. COMPUT.
Vol. 6, No. 1, March 1977

POLYNOMIAL COMPLETE CONSECUTIVE INFORMATION RETRIEVAL PROBLEMS*

LAWRENCE T. KOU†

Abstract. A set of queries Q is said to have the consecutive retrieval property with respect to a set of records R if there exists an organization of the record set (without duplication of any record) such that for every $q_i \in Q$, all relevant records can be stored in consecutive storage locations [5]. In practice, this property does not appear very often. Hence, either duplication of records is allowed so that pertinent records corresponding to any query are always stored consecutively or storing the pertinent records corresponding to a query in several blocks of consecutive storage locations is necessary so that each record is stored only once. The former gives rise to the problem of minimizing the duplication of records and the latter gives rise to the problem of minimizing the number of blocks of consecutive storage of relevant records. The computational complexity of each of these two problems is studied in this paper and both of these problems are shown to be polynomial complete in the sense of Cook [2] and Karp [8].

Key words. consecutive retrieval, cost graph, incidence matrix, Hamiltonian path, polynomial complete

1. Introduction and summary. A set of queries Q is said to have the consecutive retrieval property with respect to a set of records R if there exists an organization of the record set (without duplication of any record) such that for every $q_i \in Q$, all relevant records can be stored in consecutive storage locations. In linear storage systems (e.g. tape, surface of drum), if the query set Q has the consecutive retrieval property with respect to the record set R, then storing the pertinent records in consecutive storage locations will provide a file organization requiring small storage space and efficient retrieval time. Let the query set Q be $\{q_1, q_2, \cdots, q_m\}$ and the record set R be $\{r_1, r_2, \cdots, r_n\}$. The relationship between Q and R is conveniently represented by an $n \times m$ 0-1 matrix B. The (i, j)th entry of B is 1 iff record r_i is pertinent to query q_j. This matrix is called the record-query incidence matrix.

$$
\begin{array}{r|ccccc|}
 & q_1 & q_2 & q_3 & \cdots & q_m \\
r_1 & 1 & 1 & 0 & & 1 \\
r_2 & 0 & 1 & 0 & \cdots & 1 \\
B = r_3 & 1 & 0 & 1 & & 1 \\
\cdot & & \cdot & & & \cdot \\
\cdot & & \cdot & & & \cdot \\
\cdot & & \cdot & & & \cdot \\
r_n & 1 & 0 & 1 & & 0
\end{array}.
$$

* Received by the editors December 24, 1974.

† IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.

It should be clear that Q has the consecutive retrieval property with respect to R iff there exists a permutation of the rows of B such that the 1's in each column appear in consecutive positions. The problem of finding such a permutation, if it exists, was first solved by Fulkerson and Gross in their study of incidence matrix and interval graphs [4]. A different solution was given by Eswaran in his study of consecutive information retrieval [3]. If B is $n \times m$ and m is bounded by a polynomial of n, algorithms that have time bound $O(p(n))$ for some polynomial p can be found in [3] and [4].

However, not all pairs of Q and R have the consecutive retrieval property [3], [5], [6], [7]. As a matter of fact, in most practical cases, the pair (Q, R) does not have the consecutive retrieval property. Hence, in practice, either duplication of records is allowed so that pertinent records corresponding to any query are always stored consecutively or storing the pertinent records corresponding to a query in several blocks of consecutive storage locations is necessary so that each record is stored only once. The former gives rise to a problem of minimizing storage space (minimizing duplication of records) subjected to a fixed retrieval time and the latter gives rise to a problem of minimizing retrieval time (minimizing blocks of consecutive storage) subjected to a fixed storage space. These two problems can be stated formally as follows:

(A) *The problem of minimizing the duplications of records*:

Given an $n \times m$ incidence matrix B, let $Q_j = \{r_i | b_{ij} = 1\}$ for $1 \leqq j \leqq m$. Find the minimum length sequence x in the alphabet $R = \{r_1, r_2, \cdots, r_n\}$ such that for every $j = 1, 2, \cdots, m$, there is a consecutive subsequence of x consisting of exactly all the elements in Q_j.

(B) *The problem of minimizing the number of blocks of consecutive storage of relevant records*:

Given an $n \times m$ incidence matrix B, find a permutation of B such that the total number of blocks of consecutive 1's in the columns of B is minimized.

It is shown in this paper that both of these problems are polynomial complete. Loosely speaking, it implies that if one can find an efficient algorithm to solve one of these two problems then many known difficult problems (e.g., Hamiltonian circuit problem, job scheduling problem, traveling salesman problem, to name a few) would all have efficient algorithms to solve them—an unlikely event. In view of these negative results and the increasing need for efficient file organization techniques, good heuristic approaches for the problems seem to be necessary and acceptable.

2. Cost graph of incidence matrix. A cost graph referred to here is simply a complete digraph (all selfloops are ignored in this paper) with nonnegative integer cost associated with each edge in the graph. Given an $n \times m$ incidence matrix B, the cost graph G of B is the complete digraph with the set of vertices $V = \{1, 2, \cdots, n\}$ and the set of edges $E = \{[i, j] | i \neq j, i, j \in V\}$ together with a cost function F: $E \to I$ (I is the set of nonnegative integers) such that for all $[i, j] \in E$, $f([i, j]) = \sum_{s=1}^{m} b_{is} * b_{js}$ where b_{ij} is the (i, j)th entry in B and $*$ is a binary operation defined by $0 * 0 = 0$, $0 * 1 = 0$, $1 * 0 = 1$ and $1 * 1 = 0$. Generally, a cost graph G is denoted by $G = (V, E, f)$.

Example 1. Given

$$B=\begin{pmatrix}1 & 1 & 0 & 1 & 0\\ 0 & 1 & 1 & 1 & 0\\ 0 & 0 & 0 & 1 & 1\\ 1 & 0 & 1 & 0 & 1\end{pmatrix}.$$

The cost graph G is shown in Fig. 1.

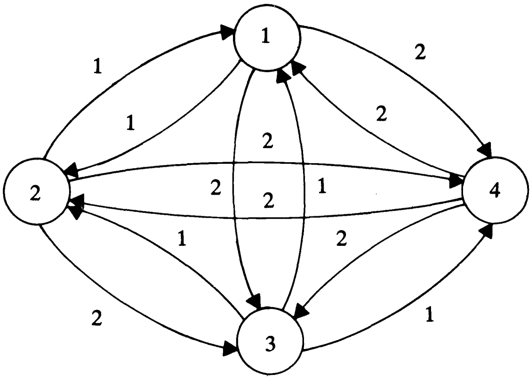


FIG. 1. *Cost graph for B in Example* 1

For any incidence matrix, there is a unique cost graph associated with it. However, not every cost graph has a corresponding incidence matrix. A simple exercise will show that the cost graph in Fig. 2 has no corresponding incidence matrix. Given a cost graph G, if there exists an incidence matrix B whose associated cost graph is G, then G is said to be 0-1 matrix realizable. A cost graph $G=(V, E, f)$ is said to be symmetric if $f([i, j])=f([j, i])$ for $i\neq j$ and $i, j\in V$. The following two theorems give sufficient conditions for cost graphs to be 0-1 matrix realizable.

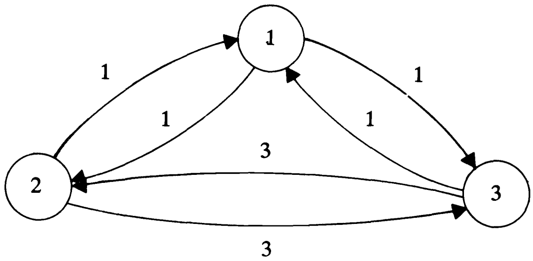


FIG 2. *A cost graph corresponding to no incidence matrix*

THEOREM 1. *Let $G_n=(V,E,f)$ be a symmetric cost graph with n vertices, $n\geqq 3$. If edges $[1,2]$ and $[2,1]$ have cost $n-1$ while every other edge has cost $n-2$, then there exists an $n\times m_n$ incidence matrix B_n such that*

(i) *B_n realizes G_n;*

(ii) *$m_n=n(n-1)/2+1$;*

(iii) *each row of B_n contains $n-1$ 1's.*

Proof. The proof is given by induction on n. For $n=3$, let

$$B_3=\begin{pmatrix}1&0&1&0\\0&1&0&1\\0&0&1&1\end{pmatrix}.$$

Now assume the theorem holds for $n=k$. Then, for $n=k+1$, consider

$$B_{k+1}=\left(\begin{array}{ccccc|ccccc}\multicolumn{5}{c|}{B_k}&\multicolumn{5}{c}{I_k}\\\hline 0&0&\cdots&0&0&1&1&1&\cdots&1\end{array}\right)$$

where I_k is a $k\times k$ identity matrix.

Part (i).

$$f([1,2])=\sum_{s=1}^{m_{k+1}} b_{1s}*b_{2s}\qquad\text{(by definition)}$$

$$=\sum_{s=1}^{m_k} b_{1s}*b_{2s}+\sum_{s=m_k+1}^{m_{k+1}} b_{1s}*b_{2s}$$

$$=(k-1)+1\qquad\text{(by the induction hypothesis (i) is true for } n=k)$$

$$=k.$$

Similarly, $f([2,1])=k$.

For $1\leqq i\leqq k$, $1\leqq j\leqq k$, $i\neq j$, and $[i,j]\neq[1,2]$ or $[2,1]$,

$$f([i,j])=\sum_{s=1}^{m_{k+1}} b_{is}*b_{js}\qquad\text{(by definition)}$$

$$=\sum_{s=1}^{m_k} b_{is}*b_{js}+\sum_{s=m_k+1}^{m_{k+1}} b_{is}*b_{js}$$

$$=(k-2)+1\qquad\text{(by the induction hypothesis (i) is true for } n=k)$$

$$=k-1.$$

Furthermore, for all $i=1,2,\cdots,k$,

$$f([k+1,i])=\sum_{s=1}^{m_{k+1}} b_{(k+1)s}*b_{is}\qquad\text{(by definition)}$$

$$=\sum_{s=1}^{m_k} b_{(k+1)s}*b_{is}+\sum_{s=m_k+1}^{m_{k+1}} b_{(k+1)s}*b_{is}$$

$$=0+(k-1)\qquad\text{(by the construction of row } k+1)$$

$$=k-1,$$

$$\begin{aligned} f([i,k+1]) &= \sum_{s=1}^{m_{k+1}} b_{is} * b_{(k+1)s} \qquad \text{(by definition)} \\ &= \sum_{s=1}^{m_k} b_{is} * b_{(k+1)s} + \sum_{s=m_k+1}^{m_{k+1}} b_{is} + b_{(k+1)s} \\ &= (k-1)+0 \qquad \text{(by the induction hypothesis (iii) is true for } n=k \\ &\qquad\qquad\qquad \text{and by the construction of row } k+1) \\ &= k-1. \end{aligned}$$

Hence, for $n=k+1$, B_n realizes G_n.

Part (ii),

$$\begin{aligned} m_{k+1} &= m_k + k \\ &= \frac{k(k-1)}{2}+1+k \qquad \text{(by the induction hypothesis (ii) is true for } n=k) \\ &= \frac{(k+1)k}{2}+1. \end{aligned}$$

Hence, for $n=k+1$, $m_n = n(n-1)/2+1$.

Part (iii). In B_{k+1}, for $1 \leqq i \leqq k$, row i contains $k-1$ 1's in the first m_k entries due to B_k and contains one 1 in the last k entries due to I_k. So for $1 \leqq i \leqq k$, row i of B_{k+1} contains k 1's. Row $k+1$ contains k 1's by construction. Hence, for $n=k+1$, each row of B_n contains $n-1$ 1's.

The proof by induction is thus completed.

Remark. If the cost graph is such that the only two edges that have individual cost $n-1$ are $[i,j]$ and $[j,i]$ instead of $[1,2]$ and $[2,1]$, simply interchanging row 1 and row 2 respectively with row i and row j will give a realization of the corresponding new cost graph.

THEOREM 2. *Let $G=(V,E,f)$ be a symmetric cost graph with n vertices, $n \geqq 3$. Let u be a positive integer, $1 \leqq u \leqq n(n-1)/2$. If $S = \{[i_1,j_1],[j_1,i_1],[i_2,j_2],[j_2,i_2],\cdots,[i_u,j_u],[j_u,i_u]\}$ is a set of $2u$ edges which have cost $u(n-2)+1$ each while every other edge in G has cost $u(n-2)$, then there exists an $n \times m$ incidence matrix B such that*

(i) *B realizes G*;

(ii) $m = u\left(\frac{n(n-1)}{2}+1\right)$;

(iii) *each row in B contains $u(n-1)$ 1's.*

Proof. Let G_k, $1 \leqq k \leqq u$, be a symmetric cost graph with n vertices such that only edges $[i_k,j_k]$ and $[j_k,i_k]$ have cost $n-1$ each while every other edge has cost $n-2$. By Theorem 1, G_k is 0-1 matrix realizable. Let B_k be the incidence matrix constructed for G_k as in Theorem 1. Now consider

$$B = (B_1 \quad B_2 \quad \cdots \quad B_u).$$

The corresponding cost graph of B is obviously the superposition of cost graphs $G_1, G_2, \cdots, G_u$ (since the cost functions are additive). The theorem follows immediately from the construction of B and Theorem 1.

3. Hamiltonian paths in the cost graph of an incidence matrix. Let B be an $n \times m$ incidence matrix and $G=(V, E, f)$ be the corresponding cost graph. A Hamiltonian path in G is a simple path in G that includes every vertex exactly once. A Hamiltonian path in G can be specified by a sequence of n vertices, $(i_1, i_2, \cdots, i_n)$, where the $i_1, i_2, \cdots, i_n$ are all distinct. The cost of a Hamiltonian path in G is the sum over the costs of the edges on the path. The following lemmas give the relationship between the cost of a Hamiltonian path in G and the total number of consecutive 1's in the columns of B.

LEMMA 1. *Let B be an $n \times m$ incidence matrix and $G=(V, E, f)$ be the corresponding cost graph. Then the cost of the Hamiltonian path $(1, 2, \cdots, n)$ is k if and only if the total number of blocks of consecutive* 1*'s in the columns of B is $k+c$, where c is the number of* 1*'s in the n-th row of B.*

Proof. Let N be the total number of blocks of consecutive 1's in the columns of B and N_i be the total number of blocks of consecutive 1's that end at row i of B. Obviously,

$$N = N_1 + N_2 + \cdots + N_n.$$

By the definition of the associated cost graph, it should be clear that, for $1 \leqq 1 < n$, $N_i = k_i$ in B iff $f([i, i+1]) = k_i$ in G. On the other hand, $N_n = c$. Hence,

$$\begin{aligned} &\text{the cost of the Hamiltonian path } (1, 2, \cdots, n) \\ &= f([1, 2]) + f([2, 3]) + \cdots + f([n-1, n]) \\ &= N_1 + N_2 + \cdots + N_{n-1} \\ &= N - c. \end{aligned}$$

The proof is thus completed.

LEMMA 2. *Let B be an $n \times m$ incidence matrix and $G=(V, E, f)$ be the corresponding cost graph. Then G has a Hamiltonian path of cost k if and only if there exists an $n \times n$ permutation matrix P such that the total number of blocks of consecutive* 1*'s in the columns of PB is $k+c$, where c is the number of* 1*'s in the n-th row of PB.*

Proof. Since each Hamiltonian path $(i_1, i_2, \cdots, i_n)$ in G corresponds in a one-to-one manner to a permutation of rows in B, the proof of this lemma follows immediately from Lemma 1.

4. Polynomial completeness of general consecutive retrieval problems. Let NP be the class of languages that can be accepted by a nondeterministic polynomial time bounded Turing machine. A language L_1 is polynomially reducible to a language L_2 (written as $L_1 \propto L_2$) iff there exists a deterministic polynomial time bounded Turing machine which will convert each string x in the alphabet of L_1 into a string y in the alphabet of L_2 such that $x \in L_1$ iff $y \in L_2$. A language L is polynomially complete iff L is in NP and every language in NP is polynomially reducible to L. A problem that requires a yes or no answer can be considered as a language such that a string x is in the language iff an instance of the problem that has a yes answer is encoded into the string x. A yes or no problem P_1 is said to be polynomially reducible to a yes or no problem P_2 iff the corresponding languages L_1, L_2, respectively, are such that $L_1 \propto L_2$. A yes or no problem is polynomially complete iff the corresponding language is polynomial complete. The reader is referred to [1], [2] and [8] for the discussions of polynomial complete problems, the polynomial reducibility and the encoding of problems onto Turing tapes.

In the following, several yes or no problems are introduced first and all of them will be shown to be polynomial complete.

Problem 1. *Given*: an undirected graph $G=(V,E)$. (Without loss of generality it is assumed that $|V|=|\{1,2,\cdots,n\}|=n\geqq 3$ and that G is not a complete graph).

Question. Is there a Hamiltonian path in G?

Problem 2. *Given*: a cost graph $G=(V,E,f)$ and a positive integer u such that

(i) $V=\{1,2,\cdots,n\}$ and $n\geqq 3$;

(ii) $1\leqq u\leqq n(n-1)/2$;

(iii) there exists a set S of $2u$ edges in G, $S=\{[i_1,j_1],[j_1,i_1],[i_2,j_2],[j_2,i_2],\cdots,[i_u,j_u],[j_u,i_u]\}$ such that $[p,q]\in S\Rightarrow f([p,q])=u(n-2)+1$ and $[p,q]\in E$, $[p,q]\notin S\Rightarrow f([p,q])=u(n-2)$.

Question. Is there a Hamiltonian path in G such that its cost is $u(n-1)(n-2)$?

Problem 3. *Given*: an $n\times m$ incidence matrix B and a nonnegative integer k.

Question. Does there exist an $n\times n$ permutation matrix P such that $\#(PB)=k$, where $\#(PB)$ denotes the total number of blocks of consecutive 1's in the columns of PB?

Problem 4. *Given*: a finite set $R=\{r_1,r_2,\cdots,r_p\}$, a family of subsets F, $F=\{Q_j|1\leqq j\leqq q, Q_j\subseteq R\}$ and a nonnegative integer k.

Question. Does there exist a string x of length k in the alphabet R such that for every $j=1,2,\cdots,q$ there is a consecutive subsequence of x consisting of exactly all the elements in Q_j?

Problems of whether a Hamiltonian path exists in an undirected or a directed graph have been shown to be polynomial complete in [8]. Although the original problems concerned Hamiltonian circuits rather than Hamiltonian paths, almost identical proofs as those shown in [8] can be constructed to show that the Hamiltonian path problem is polynomial complete. In the following, Problems 2, 3 and 4 are all shown to be polynomial complete.

THEOREM 3. *Problem* 2 *is polynomial complete.*

Proof. The language L corresponding to Problem 2 is certainly in *NP*. A polynomial time bounded nondeterministic Turing machine can be constructed such that it will guess a correct Hamiltonian path and then check if the cost of the path is equal to $u(n-1)(n-2)$. It remains to show that every language in *NP* is polynomially reducible to L. Since Problem 1 is polynomial complete, it is sufficient to show that Problem $1\propto$ Problem 2.

Let the undirected graph $G=(V,E)$ be an instance for Problem 1. A polynomial time bounded deterministic Turing machine can be constructed to do the following:

(i) set $u=\dfrac{n(n-1)}{2}-|E|$;

(ii) construct a cost graph $G_1=(V_1, E_1, f)$ such that $V_1=V$ and for $i\neq j$, if the undirected pair $\{i,j\}\notin E$, set $f([i,j])=f([j,i])=u(n-2)+1$, else set $f([i,j])=f([j,i])=u(n-2)$.

G_1 is an instance of Problem 2. Furthermore, by the construction of G_1, G has a Hamiltonian path $(i_1, i_2, \cdots, i_n)$ if and only if the cost of the path in G_1 is $u(n-1)(n-2)$. The proof is thus completed.

THEOREM 4. *Problem* 3 *is polynomial complete.*

Proof. The language L corresponding to Problem 3 is certainly in NP. A polynomial time bounded nondeterministic Turing machine can be constructed to guess a correct permutation matrix P and then check to see if $\#(PB)=k$. Given an instance of Problem 2, by Theorem 2, a polynomial time bounded deterministic Turing machine can be constructed to set the value of k equal to $u(n-1)^2$ and assign an $n\times m$ incidence B such that

(i) B realizes G;

(ii) $m=u\left(\frac{n(n-1)}{2}+1\right)$;

(iii) each row in B contains $u(n-1)$ 1's.

This is an instance of Problem 3. Furthermore, by the construction of B and Lemma 2, there exists an $n\times n$ permutation matrix P such that $\#(PB)=u(n-1)(n-2)+u(n-1)=u(n-1)^2$ if and only if the cost graph G has a Hamiltonian path with cost equal to $u(n-1)(n-2)$. Therefore, Problem $2\propto$ Problem 3. The proof is thus completed.

THEOREM 5. *Problem* 4 *is polynomial complete.*

Proof. It is easy to see that the language L corresponding to Problem 4 is in NP. In the following, it will be shown that Problem $1\propto$ Problem 4.

Let the undirected graph $G=\{V,E\}$ be an instance of Problem 1. A polynomial time bounded deterministic Turing machine can be constructed to do the following:

(i) set $R=E\cup V$;

(ii) set $F=\{Q_1, Q_2, \cdots, Q_n\}$ where $Q_i=\{\{i,j\}|\{i,j\}\in E\}\cup\{i\}$ for $i=1,2,\cdots,n$;

(iii) set $k=1-n+\sum_{i=1}^n|Q_i|=2|E|+1$.

This is an instance of Problem 4. By the construction of Q_i's, the following are obvious:

(i) for all $i,j(i\neq j)$, $Q_i\not\supseteq Q_j$;

(ii) for all $i,j(i\neq j)$, $Q_i\cap Q_j=\{\{i,j\}\}$ if and only if $\{i,j\}\in E$.

Therefore, there exists a Hamiltonian path in G if and only if there exists a string x of length k such that for $i=1,2,\cdots,n$ the elements of Q_i appear consecutively in x. Hence, Problem $1\propto$ Problem 4. The proof is thus completed.

Remark. In Theorem 4, if $\#(PB)=k=u(n-1)^2$, then for any $n\times n$ permutation matrix P', $\#(PB)\leqq\#(P'B)$. Also, in Theorem 5, if the length of x equals $k=1-n+\sum_{i=1}^n|Q_i|$, then x is the minimum length string in the alphabet R such that for $j=1,2,\cdots,n$ elements of Q_j appear consecutively in the string.

Acknowledgment. The author is obliged to Professor John E. Hopcroft for reading the manuscript of this paper and for many valuable discussions. Special gratitude is due the referees for their thorough review and, particularly for their pointing out a mistake in the proof of Theorem 5 in the original version of this paper.

REFERENCES

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

[2] S. A. COOK, *The complexity of theorem proving procedures*, Proc. 3rd ACM Conf. on Theory of Computing, May 1970.

[3] K. P. ESWARAN, *Consecutive retrieval information system*, Ph.D. thesis, Electrical Engrg. and Computer Sci. Dept., Univ. of Calif., Berkeley, 1973.

[4] D. R. FULKERSON AND D. A. GROSS, *Incidence matrices and interval graphs*, Pacific J. Math., 15 (1965), no. 3.

[5] S. P. GHOSH, *File organization: The consecutive retrieval property*, Comm. ACM, 15 (1972), no. 9.

[6] ———, *Consecutive storage of relevant records with redundancy*, IBM Res. Rep. RJ 933, Yorktown Heights, NY, 1972.

[7] ———, *On the theory of consecutive storage of relevant records*, Information Sci., 6 (1973).

[8] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. Miller and J. Thatcher, eds., Plenum Press, New York, 1972.

[9] S. SAHNI, *Some related problems from network flows, game theory and integer programming*, Proc. 13th Ann. Symp. on Switching and Automata Theory, Oct. 1972.

[10] R. SETHI, *Complete register allocation problems*, 5th Ann. ACM Symposium on Theory of Computing, May 1973.

[11] J. D. ULLMAN, *Polynomial complete scheduling problems*, ACM 4th Symp. on Operating System Principles, October 1973.

SIAM J. COMPUT.
Vol. 6, No. 1, March 1977

ON THE COMPLEXITY OF LOCAL SEARCH FOR THE TRAVELING SALESMAN PROBLEM*

CHRISTOS H. PAPADIMITRIOU AND KENNETH STEIGLITZ†

Abstract. It is shown that, unless $P = NP$, local search algorithms for the traveling salesman problem having polynomial time complexity per iteration will generate solutions arbitrarily far from the optimal.

Key words. traveling salesman problem, local search, complexity, NP-complete

1. Introduction. The traveling salesman problem (TSP) can be stated as follows: given r cities and $(r-1)r/2$ nonnegative integers denoting the distances between all pairs of cities, we are required to find a *tour*, that is, a closed path passing through each city exactly once, so that the total traversed distance is minimal. Despite the simplicity of its statement, the TSP is apparently a very hard problem and has attracted a large number of researchers. Although no efficient algorithm for its solution has been found (and no nontrivial lower bound of its complexity has been proved) a number of different lines of attack have been proposed. A class of heuristics known as local search algorithms [5], [6], [11], [12] have been particularly successful in generating good solutions for large problems by a reasonable computational effort. A local search algorithm (to be more formally defined later) starts with an essentially random tour and, by searching a set of tours which are considered "neighbors" of the former, either finds a neighbor with improved cost and uses it as a new starting point or, if this is not possible, terminates. The solution generated by this technique is called a *local optimum*. Tours of minimum length are referred to as *global optima*. Local optima may or may not necessarily be global optima, depending on the particular neighborhood structure used by the algorithm. Local search algorithms generating only global optima are called *exact*.

We will be particularly interested in the complexity of the problem of searching the neighborhood of a tour in order either to find an improvement or show this tour to be a local optimum. By "complexity of local search" the above mentioned complexity is understood—and not the complexity of the whole algorithm, which heavily depends on the number of iterations necessary. In particular we will examine the computational requirements of local search algorithms for the TSP, when certain restrictions are imposed on the quality of the obtained local optima.

The notion of a combinatorial optimization problem with a numerical input (COPNI) is introduced. This class, which appears to be a restriction of the subset problems discussed by [10], includes several well-known problems such as the TSP and instances of the problem of job scheduling with deadlines (JSD). A particular COPNI is exhibited in which the minimal exact neighborhood, although

* Received by the editors September 24, 1975.

† Department of Electrical Engineering, Princeton University, Princeton, New Jersey 08540. This work was supported by the National Science Foundation under Grant GK-42048 and by the U.S. Army Research Office—Durham under Contract DAHC04-69-C0012.

exponential in size, can be searched in linear time. This counterexample shows that the cardinality of the minimal exact neighborhood is not a lower bound for the complexity of exact local search.

In fact, if the exact local search problem were of provably exponential complexity, this would be a rather remarkable result, since exact local search for the TSP is one of those tasks that are made very easy if nondeterministic computations are permitted. In the light of this observation we can think of the question, whether exact local search for the TSP can be done in a polynomial amount of time per iteration, as a part of the presently unsettled $P = NP$ question. In fact it is shown that, unless $P = NP$, each iteration of an exact local search algorithm for the TSP requires more than a polynomial number of steps.

A stronger result is also shown along the same lines. It is proved that, if a local search algorithm requires only a polynomial amount of time per iteration, the local optima thus obtained can be arbitrarily far from the optimum, unless, of course, $P = NP$. The above result suggests that a large class of efficient heuristics [5], [6], [11], [12] yield local optima of no guaranteed accuracy whatsoever.

2. Combinatorial optimization problems with numerical input. The set of nonnegative integers is denoted by Z^+. For $n \in Z^+$ we shall denote by $\bar{n}$ the set $\{1, 2, \cdots, n\}$.

DEFINITION. A combinatorial optimization problem with numerical input (COPNI) is a pair (n, F), where $n \in Z^+$ is the *dimension* of the problem and F, a subset of $2^{\bar{n}}$, is the set of *feasible solutions*. We will require that there exists at least one feasible solution and that no feasible solution is properly contained in another.

An *instance* of the COPNI (n, F) is a function (numerical input) $c: \bar{n} \to Z^+$. In order to solve an instance c of the COPNI (n, F) we are required to find a feasible solution $f \in F$ such that $c(f) = \sum_{j \in f} c(j)$ is minimal.

Note that the feasibility of a solution is not affected by the numerical input. On the other hand the noncontainment requirement for the feasible solutions can be easily seen to be equivalent to the condition, that for each $f \in F$ there exists an instance c for which f is uniquely optimal.

There is an interesting geometric interpretation of COPNI's: every feasible solution in F corresponds (in the obvious manner) to a vertex of the n-dimensional hypercube. Hence the convex hull of these vertices is an equivalent representation of the COPNI. Since an instance of the COPNI is essentially a linear functional, it follows that solving an instance of a COPNI is equivalent to minimizing a linear functional over the vertices of a convex polytope. For a further discussion of this analogy, see [7].

Examples. The TSP with r cities is a COPNI with $n = \binom{r}{2}$ and with F being the set of all possible tours represented as sets of r intercity links.

The problem of job scheduling with deadlines (JSD) [4] is a COPNI. Here we have a set $\bar{n}$ of jobs and for each job $j \in \bar{n}$ we have the deadline D_j and the execution time T_j. A subset f of $\bar{n}$ is feasible if all jobs in $\bar{n} - f$ can be executed on a single processor within their deadlines, and no subset of $\bar{n}$ properly containing $\bar{n} - f$ enjoys this property.

In the case of JSD the values of c can be thought of as rewards obtained for executing a job within its deadline, and our goal is to minimize the rewards lost. It should be emphasized that, unlike the formulation in [4], the numbers $\{D_j\}$ and $\{T_j\}$ are not considered as a numerical input here.

The Steiner tree problem, the max flow problem, the minimal spanning tree problem and many others can be formulated as COPNI's.

DEFINITION. A *neighborhood structure* for the COPNI (n, F) is a function $N: F \rightarrow 2^F$.

Informally, N assigns to each feasible solution f its *neighborhood* $N(f)$. We will also informally describe a *local search algorithm* for the COPNI (n, F) and the neighborhood structure N as a deterministic algorithm with input $(f_0; c)$, where $f_0 \in F$ and c is an instance of (n, F). The algorithm is described below in terms of the function IMPROVE (f, c) which, when invoked, returns some $s \in N(f)$ such that $c(s) < c(f)$, if such an s exists, and returns 'no' otherwise.

```
$f := f_0$;
**while** IMPROVE $(f, c) \neg=$ 'no' **do**
  $f :=$ IMPROVE $(f, c)$;
**return** $f$
```

The output of this algorithm is called a *local optimum* with respect to N for the instance c of (n, F). The performance of a local search algorithm depends on the complexity of the function IMPROVE, the number of iterations (executions of the **while** loop) and the quality of the local optima. The neighborhood structure affects all the above factors. In particular N is *exact* if all local optima with respect to N are also global optima. For example, if $N(f) = F$ for all $f \in F$, N will be trivially exact.

The following characterization has been adapted from [10]:

THEOREM 1. *In a* COPNI (n, F) *there exists a unique minimal exact neighborhood structure given by*

$$\hat{N}(f) = \left\{ s \in F\text{: } \begin{array}{l} \textit{for some instance } c, s \textit{ is uniquely} \\ \textit{optimal with } f \textit{ second to optimal} \end{array} \right.$$

The exact nature of the map $\hat{N}$ for the case of the TSP is not known. In fact, the results in [7] suggest that there is no concise, algorithmic-oriented characterization of $\hat{N}$ for the TSP. However, the authors of [13] have shown that for an r city TSP, $\hat{N}$ consists of sets of cardinality at least $((r-2)/2)!$. They continue by arguing that the exponential size of $\hat{N}$ implies that exact enumerative local search for the TSP must be inefficient. The following fact demonstrates that this argument is not valid when nonenumerative (data-dependent) search is allowed:

FACT. *There exists a* COPNI (n, F) *and an* $f \in F$ *such that* $\hat{N}(f)$ *is exponential in size but can be searched in* $O(n)$ *time.*

Proof. Consider the JSD with n odd, $D_i = (n-1)/2$ for $i = 1, 2, \cdots, n$, $T_1 = (n-1)/2$, and $T_j = 1, j = 2, 3, \cdots, n$. The set of feasible solutions is

$$F = \{f\} \cup F',$$

where $f=\{2,3,\cdots,n\}$ and

$$F'=\{s \text{ subset of } \bar{n}: 1\in s \text{ and } |s|=(n+1)/2\}.$$

Consider any $s\in F'$. We can define an instance c_s as follows

$$c_s(j)=\begin{cases}(N-3)/2 & \text{if } j=1,\\ 0 & \text{if } j\neg=1 \text{ and } j\in s,\\ 1 & \text{otherwise.}\end{cases}$$

It can be easily verified that, for this instance, s is uniquely optimal (with cost $(n-3)/2$) with f second to optimal (cost $(n-1)/2$). Hence by Theorem 1, $s\in\hat{N}(f)$ and consequently $\hat{N}(f)=F'$. The cardinality of $\hat{N}(f)$ is approximately equal to $.8n^{-1/2}2^n$.

Yet for any instance c, $\hat{N}(f)$ can be searched in linear time. To see this, let t be the set of jobs in $\{2,3,\cdots,n\}$ having the $(n-1)/2$ largest costs. The optimum is either f or $\bar{n}-t$, depending on whether or not $\sum_{j\in t}c(j)<c(1)$. Consequently in order to search $\hat{N}(f)$ we only need to find the $(n-1)/2$ jobs in $\{2,3,\cdots,n\}$ having the largest cost, and compare the sum of their costs to $c(1)$. But this can be done in $O(n)$ time by using the median algorithm of [2]. □

The idea behind this counterexample is that the minimal exact neighborhood is a data-independent set, whereas data can be used very efficiently in order to facilitate its search. As we will see in the next section there is little hope that something similar can be done in the case of the TSP.

3. The complexity of exact and approximate local search. For the purpose of relating the complexity of local search to the $P=NP$ question, we now show certain related languages to be NP-complete. We assume the existence of a function e mapping graphs, digraphs, paths, TSP tours and instances to strings in $(0,1)^*$. A wide variety of "reasonable" encodings would suffice for our purposes.

DEFINITION. Let V be the set of nodes of a graph (V,E) (resp. a digraph (V,E')) and let $(v^1,v^2,\cdots,v^n)$ be a permutation of V such that (v^i,v^{i+1}) is an edge (resp. a directed edge) for $i=1,2,\cdots,n-1$. If (v^n,v^1) is an edge (resp. directed edge), then $(v_1,\cdots,v_n,v_1)$ is an undirected Hamiltonian circuit (UHC) (resp. directed Hamiltonian circuit (DHC)). Otherwise, if (v^n,v^1) is not an edge (resp. directed edge) then $(v^1,\cdots,v^n)$ is an undirected Hamiltonian path (UHP) (resp. directed Hamiltonian path (DHP)). Note that, by the above definition, no part of a Hamiltonian circuit is a Hamiltonian path.

In [4] the problems of determining whether a given graph (directed or undirected) has a Hamiltonian circuit are shown to be NP-complete. We show that they remain NP-complete even if a serious restriction is imposed on their domains. In particular one would expect that the search for a Hamiltonian circuit in a graph would be facilitated considerably, if we were given a Hamiltonian path. The next two theorems suggest that this is not the case.

The restricted directed Hamiltonian circuit problem is the recognition problem for the following language:

$$\text{RDHC}=\{e(G);e(P):P \text{ is a DHP in } G \text{ and } G \text{ has a DHC}\}.$$

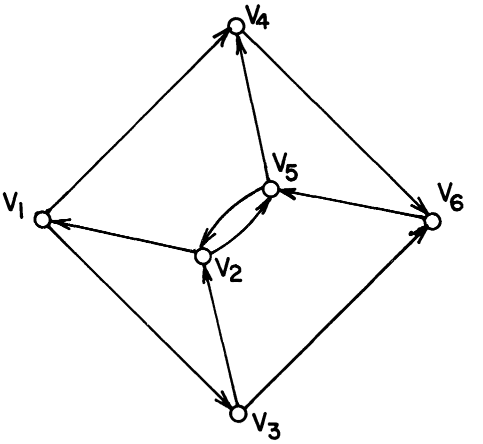


FIG. 1 *The digraph H*

THEOREM 2. RDHC *is* NP-*complete.*

For the proof of Theorem 2, the following lemma is needed:

LEMMA. *Let the digraph H (shown in Fig. 1) be a subgraph of a digraph G, such that edges of $G-H$ enter H only at v_1 or v_3 and leave H at v_4 or v_6 only. Then, if G has a* DHC *C, one of the paths* $(v_1, v_3, v_2, v_5, v_4, v_6)$ *or* $(v_3, v_6, v_5, v_2, v_1, v_4)$ *is a part of C.*

Proof of Lemma. Let C enter H at v_1. Then for some node u of $G-H$ one of the following six paths is a part of C:

1. (v_1, v_4, u),
2. (v_1, v_4, v_6, u),
3. (v_1, v_3, v_6, u),
4. $(v_1, v_3, v_2, v_5, v_4, u)$,
5. $(v_1, v_3, v_6, v_5, v_4, u)$,
6. $(v_1, v_3, v_2, v_5, v_4, v_6, u)$.

In the first five cases it can be easily verified that there is no way for C to pass through the unvisited nodes of H, contrary to our assumption that C is Hamiltonian. Consequently if C enters H at v_1, $(v_1, v_3, v_2, v_5, v_4, v_6)$ is a part of C. If C enters H at v_3, then, again, for some node u of $G-H$ one of the following seven paths is a part of C:

1. (v_3, v_2, v_1, v_4, u),
2. $(v_3, v_2, v_1, v_4, v_6, u)$,
3. (v_3, v_2, v_5, v_4, u),
4. $(v_3, v_2, v_5, v_4, v_6, u)$,
5. (v_3, v_6, u),
6. (v_3, v_6, v_5, v_4, u),
7. $(v_3, v_6, v_5, v_2, v_1, v_4, u)$.

Again, if one of the first six paths is indeed a part of C, C cannot visit the remaining nodes of H. Hence if C enters H at v_3, $(v_3, v_6, v_5, v_2, v_1, v_4)$ is a part of C, which completes the proof of the lemma. □

Proof of Theorem 2. We reduce the DHC problem to RDHC. Let $G = (V, E)$ be an instance of the DHC problem. We will construct a digraph $G' = (V', E')$ with a DHP P, such that G' has a DHC iff G has a DHC.

We let $V = \{v^1, v^2, \cdots, v^n\}$ and $V' = \{v_1^1, v_2^1, \cdots, v_6^1, v_1^2, \cdots, v_6^n\}$. For each $j \leqq n$ we connect the nodes $\{v_1^j, v_2^j, \cdots, v_6^j\}$ as $\{v_1, v_2, \cdots, v_6\}$ are connected in H, and we call the resulting subgraph H^j. Moreover for each edge $(v^i, v^j) \in E$ we add the edge (v_6^i, v_1^j) to E'. We also add the edges (v_4^i, v_3^{i+1}), $i = 1, 2, \cdots, n-1$, to E'.

Obviously G' has a DHP, namely $P = (v_3^1, v_6^1, v_5^1, v_2^1, v_1^1, v_4^1, v_3^2, v_6^2, \cdots, v_1^n, v_4^n)$. Moreover if G has a DHC $(w^1, w^2, w^3, \cdots, w^n, w^1)$, then G' also has a DHC, namely $(w_1^1, w_3^1, w_2^1, w_5^1, w_4^1, w_6^1, w_1^2, \cdots, w_6^n, w_1^1)$.

Conversely, suppose that G' has a DHC C. Suppose that for some i, C enters H^i at v_3^i. By the lemma, $(v_3^i, v_6^i, v_5^i, v_2^i, v_1^i, v_4^i)$ is a part of C. Since v_3^{i+1} is the only node in $G' - H^i$ which succeeds v_4^i, it follows that C will enter H^{i+1} at v_3^{i+1}. Hence the same argument can be applied to H^{i+1}. Inductively, we can assume that C enters H^n at v_3^n. By the lemma, $(v_3^n, v_6^n, v_5^n, v_2^n, v_1^n, v_4^n)$ will be a part of C. But there is no node in $G - H^n$ which succeeds v_4^n. Consequently C is not a DHC as supposed.

From the above contradiction we deduce, that for no $i \leqq n$ will C enter H^i at v_3^i, and hence C is equal to $(w_1^1, w_3^1, w_2^1, w_5^1, \cdots, w_6^n, w_1^1)$ for some DHC $(w_1, w_2, \cdots, w_n, w_1)$ of G. Consequently G has a DHC iff G' has a DHC, and the proof is completed. (The straightforward verification of the facts that the problem is in NP and that the reduction is a polynomial-time one has been omitted). □

Similarly we define the restricted undirected Hamiltonian circuit problem to be the recognition problem of the language

$$\text{RUHC} = \{e(G); e(P) : P \text{ is a UHP in } G \text{ and } G \text{ has an UHC}\}.$$

THEOREM 3. RUHC *is* NP-*complete.*

Proof. We reduce the RDHC to it. The construction is identical to the one used in the proof of the NP-completeness of the ordinary UHC problem [1], [4]. It is an elementary observation that the construction preserves the existence of a Hamiltonian path. □

An interesting side problem of the TSP is the following: given an instance c and an edge (i, j), does (i, j) appear in some optimal tour? This problem is also NP-complete. To show this, we define the language

$$M = \left\{ \begin{array}{ll} e(c); e(i, j): & \text{the edge } (i, j) \text{ does not appear in any} \\ & \text{optimal tour of the instance } c \text{ of the TSP} \end{array} \right\}.$$

THEOREM 4. M *is* NP-*complete.*

Proof. We reduce the RUHC to it. Let $(G; P)$ be an instance of the RUHC, where $P = (w_1, w_2, \cdots, w_n)$ is a UHP. Let c be an instance of the TSP such that $c(w_i, w_j) = 2$ if (w_i, w_j) is not an edge of G, and $c(w_i, w_j) = 1$ otherwise. If $(G, P) \in \text{RUHC}$, then G has a UHC and hence (w_1, w_n) (which, by definition of a UHP, corresponds to a missing edge of G) will not appear in any optimal tour of c. Conversely, if (w_1, w_n) does not appear in any optimal tour of c, then the tour corresponding to P is suboptimal and hence G has a UHC. Consequently $(c, (i, j)) \in M$ iff $(G, P) \in$ RUHC. □

We now define the following language:

$$L_0 = \{e(c);\, e(f) : f \text{ is a suboptimal tour for the instance } c\}.$$

It can be argued that L_0 adequately captures the complexity per iteration of the exact local search problem for the TSP, since the recognition problem for L_0 can be solved by one call of the function IMPROVE (c, f) of any exact local search algorithm. Hence the following result suggests that exact local search for the TSP could require iterations of complexity more than polynomial:

THEOREM 5. *L_0 is* NP-*complete.*

Proof. We reduce RUHC to it. Let $(G = (V, E); P)$ be an instance of the RUHC problem. Let c be an instance of the TSP with $|V|$ cities, such that $c(v, u) = 1$ if $(v, u) \in E$ and $c(v, u) = 2$ otherwise. Let f be the tour corresponding to the path P. Then $(G, P) \in$ RUHC iff $(c, f) \in L_0$. □

Let ε be any positive real number, and c an instance of the COPNI (n, F), with optimal feasible solution s. A feasible solution $f \in F$ is called *ε-approximate* [9] if $(c(f) - c(s))/c(s) \leqq \varepsilon$. Otherwise f is called *ε-suboptimal*. In a similar way to L_0, the following language is defined for $\varepsilon > 0$:

$$L_\varepsilon = \{e(c);\, e(f) : f \text{ is an } \varepsilon\text{-suboptimal tour for the instance } c\}.$$

THEOREM 6. *L_ε is* NP-*complete for all $\varepsilon > 0$.*

Proof. Let $(G = (V, E), P)$ be an instance of the RUHC problem. Let c be the instance of the $|V|$-city TSP with $c(v, u) = 1$ if $(v, u) \in E$ and $c(v, u) = 2 + |V|\varepsilon$ otherwise. f is again the tour corresponding to P. It can be easily seen that $(G, P) \in$ RUHC iff $(c, f) \in L_\varepsilon$. □

We say that a local search algorithm is *ε-approximate* if all local optima produced by this algorithm are ε-approximate. The following theorem suggests that local search algorithms for the TSP with iterations requiring only a polynomial amount of time (such as the ones proposed by [5], [6], [11], [12]) will yield local optima of no guaranteed accuracy.

THEOREM 7. *If $P \neq NP$, local search algorithms having polynomial complexity per iteration cannot be ε-approximate for any $\varepsilon > 0$.*

Proof. It suffices to show how, by using the function IMPROVE of an ε-approximate local search algorithm, we can solve the RUHC problem. Given an instance $(G = (V, E), P)$ of this problem, we construct, as before, the instance c of the $|V|$-city TSP with $c(u, v) = 1$ if $(u, v) \in E$, and $c(u, v) = 2 + |V|\varepsilon$ otherwise, and f, the tour corresponding to the Hamiltonian path P. f has cost $|V|(1 + \varepsilon) + 1$; moreover there is no tour of better cost, unless G has a UHC. Hence $(G, P) \in$ RUHC iff IMPROVE $(f, c) \neq$ 'no'. □

It should be emphasized that Theorem 7 and its implications are valid when no additional restrictions are imposed on the instances of the TSP considered. For example, if a "natural" constraint—the triangle inequality—holds among the intercity distances, there are polynomial-time algorithms (not necessarily of iterative nature) yielding 1-approximate [8] and $\frac{1}{2}$-approximate [3] solutions.

REFERENCES

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

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

[3] N. CHRISTOFIDES, Private communication, March 1976.

[4] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–103.

[5] S. LIN, *Computer solutions of the traveling salesman problem*, Bell System Tech. J., 44 (1965), pp. 2245–2269.

[6] S. LIN AND B. W. KERNIGHAN, *An effective heuristic algorithm for the traveling salesman problem*, Operations Res., 21 (1973), pp. 498–516.

[7] C. H. PAPADIMITRIOU, *The complexity of the local structure of certain convex polytopes*, Proc. 1976 Conf. on Information Systems and Sci., Johns Hopkins Univ., Baltimore, Md., March 31–April 2, 1976, pp. 47–51.

[8] D. J. ROSENKRANTZ, R. E. STEARNS AND P. M. LEWIS, *Approximate algorithms for the traveling salesperson problem*, IEEE 15th Annual Symp. on Switching and Automata Theory, Univ. of New Orleans, New Orleans, La., Oct. 14–16, 1974, pp. 33–42.

[9] S. SAHNI AND T. GONZALES, *P-complete approximation problems*, J. Assoc. Comput. Mach., 23 (1976), pp. 555–565.

[10] S. L. SAVAGE, P. WEINER AND M. J. KRONE, *Convergent local search*, RR #14, Dept. of Comput. Sci., Yale Univ., New Haven, Conn., 1973.

[11] S. REITER AND G. S. SHERMAN, *Discrete optimizing*, SIAM J. Appl. Math., 13 (1965), pp. 864–889.

[12] K. STEIGLITZ AND P. WEINER, *Some improved algorithms for computer solution of the traveling salesman problem*, Proc. 6th Ann. Allerton Conf. on Circuit and System Theory, Univ. of Ill., Urbana, Ill., Oct. 1968, pp. 814–821.

[13] P. WEINER, S. L. SAVAGE AND A. BAGGHI, *Neighborhood search algorithms for guaranteeing optimal traveling salesman tours must be inefficient*, J. Comput. System Sci., 12 (1976), pp. 25–35.

SIAM J. COMPUT.
Vol. 6, No. 1, March 1977

A FAST MONTE-CARLO TEST FOR PRIMALITY*

R. SOLOVAY† AND V. STRASSEN‡

Abstract. Let n be an odd integer. Take a random number a from a uniform distribution on the set $\{1, 2, \cdots, n-1\}$. If a and n are relatively prime, compute the residue $\varepsilon \equiv a^{(n-1)/2} (\mathrm{mod}\ n)$, where $-1 \leqq \varepsilon < n-2$, and the Jacobi symbol $\delta = (a/n)$. If $\varepsilon = \delta$, decide that n is prime. If either gcd $(a, n) > 1$ or $\varepsilon \neq \delta$ decide that n is composite. Obviously, if n is prime, the decision made will be correct. We will show below, that for composite n the probability of an incorrect decision is $\leqq 1/2$. The number of multiprecision operations needed for the whole procedure is $<6 \log_2 n$. m-fold repetition using independent random numbers yields a Monte-Carlo test for primality with error probabilities 0 (if n is prime) and $<2^{-m}$ (if n is composite) and with multiprecision arithmetic cost $<6m \log_2 n$.

Key words. Monte-Carlo tests, primality

1. Cost of the procedure. By a multiprecision operation we mean an arithmetic operation or a division with remainder of two numbers $<n^2$. To decide whether a and n are relatively prime, we compute (a, n) by Euclid's algorithm. This can be done with approximately $1.5 \log_2 n$ multiprecision operations (see Knuth [1, p. 320]). Computing ε can be done by $1.25 \log_2 n$ multiplications each followed by a reduction mod n, i.e., by $2.5 \log_2 n$ multiprecision operations (Knuth [1, p. 409]). We compute δ with the help of the reciprocity law for Jacobi symbols ([2, p. 79]). This is about as hard as computing (a, n). The total number of multiprecision operations of the procedure can therefore be estimated from above by $6 \log_2 n$.

2. Error probability. If n is prime, the procedure obviously reaches a correct decision. Let n be composite. The set

$$G = \left\{ a + (n) | a \in \mathbb{Z} \ \& \ (a, n) = 1 \ \& \ a^{(n-1)/2} \equiv \left(\frac{a}{n}\right) (\mathrm{mod}\ n) \right\}$$

is a subgroup of the group of units $\mathbb{Z}_n^\times$ of $\mathbb{Z}_n$. Therefore it suffices to show $G \neq \mathbb{Z}_n^\times$ (for this implies $|G| \leqq \frac{1}{2}|\mathbb{Z}_n^\times| \leqq (n-1)/2$, so that at most $\frac{1}{2}$ of the numbers between 1 and $n-1$ will lead to the decision that n is prime).

By the way of contradiction assume

$$a^{(n-1)/2} \equiv \left(\frac{a}{n}\right) (\mathrm{mod}\ n) \tag{1}$$

for all $a \in \mathbb{Z}$ relatively prime to n. If $n = p^e$ is a prime power, we get from (1)

$$a^{pe-1} \equiv 1\ (\mathrm{mod}\ p^e)$$

for all a not divisible by p. Since $\mathbb{Z}_{p^e}^\times$ is cyclic of order $p^{e-1}(p-1)$ we have

$$p^{e-1}(p-1) | p^e - 1$$

* Received by the editors June 12, 1974, and in final revised form October 6, 1975.

† IBM Watson Laboratory, Yorktown Heights, New York 10598.

‡ Seminar fur Angewandte Mathematik, Universität Zurich, 8032 Zurich, Switzerland.

and therefore $e \leqq 1$, which is impossible since n is composite. Thus n has a nontrivial factorization $n = r \cdot s$ with $(r, s) = 1$. Equation (1) implies

$$a^{(n-1)/2} \equiv \pm 1 \pmod{n} \tag{2}$$

for all a relatively prime to n. We claim that in fact

$$a^{(n-1)/2} \equiv 1 \pmod{n} \tag{3}$$

for such a. Otherwise there is an a with $a^{(n-1)/2} \equiv -1 \pmod{n}$. Since r and s are relatively prime we can apply the Chinese remainder theorem and find b with $b \equiv 1 \pmod{r}$, $b \equiv a \pmod{s}$. Then

$$b^{(n-1)/2} \equiv 1 \pmod{r}, \qquad b^{(n-1)/2} \equiv -1 \pmod{s},$$

in contradiction to (2). Equation (3) implies

$$\left(\frac{a}{n}\right) = 1$$

for all a relatively prime to n, which is impossible.

Remarks 1. Our result should not be confused with assertions as to n being prime or not which are correct with high probability given that n is a random number sampled from the uniform distribution on a sufficiently large segment of the integers. Under such a hypothesis it is reasonable to decide that n is composite without even looking at it. The probability of error may be further substantially reduced by checking, e.g., whether

$$2^n \equiv 2 \pmod{n}$$

(see Erdös [3]).

2. Perhaps it is useful to measure the complexity of a Monte-Carlo test (with one probability of error = 0 as above) by a single quantity. If the test has error probability α and cost t, we suggest $t/(-\log \alpha)$ as such a measure, since this is invariant under independent repetition.

Acknowledgment. It is a pleasure to thank Ernst Specker for interesting discussions about the subject.

REFERENCES

[1] D. E. KNUTH, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, Addison-Wesley, Reading, Mass., 1969.

[2] I. NIVEN AND H. S. ZUCKERMAN, *An Introduction to the Theory of Numbers*, John Wiley, New York, 1966.

[3] P. ERDÖS, *On almost primes*, Amer. Math. Monthly, 57 (1950), pp. 404–407.

SIAM J. COMPUT.
Vol. 6, No. 1, March 1977

COMMENTS ON F. HADLOCK'S PAPER: "FINDING A MAXIMUM CUT OF A PLANAR GRAPH IN POLYNOMIAL TIME"*

K. AOSHIMA AND M. IRI†

Although the final result stated in the above-mentioned paper [1] is correct, the author's arguments leading to it seem to want some correction.

In Theorem 2 he states that

> An edge set D is an odd-circuit cover of a planar graph G if and only if the corresponding edge set P is an odd-vertex pairing for the geometric dual G_D of G.

This statement is not valid, because he defines both "odd-circuit covers" and "odd-vertex pairings" with regard to the "removal" of the edges in consideration (which operation apparently means "opening the edges" in another terminology) and because the operation of removal (i.e., opening) of edges in a graph G corresponds to that of "contraction" (i.e., shortening) of the corresponding edges in the geometrical dual G_D. (It is easy to find a counterexample to the theorem.) Thus, we should replace the term "odd-vertex pairing" by another concept, e.g., "odd-vertex cover", which is defined as a set of edges such that their "contraction (shortening)" yields a graph all of whose vertices are of even degree.

Fortunately, however, the family of minimum odd-vertex covers coincides with that of minimum odd-vertex pairings, whereas the family of odd-vertex covers properly includes that of odd-vertex pairings. (Hence, Hadlock's final result remains correct.) In fact, it is almost obvious that an odd-vertex pairing is an odd-vertex cover, and it can be proved, as will be done below, that a *minimum* odd-vertex cover is an odd-vertex pairing.

In order to prove this last fact, let us consider a minimum odd-vertex cover C and the partial graph G_C of the original graph G which has C as the edge set.

First of all, we note that G_C is a forest, because, if there were a circuit in G_C, $C-\{e\}$ would remain an odd-vertex cover where e is any edge of the circuit.

Secondly, we can show that the parity of the degree of each vertex in G_C is the same as that of the corresponding vertex in G. In fact, the vertex of G corresponding to an isolated vertex in G_C is obviously of even degree. Let us consider a vertex v of odd (positive even) degree of G_C and the corresponding vertex $\bar{v}$ of G. Since C is an odd-vertex cover of G, the number of those vertices of odd degree of G which correspond to the vertices of the connected component of G_C containing v must be even. Let us then remove (i.e., open) all the edges incident to v in G_C, and denote by $G_{C1}, G_{C2}, \cdots$ the connected components of the resulting graph which contain, respectively, the vertices adjacent to v in G_C. If the parity of the degree of v in G_C did not coincide with that of $\bar{v}$ in G, there should exist at least one G_{Ci}, say G_{C1}, which contains an even number of the vertices corresponding to the vertices of odd degree of G (because there are an odd (even) number of G_{Ci}'s, $\bar{v}$ is of even (odd) degree in G and the total number of those vertices in the connected

* This Journal, 4 (1975), pp. 221–225. Received by the editors December 12, 1975.

† Department of Mathematical Engineering and Instrumentation Physics, Faculty of Engineering, University of Tokyo, Bunkyo-ku, Tokyo, Japan.

component of G_C containing v which correspond to vertices of odd degree in G must be even). Hence if we removed from C the edge which connects v and its adjacent vertex in G_{C1}, we should have another smaller odd-vertex cover, which would contradict the minimality of C.

Finally, by virtue of the famous theorem of L. Euler on the unicursal problem, we can decompose C into a set of edge-disjoint paths which establishes a pairwise correspondence among the vertices of odd degree of G. Therefore, C is an odd-vertex pairing.

REFERENCE

[1] F. HADLOCK, *Finding a maximum cut of a planar graph in polynomial time*, this Journal, 4 (1975), pp. 221–225.

SIAM J. COMPUT.
Vol. 6, No. 1, March 1977

A GRAPH-THEORETIC CHARACTERIZATION OF THE $\mathbf{PV_{chunk}}$ CLASS OF SYNCHRONIZING PRIMITIVES*

PETER B. HENDERSON† AND YECHEZKEL ZALCSTEIN‡

Abstract. Many of the process synchronization problems studied in the literature are of the form of a conjunction of finitely many conditions of the type "process p_i blocks process p_j". Such problems may be expressed as directed graphs whose nodes represent the processes and where there is an edge from node i to node j if and only if process p_i blocks process p_j. We characterize the class of graphs which correspond to the system of synchronizing primitives of Vantilborgh and van Lamsweerde in terms of a normal form representation and present an efficient algorithm for determining whether an arbitrary graph is in this class.

Key words. synchronization problem, PV_{chunk}, PV_c-definable graph, normal form, interval graph

Introduction. Vantilborgh and van Lamsweerde [14] have introduced an extension of Dijkstra's PV system of primitives [4] (dubbed PV_{chunk} or PV_c in [8]) allowing the semaphores to be updated by "chunks" that are arbitrary positive integers.

An analysis of the purely parallel behavior of systems of processes has been undertaken in [8]–[10]. Herein, we characterize, in a sense to be made precise below, the purely parallel behavior of PV_{chunk} systems (in the terminology of Lipton [8], the exclusion slices implicitly definable by PV_{chunk} systems). This solves a problem left open in [8] and [10].

1. Statement of the problem.

1. Statement of the problem. A *purely parallel program* $\mathcal{P}$ is a program of the form

$$\textbf{parbegin } Q_1 // \cdots // Q_m \textbf{ parend}$$

where **parbegin** $\cdots$ **parend** is Dijkstra's parallel block notation [4] and each Q_i is a single statement of the form [8]

$$\textbf{when } B(x_1, \cdots, x_n) \wedge b_i = 1 \textbf{ do } \Theta(x_1, \cdots, x_n);\ b_i \leftarrow 0;$$

where $x_1, \cdots, x_n$ are the program variables of $\mathcal{P}$, $B(x_1, \cdots, x_n)$ is a predicate, b_i is a Boolean variable distinct from $x_1, \cdots, x_n, b_1, \cdots, b_{i-1}, b_{i+1}, \cdots, b_m$ and $\Theta(x_1, \cdots, x_n)$; $b_i \leftarrow 0$; is a simultaneous assignment of the form

$$x_1 \leftarrow \Theta_1(x_1, \cdots, x_n); \cdots ; x_n \leftarrow \Theta_n(x_1, \cdots, x_n);\ b_i \leftarrow 0.$$

In use, assignments of the form $x_j \leftarrow x_j$ will be deleted. Each Q_i is called a (purely parallel) *process*. Note that in a purely parallel program $\mathcal{P}$ each statement (corresponding to a process) can execute at most once.

* Received by the editors April 23, 1975, and in revised form February 26, 1976.

† Department of Computer Science, State University of New York at Stony Brook, Stony Brook, New York 11790. The work of this author was supported in part by the National Science Foundation under Grant ENG75-10228.

‡ Department of Computer Science, State University of New York at Stony Brook, Stony Brook, New York 11790. The work of this author was supported in part by the National Science Foundation under Grant DCR75-01998.

A *purely parallel system of processes* is a pair $(\mathcal{P}, I)$, where $\mathcal{P}$ is a purely parallel program and I is the vector of initial values of the program variables $x_1, \cdots, x_n$ and Boolean variables b_i, $i=1,2,\cdots,m$. Furthermore, it is required that for each Q_i of the form

$$\textbf{when } B(x_1,\cdots,x_n)\wedge b_i=1 \textbf{ do } \Theta(x_1,\cdots,x_n);\ b_i\leftarrow 0;$$

the predicate $B(x_1,\cdots,x_n)\wedge b_i=1$ is true at I.

The concept of a purely parallel system of processes is a formalization of the notion of a synchronization problem with a fixed number of processes. For example, a purely parallel system of processes $(\mathcal{P}, I)$ which captures the underlying behavior of the reader-writer synchronization problem [2] with k independent reader processes $R_1, R_2, \cdots, R_k$ and writer process W is:

```
parbegin $W//R_1//R_2// \cdots //R_k$ parend with
$W$: when $x_1=0\wedge x_2=0\wedge b_0=1$ do $x_2\leftarrow x_2+1$; $b_0\leftarrow 0$;
$R_i$: when $x_2=0\wedge b_i=1$ do $x_1\leftarrow x_1+1$; $b_i\leftarrow 0$;
```

for $i=1,2,\cdots,k$ and $I=(x_1,x_2,b_0,b_1,\cdots,b_k)=(0,0,1,1,\cdots,1)$.

It should be noted that because of the restriction to a fixed number of processes, the original reader–writer problem [2], where the number of processes is potentially infinite, cannot be accommodated in this framework.

A *state* of $(\mathcal{P}, I)$ is the vector of values of the program variables $x_1,\cdots,x_n$ and the Boolean variables b_i, $i=1,2,\cdots,m$. Let Z be the set of states of $(\mathcal{P}, I)$. Each process Q_i of the form

$$\textbf{when } B(x_1,\cdots,x_n)\wedge b_i=1 \textbf{ do } \Theta(x_1,\cdots,x_n);\ b_i\leftarrow 0;$$

defines a partial function $\bar{Q}_i: Z\to Z$ as follows:

If the predicate $B(x_1,\cdots,x_n)\wedge b_i=1$ is true at $z\in Z$, then $\bar{Q}_i(z)$ is the resulting state after simultaneously executing all assignments $b_i\leftarrow 0$ and $x_j\leftarrow\Theta_j(x_1,\cdots,x_n)$, $j=1,2,\cdots,n$. If the predicate $B(x_1,\cdots,x_n)\wedge b_i=1$ is false at $z\in Z$, then the function $\bar{Q}_i(z)$ is undefined at z.

A sequence $Q_{i_1}\cdots Q_{i_k}$ of processes defines a partial function $\bar{Q}_{i_1}\circ\cdots\circ\bar{Q}_{i_k}: Z\to Z$ by functional composition; that is,

$$(z)\bar{Q}_{i_1}\circ\cdots\circ\bar{Q}_{i_k}=\bar{Q}_{i_k}(\cdots(\bar{Q}_{i_2}(\bar{Q}_{i_1}(z)))\cdots)\quad\text{for } z\in Z.$$

A *computation* of $(\mathcal{P}, I)$ is a sequence $Q_{i_1}\cdots Q_{i_k}$ of processes such that the partial function $\bar{Q}_{i_1}\circ\cdots\circ\bar{Q}_{i_k}$ is defined at I. Note that it follows from the definition of a purely parallel system of processes that no process can occur twice in a computation. The *behavior* of $(\mathcal{P}, I)$ is the (necessarily finite) set of computations of $(\mathcal{P}, I)$. The concept of the behavior of a purely parallel system of processes was introduced by Lipton [8], where it was termed "slice definable by a system of processes".

Let $(\mathcal{P}, I)$ be a purely parallel system of processes. We will say that a process Q_i *blocks* process Q_j provided that Q_iQ_j is not a computation of $(\mathcal{P}, I)$. It follows from the definition of a purely parallel program that any process Q_i blocks itself.

Consider the subclass $\mathscr{C}$ of the class of purely parallel systems of processes for which $Q_{i_1} \cdots Q_{i_k}$ is a computation if and only if for all j and l such that $1 \leqq j < l \leqq k$, Q_{i_j} does not block Q_{i_l}. In the terminology of Lipton [8], the class $\mathscr{C}$ corresponds to the "exclusion slices definable by a system of processes".

Many of the process synchronization problems studied in the literature are of this type. Examples are:

1. the "reader-writer problem" [2], in which "a writer (process) blocks all readers (processes) and all other writers", and "each reader blocks all writers" (note that a reader cannot block another reader), and

2. the "five dining philosophers problem" [5], where each of the five processes blocks exactly two other distinct processes and in turn is blocked only by these two processes.

Let $(\mathscr{P}, I)$ be in class $\mathscr{C}$. Utilizing the relation "blocks" on the processes of $(\mathscr{P}, I)$, the behavior of $(\mathscr{P}, I)$ may be expressed by a directed graph $G = (N, E)$, where node set N corresponds with the processes of $(\mathscr{P}, I)$ and E corresponds to the relation "blocks" on these processes (i.e. there is an edge (i, j) in E if and only if process Q_i blocks process Q_j). Accordingly, $(\mathscr{P}, I)$ *defines* graph G provided that there is a one-to-one correspondence $\gamma: \{Q_1, \cdots, Q_m\} \to N$ so that $Q_{i_1} \cdots Q_{i_k}$ is a computation of $(\mathscr{P}, I)$ if and only if $(\gamma(Q_{i_j}), \gamma(Q_{i_l})) \notin E$, for all $1 \leqq j < l \leqq k$. With this correspondence, we say that graph G is *definable* by the system $(\mathscr{P}, I)$.

The graphs definable by many systems of processes have been characterized [8], [10]. For example, the class of graphs definable by PV are all disjoint unions of complete graphs [8, pp. 86, 90] and the class of graphs definable by any of PV-multiple, up/down, or vector addition systems is the class of symmetric graphs (a graph is said to be symmetric when the relation E is symmetric) [10, § 5]. Also, vector replacement systems or Petri nets define all (directed) graphs [10, § 5]. In contrast with these results, the characterization of the $\mathrm{PV_c}$-definable graphs turns out to be considerably more intricate.

A purely parallel system of processes $(\mathscr{P}, I)$ is a purely parallel $\mathrm{PV_{chunk}}$ (abbreviated $\mathrm{PV_c}$) system of processes provided that (a) there is a distinguished nonempty subset $\mathscr{S}$ of the program variables (the variables in $\mathscr{S}$ are called *semaphores*) and (b) there are only statements of the form:

1. **when** $S \geqq a \wedge b_i = 1$ **do** $S \leftarrow S - a$; $b_i \leftarrow 0$;

2. **when** true $\wedge\, b_j = 1$ **do** $S \leftarrow S + c$; $b_j \leftarrow 0$;

where a and c are positive integers and $S \in \mathscr{S}$. A statement of type 1 is denoted by $P[S, a]$, a "delay" or "wait" primitive in [14]. Statements of type 2 are denoted by $V[S, c]$, the "completion" primitives in [14].

It is important to note that if graph $G = (N, E)$ is defined by the $\mathrm{PV_c}$ system $(\mathscr{P}, I)$, then a node x in G which corresponds to a $V[\cdot, \cdot]$ process in $\mathscr{P}$ is necessarily disconnected from all other nodes in G since a $V[\cdot, \cdot]$ process can neither block nor be blocked by any other process in $\mathscr{P}$. Hence, without loss of generality we consider only the class of graphs defined by the class of purely parallel $\mathrm{PV_c}$ systems of $P[\cdot, \cdot]$ processes. As an illustrative example, consider the purely parallel system $(\mathscr{P}, I)$ with $I = S = 3$ and program $\mathscr{P}$ below:

parbegin $Q_1{:}\,P[S, 3]//Q_2{:}\,P[S, 2]//Q_3{:}\,V[S, 1]$ **parend**.

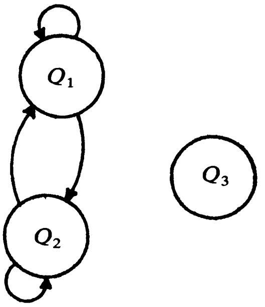


FIG. 1. *Graph definable by* PV_c *system* $(\mathcal{P}, I)$

This system defines the graph in Fig. 1.

It is easy to see that a graph definable by a PV_c system—termed a PV_c-*definable graph*—is a disjoint union of graphs definable by PV_c systems with a single semaphore. Thus in the remainder of this paper, "PV_c-definable" will mean "PV_c-definable using a single semaphore". In addition, in [10, Corollary 5.2.1] it is shown that the relation "blocks" on $P[\cdot, \cdot]$ processes is symmetric; therefore, any PV_c-definable graph may be considered as an undirected graph. In the sections following all graphs are assumed to be undirected. Also, since the relation blocks is reflexive (i.e. $(\gamma(Q_i), \gamma(Q_i)) \in E$ for $i = 1, 2, \cdots, m$), without loss of generality we consider only irreflexive graphs (no node is adjacent to itself) and implicitly assume that each process blocks itself. This assumption simplifies the presentation in the following sections.

Let $(\mathcal{P}, I)$ be a purely parallel PV_c system of processes with a single semaphore S whose initial value is t. Then the definition of a purely parallel PV_c system implies $Q_{i_1}Q_{i_2}\cdots Q_{i_k}$ is a computation, with each Q_{i_j} being a $P[S, a_j]$ process, if and only if $\sum_{j=1} a_j \leqq t$. With respect to the class of PV_c-definable graphs, the latter condition is equivalent to stating that the sets of nodes $\{\gamma(Q_{i_1}), \cdots, \gamma(Q_{i_k})\}$ in G is totally disconnected (i.e. $\gamma(Q_{i_l})$ is not adjacent to node $\gamma(Q_{i_j})$ for any $1 \leqq l \neq j \leqq k$). It is precisely this class of graphs which is characterized in this paper.

In view of the preceding statements, the problem we pose and solve can be stated as a graph-theoretic problem. Therefore, the bulk of this paper pursues a self-contained graph-theoretic development and can be read without any knowledge of synchronizing primitives.

2. Fundamental graph theory concepts.

This section introduces the fundamental concepts from graph theory required in this paper.

DEFINITION. A *graph* $G = (N, E)$ consists of a finite set of *nodes* N and a symmetric relation E on N. Each unordered pair x, y of nodes in E is an *edge* of G and the nodes x and y are said to be *adjacent*.

A graph G is *irreflexive* provided that the relation E is irreflexive (i.e. for all $x \in N$, $(x, x) \notin E$). Henceforth, all graphs will be assumed to be irreflexive.

A graph $G = (N, E)$ is a *labeled graph* when a positive integer is associated with each node in G. Let $l(x)$ denote the label for $x \in N$. The *cardinality* of graph G, denoted by $|G|$, is the cardinality of the set N.

A *subgraph* $s(N')=(N',E')$ of G is the graph for which $N'\subseteq N$ and $E'=E\cap(N'\times N')$.

A *path* in G is a sequence of nodes $(x_1, x_2, \cdots, x_k)$ in which all nodes are distinct and there is an edge (x_i, x_{i+1}) for all $1\leqq i<k$. A graph is *connected* if there is a path between every pair of nodes. A *connected component* of graph $G=(N,E)$ is a connected subgraph $s(N')$ of G such that there is no path from nodes in N' to nodes in $N-N'$. A connected component is *nontrivial* provided that $|N'|>1$.

The *degree* of a node x in G, denoted $\deg(x)$, is the number of nodes adjacent to x. The set of all nodes adjacent to x is denoted by ADJ (x). If all nodes in G have degree $|G|-1$, then G is said to be a *complete graph*. A *complete subgraph* or *clique* of G may be similarly defined. A graph is *totally disconnected* when the relation E is $\varnothing$. A graph G is called a *proper graph* if it is connected but *not* complete.

We now make precise the concept of a PV$_c$-definable graph as discussed in the previous section.

DEFINITION. An irreflexive graph $G=(N,E)$ is PV$_c$-*definable* if and only if there is a positive integer t (the threshold) such that the nodes of G can be labeled by positive integers $\leqq t$, so that given a set F of nodes, the subgraph $s(F)$ is totally disconnected if and only if $\sum_{z\in F} l(z)\leqq t$. Any complete graph (possibly empty) is PV$_c$-definable (let $t=1$ and $l(x)=1$ for each $x\in N$). Any labeling of G which satisfies the aforementioned constraints is said to be an *admissible labeling.*

Example 1. The following graphs (see Fig. 2) are PV$_c$-definable with the labeling shown.

Example 2. The following graphs (see Fig. 3) are *not* PV$_c$-definable. Graph Fig. 3(a) represents the "dining philosophers" synchronization problem and is shown *not* to be PV$_c$-definable in [8, Thm. 9, p. 89] by an ad hoc argument. That the graphs in Fig. 3(b) and 3(c) are not PV$_c$-definable follows from Lemma 1 and Theorem 1 respectively.

3. Properties of PV$_c$-definable graphs. Several properties of PV$_c$-definable graphs proven in Lemmas 1 through 6 will be utilized subsequently to characterize all PV$_c$-definable graphs. Also, these results are prerequisite to the derivation of an algorithm to determine whether an arbitrary graph is PV$_c$-definable.

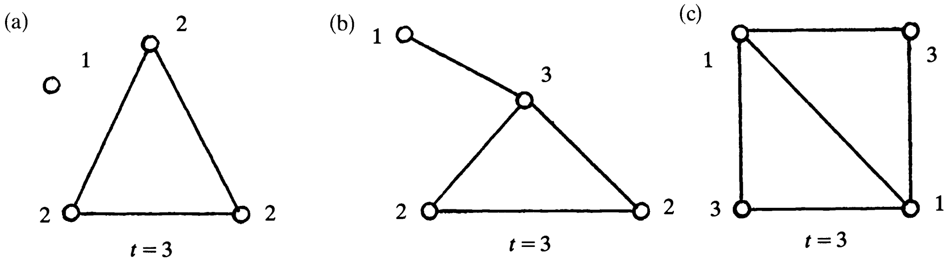


FIG. 2. *Examples of* PV$_c$-*definable graphs*

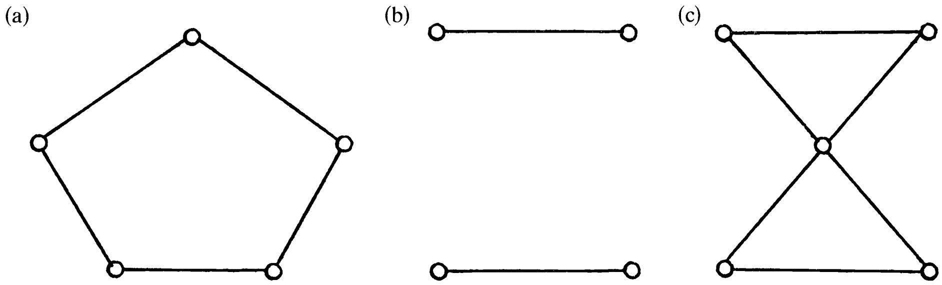


FIG. 3. *Graphs that are not* PV$_c$-*definable*

LEMMA 0. *Let G be a* PV$_c$-*definable graph. If two nodes x and y in G are adjacent, then $l(x)+l(y)>t$.*

Proof. Obvious. □

LEMMA 1. *If G is a* PV$_c$-*definable graph, then there is at most one nontrivial connected component in G.*

Proof. Assume there are at least two nontrivial connected components in G. Let x and y be adjacent nodes in one connected component, and w and z be adjacent nodes in another connected component. Applying the definition of a PV$_c$-definable graph any admissible labeling satisfies the following inequalities:

$$l(x)+l(y)>t, \tag{1}$$

$$l(w)+l(z)>t. \tag{2}$$

Also,

$$l(x)+l(w)\leqq t, \tag{3}$$

$$l(y)+l(z)\leqq t. \tag{4}$$

This is an immediate contradiction since (1) and (2) imply $l(x)+l(y)+l(w)+l(z)>2t$, and (3) and (4) imply $l(x)+l(w)+l(y)+l(z)\leqq 2t$. □

Lemma 1 implies that all PV$_c$-definable graphs consist of a possibly empty totally disconnected component and a possibly empty connected component. Several of the following lemmas concentrate on characterizing the connected component of a PV$_c$-definable graph.

Now the subgraph consisting of nodes of maximal degree in G is characterized for a PV$_c$-definable graph.

LEMMA 2. *Let G be a nonempty, connected* PV$_c$-*definable graph; then the set C_1 of the nodes in G of degree $|G|-1$ is nonempty; thus $s(C_1)$ is a nonempty clique. Furthermore, any node in G is adjacent to all nodes in C_1.*

Proof. Clearly the lemma is valid for $|G|\leqq 1$; hence assume $|G|\geqq 2$. Consider any admissible labeling of G. Let x be a node of maximal label. If a node z is *not* adjacent to x, then for all nodes y,

$$l(y)+l(z)\leqq l(x)+l(z)\leqq t$$

and z is not adjacent to any node in G. This contradicts the assumed connectivity of G. Therefore x is adjacent to all other nodes in G; thus $\deg(x)=|G|-1$. Let C_1 be the set of all nodes of degree $|G|-1$. By the above argument, C_1 is nonempty and $s(C_1)$ is clearly a clique. □

Lemma 3 concerns itself with relationships between the degrees of the nodes and their associated labels.

LEMMA 3. *Consider any admissible labeling for PV_c-definable graph G. Then for any two nodes x, y in G, $\deg(x)<\deg(y)$ implies $l(x)<l(y)$.*

Proof. Assume $l(x)\geqq l(y)$. Then for all z in G, $l(x)+l(z)\geqq l(y)+l(z)$. Hence any node adjacent to y must be adjacent to x and so $\deg(x)\geqq\deg(y)$, contrary to the hypothesis. Thus, $l(x)<l(y)$. □

LEMMA 4. *If $G=(N,E)$ is a PV_c-definable graph, then any subgraph $G'=s(N')$, $N'\subseteq N$ of G is PV_c-definable.*

Proof. Let G be a PV_c-definable graph and consider any admissible labeling of G. We claim the subgraph G' is PV_c-definable. If $F\subseteq N'$ and $s(F)$ is totally disconnected, then $s(F)$ is totally disconnected in G, implying $\sum_{z\in F} l(z)\leqq t$. Conversely, if $F\subseteq N'\subseteq N$ and $\sum_{z\in F} l(z)\leqq t$, then since G is PV_c-definable, it follows that $s(F)$ is totally disconnected in G and thus in G'. Hence, G' is PV_c-definable with the same labeling and threshold as G. □

LEMMA 5. *If $G=(N,E)$ is a proper PV_c-definable graph, and letting $C_1=\{x\in G:\deg(x)=|G|-1\}$, then $D_1=\{x\in G:\mathrm{ADJ}(x)=C_1\}$ is nonempty.*

Proof. Lemma 2 implies that $s(C_1)$ is a nonempty clique, and G being proper implies $|G|>|C_1|$.

Assume $D_1=\emptyset$. Consider the nonempty subgraph $G'=s(N-C_1)$ obtained by deleting all nodes in C_1 and all edges incident on these nodes. Since $D_1=\emptyset$ and G is connected it must be that G' consists of one or more nontrivial connected components. However, Lemma 4 implies G' is PV_c-definable; thus it follows from Lemma 1 that G' is connected. But, G' is a nonempty, connected PV_c-definable graph; thus Lemma 2 implies there is at least one node x in G' of degree $|G'|-1$. This is a contradiction, since in G, $\deg(x)=(|G'|-1)+|C_1|=|G|-1$; that is, $x\in C_1$. Hence, D_1 cannot be empty. □

LEMMA 6. *Let G be a proper PV_c-definable graph and let C_1 be defined as in Lemma 2; then the following statements regarding node x in G are equivalent.*

(1) $\mathrm{ADJ}(x)=C_1$,

(2) $\deg(x)=|C_1|$,

(3) *x has minimal degree.*

Proof. That (1) implies (2) is immediate and (2) implies (1) follows from Lemma 2. We now show that (2) implies (3). For any node y in G, $|C_1|\leqq\deg(y)$, since all nodes in C_1 are adjacent to y by Lemma 2. Thus if $\deg(x)=|C_1|$, then x has minimal degree.

To show (3) implies (2), assume x has minimal degree. For any proper PV_c-definable graph, Lemma 5 implies $D_1=\{y\in G:\mathrm{ADJ}(y)=C_1\}$ is nonempty. Thus there is a node y in D_1 with $\deg(y)=|C_1|$, and $\deg(x)\leqq\deg(y)=|C_1|$, since x is of minimal degree. But, x is in G implying $|C_1|\leqq\deg(x)$ by the above argument; therefore, $\deg(x)=|C_1|$. □

4. Characterization of PV_c-definable graphs. In this section we characterize the PV_c-definable graphs in terms of a normal form representation.

DEFINITION. A graph G is said to be in *Normal Form* (NF) if either G is empty or the nodes of G can be partitioned into sets $C_1, C_2, \cdots, C_k$, $D_0, D_1, \cdots, D_{k-1}$ for $k \geqq 1$, where each $s(C_i)$, $i = 1, 2, \cdots, k-1$ is a nonempty clique, $s(C_k)$ is a possibly empty clique, and each $s(D_i)$, $i = 0, 1, \cdots, k-1$ is a possibly empty totally disconnected subgraph, such that:

$$(*)\quad \begin{gathered} \text{For every node } x \text{ in } C_i, i = 1, 2, \cdots, k-1 \\ \text{ADJ}(x) = \left(\left(\bigcup_{j=1}^{k} C_j - \{x\}\right) \cup \left(\bigcup_{j=i}^{k-1} D_j\right)\right) \\ \text{and for every node } x \text{ in } D_i, i = 0, 1, \cdots, k-1 \\ \text{ADJ}(x) = \cup_{j=1}^{i} C_j \text{ (by convention } \cup_{j=1}^{0} C_j = \varnothing). \end{gathered}$$

Henceforth, a graph G in normal form may be denoted $((C_1, C_2, \cdots, C_k), (D_0, D_1, \cdots, D_{k-1}))$. We refer to $((C_1, C_2, \cdots, C_k), (D_0, D_1, \cdots, D_{k-1}))$ as the *normal form representation* for G. The integer k will be called the *depth* of the normal form representation. Pictorially, a graph in normal form can be viewed as a "left recursive rooted binary tree" or a "comb". For the normal form illustrated in Fig. 4, the edges are assumed to be "upward transitive" (i.e. all the adjacency relationships following from (*) are assumed implicitly).

Some properties of the normal form representation follow directly from the definition.

LEMMA 7. *In a normal form representation* $((C_1, C_2, \cdots, C_k), (D_0, D_1, \cdots, D_{k-1}))$, *if* $x, y \in C_i$, $1 \leqq i \leqq k$ *or* $x, y \in D_j$, $1 \leqq j \leqq k-1$, *then* $\deg(x) = \deg(y)$.

Proof. That $\deg(x) = \deg(y)$, for x, y in C_i, $1 \leqq i \leqq k$ is clear from the normal form definition. Also, by definition, nodes x, y in D_l, $0 \leqq l \leqq k-1$ are adjacent to and only to every node in C_j, $j = 1, 2, \cdots, l$; thus $\deg(x) = \deg(y)$. □

For a NF let $\deg(C_i)$ and $\deg(D_j)$ denote the degree of the nodes in C_i and D_j respectively.

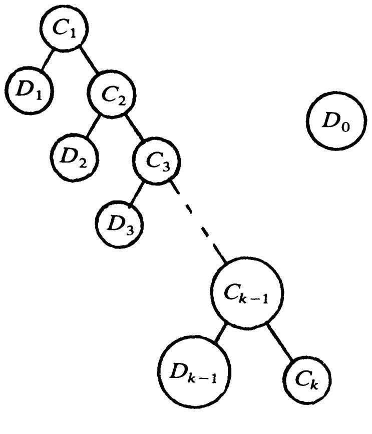


FIG. 4. *Normal form representation*

LEMMA 8. *If G has a* NF *representation* $((C_1, C_2, \cdots, C_k), (D_0, D_1, \cdots, D_{k-1}))$, *then*

(1) $\deg(D_j) = \sum_{i=1}^{j} |C_i|, j = 0, 1, 2, \cdots, k-1$,

(2) $\deg(C_j) = \deg(C_{j-1}) - |D_{j-1}|, j = 2, 3, \cdots, k$,

(3) $\sum_{i=1}^{j-1} |D_i| = \deg(C_1) - \deg(C_j), j = 1, 2, \cdots, k$.

Proof. The normal form definition implies that nodes in a nonempty D_j are adjacent only to nodes in C_i, $i = 1, 2, \cdots, j$ and (1) follows. Equalities (a) and (b) below follow from the normal form (NF) definition.

$$\text{(a)}\quad \deg(C_j) = \sum_{r=1}^{k} |C_r| - 1 + \sum_{r=j}^{k-1} |D_r|,$$

$$\text{(b)}\quad \deg(C_{j-1}) = \sum_{r=1}^{k} |C_r| - 1 + \sum_{r+j-1}^{k-1} |D_r|.$$

Subtracting (a) from (b) yields (2)

$$\text{(2)}\quad \deg(C_{j-1}) - \deg(C_j) = |D_{j-1}|.$$

Now, summing (2) for $i = 2, 3, \cdots, j$, we obtain

$$\deg(C_1) - \deg(C_j) = \sum_{i=2}^{j} |D_{i-1}| = \sum_{i=1}^{j-1} |D_i|. \qquad \square$$

LEMMA 9. *A graph G which has a normal form representation* $((C_1, C_2, \cdots, C_k), (D_0, D_1, \cdots, D_{k-1}))$ *is* PV_c-*definable.*

Proof. Assume G has a normal form representation. Label all nodes in the (possibly empty) set D_j using the following recursive scheme:

$$l(D_0) = 1,$$

$$l(D_j) = \sum_{i=0}^{j-1} |D_i| l(D_i) + 1, \qquad j = 1, 2, \cdots, k-1.$$

Now label nodes in set C_k (where C_k *may be empty*):

$$l(C_k) = \sum_{i=0}^{k-1} |D_i| l(D_i) + 1.$$

Let the threshold $t = 2l(C_k) - 1$ and label all nodes in set C_j by $l(C_j) = t - l(D_j) + 1$, $j = 1, 2, \cdots, k-1$. Thus each node label is positive and $\leqq t$. It follows from this definition of the labeling that $l(D_0) \leqq l(D_1) \leqq \cdots \leqq l(D_{k-1}) \leqq l(C_k) \leqq l(C_{k-1}) \leqq \cdots \leqq l(C_1) \leqq t$.

First we show that for any set of nodes F, if $s(F)$ is totally disconnected, then $\sum_{z \in F} l(z) \leqq t$.

By the definition of the NF, (a) the subgraph $s(\bigcup_{i=0}^{k-1} D_i)$ is totally disconnected, (b) any two distinct nodes in $\bigcup_{i=1}^{k} C_i$ are adjacent and (c) any node in C_i is adjacent to all nodes in D_j, $i \leqq j \leqq k-1$. Thus if $s(F)$ is totally disconnected,

$$F \subseteq \bigcup_{i=0}^{q} D_i \cup \{x\}, \quad \text{where } 0 \leqq q \leqq k-1 \text{ and } x \in C_j, \quad q+1 \leqq j \leqq k.$$

By the monotonicity property of the labeling scheme it suffices to consider only the case where

$$F = \bigcup_{i=0}^{q} D_i \cup \{x\}, \quad 0 \leqq q \leqq k-1 \text{ and } x \in C_{q+1}.$$

Hence,

$$\begin{aligned} \sum_{z \in F} l(z) &= \sum_{i=0}^{q} |D_i| l(D_i) + l(C_{q+1}) \\ &= l(D_{q+1}) - 1 + l(C_{q+1}) = t, \qquad 0 \leqq q \leqq k-2, \\ &= l(C_k) - 1 + l(C_k) = t, \qquad q = k-1. \end{aligned}$$

Now we show that $\sum_{z \in F} l(z) \leqq t$ implies $s(F)$ is totally disconnected. If $|F| = 0$ or 1, then the assertion is true; thus let $|F| \geqq 2$. By way of contradiction, assume $\sum_{z \in F} l(z) \leqq t$ and $s(F)$ is *not* totally disconnected. Hence there are nodes x, y in F for which x and y are adjacent. By hypothesis $l(x) + l(y) \leqq t$. There are two cases to be considered.

Case I. $x \in C_i$ and $y \in C_j$ for $1 \leqq i, j \leqq k$. Since $l(C_i) \geqq l(C_k)$ and $l(C_j) \geqq l(C_k)$, it suffices to show that $\{l(C_k) + l(C_k) > t\}$, which follows immediately from $t = 2l(C_k) - 1$.

Case II. $x \in C_i$ and $y \in D_j$ for $1 \leqq i \leqq j \leqq k-1$. Here $l(x) + l(y) = t - l(D_i) + 1 + l(D_j)$. But $l(D_i) \leqq l(D_j)$, since $i \leqq j$. Accordingly $l(x) + l(y) \geqq t + 1 > t$. □

The labeling utilized in the proof of Lemma 9 is illustrated in Fig. 5.

LEMMA 10. *If $G = (N, E)$ is* PV$_c$*-definable, then G has a* NF *representation.*

Proof. The proof is by induction on $|G|$. The lemma is true if $|G| \leqq 1$. Assume all PV$_c$-definable graphs with at most $(n-1)$ nodes have a NF representation. Let $G = (N, E)$ be a PV$_c$-definable graph having n nodes. Let

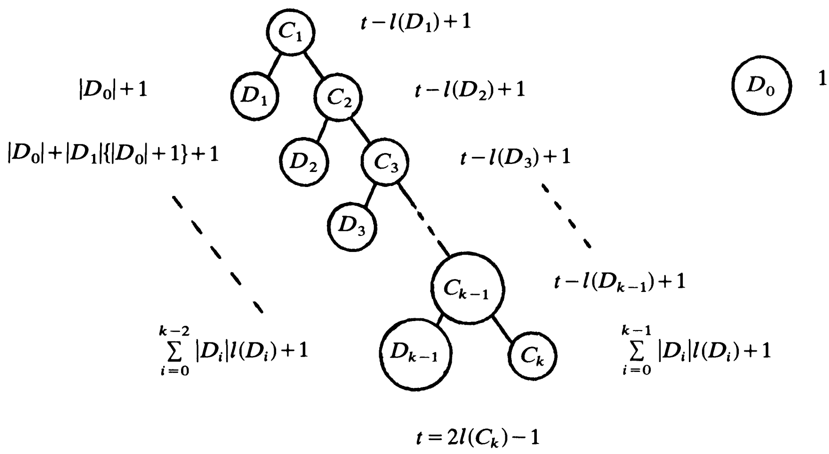


FIG. 5. *Admissible normal form labeling*

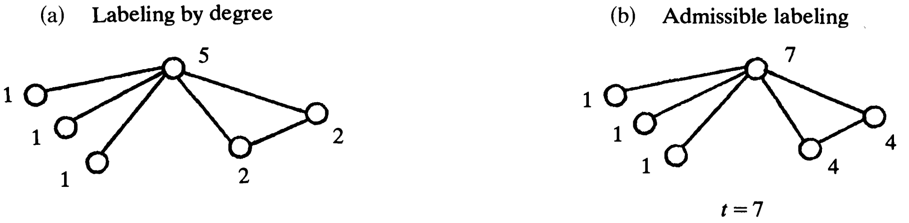


FIG. 6. *Counterexample to degree labeling*

$D_0=\{x \in G\colon \mathrm{ADJ}(x)=\emptyset\}$. If $D_0 \neq \emptyset$, then let $x \in D_0$ and $G'=S(N-\{x\})$. Lemma 4 implies G' is $\mathrm{PV_c}$-definable and the induction hypothesis implies G' has a NF representation $((C_1,\cdots,C_k), (D_0-\{x\}, D_1,\cdots,D_{k-1}))$. That $((C_1,\cdots,C_k), (D_0, D_1,\cdots,D_{k-1}))$ is a NF representation for G is immediate from the NF definition.

If $D_0=\emptyset$, then G is connected and Lemma 2 implies the set of nodes $C_1=\{x\in G\colon \deg(x)=|G|-1\}$ form a nonempty clique $s(C_1)$. If G is complete, it has a NF representation $((C_1),(\emptyset))$; otherwise, G is proper $\mathrm{PV_c}$-definable and Lemma 5 implies the set of nodes $D_1=\{x\in G: \mathrm{ADJ}(x)=C_1\}$ is nonempty. Let $x\in D_1$ and let $G'=s(N-\{x\})$. Lemma 4 implies G' is $\mathrm{PV_c}$-definable, thus by the induction hypothesis G' has a NF representation $((C_1,\cdots,C_k), (\emptyset, D_1-\{x\}, D_2,\cdots,D_{k-1}))$. That $((C_1, C_2,\cdots,C_k), (\emptyset, D_1,\cdots,D_{k-1}))$ is a NF representation for G follows immediately from the NF definition. □

Remark. Lemmas 10, 7 and 3 may appear to imply that labeling each node with its degree is an admissible labeling for a connected $\mathrm{PV_c}$-definable graph. In general this labeling technique fails as the counterexample in Fig. 6 demonstrates (i.e. in Fig. 6(a), $t \geqq 5$, but $2+2<5$).

Lemma 9 and Lemma 10 imply Theorem 1.

THEOREM 1. *A graph G is* $\mathrm{PV_c}$*-definable if and only if G has a normal form representation.*

It follows from Theorem 1 that the class of $\mathrm{PV_c}$-definable graphs is a subclass of the class of interval graphs. The latter being a class which arises in numerous applications [13].

A graph $G=(N,E)$ is said to be an *interval graph* if there is a one-to-one correspondence ψ between the vertices of G and a set of open intervals on the real line so that $(x,y)\in E$ if and only if $\psi(x)\cap\psi(y)\neq\emptyset$ [6].

COROLLARY 1. *The class of* $\mathrm{PV_c}$*-definable graphs is a proper subclass of the class of interval graphs.*

Proof. Let G be a $\mathrm{PV_c}$-definable graph with NF representation $((C_1, C_2,\cdots,C_k), (D_0, D_1,\cdots,D_{k-1}))$. Figure 7 illustrates a set of intervals corresponding to the nodes of G which satisfy the definition of an interval graph. That the inclusion is proper follows from Lemma 1 and the graph in Fig. 3(b). □

A normal form representation of a graph may not be unique. Figure 8 shows two normal form representations for the same graph. This nonuniqueness can be eliminated by defining a stronger normal form.

DEFINITION. A normal form representation $((C_1,\cdots,C_k), (D_0,\cdots,D_{k-1}))$ is said to be *minimal* provided that $|C_k|\neq 1$ and there exists

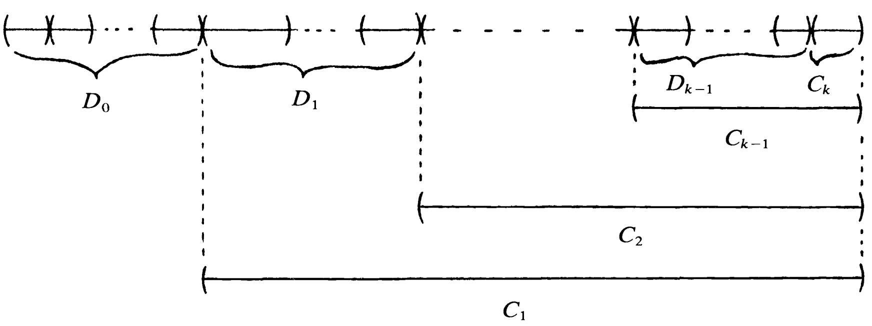


FIG. 7. *Intervals for the* NF *representation*

no other normal form representation $((C'_1, \cdots, C'_{k'}), (D'_0, \cdots, D'_{k'-1}))$ for which $k' < k$.

Lemma 11 demonstrates some important properties of the minimal NF representation.

LEMMA 11. *The following statements regarding the nonempty* $\mathrm{PV_c}$-*definable graph G are equivalent.*

(1) $((C_1, C_2, \cdots, C_k), (D_0, D_1, \cdots, D_{k-1}))$ *is a minimal* NF *representation for G*;

(2) $D_j \neq \emptyset$ *for all* $j = 1, 2, \cdots, k-1$, $|C_k| \neq 1$ *and* $C_k = \emptyset$ *implies* $|D_{k-1}| \geqq 2$;

(3) $\deg(D_0) < \deg(D_1) < \cdots < \deg(D_{k-1}) < \deg(C_k) < \cdots < \deg(C_1)$; *if* $C_k = \emptyset$, *then* $\deg(C_k)$ *is deleted from the inequality*; *if* $D_j = \emptyset$, *then* $\deg(D_j)$ *is deleted from the inequality*;

(4) $\deg(x) = \deg(y)$ *if and only if* $x, y \in C_i$, $1 \leqq i \leqq k$ *or* $x, y \in D_j$, $0 \leqq j \leqq k-1$.

Proof. First we show (1) implies (2). Assume $((C_1, \cdots, C_k), (D_0, D_1, \cdots, D_{k-1}))$ is a minimal NF representation for G. By way of contradiction,

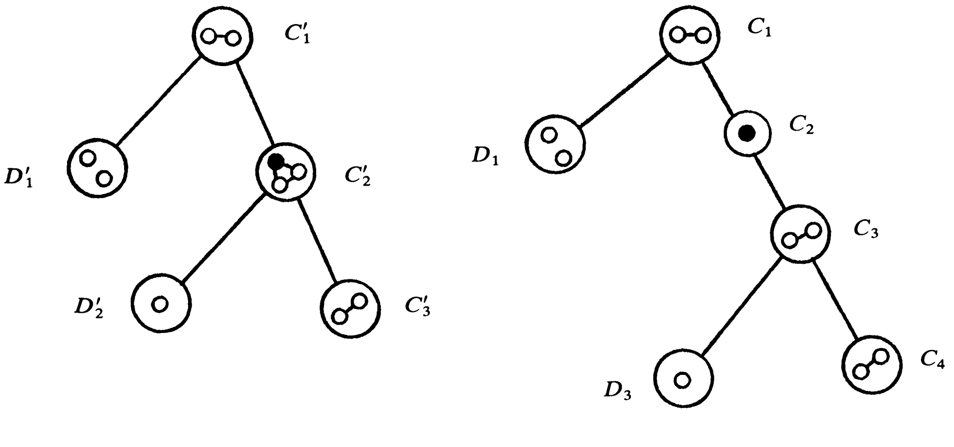


FIG. 8. *Two normal form representations for the same graph*

assume $D_i = \emptyset$ for some i, $1 \leqq i \leqq k-1$. Let $x \in C_i$ and $y \in C_{i+1}$. Let $C = \bigcup_{j=1}^k C_j$; then the NF definition implies

$$\mathrm{ADJ}\,(x) = (C - \{x\}) \cup \left(\bigcup_{j=i}^{k-1} D_j \right),$$

$$\mathrm{ADJ}\,(y) = (C - \{y\}) \cup \left(\bigcup_{j=i+1}^{k-1} D_j \right).$$

Since $D_i = \emptyset$, $\mathrm{ADJ}\,(x) = \mathrm{ADJ}\,(y)$. This is true for all $x \in C_i$, $y \in C_{i+1}$, $i < k-1$ and for $i = k-1$ in case $C_k \neq \emptyset$.Hence $((C_1, \cdots, C_{i-1}, C_i \cup C_{i+1}, \cdots, C_k)$, $(D_0, \cdots, D_{i-1}, D_{i+1}, \cdots, D_{k-1}))$ is a NF representation for G with depth $(k-1)$. If $C_k = \emptyset$ and $D_{k-1} = \emptyset$, the contradiction is immediate. When $C_k = \emptyset$ and $|D_{k-1}| = 1$, $((C_1, \cdots, C_{k-1} \cup \{x\}), (D_0, \cdots, D_{k-2}))$, where $x \in D_{k-1}$ is a NF representation with smaller depth.

That (2) implies (3) follows from the NF definition, which implies:

(a) $\deg(D_i) = \sum_{j=1}^{i} |C_j| < \sum_{j=1}^{i+1} |C_j| = \deg(D_{i+1}), \qquad i = 0, 1, \cdots, k-2,$

(b) $\deg(D_{k-1}) = \sum_{j=1}^{k-1} |C_j| < \sum_{j=1}^{k} |C_j| - 1 = \deg(C_k), \quad \text{when } |C_k| \geqq 2,$

(c) $\deg(D_{k-1}) = \sum_{j=1}^{k-1} |C_j| < \sum_{j=1}^{k-1} |C_j| - 1 + |D_{k-1}| = \deg(C_{k-1}), \quad \text{when}$

$C_k = \emptyset$ implying $|D_{k-1}| \geqq 2$,

(d) $\deg(C_{i+1}) = (|C| - 1) + \sum_{j=i+1}^{k-1} |D_j| < (|C| - 1) + \sum_{j=i}^{k-1} |D_j| = \deg(C_i),$

$i = 1, 2, \cdots, k-1$ since $|D_i| \geqq 1$.

In view of Lemma 7, it is immediate that (3) implies (4).

We show (4) implies (1) by contradiction. Assume (4) is true and $((C_1', \cdots, C_l'), (D_0', \cdots, D_{l-1}'))$ is a minimal NF representation for G where $l < k$. By a pigeon-hole argument, there exist at least two distinct nodes x, y for which $\deg(x) \neq \deg(y)$ and $x, y \in C_i'$, $1 \leqq i \leqq l$ or $x, y \in D_j'$, $0 \leqq j \leqq l-1$. The contradiction follows from Lemma 7. □

COROLLARY 2. *The minimal* NF *representation for a* PV$_c$*-definable graph is unique.*

Lemma 11 and Corollary 2 imply:

COROLLARY 3. *A* PV$_c$*-definable graph G is determined, up to isomorphism, by the $2k$-tuple of nonnegative integers* $(|C_1|, |C_2|, \cdots, |C_k|, |D_0|, |D_1|, \cdots, |D_{k-1}|)$, *where* $((C_1, C_2, \cdots, C_k), (D_0, D_1, \cdots, D_{k-1}))$ *is the minimal* NF *representation for G.*

Thus, this $2k$-tuple of nonnegative integers is a complete set of isomorphism invariants [7] for the class of PV$_c$-definable graphs.

The definition of a PV$_c$-definable graph postulates the existence of a graph labeling which satisfies a condition over all totally disconnected subgraphs.

Applying Theorem 1, we proceed to show that this global condition is equivalent to a seemingly weaker local condition, which considers only pairs of nodes.

LEMMA 12. *A graph $G=(N, E)$ is* PV_c*-definable if and only if there is a labeling of G and a positive integer t such that $(x, y)\in E$ if and only if $l(x)+l(y)>t$. (This labeling may not constitute an admissible labeling of G.)*

Proof. Necessity follows from Lemma 0. We prove sufficiency by induction on $|G|$. The lemma is true when $|G|\leqq 1$. Assume the hypothesis is true for all graphs having at most $(n-1)$ nodes. Let $G=(N, E)$ be a labeled graph with $|G|=n$. Let x and y be nodes in G with minimal and maximal label respectively. If $l(x)+l(y)\leqq t$, then $\mathrm{ADJ}\,(x)=\varnothing$. Let $G'=s(N-\{x\})$. By the induction hypothesis G' is PV_c-definable and hence has a NF representation $((C'_1, \cdots, C'_k), (D'_0, \cdots, D'_{k-1}))$. But $\mathrm{ADJ}\,(x)=\varnothing$, and thus, G has NF representation $((C'_1, \cdots, C'_k), (D'_0\cup\{x\}, D'_1, \cdots, D'_{k-1}))$ implying, by Theorem 1 that G is PV_c-definable. If $l(x)+l(y)>t$, then G is connected. Let $G'=s(N-\{y\})$ which is PV_c-definable by the induction hypothesis. There are two cases to be considered.

Case I. If G' is connected, then let $((C'_1, C'_2, \cdots, C'_k), (D'_0, \cdots, D'_{k-1}))$ be a NF representation for G'. In G, $\mathrm{ADJ}\,(y)=N-\{y\}$; therefore, $((C'_1\cup\{y\}, C'_2, \cdots, C'_k), (D'_0, \cdots, D'_{k-1}))$ is a NF representation for G.

Case II. If G' is disconnected, let $((C'_1, \cdots, C'_k), (D'_0, D'_1, \cdots, D'_{k-1}))$ be a NF representation for G'. In G, $\mathrm{ADJ}\,(y)=N-\{y\}$; thus, $((\{y\}, C'_1, \cdots, C'_k), (\varnothing, D'_0, D'_1, \cdots, D'_{k-1}))$ is a NF representation for G. □

The main results of this section are summarized in Theorem 2.

THEOREM 2. *For a graph $G=(N, E)$, the following statements are equivalent.*

(1) *G is* PV_c*-definable,*

(2) *G has a* NF *representation,*

(3) *G has a minimal* NF *representation,*

(4) *there is a labeling of G and a positive integer t such that $(x, y)\in E$ if and only if $l(x)+l(y)>t$.*

Proof. The equivalence of (1) and (2) follows from Theorem 1. That (2) is equivalent to (3) is immediate and Lemma 12 implies the equivalence of (1) and (4). □

5. Algorithm for testing PV_c-definability. Whenever a problem of the nature discussed in the preceding sections arises, one is often interested in determining whether a given labeling of a graph is admissible or if an arbitrary graph is PV_c-definable. In this section we present an efficient algorithm to determine if a given graph is PV_c-definable.

A recursive algorithm for checking PV_c-definablility and constructing the minimal NF for a graph $G=(N, E)$ is implicit in the definition of the NF. The algorithm proceeds as follows:

Let $D_0=\{x\in G:\mathrm{ADJ}\,(x)=\varnothing\}$ and $G'=s(G-D_0)$.

Initially, let $i=1$ and $G_i=G'$. If $G_i=(N_i, E_i)$ is empty, then G is PV_c-definable. Otherwise, determine $C_i=\{x\in G_i:\deg(x)=|G_i|-1\}$. If $C_i=\varnothing$, then Lemma 2 implies G is not PV_c-definable. Otherwise, determine $D_i=\{x\in G:\mathrm{ADJ}(x)=C_i\}$. If $D_i=\varnothing$, then by Lemma 5, G is not PV_c-definable. Apply the above steps recursively to the new graph $G_{i+1}=s(N_i-\{C_i\cup D_i\})$.

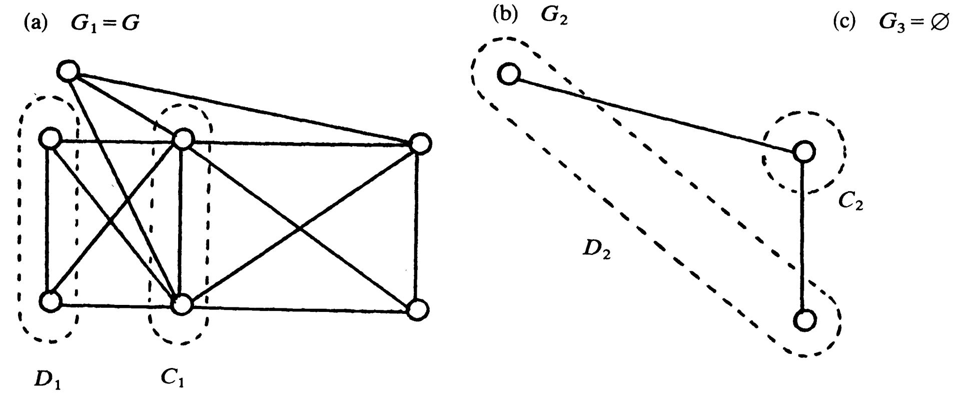


FIG. 9. *Example of the algorithm*

Figure 9 illustrates the execution of the algorithm. Observe that Lemma 11 implies this algorithm directly determines the minimal NF representation $((C_1, \cdots, C_k), (D_0, D_1, \cdots, D_{k-1}))$ for G. That is, any two nodes with the same degree are necessarily in the same set C_i or D_j.

Given the degree of each node in a graph G, Algorithm A, to be presented below, tests for $\mathrm{PV_c}$-definability of arbitrary graph G in time $O(|G|)$. Note that given only graph G (adjacency matrix or adjacency list), we require $O(|G|^2)$ time to find the degree of each node.

For any graph G, let $N[j]$ denote the cardinality of the set $\{x \in G : \deg(x) = j\}$. That is, $N[j]$ equals the number of nodes with degree j. Algorithm A is presented in an ALGOL programming language [3]. Comments are denoted by $\{\cdots\}$.

ALGORITHM A. Test graph $G = (N, E)$ for $\mathrm{PV_c}$-definability.

Input. The integers $d(1), d(2), \cdots, d(m)$, $m \geqq 1$, specifying the degree of each node in finite graph G.

Output. 1. **Accept** G, if G is $\mathrm{PV_c}$-definable.
2. **Reject** G, if G is **not** $\mathrm{PV_c}$-definable.

Method.

```
begin integer $m$;
  string(3) definable;
  input($m$);
  begin integer deg_$ci$, deg_$di$; {deg_$ci$ = deg ($C_i$), deg_$di$ = deg ($D_i$)}
    integer array $d[1::m]$, $N[0::m]$, {$N[i]$ = # nodes of degree $i$}
    for $i := 1$ until $m$ do input ($d(i)$);
    for $i := 0$ until $m$ do $N[i] := 0$;
    for $i := 1$ until $m$ do $N[d[i]] := N[d[i]] + 1$;
    deg_$ci$ := $m - N[0] - 1$; definable := "yes"; {$G' = s(N - D_0)$}
    if deg_$ci = -1$ then deg_$di$ := 0 else deg_$di$ := $N$[deg_$ci$];
    while (deg_$ci$ ≧ deg_$di$) and (definable = "yes") do
    begin {nodes of degree $|G'| - 1$ form a nonempty clique $C_i$ (Lemma 2)
        and $|D_i| \neq 0$ (Lemma 5).}
      if $N$[deg_$ci$] = 0 or $N$[deg_$di$] = 0 then definable := "no"
```

```
            else begin
              deg_ci := deg_ci − N[deg_di]; {deg ($C_{i+1}$) = deg ($C_i$) − $|D|$}
              deg_di := N[deg_ci] + deg_di; {deg ($D_{i+1}$) = $|C_{i+1}|$ + deg ($D_i$)}
            end;
          end;
      end;
      if definable = "yes" then output("accept") else output("reject");
    end.
```

Before proving the correctness of Algorithm A, we illustrate the action of the algorithm in Example 3.

Example 3. Operation of Algorithm A.

(a) Let G be the normal form shown in Fig. 10a with Table 1 below:

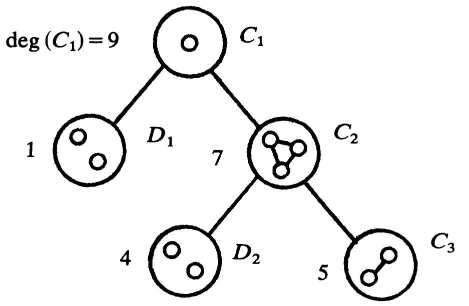


FIG. 10a. *Initially* deg_$ci = m - 1 = 9$ *and* deg_di = *number of nodes of degree* $9 = 1$. *The algorithm halts, accepting* G *on the third iteration, when* deg (C_3) = $5 \ngeq 6$ = deg (D_3).

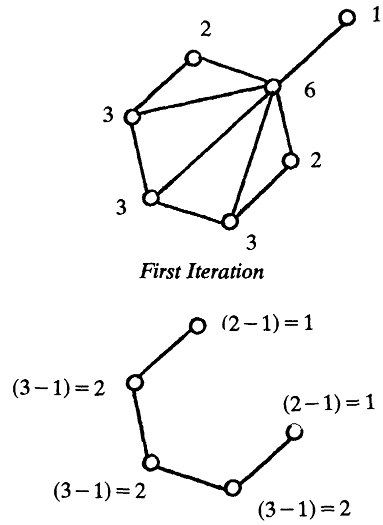
TABLE 1

	Degree	Number of nodes
deg (C_1) =	9	1
	8	0
deg (C_2) =	7	3
deg (D_3) =	6	0
deg (C_3) =	5	2
deg (D_2) =	4	2
	3	0
	2	0
deg (D_1) =	1	2

(b) Let G be the graph shown in Fig. 10b with Table 2 on the next page.

LEMMA 13. *Algorithm* A *terminates.*

Proof. By assumption graph G is finite; thus m is finite. In each iteration of the **while** loop the integer variable deg_ci is monotone decreasing and the positive variable deg_di cannot decrease. □



First Iteration

Second Iteration

FIG. 10b

TABLE 2

	Degree	Number of nodes
$\deg(C_1)=$	6	1
$\deg(C_2)=$	5	0*
	4	0
	3	3
	2	2
$\deg(D_1)=$	1	1

* The algorithm rejects G on the second iteration, since the remaining subgraph is nonempty and $|C_2|=0$.

Lemma 14 relates the computational parameters in Algorithm A with the results of the previous sections.

LEMMA 14. *Given a graph $G=(N, E)$, if* deg_*ci* ≧ deg_*di and definable* = "*yes*" *prior to the* i*th iteration of Algorithm* A, *then the following assertions are valid*:

(i) *For each iteration*, j, $1 \leqq j < i$

(a) *the set* $C_j=\{x \in G\colon \deg(x)=\text{deg\_}ci\} \neq \varnothing$,

(b) *the set* $D_j=\{x \in G\colon \deg(x)=\text{deg\_}di\} \neq \varnothing$.

(ii) *If G is* PV$_c$-*definable with minimal* NF $((C_1, \cdots, C_k), (D_0, \cdots, D_{k-1}))$ *then*

(a) deg_*ci* = deg (C_i) *and* $C_i=\{x \in G\colon \deg(x)=\text{deg\_}ci\}$,

(b) deg_*di* = deg (D_i) *and* $D_i=\{x \in G\colon \deg(x)=\text{deg\_}di\}$.

(iii) *The subgraph* $G'=s(N-\{C_1\cup\cdots\cup C_{i-1}\cup D_0\cup\cdots\cup D_{i-1}\})$ *is* not *a clique.*

Proof. The sets C_j and D_j for $j=1,2,\cdots,i-1$ are clearly nonempty since definable = "yes" prior to the ith iteration.

We prove (ii) and (iii) by induction on i, the number of iterations of the **while** loop. Let $G'=s(N-D_0)$, where $D_0=\{x\in G\colon \deg(x)=0\}$.

Basis. $i=1$. Prior to the first iteration, $\deg\_ci=m-N[0]-1=|G'|-1$ and $\deg\_di=N[|G'|-1]$. By assumption G is PV_c-definable and Lemma 4 implies G' is PV_c-definable. Hence, Lemma 2 implies (ii–a) and Lemma 5 implies (ii–b).

For (iii), if G' were a complete graph, then $\deg\_ci=|G'|-1$ and $\deg\_di=N[|G'|-1]=|G'|$ which is a contradiction, since by assumption $\deg\_ci\geqq\deg\_di$.

Induction step. Assume the assertions are true prior to the ith iteration. Now, assume $\deg\_ci\geqq\deg\_di$ and definable = "yes" prior to the $(i+1)$st iteration.

If G' is PV_c-definable with minimal NF $((C_1,\cdots,C_k),(D_0,\cdots,D_{k-1}))$, then (ii–a) applying the induction hypothesis

$$\begin{aligned}\deg\_ci &:= \deg\_ci-N[\deg\_di];\\ &= \deg(C_i)-N[\deg(D_i)]\\ &= \deg(C_i)-|D_i|\\ &= \deg(C_{i+1})\quad\text{by Lemma 8,}\end{aligned}$$

(ii–b) likewise,

$$\begin{aligned}\deg\_di &:= N[\deg\_ci]+\deg\_di;\\ &= N[\deg(C_{i+1})]+\deg(D_i)\\ &= |C_{i+1}|+\deg(D_i)\\ &= |C_{i+1}|+\sum_{j=1}^{i}|C_j|\quad\text{by Lemma 8}\\ &= \deg(D_{i+1})\qquad\text{by the NF definition.}\end{aligned}$$

(iii) Assume $G'=s(N-\{C_1\cup\cdots\cup C_i\cup D_0\cup\cdots\cup D_i\})$ is a clique. If G' is a clique, then G is PV_c-definable with the NF representation $((C_1,\cdots,C_i,G'),(D_0,\cdots,D_i))$. Prior to the $(i+1)$st iteration, (ii–a) and (ii–b) above imply $\deg\_ci=\deg(C_{i+1})$ and $\deg\_di=N[\deg(C_{i+1})]+\deg(D_i)$. By assumption $\deg\_ci\geqq\deg\_di$; hence

$$\deg(C_{i+1})\geqq N[\deg(C_{i+1})]+\deg(D_i)$$

or

$$(\xi)\qquad \{\deg(C_{i+1})-\deg(D_i)\}\geqq N[\deg(C_{i+1})].$$

However, in subgraph G', the left-hand side of (ξ) is $\deg(G')$ and the right-hand side is $N[\deg(G')]$. Thus, $\deg(G')\geqq N[\deg(G')]$ which is a contradiction to the assumption that G' is a clique (i.e. $\deg(G')=|G'|-1<N[|G'|-1]=N[\deg(G')]$). □

THEOREM 3. *Algorithm* A *accepts finite graph* $G=(N, E)$ *if and only if* G *is* PV_c-*definable.*

Proof. Assume G is PV_c-definable. Hence, G has a minimal NF representation $((C_1, \cdots, C_k), (D_0, \cdots, D_{k-1}))$. Initially Algorithm A considers the subgraph $G'=s(N-D_0)$; that is, $\deg\_ci=m-N[0]-1$. Lemma 14 implies the algorithm cannot terminate with definable = "no" prior to the kth iteration. The remaining subgraph $G'=C_k$ is a clique (vacuously if $C_k=\emptyset$); thus a contrapositive argument using Lemma 14 (iii) implies $\deg\_ci<\deg\_di$ and the algorithm accepts G.

Algorithm A rejects G only if $N[\deg\_ci]=0$ or $N[\deg\_di]=0$. Lemma 14 (iii) implies that $G'=s(N-\{C_1\cup\cdots\cup C_{i-1}\cup D_0\cup\cdots\cup D_{i-1}\})$ is not a clique and hence $G'\neq\emptyset$. We claim G' is not a PV_c-definable graph if either $N[\deg\_ci]=0$ or $N[\deg\_di]=0$, thus by Lemma 4 G is not PV_c-definable.

When $N[\deg\_ci]=0$, Lemma 14 (ii–a) and Lemma 2 suffice to prove that G' is not PV_c-definable. When $N[\deg\_di]=0$, Lemma 14 (ii–b) and Lemma 5 imply G' is not PV_c-definable. □

The running time of Algorithm A is $O(|N|)$. This is clear since $m=|N|$ and all **for** loops and the **while** loop are iterated at most m times. Also, minor modifications to the algorithm could be made so that the minimal NF representation $((C_1, \cdots, C_k), (D_0, \cdots, D_{k-1}))$ for G was determined explicitly. This is accomplished by listing, at the beginning of the **while** loop, (a) the loop iteration number $i=1, 2, \cdots, k$, (b) $\deg\_ci=\deg(C_i)$ and (c) $\deg\_di=\deg(D_i)$. Since the degree of each node is unique the sets C_i and D_{i-1}, $i=1, 2, \cdots, k$ may easily be determined from the numbers listed.

The careful reader may have observed that the set $\{\deg(x): x\in N\}$ is another complete set of isomorphism invariants for the class of PV_c-definable graphs (refer to the remark following Corollary 3).

6. Discussion and conclusion. In this paper, we have characterized—in terms of a normal form representation—the class of graphs which are definable by the class of purely parallel PV_c systems of processes.

A normal form representation can be interpreted as a hierarchial structure of asynchronous processes where each process in C_i can block (i) any process in D_j, $j\geqq i$ and (ii) any process in C_j, for all $1\leqq j\leqq k$, while the processes in $\bigcup_{i=0}^{k} D_i$ cannot block each other. One can think of this situation as a "reader–writer" problem where there is a hierarchy of writers (the C_i's) and a hierarchy of readers (the D_j's). In the case $k=2$ with D_0 and C_2 empty, the normal form represents the classical reader–writer problem [2].

A possible application of a hierarchical ordering described by the normal form representation is to synchronizing a shared data base that is accessed by several processes that can read or write into nested segments of the data base. An example is presented below:

Example 4. Consider the following process synchronization problem. There are seven sequential (not purely parallel) processes $Q_1, Q_2, \cdots, Q_7$ which require access to specified segments of a common memory M. The memory M

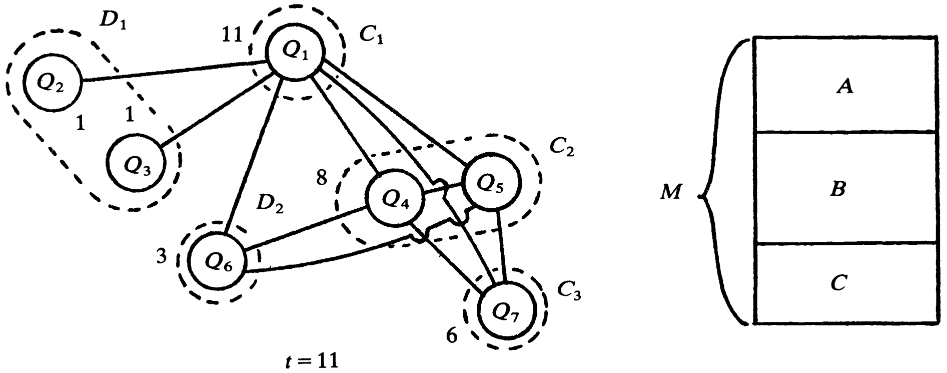


FIG. 11. *Blocking relation for Example* 4

is partitioned into three segments A, B and C. The memory access requirements for each process are:

Q_1—can only write into all of memory M

Q_2—can only read from memory segment C

Q_3—can only read from memory segment C

Q_4—can only write into memory segment $B \cup A$

Q_5—can only write into memory segment $B \cup A$

Q_6—can only read from memory segment B

Q_7—can only write into memory segment A.

Figure 11 illustrates a NF representation whose nodes correspond to the processes $Q_1, \cdots, Q_7$ and edges correspond to the block relationship on these processes.

A program, which "solves" this synchronization problem is given below. Initially $S = 11$.

```
parbegin $Q_1 // Q_2 // \cdots // Q_7$ parend where

$Q_1$: $P[S, 11]$          $Q_2$: $P[S, 1]$     $Q_3$: $P[S, 1]$     $Q_4$: $P[S, 8]$
    WRITE [$M$]            READ [$C$]        READ [$C$]        WRITE [$B \cup A$]
    $V[S, 11]$             $V[S, 1]$         $V[S, 1]$         $V[S, 8]$

$Q_5$: $P[S, 8]$           $Q_6$: $P[S, 3]$     $Q_7$: $P[S, 6]$
    WRITE [$B \cup A$]     READ [$B$]        WRITE [$A$]
    $V[S, 8]$              $V[S, 3]$         $V[S, 6]$
```

In conclusion, we expect that our results and techniques will be helpful in attacking the major problem left open in [8] and [10] concerning the relation between PV chunk and PV multiple systems of processes.

Note. Martin Golumbic has pointed out that the PV_c-definable graphs are a subclass of the permutation graphs. After this paper was submitted for publication, the PV_c-definable graphs (under the name of threshold graphs) were introduced and studied by V. Chvatal and P. L. Hammer in *Aggregation of inequalities in integer programming*, Tech. Rep. STAN-CS-75-518, Stanford University, August 1975.

Acknowledgments. We are indebted to Alan Tucker for suggestions which contributed to simplifying the proof of Theorem 1. We would also like to thank S. C. Eisenstat, R. J. Lipton and L. Snyder for a critical reading of a preliminary draft of this paper. In addition, we are very grateful to the anonymous referees for their helpful comments and suggestions.

Note added in proof. The notion of a PV_c-definable graph can be generalized, replacing the function $\sum_{z \in F} l(z)$ by a function monotone in each variable. It can be shown, however, that with this generalized definition, one still obtains exactly the class of graphs discussed in this paper.

REFERENCES

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

[2] P. J. COURTOIS, F. HEYMANS AND D. L. PARNAS, *Concurrent control with "readers" and "writers"*, Comm. ACM, 14 (1971), pp. 667–668.

[3] O.-J. DAHL, E. W. DIJKSTRA AND C. A. R. HOARE, *Structured Programming*, Academic Press, New York, 1972.

[4] E. W. DIJKSTRA, *Cooperating sequential processes*, Programming Languages, F. Genuys, ed., Academic Press, New York, 1968, pp. 43–112.

[5] ———, *Hierarchical ordering of sequential processes*, Operating Systems Techniques, C. A. R. Hoare and R. H. Perrott, eds., Academic Press, New York, 1972, pp. 72–93.

[6] D. R. FULKERSON AND O. A. GROSS, *Incidence matrices and interval graphs*, Pacific J. Math., 15 (1965), pp. 835–855.

[7] F. HARARY, *Graph Theory*, Addison-Wesley, Reading, Mass., 1969.

[8] R. J. LIPTON, *On synchronization primitive systems*, Tech. Rep. 22, Computer Science Department, Yale University, New Haven, Conn., 1973.

[9] ———, *Limitations of synchronization primitives with conditional branching and global variables*, Proc. Sixth ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1974, pp. 230–241.

[10] R. J. LIPTON, L. SNYDER AND Y. ZALCSTEIN, *A comparative study of models of parallel computation*, Conf. Record of the Fifteenth Annual IEEE Symp. on Switching and Automata Theory, IEEE, New York, 1974, pp. 145–155.

[11] R. J. LIPTON, *Reduction: A new method for proving properties of systems of processes*, Conf. Record of the Second Annual ACM Symp. on Principles of Programming Languages, Association for Compting Machinery, New York, 1975, pp. 78–86.

[12] R. J. LIPTON AND R. TUTTLE, *A synchronization anomaly*, Information Processing Lett. 3 (1975), pp. 65–66.

[13] F. S. ROBERTS, *Discrete Mathematical Models, with Applications to Social, Biological and Environmental Problems*, Prentice-Hall, Englewood Cliffs, N.J., to appear.

[14] H. VANTILBORGH AND A. VAN LAMSWEERDE, *On an extension of Dijkstra's semaphore primitives*, Information Processing Lett., 1 (1972), pp. 181–186.

SIAM J. COMPUT.
Vcl. 6, No. 1, March 1977

EVALUATING RELATIONAL EXPRESSIONS WITH DENSE AND SPARSE ARGUMENTS*

T. G. SZYMANSKI AND J. D. ULLMAN†

Abstract. We consider expressions whose arguments are relations and whose operators are chosen from among $\cup$, $\circ$, $*$, and $^{-1}$. We further assume that operands may be designated "sparse" or "dense", in a manner to be made formal subsequently. Our aim is to determine whether the evaluation of such an expression is

(a) as hard as general transitive closure,
(b) as hard as transitive closure for sparse graphs,
(c) as hard as finding connected components of an undirected graph.

Key words. computational complexity, relation, graph, sparse relation, sparse graph, composition, transitive closure

1. The model.

Let us consider expressions with operators union ($\cup$), composition ($\circ$), reflexive and transitive closure ($*$) and inverse ($^{-1}$), whose operands are relation symbols chosen from two alphabets $\mathcal{S}$ (the *sparse* relations) and $\mathcal{D}$ (the *dense* relations). Intuitively, a relation obtained by some mechanism is said to be "sparse" if the number of pairs in the relation forms a useful upper bound on the "size" of the relation for the purpose of complexity analysis. If only the square of the domain size is a useful bound then the relation is "dense." We shall make these ideas formal shortly.

Example 1. The relations ρ, μ and λ on the grammar symbols of a context free grammar are defined as follows. (See [4] for additional details.)

(i) $A\rho X$ if there is a production of the form $A \rightarrow \alpha X\beta$ for some strings of symbols α and β, where β is the empty string or derives the empty string.

(ii) $A\lambda X$ if there is a production of the form $A \rightarrow \alpha X\beta$ for some strings α and β, where α is or derives the empty string.

(iii) $X\mu Y$ if there is a production of the form $A \rightarrow \alpha X\beta Y\gamma$ for some nonterminal A and strings α, β and γ, where β either is or derives the empty string.

The "follows" relation is defined to be $(\rho^{-1})^*\mu\lambda^*$.

For complexity analysis we usually take the "size" of a grammar problem to be the length of all the productions of the grammar, written out. By rules (i) and (ii), the relations ρ and λ for a grammar of size n can have no more than n pairs, since each position of each production right side gives rise to at most one pair for each relation. Thus, ρ and λ may be regarded as sparse. In contrast, a grammar of size n can give rise to on the order of n^2 pairs in the μ relation. Consider, for example, a grammar with productions $S \rightarrow A_1A_2 \cdots A_k$ and $A_i \rightarrow \varepsilon$ for all i, $1 \leqq i \leqq k$. The size of this grammar is $O(k)$, yet $A_i\mu A_j$ for all $i<j$, so μ has $O(k^2)$ pairs. However, the domain of μ, the set of grammar symbols, is also $O(k)$, so no relation on this domain can have more than $O(k^2)$ pairs. We thus regard μ as dense for our purposes.

* Received by the editors December 1, 1975, and in revised form June 15, 1976.

† Department of Electrical Engineering and Computer Science, Princeton University, Princeton, New Jersey 08540. This work was supported by the National Science Foundation under Grant DCR74-21939.

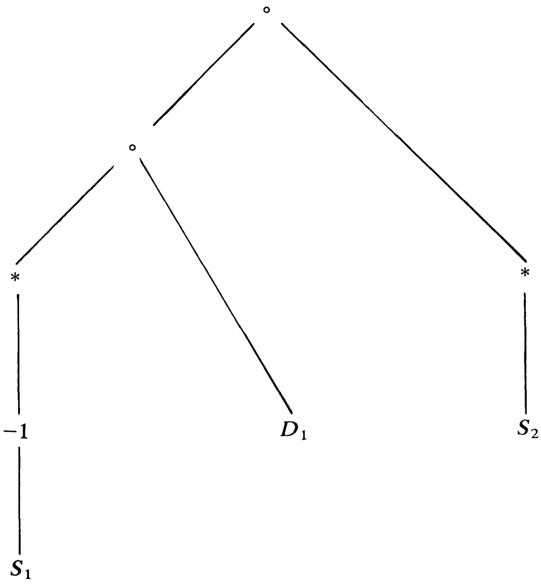


FIG. 1. *Expression tree*

If we use S_1 and S_2, symbols in $\mathcal{S}$, for ρ and λ, and we use D_1 in $\mathcal{D}$ for μ, then the "follows" relation can be represented by the expression tree shown in Fig.1. □

Our goal is to characterize the complexity of evaluating relational expressions in terms of the following parameters of a set of argument relations:

1. n, the number of elements in the domains and ranges of the arguments,

2. e, the sum of the sizes of the arguments, where the *size* of a relation is the number of pairs in that relation,

3. e_s, the sum of the sizes of those arguments designated as "sparse," that is, those arguments associated with operand symbols in $\mathcal{S}$.

We shall henceforth assume that for any set of arguments, both e and e_s are at least as great as n.

Our results involve the classification of expressions into three categories, (i) those expressions which may be evaluated in $O(e)$ time and hence are very easy to compute, (ii) those expressions which may be evaluated in $O(ne_s)$ and whose complexity is therefore independent of the number of pairs in their dense arguments, and (iii) those expressions which require as much time for evaluation as the composition of arbitrary relations. The best known algorithm for this latter problem requires $O(\min(n^{2.81}, ne))$ time and is equivalent in complexity to the problem of computing the transitive closure of an arbitrary relation (see [1], [2], [3], [6]).

The motivation for separating the operands of an expression into two disjoint classes stems from the fact that in many applications of interest (see [4] for some specific examples) certain arguments are known always to have at most $O(n)$

pairs. If these arguments correspond to operands which are designated "sparse" and if the expression in question has an $O(ne_s)$ evaluation algorithm, then $O(n^2)$ time suffices for evaluating the expression on the relations of interest.

We shall now give a precise meaning to the statement that an expression is "hard."

DEFINITION 1. The evaluation of an expression $\mathscr{E}$ is said to be *as hard as composition* if the existence of an $f(n, e)$ time bounded algorithm for $\mathscr{E}$ implies that for some constant c there exists an $O(f(cn, ce))$ time bounded algorithm for composing arbitrary relations.

The reader should note that the equivalence of composition and transitive closure, as far as computation time is concerned, implies that no expression over the operators $(\circ, \cup, {}^*, {}^{-1})$ is strictly harder than composition.

2. A class of easy expressions. In this section we shall show that any expression known to yield an equivalence relation on a particular set of arguments may be evaluated in $O(e)$ time; the problem essentially reduces to finding connected components on an undirected graph. In fact we can prove a slightly stronger result.

DEFINITION 2. A relation R is said to have *property* P if aRd, cRd and cRb imply aRb. That is, the images of any two domain elements are either disjoint or identical.

THEOREM 1. *Let $\mathscr{E}$ be any expression over operators $\circ$, $\cup$, * and ${}^{-1}$; let R be the relation produced by applying $\mathscr{E}$ to some list of arguments with parameters n and e. Suppose that R is known to have property* P. *Then R may be computed in $O(e)$ time, in the sense that we can in this amount of time build a structure from which the answer to "aRb?" can be obtained in constant time.*

Proof. By [4] we can construct for R a directed *representation graph*, $G = (V, E)$ with the following properties:

(i) There are disjoint subsets $\mathscr{I}$ and $\mathscr{O}$ of V, such that for each a in the domain of R there is a node $\mathscr{I}(a)$ in $\mathscr{I}$ "representing" a. Similarly, for each b in the range of R there is a representative $\mathscr{O}(b)$ in $\mathscr{O}$.

(ii) The number of nodes of G is $O(n)$ and the number of edges is $O(e)$. G can be constructed from $\mathscr{E}$ and its arguments in $O(e)$ time.

(iii) No edge enters $\mathscr{I}$ or leaves $\mathscr{O}$.

(iv) There is a path from $\mathscr{I}(a)$ to $\mathscr{O}(b)$ if and only if aRb.

The following steps suffice to compute R in $O(e)$ time.

1. Construct $G = (V, E)$, the representation graph for R.

2. Eliminate all nodes not reachable from a member of $\mathscr{I}$ or which do not reach a member of $\mathscr{O}$. This can be done by a straightforward application of depth-first search (see [1], [5], e.g.). Call the result G'.

3. From G' construct G'', an undirected graph having an edge (v, w) if and only if G' has an edge $v \to w$ or $w \to v$.

4. Find connected components of G''. Then aRb if and only if $\mathscr{I}(a)$ and $\mathscr{O}(b)$ are in the same connected component of G''.

To show that the above works we must prove that aRb if and only if there is a path from $\mathscr{I}(a)$ to $\mathscr{O}(b)$ in G''. The hard part is to show that if there is a sequence of nodes $v_1, v_2, \cdots, v_k$ of G', such that $v_1 = \mathscr{I}(a)$, $v_k = \mathscr{O}(b)$, and for $1 \leqq i < k$, either

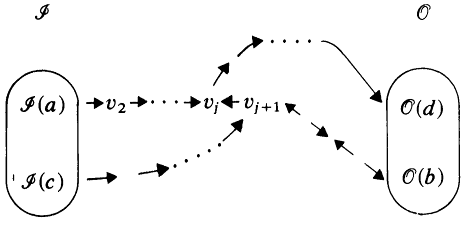


FIG. 2. *Paths in G'*

$v_i \rightarrow v_{i+1}$ or $v_{i+1} \rightarrow v_i$, then there is another path $\mathcal{I}(a) \rightarrow w_2 \rightarrow w_3 \rightarrow \cdots \rightarrow w_r \rightarrow \mathcal{O}(b)$ in G', so aRb.

We proceed by induction on the number of values of i such that $v_i \rightarrow v_{i+1}$ in G' is false. The basis, zero, is trivial, as $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k$ is then a path in G. For the induction, let j be the smallest value for i for which $v_i \rightarrow v_{i+1}$ is false. Then $v_{j+1} \rightarrow v_j$.

By step 2 of the algorithm, we can find c and d such that there are paths from $\mathcal{I}(c)$ to v_{j+1} and from v_j to $\mathcal{O}(d)$ in G', as shown in Fig. 2. Then there are in G', (and hence in G) paths from $\mathcal{I}(a)$ to $\mathcal{O}(d)$ and from $\mathcal{I}(c)$ to $\mathcal{O}(d)$. Moreover, by the inductive hypothesis, a path from $\mathcal{I}(c)$ to $\mathcal{O}(b)$ exists in G. Thus aRd, cRd and cRb, so aRb follows by property P. □

COROLLARY. *Let $\mathcal{E}$ be an expression over operators $\circ$, $\cup$, * and $^{-1}$, and let R be the result of applying $\mathcal{E}$ to a set of arguments with parameter e. Suppose that R is known to be an equivalence relation. Then R may be computed in $O(e)$ time in the sense of Theorem* 1.

Proof. Every equivalence relation has property P. □

Example 2. It is not necessary that $\mathcal{E}$ satisfy the condition of Theorem 1 or its corollary for every value of its arguments. The algorithm of Theorem 1 will work on any particular set of arguments that happen to make $\mathcal{E}$ have property P. Thus, if $G=(V,E)$ is any undirected graph, we can determine in $O(|E|+|V|)$[1] time a relation such that vRw if and only if there is a path from v to w of even length. That is, $R=(E\circ E)^*$. Note that the expression $(E\circ E)^*$ is only an equivalence relation if E is symmetric, as it will be for an undirected graph. We could also write the formula $R=((E\cup E^{-1})\circ(E\cup E^{-1}))^*$ and be assured that R was an equivalence relation independent of E. □

Theorem 1 is slightly more general than its corollary in that certain expressions can be shown to yield relations which are not necessarily equivalence relations but which always have property P. The simplest such expression known to us is

$$(R\circ R^{-2}\circ R\circ R^{-1}\circ R^2\circ R^{-1}\circ R\circ R^{-2}\circ R)^*.$$

3. A tool for proving expressions hard. In this section we shall develop a theorem which may be used to prove that the evaluation of certain relational expressions requires asymptotically as much time as the composition of arbitrary relations. Although the theorem can be stated and proved for expressions with

[1] $|A|$ is the number of members of set A.

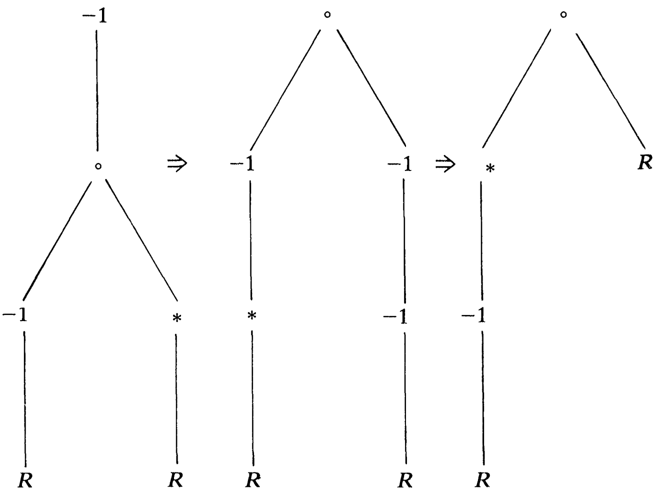


FIG. 3. *Transformation to normal form*

multiple operands, we shall develop the simpler case of single operand expressions. This less general result will still be sufficiently powerful to handle the applications in § 4.

DEFINITION 3. We say that an expression $\mathscr{E}$ is in *normal form* if in the tree corresponding to $\mathscr{E}$, all $^{-1}$ operators are parents of leaves.

Any expression over the operators with which we are concerned may be uniquely put into normal form by repeated application of the following identities:

$$(\mathscr{E}_1 \cup \mathscr{E}_2)^{-1} = \mathscr{E}_1^{-1} \cup \mathscr{E}_2^{-1},$$
$$(\mathscr{E}_1 \circ \mathscr{E}_2)^{-1} = \mathscr{E}_2^{-1} \circ \mathscr{E}_1^{-1},$$
$$(\mathscr{E}_1^*)^{-1} = (\mathscr{E}_1^{-1})^*,$$
$$(\mathscr{E}_1^{-1})^{-1} = \mathscr{E}_1.$$

Example 3. The expression $(R^{-1} \circ R^*)^{-1}$ may be transformed to normal form as shown in Fig. 3. □

It should be clear that the transformation of an arbitrary expression to normal form can be performed efficiently and can at most double the number of operators in the expression. Moreover, the evaluation of the new expression is as hard as composition if and only if the evaluation of the original expression was as hard as composition.

DEFINITION 4. Let $\mathscr{E}$ be an expression with one operand, in normal form. The set of *path schemes* of $\mathscr{E}$, denoted $P(\mathscr{E})$, is the set of strings over the two symbol alphabet $\{R, R^{-1}\}$ defined by interpreting $\mathscr{E}$ as a regular expression over this alphabet.

A path scheme can naturally be interpreted as a sequence of "go forward" and "go backward" instructions.

If we let S be a relation, we may take a path scheme $p_1p_2 \cdots p_m$ for $\mathscr{E}$, and starting at some s in the domain of S, walk the edges of S. At the ith step of the walk we follow an edge forward if p_i is R, i.e., a "go forward" instruction, and we follow an edge backward if p_i is R^{-1}, i.e., a "go backward" instruction. After following "instructions" $p_1p_2 \cdots p_m$ we arrive at some node t of S's graph; there could be many such t's possible. We can show that the t's we can reach by this process are exactly the objects for which $s\mathscr{E}(S)t$, where $\mathscr{E}(S)$ is expression $\mathscr{E}$ with actual parameter S substituted for formal parameter R. We make these statements precise with the next definition and lemma.

DEFINITION 5. Let S be any relation. A sequence of S's domain[2] elements, $v_1, v_2, \cdots, v_r$ is said to be a *path induced by* $\mathscr{E}$ if there exists a path scheme Π in $P(\mathscr{E})$ such that

(i) $\Pi = p_1p_2 \cdots p_{r-1}$, where each p_i is either R or R^{-1},

(ii) v_iSv_{i+1} whenever $p_i = R$,

(iii) $v_iS^{-1}v_{i+1}$ whenever $p_i = R^{-1}$.

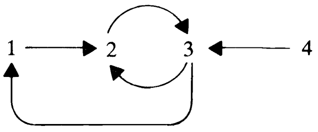
There is a natural correspondence between paths induced in S by $\mathscr{E}$ and the relation $\mathscr{E}$ (S). This correspondence is described in the following lemma.

LEMMA 1. *Let $\mathscr{E}$ be a normal form expression with a single operand. Let S be a relation and let s and t be elements of the domain of S. Then $s\mathscr{E}(S)t$ if and only if $\mathscr{E}$ induces a path from s to t in S.*

Proof. The proof is a simple induction on the number of occurrences of the operators $\circ$, $*$ and $\cup$ in the normal form expression $\mathscr{E}$. □

Example 4. Let us consider the single operand expression $\mathscr{E} = [R \circ (R \cup R^{-1})]^*$. The set of path schemes $P(\mathscr{E})$ of $\mathscr{E}$ is precisely the set of even length strings whose odd positions are R and whose even positions are either R or R^{-1}.

Now consider the relation S depicted by the following directed graph.



The path 1 2 1 is induced in S by the path scheme RR^{-1} and so we conclude by Lemma 1 that $1\mathscr{E}(S)1$. The path 1 2 3 is induced by both RR and RR^{-1} and so $1\mathscr{E}(S)3$. Finally, $1\mathscr{E}(S)4$ because the path 1 2 3 1 2 3 4 is induced by the path scheme $RR^{-1}RRRR^{-1}$. □

The motivation for considering path schemes is that we would like to show composition of relations to be embedded in particular expressions $\mathscr{E}$. The path scheme provides a useful formalism for discussing such embeddings.

The next theorem is the main result of this section. It says that if every path induced by an expression $\mathscr{E}$ in a particular relation S is of a certain form, then any algorithm for evaluating $\mathscr{E}$ on an arbitrary argument R requires as much time as the composition of arbitrary relations of the same size as R.

THEOREM 2. *Let $\mathscr{E}$ be an expression with a single operand. Suppose there exists a relation S with domain elements s, t, x, $\bar{x}$, y and $\bar{y}$ such that*

[2] Throughout the rest of this paper, we shall use the term "domain" to denote that set which is the union of the domain and range of the relation in question.

1. *$\mathscr{E}$ induces at least one path in S from s to t;*

2. *for every induced path $v_1 \cdots v_r$ with $v_1 = s$ and $v_r = t$ there exists a* unique *pair of integers p and q such that*

(a) $1 \leqq p < q < r$,

(b) *one of v_p and v_{p+1} is x, the other is $\bar{x}$,*

(c) *one of v_q and v_{q+1} is y, the other is $\bar{y}$.*

Then the evaluation of $\mathscr{E}$ is as hard as composition.

Proof. Let M be the domain of S, and let R_1 and R_2 be arbitrary relations with domain N and parameters n and e. Construct relation T with domain $M \times N$ such that for all i, j, k, l in N,

$$\begin{aligned}
\langle x, i\rangle T\langle \bar{x}, j\rangle &\quad \text{iff} \quad iR_1j \text{ and } xS\bar{x},\\
\langle \bar{x}, j\rangle T\langle x, i\rangle &\quad \text{iff} \quad iR_1j \text{ and } xS^{-1}\bar{x},\\
\langle y, j\rangle T\langle \bar{y}, k\rangle &\quad \text{iff} \quad jR_2k \text{ and } yS\bar{y},\\
\langle \bar{y}, k\rangle T\langle y, j\rangle &\quad \text{iff} \quad jR_2k \text{ and } yS^{-1}\bar{y},\\
\langle z, l\rangle T\langle \bar{z}, l\rangle &\quad \text{iff} \quad zS\bar{z} \text{ and the pair } \langle z, \bar{z}\rangle\\
&\qquad\qquad \text{is none of } \langle x, \bar{x}\rangle, \langle \bar{x}, x\rangle, \langle y, \bar{y}\rangle, \text{ or } \langle \bar{y}, y\rangle.
\end{aligned}$$

We claim that $\langle s, i\rangle \mathscr{E}(T)\langle t, k\rangle$ if and only if $iR_1 \circ R_2k$.

Intuitively, T allows us to walk around the graph of S at will, using the first component of its elements. However, the fact that we are concerned with $\mathscr{E}(T)$ will force us to follow the instructions of some path scheme of $\mathscr{E}$ when determining whether to go forward or backward along edges of S. The second component of the elements cannot change unless we traverse the edge $(x, \bar{x})$ or the edge $(y, \bar{y})$ in one direction or the other. When we traverse the former, we apply R_1 to the second component, and when we traverse the latter we apply R_2. The fact that every path induced by $\mathscr{E}$ on S involves exactly one traversal of each of the edges $(x, \bar{x})$ and $(y, \bar{y})$ in that order assures us that we apply $R_1 \circ R_2$ to the second component.

Suppose first that iR_1jR_2k for some j. Let $\Pi = v_1v_2 \cdots v_r$ be any path induced in T by $\mathscr{E}$ with $v_1 = s$ and $v_r = t$. Such a path exists by condition 1 of the hypothesis. Let p and q be the unique integers of condition 2 of the hypothesis and consider the sequence

$$\begin{aligned}
&\Pi' = \langle v_1, l_1\rangle \cdots \langle v_r, l_r\rangle\\
&\text{with} \quad l_1 = \cdots = l_p = i,\\
&\qquad\quad l_{p+1} = \cdots = l_q = j,\\
&\qquad\quad l_{q+1} = \cdots = l_r = k.
\end{aligned}$$

It is easy to verify that Π' is induced in T by the same path scheme that induced Π in S. By Lemma 1, we conclude that

$$\langle s, i\rangle \mathscr{E}(S)\langle t, k\rangle.$$

On the other hand, suppose that

$$\langle s, i\rangle \mathscr{E}(S)\langle t, k\rangle.$$

By Lemma 1, there exists some path

$$\Pi = \langle v_1, l_1\rangle \cdots \langle v_r, l_r\rangle$$

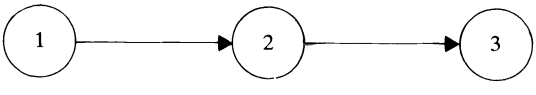
induced in T from $\langle s, i\rangle$ to $\langle t, k\rangle$. By definition of T, $\Pi' = v_1 \cdots v_r$ must be a path induced in S by $\mathscr{E}$. Let p and q be as in condition 2. The definition of T then implies that for some k,

$$l_1 = \cdots = l_p = i,$$
$$l_{p+1} = \cdots = l_q = j,$$
$$l_{q+1} = \cdots = l_r = k.$$

But then we have iR_1jR_2k, so $iR_1 \circ R_2k$ as was to be shown.

To complete the proof of the theorem, observe that the domain of T has $|M| \cdot n$ elements and T itself has at most $2e + |M|n$ pairs. Therefore T has at most cn domain elements and ce pairs for some constant c depending only on S. Thus any algorithm for evaluating $\mathscr{E}$ on arbitrary arguments in $f(n, e)$ time can be used to compose arbitrary relations in $O(f(cn, ce))$ time. □

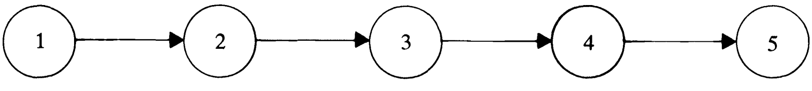
Example 5. Let us show the known result that transitive closure is as hard as composition. Consider the single operand expression $\mathscr{E} = R^*$ and the relation S depicted by the graph below.



$\mathscr{E}$ induces precisely one path in S from 1 to 3, namely, the path 1 2 3. We may therefore apply Theorem 2 with

$$s = x = 1, \quad \bar{x} = y = 2, \quad \bar{y} = t = 3. \quad □$$

Example 6. Consider the more complicated expression $\mathscr{E} = [R \circ (R \cup R^{-1})]^*$ of Example 4 and the graph S depicted below.



The set of paths induced in S by $\mathscr{E}$ may be expressed as the regular set $(12)^*123(43)^*45$. Theorem 2 may therefore be applied with $s = 1$, $x = 2$, $\bar{x} = 3$, $y = 4$, $\bar{y} = t = 5$ to show that $\mathscr{E}$ is hard. □

4. Expressions without $^{-1}$ or without *. Since Theorem 1 says any equivalance relation is easy, we must look elsewhere for hard expressions. One interesting special case is expressions without inverse, which may not in general be symmetric, and another is expressions without *, which need not be transitive. In each of these cases we can divide expressions into two classes, those which are equivalent in complexity to general composition, and those which have $O(ne_s)$ algorithms, that is, are no harder than sparse transitive closure. The division can be expressed as follows.

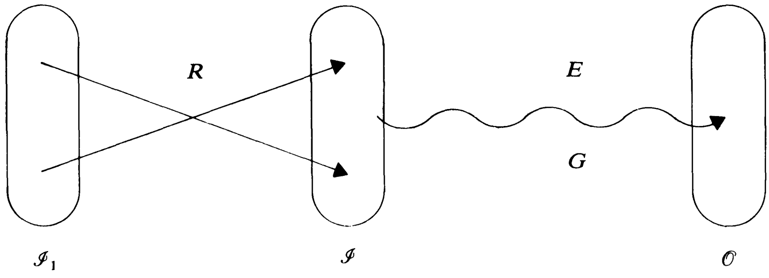


FIG. 4. *Construction for* $R \circ E$

DEFINITION 6. An expression $\mathscr{E}$ with expression tree T is *simple* if

(i) no dense argument has an ancestor in T labeled *,

(ii) no two dense arguments have a lowest common ancestor labeled $\circ$.

Example 7. The expression tree of Fig. 1 is simple. Since there is only one dense argument, rule (ii) of Definition 6 must be satisfied. Inspection shows rule (i) satisfied, since both ancestors of D_1 are labeled $\circ$. If the argument S_1 were replaced by a dense argument, say D_1 or D_2, then both rules (i) and (ii) would be violated. □

LEMMA 2. *Let E be the relation resulting from the application of some expression $\mathscr{E}$ to a set of arguments with parameters n and e. Let R be a relation whose domain is of size $O(n)$. Then the relations $E \circ R$ and $R \circ E$ can be found in $O(ne)$ time, that is, in time independent of the number of pairs in R.*

Proof. We consider $R \circ E$ only. The proof for $E \circ R$ is a mirror image of what follows. Construct the representation graph G for E from $\mathscr{E}$ and its arguments. Create a new set $\mathscr{I}_1$ of nodes corresponding to the domain of R. Put an edge from $\mathscr{I}_1(a)$ to $\mathscr{I}(b)$ if and only if aRb, as shown in Fig. 4.

Construct a table $T(x, y)$ where $x \in \mathscr{I}_1$ and y is a node of G, such that $T(x, y) = \textbf{true}$ if and only if there is a path from x to y. Initialize $T(x, y) = \textbf{true}$ if and only if $y \in \mathscr{I}$ and $[\mathscr{I}_1^{-1}(x)]R[\mathscr{I}^{-1}(y)]$. Then use a standard stack oriented table filling algorithm to complete T in $O(ne)$ time. The formal procedure is given in the following algorithm.

TABLE FILLING ALGORITHM.

```
procedure INSERT($x$, $y$); /* set $T(x, y)$ to true and stack $(x, y)$ if $T(x, y)$ was not
previously true. */
if $T(x, y)$ = false then begin
        $T(x, y)$ = true;
        push $(x, y)$ onto STACK
    end;
/* main program follows */
    begin
    (1) for $x$ in $\mathscr{I}_1$ and node $y$ of $G$ do $T(x, y)$ = false;
    (2) for all $a$ and $b$ such that $aRb$ do INSERT ($\mathscr{I}_1(a)$, $\mathscr{I}(b)$);
        /* above steps initialize $T$ */
        while STACK not empty do
        begin
    (3)     pop top pair $(x, y)$ from STACK;
```

```
(4)        for each edge $y \rightarrow z$ in $G$ do INSERT($x$, $z$)
       end
   end
```

We may easily check that the above algorithm never puts (x, y) on STACK twice. It therefore terminates. An easy inductive argument shows that $T(x, y)$ is set to **true** if and only if there is some z in $\mathscr{I}$ for which $\mathscr{I}_1^{-1}(x)R\mathscr{I}^{-1}(z)$ and there is a path in G from z to y. Thus, if $y \in \mathscr{O}$, we have $T(x, y) =$ **true** if and only if there is a z in $\mathscr{I}$ for which $\mathscr{I}^{-1}(z)E\mathscr{O}^{-1}(y)$. It is thus straightforward to see the correctness of the algorithm.

For the timing, note that steps (1) and (2) of the algorithm are each $O(n^2)$, since INSERT takes constant time. Step (3) is $O(n^2)$ since pairs are never placed on STACK twice. No y can be the second component of a pair selected in step (3) more than $O(n)$ times, for the same reason. Thus the time spent in step (4) for fixed y is no more than $O(nd_y)$ where d_y is the out-degree of y in G. The total time spent in step (4) over all possible y's is thus $O(ne)$. □

THEOREM 3. *Any simple expression $\mathscr{E}$ may be evaluated in $O(ne_s)$ steps.*

Proof. The proof is a construction. We evaluate, in a bottom up fashion, all nodes v which have one or more dense descendants. If there are no dense arguments, we first construct the representation graph for $\mathscr{E}$ applied to the given set of arguments as explained in Theorem 1. We then follow all paths from the input set of the representation graph to the output set, thereby evaluating $\mathscr{E}$. This process takes $O(ne_s)$ time.

Assume that there are dense arguments. First we observe that by rule (i) in Definition 6 every node with a dense descendant is labeled $\cup$, $\circ$, or $^{-1}$.

Case 1. The label of v is $\cup$. Let the subexpression represented by v be $\mathscr{E}_1 \cup \mathscr{E}_s$. Then $\mathscr{E}_1$ and $\mathscr{E}_2$ have been evaluated or are single arguments. The union of their values may easily be taken in $O(n^2)$ steps.

Case 2. The label of v is $\circ$. It is not possible that both children of v have dense descendants by rule (ii) of Definition 6., Thus, assume without loss of generality that the subexpression represented by v is of the form $\mathscr{E}_1 \circ \mathscr{E}_2$, where $\mathscr{E}_1$ has a dense argument and $\mathscr{E}_2$ does not. Then either $\mathscr{E}_1$ is a single dense argument or an expression which has already been evaluated. In either event, let R be the value of $\mathscr{E}_1$. Then $R \circ \mathscr{E}_2$ can be evaluated in $O(ne_s)$ steps by Lemma 2.

Case 3. The label of v is $^{-1}$. It is easy to invert the relation represented by the subtree with root v in $O(n^2)$ steps.

When we evaluate the root, we have evaluated the entire expression. Since evaluation consists of a fixed number of steps, each of which is $O(ne)$, we have our result. □

COROLLARY. *If all sparse arguments of a simple expression $\mathscr{E}$ have size $O(n)$, then $O(n^2)$ steps suffice to evaluate $\mathscr{E}$, independent of the number of pairs in the dense arguments.*

One can argue that without parametrizing the problem further, the above corollary is optimal for simple expressions, since $O(n^2)$ time could be needed to print the answer.

Example 8. In [4], evaluation of the expression $(\rho^{-1})^*\mu\lambda^*$ was done efficiently for dense μ and sparse ρ and λ, by expressing μ in terms of operations applied to sparse relations. However, Theorem 3 implies that the evaluation could have been done directly with dense μ, since the expression is simple when ρ and λ are sparse. However, certain other expressions in [4] yield nonsimple expressions if the substitution for μ is not made. □

Now, let us consider nonsimple expressions. Clearly, no expression over the four operators we have been considering is harder than general transitive closure. We establish lower bounds for the special cases mentioned at the beginning of the section.

Our first task is to reduce the evaluation of arbitrary expressions over operators $\circ$, $\cup$, $*$ and $^{-1}$ to expressions with one dense and no sparse operators.

DEFINITION 7. Let $\mathscr{E}$ be an expression with at least one dense argument. The *pruning* of $\mathscr{E}$ is constructed as follows.

1. Substitute D for every dense argument of $\mathscr{E}$.

2. For each node in the expression tree for $\mathscr{E}$ having only sparse descendants, but whose parent has a dense descendant, substitute a leaf labeled I, the identity relation.

3. Propagate I's up the tree by using the following relations, always replacing left sides by right.

$$\begin{array}{rll}
\text{(i)} & I\circ E & =E, \\
\text{(ii)} & (I\cup E)^* & =E^*, \\
\text{(iii)} & (I\cup E)^{-1} & =I\cup E^{-1}, \\
\text{(iv)} & (I\cup E_1)\circ E_2 & =E_2\cup E_1\circ E_2, \\
\text{(v)} & E_1\circ(I\cup E_2) & =E_1\cup E_1\circ E_2, \\
\text{(vi)} & (I\cup E_1)\circ(I\cup E_2) & =I\cup(E_1\cup E_2\cup E_1\circ E_2), \\
\text{(vii)} & (I\cup E_1)\cup E_2 & =I\cup(E_1\cup E_2), \\
\text{(viii)} & (I\cup E_1)\cup(I\cup E_2) & =I\cup(E_1\cup E_2).
\end{array}$$

It is elementary to show that the result is an expression $\mathscr{E}$, which either has no instances of I or which is of form $I\cup\mathscr{E}''$, where $\mathscr{E}''$ has no instances of I.

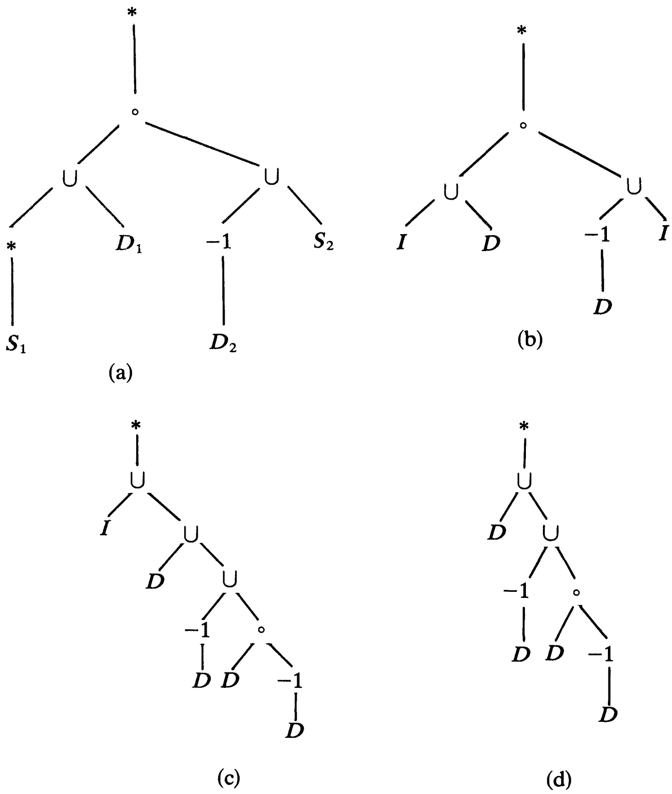
Example 9. Fig. 5(a) shows an expression $\mathscr{E}$. In Fig. 5(b) we have applied steps 1 and 2 of the pruning process (Definition 7). In Fig. 5(c) we have applied rule 3(vi), with $E_1=D$ and $E_2=D^{-1}$. In Fig. 5(d) we have applied rule 3(ii) to obtain the pruned expression. □

LEMMA 3. *Let $\mathscr{E}$ be an expression with at least one dense operand, and let $\mathscr{E}'$ be its pruning. Let $\mathscr{E}''$ be $\mathscr{E}'$ if $\mathscr{E}'$ involves no I, and let $\mathscr{E}''$ be such that $\mathscr{E}'=I\cup\mathscr{E}''$ otherwise. Then* (a) *$\mathscr{E}$ is simple if and only if $\mathscr{E}''$ is simple.* (b) *If $\mathscr{E}''$ is as hard as composition, so is $\mathscr{E}$.*

Proof. Part (a) follows from the fact that each step of the pruning process is easily seen to preserve simplicity and nonsimplicity.

For (b), suppose we are given an $f(n, e)$ time bounded algorithm to evaluate $\mathscr{E}$. We can evaluate $\mathscr{E}''$ with actual parameter D as follows. First, for each d in the domain of D, create another element d'. Then for every pair d_1Dd_2, add the pairs d_1Dd_2', $d_1'Dd_2$ and $d_1'Dd_2'$. Thus, d' is just a copy of d which behaves the same way as d. The presence of d' will help us determine whether $d\mathscr{E}''(D)d$ however. Call the new relation D'.

If we consider $\mathscr{E}$ with D' substituted for all dense arguments and I for all sparse ones, we get $\mathscr{E}'(D')$, the pruning of $\mathscr{E}$. In proof, note that any of the



(a) (b) (c) (d)

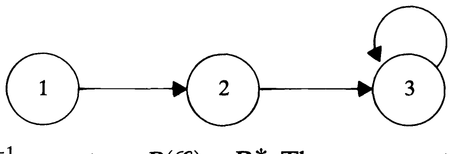
FIG. 5. *Pruning*

operators ∘, ∪, * and $^{-1}$ applied to I's only produces I as a result. Thus the nodes of $\mathscr{C}$ which are replaced by I in step 2 of the pruning algorithm (Definition 7) are sure to get I as a value anyway.

If we apply our supposed $f(n, e)$ time bounded algorithm to $\mathscr{C}'(D')$, we get either $\mathscr{C}''(D')$ or $I \cup \mathscr{C}''(D')$. In either case, $\mathscr{C}''(D)$ can be recovered from $\mathscr{C}'(D')$ by noting that $d_1\mathscr{C}'(D')d_2'$ if and only if $d_1\mathscr{C}''(D)d_2$. We thus have an $f(cn, ce)$ time algorithm to compute $\mathscr{C}''(D)$, which we supposed was as hard as composition. □

THEOREM 4. *The evaluation of any nonsimple expression over the operators* (∘, ∪, *) *is as hard as composition.*

Proof. By Lemma 3, it suffices to consider expressions with a single operand. Let $\mathscr{C}$ be such an expression and consider the relation S depicted below.

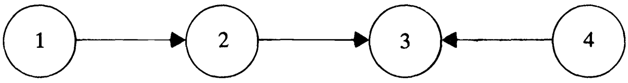


Since $\mathscr{C}$ contains no $^{-1}$ operators, $P(\mathscr{C}) \subseteq R^*$. Thus every path induced in S by $\mathscr{C}$ is of the form 1233^*. Moreover, since $\mathscr{C}$ is nonsimple, $P(\mathscr{C})$ must include at least one element of length two or more. Thus $\mathscr{E}$ induces at least one path from 1 to 3 in T. We may therefore apply Theorem 2 with $s=x=1$, $\bar{x}=y=2$ and $\bar{y}=t=3$ to conclude that $\mathscr{E}$ is hard. □

THEOREM 5. *The evaluation of any non-simple expression over the operators,* $(\circ, \cup, ^{-1})$ *is as hard as composition.*

Proof. Once again, by virtue of Lemma 3, we need only consider expressions with a single operand. Without the * operator, the set of path schemes $P(\mathscr{E})$ must be finite. Let $\Pi = p_1 \cdots p_r$ be a longest element of $P(\mathscr{E})$. Since $\mathscr{E}$ is nonsimple, r must be at least two. Consider the relation S whose domain is $\{1, 2, \cdots, r+1\}$, which is defined by $(i)S(i+1)$ if and only if p_i is R and $(i+1)S(i)$ if and only if p_i is R^{-1}. $P(\mathscr{E})$ induces exactly one path from 1 to $r+1$ in S, namely $1, 2, \cdots, r, r+1$. Thus Theorem 2 applies with $s=x=1$, $\bar{x}=2$, $y=r$ and $\bar{y}=t=r+1$. Thus $\mathscr{E}$ is hard. □

Example 10. Let us consider a specific example of the construction in Theorem 5. Suppose that $\mathscr{E}$ is $[D \circ (D \circ D^{-1} \cup D)] \cup [D \circ D^{-1} \circ D]$. Certainly $\mathscr{E}$ is nonsimple. $P(\mathscr{E})$ is the set $\{RRR^{-1}, RR, RR^{-1}R\}$. A longest path scheme Π is RRR^{-1} and the corresponding relation T is



Consider the paths induced from 1 by $\mathscr{E}$. The scheme RRR^{-1} induces paths 1234 and 1232, RR induces only 123 and $RR^{-1}R$ induces 1212. Thus $\mathscr{E}$ induces exactly one path from 1 to 4, namely 1234. □

A corollary of Theorems 4 and 5 is that the existence of an $O(ne_s)$ algorithm for any nonsimple expression over $(\cup, \circ, ^{-1})$ or $(\cup, \circ, ^*)$ implies the existence of an $O(n^2)$ algorithm for arbitrary composition, a rather unlikely possibility.

5. Summary. It has been shown that any expression involving a particular set of arguments which yields a relation having property P (a generalization of "equivalence relation") may be evaluated in $O(e)$ time. Since every equivalence relation has property P, we conclude that any equivalence relation defined as an expression over the operators $\{\cup, \circ, *, ^{-1}\}$ may be computed from the arguments of the expression in $O(e)$ time.

The "simple" expressions of § 4 were shown to be no harder than sparse transitive closure, that is, they may be evaluated in $O(ne_s)$ time.

The classes of expressions over the operators $(\cup, \circ, ^*)$ and $(\cup, \circ, ^{-1})$ were each divided into two categories, those which are "simple" and hence have $O(ne_s)$ evaluation algorithms, and those which are not simple and hence do not have $O(ne_s)$ evaluation algorithms unless arbitrary composition can be done in $O(n^2)$ time.

A general theorem was developed for showing that certain expressions require as much time for evaluation as does arbitrary composition or transitive closure. This theorem may be applied to many classes of expressions besides the ones we have considered here. Consider for example the class of expressions of the form

$$(D^{e_1} \circ D^{e_2} \circ \cdots \circ D^{e_k})^*$$

where e_i is either -1 denoting inverse, or $+1$, denoting absence of an inverse operator. These expressions are always nonsimple, yet neither Theorem 4 nor 5 applies because both the $^{-1}$ and $*$ operators may be simultaneously present. Nevertheless, Theorem 2 may be applied to show that expressions of this form are hard whenever $\sum e_i \neq 0$.

Theorems 1 and 2 of this paper give sufficient conditions for an expression being easy or hard to evaluate. In [7] we shall deal with the problem of determining when these theorems apply.

REFERENCES

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

[2] M. 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, 194 (1970), p. 524.

[3] M. J. FISCHER AND A. R. MEYER, *Boolean matrix multiplication and transitive closure*, 12th IEEE Symp. on Switching and Automata Theory, IEEE, New York, 1971, pp. 129–131.

[4] H. B. HUNT, III, T. G. SZYMANSKI AND J. D. ULLMAN, *Operations on sparse relations, with applications to grammar problems*, 15th IEEE Symp. on Switching and Automata Theory, 1974; Comm. ACM, Jan. 1977, to appear.

[5] J. E. HOPCROFT AND R. E. TARJAN, *Efficient algorithms for graph manipulation*, Comm. ACM, 16 (1973), pp. 372–378.

[6] I. MUNRO, *Efficient determination of the transitive closure of a directed graph*, Information Processing Lett., 1 (1971), pp. 56–58.

[7] T. G. SZYMANSKI, manuscript in preparation.

SIAM J. COMPUT.
Vol. 6, No. 1, March 1977

DERIVATION COMPLEXITY IN CONTEXT-FREE GRAMMAR FORMS*

SEYMOUR GINSBURG AND NANCY LYNCH†

Abstract. Let F be an arbitrary context-free grammar form and $\mathcal{G}(F)$ the family of grammars defined by F. For each grammar G in $\mathcal{G}(F)$, the derivation complexity function Φ_G, on the language of G, is defined for each word x as the number of steps in a minimal G-derivation of x. It is shown that derivations may always be speeded up by any constant factor n, in the sense that for each positive integer n, an equivalent grammar G' in $\mathcal{G}(F)$ can be found so that $\Phi_{G'}(x) \leqq |x|/n$ for all large words x, $|x|$ denoting the length of x.

Key words. complexity theory, grammar complexity, grammar forms

Introduction. In [2] the notion of a (context-free) grammar form was introduced, to model the situation where all grammars structurally close to a master grammar are being considered. The research on grammar forms to date [2], [3], [5] has been concerned with grammatical, structural, and language-theoretic problems. The present paper initiates the study of complexity-theoretic questions. Specifically, the derivation complexity function Φ_G, defined to be the minimal number of steps in a derivation of x, is examined with respect to all grammars defined by a grammar form F. It is trivial that Φ_G is at least linear and almost trivial that Φ_G is, in fact, linear. Our main result asserts that derivations may be speeded up by any constant factor n, in the sense that for each positive integer n, an equivalent grammar G' defined by F can be found so that $\Phi_{G'}(x) \leqq |x|/n$ for all large words x, $|x|$ denoting the length of x.

The basic question underlying this work is whether among the different grammar forms yielding the same family of languages, there are some which are more efficient than others. The results of this paper show that, if length of derivation is the only criterion, there is no difference among grammar forms. As will be seen, the cost of the speedup is a large increase in the size (e.g., number of productions) of the grammars used. It remains to study the resulting trade-offs.

The notion of derivation complexity was originally defined by Gladkii [6] and has been extensively studied by Book [1] for arbitrary phrase-structure grammars. Some of the results in [1] have a speedup flavor similar to ours, but the grammars in [1] accomplishing the speedup have structure very different from those of the original grammars. By carrying out our constructions within the framework of grammar forms, we preserve structure while speeding up derivations.

The paper is divided into three sections and an Appendix. Section 1 reviews grammar form concepts, defines the derivation complexity function, and determines a lower bound for it. Section 2 is concerned with proving Proposition 2.4, a special case of the main theorem. The main result itself, Theorem 3.2, is established in § 3. The proof involves first showing (Lemma 3.1) that the original grammar form may be assumed to have certain additional properties. An induction argument on the number of variables in the grammar form is then presented,

* Received by the editors November 27, 1974, and in revised form December 8, 1975.

† Computer Science Department, University of Southern California, Los Angeles, California 90007. This work was supported in part by a Guggenheim Fellowship and in part by National Science Foundation Grant GJ 42306.

with the case for one variable being exactly the situation handled in § 2. The Appendix is devoted to proving a technical combinatorial lemma, needed in § 2 to verify the main theorem for each grammar form defining the family of all context-free languages.

1. Preliminaries. In this section we first review the principal ideas relating to context-free grammar forms. Then we introduce the formalism for treating derivation complexity in grammar forms.

DEFINITION. A *(context-free) grammar form* is a 6-tuple $F = (V, \Sigma, \mathcal{V}, \mathcal{S}, \mathcal{P}, \sigma)$, where

(i) V is an infinite set of abstract symbols,

(ii) Σ is an infinite subset of V such that $V-\Sigma$ is infinite, and

(iii) $G_F = (\mathcal{V}, \mathcal{S}, \mathcal{P}, \sigma)$, called the *form grammar* (*of F*) is a context-free grammar[1] with $\mathcal{S} \subseteq \Sigma$ and $(\mathcal{V}-\mathcal{S}) \subseteq (V-\Sigma)$.

The reader is referred to [2] for motivation and further details about grammar forms.

Throughout, V and Σ are assumed to be fixed infinite sets satisfying conditions (i) and (ii) above. All context-free grammar forms considered here are with respect to this V and Σ. Also, the adjective "context-free" is usually omitted from the expression "context-free grammar form."

The purpose of a grammar form is to specify a family of grammars, each "structurally close" to the form grammar. This is accomplished by the notion of:

DEFINITION. An *interpretation* of a grammar form $F = (V, \Sigma, \mathcal{V}, \mathcal{S}, \mathcal{P}, \sigma)$ is a 5-tuple $I = (\mu, V_I, \Sigma_I, P_I, S_I)$, where

1. μ is a substitution on $\mathcal{V}^*$ such that $\mu(a)$ is a finite subset of Σ^*, $\mu(\xi)$ is a finite subset of $V-\Sigma$ for each ξ in $\mathcal{V}-\mathcal{S}$, and $\mu(\xi) \cap \mu(\eta) = \emptyset$ for each ξ and η, $\xi \neq \eta$, in $\mathcal{V}-\mathcal{S}$;

2. P_I is a subset of $\mu(\mathcal{P}) = \bigcup_{\pi \text{ in } \mathcal{P}} \mu(\pi)$, where $\mu(\alpha \to \beta) = \{u \to v / u \text{ in } \mu(\alpha), v \text{ in } \mu(\beta)\}$;

3. S_I is in $\mu(\sigma)$; and

4. $\Sigma_I(V_I)$ contains the set of all symbols in $\Sigma(V)$ which occur in P_I (together with S_I).

$G_I = (V_I, \Sigma_I, P_I, S_I)$ is called the *grammar* of I.

An interpretation is usually exhibited by indicating S_I, P_I, and (implicitly or explicitly) μ. The sets V_I and Σ_I are ordinarily not stated explicitly.

A grammar form determines a family of grammars and a family of languages as follows:

DEFINITION. For each grammar form F, $\mathcal{G}(F) = \{G_I / I \text{ an interpretation of } F\}$ is called the *family of grammars of F* and $\mathcal{L}(F) = \{L(G_I) | G_I \text{ in } \mathcal{G}(F)\}$ the *grammatical family of F*.

In this paper we are interested in studying derivation complexity in a grammar form, i.e., the derivation complexity of the grammars in $\mathcal{G}(F)$. To do this we consider the following:

[1] We assume the reader is familiar with the basic notions pertaining to context-free grammars. Here $\mathcal{V}$ is the total alphabet, $\mathcal{S}$ is the terminal alphabet, $\mathcal{P}$ is the set of productions, and σ is the start variable.

Notation. For every context-free grammar $G=(V_1, \Sigma_1, P, S)$ let Φ_G be the function on $L(G)$ in which $\Phi_G(x)$ is the minimum number of steps among all G-derivations of x,[2] for each x in $L(G)$.

Thus Φ_G is the minimum derivation function, in the sense that $\Phi_G(x)$ is the minimum number of steps necessary to derive x.

Notation. For every context-free grammar G let $\phi_G = \min\{\Phi_G(x) | x$ in $L(G), x \neq \varepsilon\}$ if G is not vacuous,[3] and $\phi_G = \infty$ otherwise.

Thus ϕ_G is the fewest number of steps needed to derive at least one non-ε word in $L(G)$.

The restriction in our definition of ϕ_G to non-ε words is needed because of the construction used in Lemma 2.1 to make finite patches on grammars.

From Lemma 2.1 of [2] we immediately get:

LEMMA 1.1. *For each grammar form F and each grammar G in $\mathscr{G}(F)$, $\phi_G \geqq \phi_{G_F}$.*

Using the above lemma we now obtain a lower bound for Φ_G.

PROPOSITION 1.2. *Let F be a grammar form and G in $\mathscr{G}(F)$. Then there exists a positive integer n so that $\Phi_G(x) \geqq \max\{\phi_{G_F}, |x|/n\}$ for*[4] *all $x \neq \varepsilon$ in $L(G)$.*

Proof. Let n be the largest number of terminal symbols on the right side of any production of G. Then for each $x \neq \varepsilon$ in $L(G)$, at least $|x|/n$ steps are needed to derive x and

$$\begin{aligned} \Phi_G(x) &\geqq \phi_G, && \text{by definition,} \\ &\geqq \phi_{G_F}, && \text{by Lemma 1.1.} \end{aligned}$$

Hence the result.

Note that the right-hand expression in the conclusion of Proposition 1.2 decreases as n increases. The question arises whether the lower bound on the right-hand side is attainable as n gets larger.

DEFINITION. A grammar form F is *minimal* if for each L in $\mathscr{L}(F)$ and each positive integer n, there exists a grammar G in $\mathscr{G}(F)$ such that $L(G)=L$ and $\Phi_G(x) \leqq \max\{\phi_{G_F}, |x|/n\}$ for all x in L.

The purpose of the present paper is to prove that every grammar form is minimal. In other words, for each grammar form F and each language L in $\mathscr{L}(F)$ a grammar G in $\mathscr{G}(F)$ can be found which speeds up the derivation as much as desired. Thus, if derivation complexity is the only criterion being considered, there is no reason to select one grammar form instead of another.

2. Minimal forms for specific grammatical families. In this section we show that all grammar forms defining the finite languages, the regular sets, the linear languages, and the context-free languages are minimal. Using this result we then prove in the next section that all grammar forms are minimal.

To establish the result about all grammar forms for the above four grammatical families we need two lemmas. The first states that if a grammar form F has an

[2] By a *G-derivation of x, with n steps,* is meant a derivation $S = w_0 \underset{G}{\Rightarrow} w_1 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} w_n = x$.

[3] A grammar G is said to be *vacuous* if $L(G) = \emptyset$ or $L(G) = \{\varepsilon\}$. A grammar form F is said to be *vacuous* if G_F is vacuous.

[4] For each word x, $|x|$ denotes its length.

interpretation I which derives all long words x in $L(G_I)$ in at most $|x|/n$ steps, then F has another interpretation defining $L(G_I)$ which derives all possible words x in $L(G_I)$ in at most $|x|/n$ steps. In other words, a "finite patch" may be made for short words.

LEMMA 2.1. *Let F be a nonvacuous form and n a positive integer. Suppose there exists a positive integer k and a grammar G in $\mathcal{G}(F)$ such that $\Phi_G(x) \leqq \max\{k, |x|/n\}$ for all x in $L(G)$. Then there exists a grammar G' in $\mathcal{G}(F)$ such that $L(G') = L(G)$ and $\Phi_{G'}(x) \leqq \max\{\phi_{G_F}, |x|/n\}$ for all x in $L(G')$.*

Proof. The argument consists of adding to G productions that generate, in ϕ_{G_F} steps, the finite number of words x in $L(G)$ for which $|x|/n < k$.

Since $F = (V, \Sigma, \mathcal{V}, \mathcal{S}, \mathcal{P}, \sigma)$ is not vacuous, there is a ϕ_{G_F}-step derivation

$$(1) \qquad \sigma = z_0 \underset{G_F}{\Longrightarrow} \cdots \underset{G_F}{\Longrightarrow} z_{\phi_{G_F}} = z$$

of a non-ε word z in $L(G_F)$. For each i, $1 \leqq i \leqq \phi_{G_F}$, let

$$(2) \qquad \beta_i \to w_i$$

be the production used in $z_{i-1} \underset{G_F}{\Rightarrow} z_i$. Since $z \neq \varepsilon$, there exists some j such that z_j contains a symbol of $\mathcal{S}$, say $w_j = w_{j1} a_j w_{j2}$, where a_j is in $\mathcal{S}$ and w_{j1}, w_{j2} are in $\mathcal{V}^*$. Note that for every i and every occurrence of a variable on the right side of the ith production in (2), there is a unique integer l such that $\beta_l \to w_l$ is the production applied to that variable in the derivation (1).

Now let $G = (V_1, \Sigma_1, P_1, S)$ and x be an arbitrary word in $L(G)$, with $|x|/n < k$. For each i, $2 \leqq i \leqq \phi_{G_F}$, let $A_{i,x}$ be a new variable (in $V - \Sigma$). Let $A_{1,x} = S$. For each x, consider the set of new productions

$$(3) \qquad \{A_{i,x} \to v_{i,x} \mid 1 \leqq i \leqq \phi_{G_F}, i \neq j\} \cup \{A_{j,x} \to v_{j1x} x v_{j2x}\},$$

where each v_{ix}, v_{j1x}, v_{j2x} is obtained from the corresponding word w_i, w_{j1}, w_{j2} by deleting all symbols in $\mathcal{S}$ and replacing each variable by the appropriate variable $A_{l,x}$. Clearly the rules in (3) derive just the word x and no production $A_{i,x} \to v_{i,x}$ can be applied to any variable $A_{i',y}$ unless $i = i' = 1$.

Let $G' = (V_2, \Sigma_1, P_2, S)$, where

$$V_2 = V_1 \cup \{A_{i,x} \mid x \text{ in } L(G), \frac{|x|}{n} < k, 2 \leqq i \leqq \phi_{G_F}\}.$$

Obviously G' is in $\mathcal{G}(F)$, $L(G') = L(G)$, and G' has a ϕ_F-step derivation for all x in $L(G) = L(G')$. Thus G' satisfies the conclusion of the lemma.

Remark. An alternative formulation of Lemma 2.1 is the following: Let F be a nonvacuous form and n a positive integer. Suppose there exists a positive integer r such that $\Phi_G(x) \leqq |x|/n$ for all x in $L(G)$ having $|x| \geqq r$. Then there exists a grammar G' in $\mathcal{G}(F)$ so that $L(G') = L(G)$ and $\Phi_{G'}(x) \leqq \max\{\phi_{G_F}, |x|/n\}$ for all x in $L(G')$.

COROLLARY 2.2. *Let $F = (V, \Sigma, \mathcal{V}, \mathcal{S}, \mathcal{P}, \sigma)$ and $F' = (V, \Sigma, \mathcal{V}', \mathcal{S}', \mathcal{P}', \sigma)$ be equivalent grammar forms,*[5] *with $\mathcal{P} \subseteq \mathcal{P}'$. If F is minimal, then so is F'.*

[5] Grammar forms F and F' are said to be *equivalent* if $\mathcal{L}(F) = \mathcal{L}(F')$.

The second lemma states that minimal grammar forms are not affected by either adding or removing "redundant" productions.

LEMMA 2.3. *Let F be a nonvacuous grammar form and $\beta \overset{*}{\underset{G_F}{\Longrightarrow}} w$ a derivation, with β a variable. Let F' be the grammar form obtained by adding to F the production $\beta \to w$. Then F is minimal if and only if F' is minimal.*

Proof. Suppose F is minimal. By Proposition 2.1 of [2], $\mathscr{L}(F') = \mathscr{L}(F)$. Hence F' is minimal by Corollary 2.2.

Now assume that F' is minimal. Let L be in $\mathscr{L}(F) = \mathscr{L}(F')$ and n be a positive integer. Since $\beta \overset{*}{\underset{G_F}{\Longrightarrow}} w$, $\beta \overset{k}{\underset{G_F}{\Longrightarrow}} w$ for[6] some nonnegative integer k. Suppose $k \geqq 1$. Then there exists $G' = (V_1, \Sigma_1, P', S)$ in $\mathscr{G}(F')$ such that $L(G') = L$ and $\Phi_{G'}(x) \leqq \max\{\phi_{G_{F'}}, |x|/(kn)\}$ for all x in L. Also, there exists a sequence $\pi_1, \cdots, \pi_k$ of productions in F which realize $\beta \overset{k}{\Longrightarrow} w$. Let $G = (V_2, \Sigma_1, P, S)$ in $\mathscr{G}(F)$ be obtained from G' as follows. Each production $A \to y$ in G' which comes from the production $\beta \to w$ in F' is replaced by a sequence of k productions, corresponding to $\pi_1, \cdots, \pi_k$, which realizes $A \overset{*}{\Longrightarrow} y$. As in the proof of Lemma 2.1, new intermediate variables are introduced in the sequence in such a way that the new sequence can only derive $A \overset{*}{\Longrightarrow} y$. Let V_2 be V_1 together with all the new variables introduced. Clearly $L(G) = L(G')$. Since every production of G' requires at most k productions of G to simulate it, $\Phi_G(x) \leqq \max\{k\phi_{G_F}, k|x|/(kn)\}$ for all x in L. By Lemma 2.1, there exists $\bar{G}$ in $\mathscr{G}(F)$ such that $L(\bar{G}) = L$ and $\Phi_{\bar{G}}(x) \leqq \max\{\phi_{G_F}, |x|/n\}$ for all x in L.

Suppose $k = 0$. Then there exists $G' = (V_1, \Sigma_1, P', S)$ in $\mathscr{G}(F')$ such that $L(G') = L$ and $\Phi_{G'}(x) \leqq \max\{\phi_{G_{F'}}, |x|/n\}$ for all x in L. Let $G = (V_2, \Sigma_1, P, S)$, where

$P = (P' - \{A \to B$ in $P' | A$ and B in $\mu(\beta)\}) \cup \{A \to y | A$ in $\mu(\beta)$, y in $V^* - \mu(\beta)$, there exist $l \geqq 1$ and $A_1, \cdots, A_l$ in $\mu(\beta)$ such that $A \to A_1$, $A_i \to A_{i+1}$, $A_l \to y$, $1 \leqq i \leqq l-1$, are in $P'\}$.

Clearly G is in $\mathscr{G}(F)$ and $L(G) = L$. Since each derivation δ' in G' of a word x in L has an obvious corresponding derivation δ in G of x, with at most the same number of steps as in δ', $\Phi_G(x) \leqq \max\{\phi_{G_{F'}}, |x|/n\}$ for each x in L. By Lemma 2.1, there exists $\bar{G}$ in $\mathscr{G}(F)$ such that $L(\bar{G}) = L$ and $\Phi_{\bar{G}}(x) \leqq \max\{\phi_{G_F}, |x|/n\}$ for all x in L.

Thus, for any value of k, there exists $\bar{G}$ in $\mathscr{G}(F)$ so that $L(\bar{G}) = L$ and $\Phi_{\bar{G}}(x) \leqq \max\{\phi_{G_F}, |x|/n\}$ for all x in L. Hence F is minimal.

We are now ready for the main result of the section.

PROPOSITION 2.4. *In each of the following cases, $F = (V, \Sigma, \mathscr{V}, \mathscr{S}, \mathscr{P}, \sigma)$ is minimal:*

(a) *$\mathscr{L}(F)$ is the family of all finite sets.*

(b) *$\mathscr{L}(F)$ is the family of all regular sets.*

(c) *$\mathscr{L}(F)$ is the family of all linear languages.*

(d) *$\mathscr{L}(F)$ is the family of all context-free languages.*

[6] By $u \overset{k}{\Rightarrow} v$ is meant that there exist $u_1, \cdots, u_{k-1}$ such that $u_0 = u \Rightarrow u_1 \Rightarrow \cdots \Rightarrow u_k = v$.

Proof. (a) Since $\mathscr{L}(F)$ is the family of all finite sets, by Theorem 2.1 of [2] there is a word w in[7] $\mathscr{S}^+$ such that $\sigma \overset{*}{\underset{G_F}{\Longrightarrow}} w$. Let F' be the grammar form obtained from F by adding the production $\sigma \to w$. By Lemma 2.3 it suffices to show that F' is minimal.

Let $L=\{x_1, \cdots, x_k\}$ be any finite set and let n be an arbitrary positive integer. If $k=0$ there is nothing to prove. Suppose $k \geqq 1$. Let $G=(\{S\} \cup \Sigma_1, \Sigma_1, P, S)$ be the grammar where Σ_1 is the set of all symbols appearing in any of the x_i, $1 \leqq i \leqq k$, and $P=\{S \to x_i | 1 \leqq i \leqq k\}$. Obviously G is in $\mathscr{G}(F')$, $L(G)=L$, and $\Phi_G(x)=1 \leqq \max\{\phi_{G_{F'}}, |x|/n\}$ for all x in L. Thus F' is minimal.

(b) Since $\mathscr{L}(F)$ is the class of all regular sets, $L(G_F)$ is an infinite set by Theorem 2.1 of [2]. By [9], there exist x_1, x_2, x_3, x_4, x_5 in $\mathscr{S}^*$ and β in $\mathscr{V}-\mathscr{S}$ such that x_5 is in $\mathscr{S}^+$, x_3x_4 is in $\mathscr{S}^+$, $\sigma \overset{*}{\underset{G_F}{\Longrightarrow}} x_1\beta x_2$, $\beta \overset{*}{\underset{G_F}{\Longrightarrow}} x_3\beta x_4$, and $\beta \overset{*}{\underset{G_F}{\Longrightarrow}} x_5$. By Lemma 2.3, there is no loss of generality in assuming that $\sigma \to x_1\beta x_2$, $\beta \to x_3\beta x_4$, and $\beta \to x_5$ are in $\mathscr{P}$. By symmetry, there is no loss in assuming $x_3 \neq \varepsilon$.

Let L be any regular set. Then $L=L(G)$ for some right-linear grammar $G=(V_1, \Sigma_1, P, S)$ in which $A \to wB$ in P, A and B variables, implies w is in Σ_1^+. Let n be an arbitrary positive integer. Let S' be a new variable,[8]

$$P_2=\{S' \to S\} \cup \{A \to wB | A, B \text{ in } V_1-\Sigma_1, A \xrightarrow[G]{\leqq n+1} wB\}$$

$$\cup \{A \to w | A \text{ in } V_1-\Sigma_1, w \text{ in } \Sigma_1^*, A \overset{\leqq n+1}{\underset{G}{\Longrightarrow}} w\},$$

and $G'=(\{S'\} \cup V_1, \Sigma_1, P_2, S')$. Clearly $L(G')=L(G)=L$. Also G' is in $\mathscr{G}(F)$. [For one can construct an interpretation (μ, G') of F for which $S' \to S$ is in $\mu(\sigma \to x_1\beta x_2)$, $A \to wB$ is in $\mu(\beta \to x_3\beta x_4)$ for every production $A \to wB$, A and B variables, in P_2, and $A \to w$ is in $\mu(\beta \to x_5)$ for every production $A \to w$, w in Σ_1^*, in P_2.] Consider any word x in L. Obviously there exists a derivation in G' of x so that except, perhaps, for the first and last productions, each production deposits at least $n+1$ terminals. Thus $\Phi_{G'}(x) \leqq 2+|x|/(n+1)$. For x sufficiently large, $2+|x|/(n+1)<|x|/n$. Hence, $\Phi_{G'}(x) \leqq |x|/n$ for all large x. By Lemma 2.1, F is minimal.

(c) Since $\mathscr{L}(F)$ is the family of all linear languages, by Theorem 2.4 of [2] there exist x_1, x_2, x_3, x_4, x_5 in $\mathscr{S}^*$ and β in $\mathscr{V}-\mathscr{S}$ such that x_3, x_4, x_5 are in $\mathscr{S}^+$ and $\sigma \overset{*}{\underset{G_F}{\Longrightarrow}} x_1\beta x_2$, $\beta \overset{*}{\underset{G_F}{\Longrightarrow}} x_3\beta x_4$, and $\beta \overset{*}{\underset{G_F}{\Longrightarrow}} x_5$. By Lemma 2.3, we may assume that $\sigma \to x_1\beta x_2$, $\beta \to x_3\beta x_4$, and $\beta \to x_5$ are in $\mathscr{P}$.

Now let L be any linear language. Then $L=L(G)$ for some linear grammar $G=(V_1, \Sigma_1, P, S)$ such that $A \to uBv$, A and B in $\mathscr{V}-\mathscr{S}$, implies uv in Σ_1^+. Let n be

[7] For each set E of words, $E^+=\bigcup_{i=1}^{\infty} E^i$.

[8] By $u \overset{\leqq k}{\underset{G}{\Longrightarrow}} v$ is meant that there exists a derivation of v from u in at most k (possibly 0) steps.

an arbitrary positive integer. Let S' be a new variable,

$$P_2 = \{S' \to S\} \cup \{A \to uBv \mid A, B \text{ in } V_1 - \Sigma_1, A \overset{\leqq n+1}{\underset{G}{\Longrightarrow}} uBv\}$$

$$\cup \{A \to w \mid A \text{ in } V_1 - \Sigma_1, w \text{ in } \Sigma_1^*, A \overset{\leqq n+1}{\underset{G}{\Longrightarrow}} w\},$$

and $G' = (\{S'\} \cup V_1, \Sigma_1, P_2, S')$. The remainder of the argument is as in part (b).

(d) As often happens in proofs about grammar forms, the case where $\mathscr{L}(F)$ is the family of all context-free languages is proved very differently from the other cases, although the statement of the result is similar. Here we cannot simply compose productions as in (b) and (c), since the resulting productions would not necessarily be in the required form. Instead, we introduce productions where possible which simulate within the required form the result of composing sequences of productions. We then use a combinatorial lemma to show that the simulating productions are sufficient for the task at hand.

Since $\mathscr{L}(F)$ is the family of all context-free languages, by Theorem 2.2 of [2] there exist x_1, x_2, x_3, x_4, x_5 in $\mathscr{S}^*$, x_6 in $\mathscr{S}^+$, and β in $\mathscr{V} - \mathscr{S}$ such that $\sigma \overset{*}{\underset{G_F}{\Longrightarrow}} x_1\beta x_2$, $\beta \overset{*}{\underset{G_F}{\Longrightarrow}} x_3\beta x_4\beta x_5$, and $\beta \overset{*}{\underset{G_F}{\Longrightarrow}} x_6$. By Lemma 2.3, we may assume that $\sigma \to x_1\beta x_2$, $\beta \to x_3\beta x_4\beta x_5$, and $\beta \to x_6$ are in $\mathscr{P}$. Intuitively this means that it suffices to prove the results for $G_F = (\{\sigma, a\}, \{a\}, \{\sigma \to \sigma\sigma, \sigma \to a\}, \sigma)$.

Now let L be any context-free language. Then there exists a grammar $G = (V_1, \Sigma_1, P_1, S)$ such that $L = L(G)$ and each production of P_1 is of the type $A \to BC$ or $A \to w$, where A, B, C are variables, w is in Σ_1^*, neither B nor C is S, and $A = S$ if $w = \varepsilon$. (Thus S never appears on the right side of any production and S is the only variable which derives ε.) Intuitively, we shall construct a grammar G' as follows. We consider a derivation of a word w in G, represented by a derivation tree. We put into G' productions which simulate the effects of G-productions used near the ends of branches. (See part 4 below.) Thus if the derivation tree of w is very wide, then these productions yield the needed speed-up. On the other hand, the derivation tree of w may be very narrow, with very little internal branching. In this case, the new productions do not speed up the derivation sufficiently. To obtain the speed-up here, we put into G' productions which "condense" long internal paths having little branching. (See part 6 below.)

Proceeding more formally, note that

1. if $A \overset{s}{\underset{G}{\Longrightarrow}} w$, where A is in $V_1 - \Sigma_1$ and w is in $\Sigma_1^*(V_1 - \Sigma_1)\Sigma_1^*$, then $s \leqq 2|w| - 2$, and

2. if $A \overset{s}{\underset{G}{\Longrightarrow}} w$, where A is in $V_1 - \Sigma_1$ and w is in Σ_1^+, then $s \leqq 2|w| - 1$.

Let n be an arbitrary positive integer. Let $G' = (V_2, \Sigma_1, P_2, S')$, where S' is a new variable, V_2 consists of the symbols of V_1 together with all variables in P_2, and P_2 is defined as follows:

3. $S' \to S$ is in P_2.

4. $A \to w$ is in P_2 if $A \overset{\leqq 112n}{\underset{G}{\Longrightarrow}} w$, where A is in $V_1-\Sigma_1$ and w is in Σ_1^*.

5. $A \to BC$ if A, B, C are in $V_1-\Sigma_1$ and $A \to BC$ is in P_1.

6. For each derivation δ: $A \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} vBw$ of at most $112n$ steps, with A, B in $V_1-\Sigma_1$ and v, w in Σ_1^*, let D_δ, E_δ, and H_δ be new variables. Let $A \to D_\delta E_\delta$, $E_\delta \to BH_\delta$, $D_\delta \to v$, and $H_\delta \to w$ be in P_2.

The indexing of the variables in part 6 is done, as in the proof of Lemma 2.1, to keep the new variables distinct, so that each new production can be used only as part of the simulation of the derivation of G for what it was intended.

From parts 2 and 4, we get

7. $A \to w$ is in P_2 if $A \overset{*}{\underset{G}{\Longrightarrow}} w$, with A in $V_1-\Sigma_1$, w in Σ_1^*, and $|w| \leqq 56n$.

It is easily seen that G' is in $\mathscr{G}(F)$. Since $P_1 \subseteq P_2$ and new productions (except $S' \to S$) in P_2 are only used to simulate productions in P_1, it follows that $L(G') = L(G) = L$. To complete the argument, it remains to show that G' has sufficiently short derivations of all words in L.

Let x be an arbitrary word in L. For each G'-derivation of x, there is associated in the obvious way [4] a G'-derivation tree[9] $T(x)$. Let $\bar{T}(x)$ be the tree obtained from $T(x)$ by deleting the root, all leaf nodes, and all edges incident to these nodes. Note that $\bar{T}(x)$ is a binary tree.[10] Consider the set of all such derivation trees $\bar{T}(x)$ for x. Let $\bar{T}_0(x)$ be one such tree with the fewest number of nodes, and let $T_0(x)$ be a tree giving rise to $\bar{T}_0(x)$. Then $\bar{T}_0(x)$ has the following three properties:

8. No two consecutive[11] internal nodes of $\bar{T}_0(x)$ have both their node names in $V_2 - V_1$.

9. Each internal node of $\bar{T}_0(x)$ with at least one daughter an internal node generates a subtree of $T_0(x)$ whose terminal word is of length $\geqq 56n$.

10. Let $n_1, \cdots, n_6$ be six consecutive internal nodes of $\bar{T}_0(x)$ and for each i, $2 \leqq i \leqq 6$, let m_i be the other daughter of n_{i-1}. Suppose that each m_i is a leaf in $\bar{T}_0(x)$ and $B_i \to w_i$ in $T_0(x)$, B_i the node name of m_i. Then $|w_2 \cdots w_6| \geqq 56n$.

For consider part 8. Let n_1 and n_2 be consecutive internal nodes of $\bar{T}_0(x)$. Let A and B be the node names of n_1 and n_2 respectively. Suppose A is in $V_2 - V_1$. Then by part 6, there is a δ such that A is E_δ and B is H_δ. By construction, $B = H_\delta$ cannot be an internal node of $\bar{T}_0(x)$.

Consider part 9. Suppose it is false. Then there exist consecutive internal nodes n_1 and n_2 of $\bar{T}_0(x)$ such that n_1 (thus n_2) generates a subtree of $T_0(x)$ whose terminal word w_1 (w_2) is of length smaller than $56n$. Let A and B be the node names of n_1 and n_2 respectively. By part 8, one of the variables, say A, is in V_1. Then $A \overset{*}{\underset{G}{\Longrightarrow}} w_1$. By part 7 $A \to w_1$ is in P_2. Replacing the subtree in $T_0(x)$ realizing

[9] Trees are viewed with the root node at the top. A node leading downward to another node is called an *internal* node. Otherwise, the node is called a *leaf*. If nodes n_1 and n_2 are jointed by an edge, with n_2 below n_1, then n_2 is called a *daughter* of n_1, and n_1 the *father* of n_2.

[10] A tree is called *binary* if each internal node has exactly two daughters.

[11] Nodes $n_1, \cdots, n_r$ are *consecutive* if each n_{i+1} is a daughter of n_i, $i \geqq 2$.

$A \overset{*}{\underset{G'}{\Rightarrow}} w_1$ by the subtree representing $A \rightarrow w_1$ gives rise to a G'-derivation tree $T_1(x)$, deriving x, with the property that $\bar{T}_1(x)$ has fewer nodes than $\bar{T}_0(x)$. (The two daughter nodes of n_1 in $\bar{T}_0(x)$ are no longer present.) This is a contradiction. A similar contradiction arises if B is in V_1. Hence part 9 holds.

Consider part 10. Suppose it is false. Let A_i, $1 \leqq i \leqq 6$, and B_j, $2 \leqq j \leqq 6$, be the node names of n_i and m_j, respectively. By part 8, either A_1 or A_2, say A_2, and either A_5 or A_6, say A_5, is in V_1. [An analogous argument holds if any of the other three possibilities occurs.] Since m_3, n_3 are daughters of n_2; m_4, n_4 are daughters of n_3; and m_5, n_5 are daughters of n_4, we have as productions in P_2, $A_2 \rightarrow A_3B_3$ or $A_2 \rightarrow B_3A_3$, $A_3 \rightarrow A_4B_4$ or $A_3 \rightarrow B_4A_4$, and $A_4 \rightarrow A_5B_5$ or $A_4 \rightarrow B_5A_5$. Suppose $A_2 \rightarrow B_3A_3$, $A_3 \rightarrow A_4B_4$, and $A_4 \rightarrow B_5A_5$ are the productions in P_2 realizing the above daughter relations. [An analogous argument holds if one of the other combinations occur.] Thus

$$A_2 \underset{G'}{\Rightarrow} B_3A_3 \underset{G'}{\Rightarrow} B_3A_4B_4 \underset{G'}{\Rightarrow} B_3B_5A_5B_4 \overset{*}{\underset{G'}{\Rightarrow}} w_3w_5A_5w_4.$$

Since A_2 and A_5 are in V_1, $A_2 \overset{*}{\underset{G}{\Rightarrow}} w_3w_5A_5w_4$. By assumption, $|w_2 \cdots w_6| < 56n$. Thus $|w_3w_5w_4| < 56n$. By part 1, there exists a derivation $\delta : A_2 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} w_3w_5A_5w_4$, of at most $112n$ steps. Replacing in T_0 the subgraph realizing $A_2 \rightarrow B_3A_3$, $B_3 \rightarrow w_3$, $A_3 \rightarrow A_4B_4$, $B_4 \rightarrow w_4$, $A_4 \rightarrow B_5A_5$, $B_5 \rightarrow w_5$ by the graph realizing $A_2 \rightarrow D_\delta E_\delta$, $E_\delta \rightarrow A_5H_\delta$, $D_\delta \rightarrow w_3w_5$, $H_\delta \rightarrow w_4$ gives rise to a G' derivation tree $T_1(x)$ for x. Then $\bar{T}_1(x)$ has two nodes fewer than $\bar{T}_0(x)$. This contradicts the minimality of $\bar{T}_0(x)$. Hence part 10 holds.

Using the above symbolism for trees, let r be a positive integer such that $\bar{T}_0(x)$ has at least two internal nodes for each x in L, $|x| \geqq r$. Clearly r exists. By Lemma 2.1, it suffices to show that $\Phi_{G'}(x) \leqq |x|/n$ for each word x in L, $|x| \geqq r$.

Let x be any word in L, with $|x| \geqq r$. Consider the following result, whose proof is in the Appendix.

LEMMA 2.5. *Let T be a binary tree with at least two internal nodes and exactly l leaf nodes. Suppose there exists a positive integer k and a weight function ω which assigns a nonnegative integer to every leaf node in such a way that the following two properties hold*:

(a) *For each internal node n_0 which has at least one daughter an internal node, $\sum_{m \text{ in } Q(n_0)} \omega(m) \geqq k$, where $Q(n_0)$ is the set of all leaf nodes in the subtree generated by n_0.*

(b) *If $n_1, \cdots, n_6$ are six arbitrary consecutive internal nodes and $m_2, \cdots, m_7$ are leaf nodes such that each m_i is a daughter of n_{i-1}, then $\sum_{i=2}^{7} \omega(m_i) \geqq k$.*
Then

$$\sum_{m \text{ a leaf}} \omega(m) \geqq \frac{kl}{28}.$$

In Lemma 2.5, let $k = 56n$, let $T = \bar{T}_0(x)$, and for each leaf node m in $\bar{T}_0(x)$ let $\omega(m) = |w|$, where A is the node name of m and $A \rightarrow w$ is the production in P_2 realizing the subtree in $T_0(x)$ generated by m. By parts 9 and 10, (a) and (b) of

Lemma 2.5 are satisfied. (There is some redundancy in (b).) By the conclusion of Lemma 2.5, $\sum_{m \text{ a leaf}} \omega(m) \geqq (56n/28)l = 2nl$, where l is the number of leaf nodes in $\bar{T}_0(x)$. Now the number of steps in any derivation realizing $T_0(x)$ is 1 (for $S' \rightarrow S$) plus the number of internal nodes of $\bar{T}_0(x)$ plus l. Since $\bar{T}_0(x)$ is a binary tree, it is easily seen that l is 1 plus the number of internal nodes. Hence the number of steps in any derivation realizing $T_0(x)$ is $2l$. But $|x| = \sum_{m \text{ a leaf}} \omega(m)$. Therefore $\Phi_{G'}(x) = 2l \leqq |x|/n$, and the proof of Proposition 2.4 is complete.

3. Minimality of arbitrary grammar forms. In the previous section we proved that all the grammar forms for some special types of grammatical families were minimal. In the present section we establish the result for all grammar forms for all grammatical families.

The argument for the main result is as follows. In § 3 of [2] a procedure was given for converting an arbitrary grammar form into an equivalent, completely reduced, sequential one. This procedure is exploited here to show that the transformation cannot convert a nonminimal grammar form into a minimal one. It is then proved that the resulting grammar form is always minimal.

LEMMA 3.1. *For every grammar form F there exists an equivalent, completely reduced,*[12] *sequential grammar form*[13] *F' such that F is minimal if F' is.*

Proof. If F is vacuous then $L(G_F) = \emptyset$ or $L(G_F) = \{\varepsilon\}$. Thus $\mathcal{L}(F) = \{\emptyset\}$ or $\mathcal{L}(F) = \{\emptyset, \{\varepsilon\}\}$. In the former case, let F' be a form with no productions, and in the latter let F' be a grammar form with the single production $\sigma \rightarrow \varepsilon$. Clearly F and F' are both minimal, and F' satisfies the conclusion of the lemma.

Suppose that F is not vacuous. For the remainder of this proof, we assume the reader is familiar with the contents of § 3 of [2]. We follow the transformation procedure given there, noting that each step of the procedure cannot change a nonminimal grammar form into a minimal one. There are five parts to consider.

(a) By the proof of Lemma 3.1 of [2] a reduced, equivalent grammar form F_a is obtained from F. Since F_a is constructed by deleting the useless productions of F, i.e., those productions involved in no derivation of a terminal word, only useless productions of each G in $\mathcal{G}(F)$ are deleted. Thus F is minimal if F_a is.

(b) By Lemma 3.2 of [2], an equivalent, reduced, noncyclic grammar form F_b is obtained from F_a. Assume F_b is minimal. Let F'' be the grammar form obtained by adding to F_b all productions $\beta \rightarrow \gamma$, β and γ variables, for which $\beta \underset{G_{F_a}}{\overset{*}{\Longrightarrow}} \gamma$. Then $\mathcal{L}(F_b) = \mathcal{L}(F_a) = \mathcal{L}(F'')$. Since each production in F_b is in F'', it follows from Corollary 2.2 that F'' is minimal. By repeated use of Lemma 2.3, F_a is minimal.

(c) By Lemma 3.3 of [2] an equivalent reduced grammar form F_c containing no production of the kind $\xi \rightarrow \eta$, ξ and η variables, is obtained from F_b. Suppose F_c is minimal. Repeating the argument in (b) above, with F_c replaced by F_a and F_a by F_b, it is easily seen that F_b is minimal.

[12] A grammar form $F = (V, \Sigma, \mathcal{V}, \mathcal{S}, \mathcal{P}, \sigma)$ is said to be *completely reduced* if G_F is reduced, there are no variables α and β in $\mathcal{V} - \mathcal{S}$ such that $\alpha \rightarrow \beta$ is in $\mathcal{P}$, and for each variable α in $\mathcal{V} - (\mathcal{S} \cup \{\sigma\})$ there exist x and y in $\mathcal{S}^*$, $xy \neq \varepsilon$, such that $\alpha \rightarrow x\alpha y$ is in $\mathcal{P}$.

[13] A context-free grammar (V_1, Σ_1, P, S) is *sequential* if the variables can be ordered $S = A_1, \cdots, A_r$ such that if $A_i \rightarrow uA_jv$ is any production in P, then $j \geqq i$. A grammar form F is *sequential* if G_F is sequential.

(d) By Lemma 3.4 of [2], an equivalent, completely reduced grammar form F_d is obtained from F_c. Assume F_d is minimal. If every variable of F_c is partially self-embedding, then F_c is minimal by Lemma 2.3. Suppose that F_c has exactly $k > 0$ variables which are not partially self-embedding. The procedure in (β) of Lemma 3.4 of [2] shows how to obtain from F_c an equivalent, reduced grammar form F'_c having exactly $k-1$ variables which are not partially self-embedding. This procedure is iterated k times until F_d is obtained. To show that F_c is minimal, it therefore suffices to prove that F_c is minimal provided that F'_c is minimal.

Assume F'_c is minimal. Let L be in $\mathscr{L}(F_c) = \mathscr{L}(F'_c)$ and n be an arbitrary positive integer. Let l be 1 plus the maximum number of times any single variable appears on the right of any single production of F_c. Then there exists a grammar G' in $\mathscr{G}(F'_c)$ such that $L(G') = L$ and $\Phi_{G'}(x) \leqq \max\{\phi_{G_{F'}}, |x|/(ln)\}$ for all x in L. By the method of construction of F'_c from F_c, there exists a grammar G in $\mathscr{G}(F_c)$ such that $L(G) = L$ and the following holds: For every G'-derivation δ' of a word x in $L(G')$ there is a G-derivation δ of x in which each step of δ' is simulated by at most l steps of δ. Then $\Phi_G(x) \leqq \max\{l\phi_{G_{F'_c}}, |x|/n\}$ for all x in L. The minimality of F_c follows from Lemma 2.1.

(e) By Theorem 3.1 of [2], an equivalent, completely reduced, sequential grammar form F' is obtained from F_d. Assume F' is minimal. Let F'_e be a grammar form obtained by adding to F_d one production of the kind $\beta \rightarrow v\gamma w$, v and w in $\mathscr{S}^*$, for all variables β and γ, $\beta \neq \gamma$, in F_d such that $\beta \overset{*}{\underset{G_{F_d}}{\Longrightarrow}} v'\gamma w'$ for some v' and w' in $\mathscr{S}^*$. By Lemma 2.3, it suffices to show that F'_c is minimal.

Let L be in $\mathscr{L}(F'_e) = \mathscr{L}(F_d) = \mathscr{L}(F')$ and n be an arbitrary positive integer. Let k be 2 plus the maximum number of variables on the right side of any production in F'. Then there exists a grammar G' in $\mathscr{G}(F')$ such that $L(G) = L$ and $\Phi_{G'}(x) \leqq \max\{\phi_{G_{F'}}, |x|/(kn)\}$. In an obvious way there exists a grammar G'_e in $\mathscr{G}(F'_e)$ such that $L(G'_e) = L(G')$ and for each G'-derivation δ' of a word x in L there corresponds a G'_e-derivation δ'_e of x in which each step of δ' is simulated by at most k steps in δ'_e. (Specifically, each production p, corresponding to a production in F', may be replaced by one production corresponding to a production in F_d, plus one production corresponding to a production of the form $\beta \rightarrow v\gamma w$ for every variable in p.) Thus $L(G'_e) = L$ and $\Phi_{G'_e}(x) \leqq \max\{k\phi_{G_{F'}}, |x|/n\}$ for all x in L. This implies the minimality of F'_e.

Combining (a)–(e), we obtain our result.

THEOREM 3.2. *Every grammar form is minimal.*

Proof. Clearly each vacuous grammar form is minimal. Consider nonvacuous grammar forms. By Lemma 3.1, we may restrict our attention to completely reduced, sequential grammar forms. The proof will be by induction on the number k of variables in the grammar form.

Suppose F is a completely reduced, sequential grammar form with just one variable. By Lemma 5.1 of [2], $\mathscr{L}(F)$ is either the family of finite, regular, linear, or context-free languages. By Proposition 2.4, F is minimal.

Now assume the theorem is true for all completely reduced, sequential grammar forms with at most k variables. Let $F = (V, \Sigma, \mathscr{V}, \mathscr{S}, \mathscr{P}, \sigma)$ be a completely reduced, sequential grammar form with $k+1$ variables. Thus, the variables in $\mathscr{V}$ can be arranged into a sequence $\sigma = \alpha_0, \cdots, \alpha_k$ so that for each production $\alpha_i \to u\alpha_j v$ in $\mathcal{P}$, $i \leqq j$. If $\mathcal{L}(F)$ is the family of context-free languages, then by Proposition 2.4 we are through. Thus assume $\mathcal{L}(F)$ is not the family of all context-free languages. We may assume similarly that F is nontrivial[14] (since otherwise F is either vacuous or generates the family of all finite sets.) By Lemma 2.3 we may assume the following:

1. For each i, $0 \leqq i \leqq k$, there exists v_i in $\mathcal{S}^+$ such that $\alpha_i \to v_i$ is in $\mathcal{P}$.

2. If there exist w_1, w_2 in $(\mathcal{V}-\{\sigma\})^+$ such that $\sigma \overset{*}{\underset{G_F}{\Longrightarrow}} w_1\sigma w_2$, then there exist x_1, x_2 in $\mathcal{S}^+$ such that $\sigma \to x_1\sigma x_2$ is in $\mathcal{P}$.

3. If there exist w in $(\mathcal{V}-\{\sigma\})^+$ such that $\sigma \overset{*}{\underset{G_F}{\Longrightarrow}} w\sigma$ $(\sigma \to \sigma w)$ then there exists x in $\mathcal{S}^+$ such that $\sigma \to x\sigma$ $(\sigma \to \sigma x)$ is in $\mathcal{P}$.

Intuitively, we proceed as follows. Consider the collection of "component" grammar forms arising from F by treating each variable of F except σ, in turn, as the start variable. Each such component form has at most k variables and thus, by induction, is minimal. Given any interpretation grammar G of F, the speed-up is accomplished by a grammar G' constructed as follows: Consider the collection of "components" of G, i.e., the parts of G which correspond to respective component forms of F. By the minimality of the component forms, each component of G may be sped up. The productions accomplishing this are then placed into G'. (See part 6 below.) In addition, productions which speed up short derivations are also placed into G'. (See part 4 below.) Similarly, productions which speed up the portion of G not part of any component of G (i.e., the portion involving variables corresponding to σ) are placed into G'. (See part 5 below.)

More formally, let $F_i = (V, \Sigma, \mathcal{V}_i, \mathcal{S}, \mathcal{P}_i, \alpha_i)$ be the grammar form in which $\mathcal{V}_i = \mathcal{S} \cup \{\alpha_j \mid \alpha_i \overset{*}{\underset{G_F}{\Longrightarrow}} w_1\alpha_j w_2$ for some w_1, w_2 in $\mathcal{V}^*\}$ and $\mathcal{P}_i$ be the set of all productions in F involving only variables in $\mathcal{V}_i$. Then F_i is nontrivial, completely reduced, sequential, and has at most k variables. For each i, $\phi_{G_{F_i}} = 1$ by assumption 1 above.

Let L be in $\mathcal{L}(F)$ and n be an arbitrary positive integer. There exists an interpretation (μ, G) of F such that $L(G) = L$. Let $G = (V_1, \Sigma_1, P, S)$. We shall modify G to obtain a new grammar G' obtaining the speedup of L by constant n. Let A be an arbitrary variable in $V_1 - (\Sigma_1 \cup \mu(\sigma))$. Then A is in $\mu(\alpha_i)$ for some $i \geqq 1$. Let $G'_A = (V'_A, \Sigma_1, P'_A, A)$, where $P'_A = P \cap \mu(\mathcal{P}_i)$ and V'_A is $\{A\} \cup \Sigma_1$ together with all the symbols appearing in productions of P'_A. Obviously G'_A is in $\mathcal{G}(F_i)$. By induction, there exists a grammar $G_A = (V_A, \Sigma_1, P_A, A')$ in $\mathcal{G}(F_i)$ such that $L(G_A) = L(G'_A)$ and $\Phi_{G_A}(x) \leqq \max\{1, |x|/(2(n+1))\} = \max\{\phi_{G_{F_i}}, |x|/(2(n+1))\}$ for all x in $L(G_A)$. There is no loss of generality in assuming that $A' = A$, A does not occur on the right side of any production in P_A, and each symbol in $V_A - (\Sigma_1 \cup \{A\})$ is a new symbol in $V - \Sigma$.

Let s be the number of elements in $\mu(\sigma)$ and r the maximum number of variables (not necessarily distinct) on the right side of any production in $\mathcal{P}$. Let $G' = (V'_1, \Sigma_1, P', S)$, where V'_1 is $\Sigma_1 \cup \{S\}$ together with all symbols in P', and P' consists of P and the following productions:

[14] A grammar form F is said to be *nontrivial* if $L(G_F)$ is infinite.

4. Foir every variable A in $V_1-\Sigma_1$ and w in Σ_1^*, with $|w|\leqq 2(n+1)(r+2)$ and $A \overset{*}{\underset{G}{\Longrightarrow}} w$, let $A \to w$ be in P'.

[Part 4 speeds up the G-derivation of short terminal words from variables.]

5. Suppose A, B are in $\mu(\sigma)$ and $A \overset{\leqq(n+1)s}{\underset{G}{\Longrightarrow}} w_1Bw_2$ for some w_1, w_2 in V_1^*. Let $B_1, \cdots, B_q$ be the variables (not necessarily distinct) of G appearing in w_1w_2 in order from left to right, and for each i suppose $B_i \overset{*}{\underset{G}{\Longrightarrow}} u_i$, where u_i is in Σ_1^* and $|u_i|\leqq 2(n+1)(r+2)$. Let $\bar{w}_1$ and $\bar{w}_2$ be the words obtained by replacing each B_i by u_i in w_1 and w_2 respectively. Then $A \to \bar{w}_1B\bar{w}_2$ is in P'.

[Part 5 speeds up G-derivations of the form $A \overset{*}{\underset{G}{\Longrightarrow}} w_1Bw_2 \overset{*}{\underset{G}{\Longrightarrow}} \bar{w}_1B\bar{w}_2$, A and B corresponding to σ, in which $A \overset{*}{\underset{G}{\Longrightarrow}} w_1Bw_2$ does not have too many steps and each variable of w_1w_2 has a G-derivation of a short terminal word.]

6. For each A in $V_1-(\Sigma_1\cup\mu(\sigma))$ let each production of P_A be in P'.

[Part 6 speeds up G-derivations of long words from variables not corresponding to σ.]

It is readily seen that G' is in $\mathscr{G}(F)$ and that $L(G')=L$. To show that F is minimal, by Lemma 2.1 it suffices to show that $\Phi_{G'}(x)\leqq|x|/n$ for all words x in L such that $|x|/(n+1)+4+2r\leqq|x|/n$. [For the number of words x in L such that $|x|/(n+1)+4+2r>|x|/n$ is finite.] Thus consider any word x in L such that $|x|/(n+1)+4+2r\leqq|x|/n$. There exists a minimal G-derivation δ of x so that productions (possibly none) of the type $A\to vBw$, A, B in $\mu(\sigma)$ and v, w in $(V_1-\mu(\sigma))^*$ are applied first; then exactly one production of the type $A\to v$, A in $\mu(\sigma)$ and v in $(V_1-\mu(\sigma))^*$, and finally productions involving only variables in $V_1-(\Sigma_1\cup\mu(\sigma))$. Three cases arise.[15]

(α) There are $(n+1)s$ consecutive productions $p_1, \cdots, p_{(n+1)s}$ in δ with a variable in $\mu(\sigma)$ on both sides such that for each variable A in $V-\mu(\sigma)$ generated by each of the p_i, the subword of x generated by A has length at most $2(n+1)(r+2)$. Then by part 5, there is a production π in G' which simulates the sequence $p_1, \cdots, p_{(n+1)s}$ as well as the derivations into subwords of x by every variable not in $\mu(\sigma)$ produced by each of the p_i. Since δ is minimal, each time a variable in $\mu(\sigma)$ is repeated, either a terminal symbol or a variable deriving (in δ) a terminal symbol is produced. Since there are only s distinct variables in $\mu(\sigma)$ and the production π simulates the effect of all the productions alluded to above, π deposits at least $n+1$ symbols.

(β) Fewer than $(n+1)s$ consecutive productions as in (α) occur, followed by a production p: $A\to vBw$ such that A, B are in $\mu(\sigma)$, v, w are in $(V_1-\mu(\sigma))^*$, and $|u|>2(n+1)(r+2)$ for some subword u of x generated by some variable in vw. As in (α), there is a production π of G' which simulates the sequence of productions preceding p, plus the derivation of variables not in $\mu(\sigma)$ into subwords of x. Let

[15] We implicitly use the fact that since $\mathscr{L}(F)$ is not the family of all context-free languages, it follows from Theorem 2.2 of [2] that there are no words u_1, u_2, u_3 in $\mathscr{V}^*$ such that $\sigma \overset{*}{\underset{G_F}{\Longrightarrow}} u_1\sigma u_2\sigma u_3$.

$B_1, \cdots, B_q$ be the occurrences of the variables in vw, in order, and let $x_1, \cdots, x_q$ be the corresponding subwords of x which the B_i generate. Note that $q \leqq r$. For each i such that $|x_i| \leqq 2(n+1)(r+2)$, there is a production $B_i \to x_i$ in P', by part 4. For each i such that $|x_i| > 2(n+1)(r+2)$, there is a sequence of at most $|x_i|/(2(n+1))$ productions in P' which converts B_i into x_i by part 6. Thus, the simulation of the sequence of productions, plus p, plus the derivation into subwords of x of all generated variables not in $\mu(\sigma)$, requires at most 1 (for the initial sequence of productions)$+1$ (for p)$+r$ (for expanding all B_i such that $|x_i| \leqq 2(n+1)(r+2)$) $+\sum_i^q |x_i|/(2(n+1))$ (for i such that $|x_i| > 2(n+1)(r+2)$) productions of G', i.e., at most $2+r+\sum_{i=1}^q |x_i|/(2(n+1))$ productions. Since $\sum_{i=1}^q |x_i| > 2(n+1)(r+2)$, the number of productions is at most $\sum_{i=1}^q |x_i|/(n+1)$. Since at least $\sum_{i=1}^q |x_i|$ terminals are deposited by these productions, it follows that at least $n+1$ terminal symbols are deposited for each production of G' used (although each production may not itself deposit $n+1$ terminals.)

(γ) Fewer than $(n+1)s$ consecutive productions as in (α) occur, followed by a production $p: A \to v$, where A is in $\mu(\sigma)$ and v is in $(V_1 - \mu(\sigma))^*$. As in (β), one production of G' simulates the sequence of productions preceding p, plus the derivation of the variables not in $\mu(\sigma)$ into subwords of x. Let $B_1, \cdots, B_q$ be the sequence of occurrences of variables, in order, and $x_1, \cdots, x_q$ the corresponding subwords of x. As in (β), the total number of productions needed to simulate the initial productions, plus p, plus the derivation of all generated variables into subwords of x, is at most $2+r+\sum_{i=1}^q |x_i|/(2(n+1))$. If $\sum_{i=1}^q |x_i| > 2(n+1)(r+2)$, then as in case (β), $n+1$ terminals are deposited for every production of G' used.

We now apply (α) and (β) to δ in the obvious way until (γ) arises. The number of applications of productions in G' is at most $\sum_{i=1}^q |x_i|/(n+1)$, where the x_i are the subwords of x derived from the variables not in $\mu(\sigma)$. Suppose that $\sum_{i=1}^q |x_i| > 2(n+1)(r+2)$ for the x_i arising in (γ). Then $\Phi_{G'}(x) \leqq |x|/(n+1) < |x|/n$. Suppose that $\sum_{i=1}^q |x_i| \leqq 2(n+1)(r+2)$ for the x_i arising in (γ). Then

$$\begin{aligned}\Phi_{G'}(x) &\leqq \frac{|x|}{n+1} \quad \text{(from } (\alpha) \text{ and } (\beta)\text{)}\\ &\quad +2+r+\frac{\sum_{i=1}^q |x_i|}{2(n+1)} \quad \text{(for the } x_i \text{ arising in } (\gamma)\text{)}\\ &\leqq \frac{|x|}{n+1}+2+r+2+r\\ &= \frac{|x|}{n+1}+4+2r\\ &\leqq \frac{|x|}{n}, \quad \text{by hypothesis on } x.\end{aligned}$$

Hence the result.

The basic question we are interested in is whether some grammar forms are "more efficient" than others, either for representing particular languages or for their entire language families. By our main result, this question has a negative answer if our measure of complexity is derivation length. (That is, each grammar form has the power of expressing each language in its grammatical family as efficiently as liked.) Of course, derivation length is not the only criterion for judging the efficiency of a grammar. Other possibilities are "size" measures (e.g., total number of symbols needed to represent the grammar or number of productions in the grammar), as studied, for example, in [7], [8]. The reader will note that the cost of the speedup using our construction is a large increase in the size of the grammar: if $S(n)$ is the size of the grammar constructed to accomplish speedup of a language by constant n, then $S(n)$ can be roughly equal to $S(1)^{kn}$, where k is a constant depending on the form and language. It remains to study comparative efficiency of forms with respect to size measures, and to examine trade-offs between the two types of measures.

Appendix. We now establish Lemma 2.5. Suppose there are at least $l/7$ leaf nodes m with the property that

(A.1) for some leaf $m' \neq m$, m and m' are daughters of the same father. Then there are at least $l/14$ pairs of distinct leaf nodes, the two nodes in each pair having a common father. Thus there are at least $l/14$ such fathers, and since T has at least two internal nodes, $l/28$ fathers of such fathers. Each such father of a father is an internal node with at least one daughter an internal node. By (a) of the hypothesis, the sum of the weights below each such father of a father is at least k. Thus $\sum_{m \text{ a leaf}} \omega(m) > k(l/28)$.

Suppose there are $e < l/7$ leaf nodes m satisfying (A.1). Call an internal node both of whose daughters are internal nodes a *branch* node. Let i be the number of internal nodes and b the number of *branch* nodes. Since T is a binary tree, it is readily seen that $l = i + 1$ and $e/2 = b + 1$. Now remove all branch nodes and their incident edges from the tree T, obtaining a graph with g connected components, $\Gamma_1, \cdots, \Gamma_g$. Clearly

$$g \leqq 2b + 1 = e - 1 < \frac{l}{7}.$$

Let n be the number of original internal nodes in all the g components. Then

$$n = i - b = l - \frac{e}{2} > \frac{13}{14} l.$$

Also observe that each component is one of the following two types:

(A.2) For some $r \geqq 1$, the nodes are $\{n_i, m_i | 1 \leqq i \leqq r\} \cup \{m'_r\}$, where for each i, $1 \leqq i \leqq r$, m_i is a daughter of n_i, and for each i, $1 \leqq i \leqq r - 1$, n_{i+1} is a daughter of n_i. Also, m'_r is a daughter of n_r. In addition, $n_1, \cdots, n_r$ are internal nodes of T but not branch nodes, and each m_i, m'_r is a leaf node of T.

(A.3) For some $r \geqq 1$, the nodes are $\{n_i, m_i | 1 \leqq i \leqq r\}$, where for each i, $1 \leqq i \leqq r$, m_i is a daughter of n_i, and for each i, $1 \leqq i \leqq r - 1$, n_{i+1} is a daughter of n_i. In addition, $n_1, \cdots, n_r$ are internal nodes of T but not branch nodes, and each m_i is a leaf node of T.

Define a 6-chain as a 6-tuple $(n_1, \cdots, n_6)$ in which each n_i is an internal, nonbranch node of T, and n_j is a daughter of n_{j-1} for all $j \geqq 2$. All 6 nodes of a 6-chain are in some common component since no n_i is a branch node. For each i,

$1 \leqq i \leqq g$, let a_i be the number of original internal nodes in Γ_i. Then there is a set of [16]

$$\begin{aligned}\sum_{i=1}^{g} \lfloor a_i/6 \rfloor &\geqq \frac{1}{6}\sum_{1}^{g} a_i - \frac{5}{6}g \\ &= \frac{1}{6}(n-5g) \\ &> \frac{1}{6}\left(\frac{13}{14}l - \frac{5}{7}l\right) \\ &= l/28\end{aligned}$$

pairwise disjoint 6-chains (i.e., 6-chains having no elements in common). Since every internal node not a branch node has a daughter which is a leaf, it follows from (b) of the hypothesis that the sum of the weights of the leaf nodes which are daughters of nodes in a given 6-chain is at least k. Since there are at least $l/28$ disjoint 6-chains, the sum of the weights of the leaf nodes in T is at least $kl/28$, completing the proof of Lemma 2.5.

REFERENCES

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

[2] A. B. Cremers and S. Ginsburg, *Context-free grammar forms*, Ibid., 11 (1975), pp. 86–117.

[3] A. B. Cremers, S. Ginsburg and E. H. Spanner, *The structure of context-free grammatical families*, submitted.

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

[5] S. Ginsburg and E. H. Spanier, *Substitution of context-free grammar forms*, Acta Math., 5 (1975), pp. 377–386.

[6] A. Gladkii, *On the complexity of derivations in phase-structure grammars*, Algebra i Logika Sem., 3 (1964), pp. 29–44.

[7] J. Gruska, *On the size of context-free grammars*, Kybernetica (Prague), 8 (1972), pp..213–218.

[8] A. R. Meyer and M. J. Fischer, *Economy of description by automata, grammars and formal systems*, Twelfth Annual Symp. on Switching and Automata Theory, 1971, pp. 188–191.

[9] S. Scheinberg, *Note on the Boolean properties of context-free languages*, Information and Control, 3 (1960), pp. 372–375.

[16] For each number t, $\lfloor t \rfloor$ is the largest integer less than or equal to t.

SIAM J. COMPUT.
Vol. 6, No. 1, March 1977

TWO ALGORITHMS FOR GENERATING WEIGHTED SPANNING TREES IN ORDER*

HAROLD N. GABOW†

Abstract. Two algorithms for generating spanning trees of a connected graph in order of increasing weight are presented. The first generates the K smallest weight trees, where K can be specified in advance or during execution of the algorithm. The run time is $O(KE\alpha(E, V)+E \log E)$ and the space is $O(K+E)$; here V is the number of vertices, E is the number of edges, and α is Tarjan's inverse of Ackermann's function and is very slow-growing. The algorithm uses a minimum weight spanning tree as a "reference tree", and exchanges edges to derive other trees. The second algorithm, a modification of the first, generates all spanning trees of the graph, in order. If N is the number of spanning trees, the time is $O(NE)$ and the space is $O(N+E)$.

Key words. weighted spanning trees, edge exchange, set merging, branch and bound

1. Introduction. In many practical situations, one wants to generate the spanning trees of a graph in order of increasing weight. For example, consider this electrical wiring problem: n pins must be wired together with as little wire as possible; further, the wiring must satisfy complicated and diverse constraints, e.g., at most k wires can meet any pin, no two wires can be too close, etc. One way to solve the problem is to generate spanning trees in order of increasing length (weight) until a tree satisfying the constraints is found.

This paper presents an algorithm for generating the K smallest spanning trees of a connected graph, in order of increasing weight. K may be known in advance or specified as trees are generated. The algorithm requires time $O(KE\alpha(E, V)+E \log E)$ and space $O(K+E)$. Here V is the number of vertices, E is the number of edges, and α is Tarjan's inverse of Ackermann's function and is very slow-growing. A previously known algorithm for this problem [2] has slower run time, $O(KV^2+KV \log K)$. The algorithm works by using a minimum weight spanning tree as a "reference tree" [6], and exchanging edges to find other trees. It uses fast set merging algorithms [10], and a bounding technique for partitioning the solution space [5].

The algorithm is modified for the problem of generating all spanning trees of a connected graph, in order of increasing weight. The run time is $O(NE)$ and the space is $O(N+E)$, where N is the number of spanning trees of the graph. Efficient algorithms for generating all spanning trees of a graph without weights [7], [8] can be applied to this problem. The best known algorithm gives time bound $O(NE)$ and space bound $O(NV+E)$. Since N can be large, storage may be a limiting factor. So the improvement in space is significant.

Section 2 gives definitions from graph theory. Section 3 gives a result on edge exchanges that is the basis of the algorithm. Section 4 presents the algorithm, and analyzes time and space requirements. Section 5 describes the modified algorithm. Section 6 summarizes computational experience.

* Received by the editors September 12, 1975, and in revised form February 9, 1976.

† Department of Computer Science, University of Colorado, Boulder, Colorado 80309. This work was supported in part by the National Science Foundation under Grant GJ 36461.

2. Definitions. A *graph* G consists of a finite set of V vertices and a finite set of E edges. An *edge* is an (unordered) set of two distinct vertices. The edge containing vertices v and w is denoted (v, w); it *joins* v and w.

A *subgraph* of G is a graph whose vertices and edges are in G. If H is a collection of edges on the vertices of G, graph $G–H$ consists of the vertices of G and all edges in G except H; graph $G \cup H$ consists of all vertices of G and all edges in G or H.

A *path from* v_1 *to* v_n is a sequence of edges $(v_1, v_2), (v_2, v_3), \cdots, (v_{n-1}, v_n)$. If all vertices v_i, $1 \leqq i \leqq n$, are distinct, the path is *simple*. A graph is *connected* if there is a path between any two distinct vertices. A *bridge* is an edge (v, w) that is in every path from v to w.

A *tree* is a connected graph, where any two distinct vertices are joined by a unique simple path. A *spanning tree* of G is a tree that is a subgraph containing all vertices of G.

A *rooted tree* is a tree T with one vertex r chosen as the *root*. Let v be a vertex in T, and let $v = v_0, v_1, \cdots, v_n = r$ be the sequence of vertices in the simple path from v to r. Any vertex v_i, for $0 \leqq i \leqq n$, is an *ancestor* of v; v_1 is the *father* of v; v is a *son* of v_1; the *depth* of v is n. The *first common ancestor* of two vertices v and w is the ancestor of both v and w that has greatest depth. Two distinct sons of the same vertex are *brothers*. In an *oriented tree*, the sons of a vertex are ordered from left to right.

In a *weighted graph*, every edge e has a number, $w(e)$, called its *weight*. If H is a set of edges (such as a spanning tree), the *weight of* H is $\sum_{e \in H} (e)$.

3. *T*-exchanges. This section describes how spanning trees can be derived by exchanging edges. T-exchanges are defined and a useful property is proved.

Let T be a spanning tree of graph G. A *T-exchange* is a pair of edges e, f where $e \in T$, $f \notin T$, and $T-e \cup f$ is a spanning tree. The *weight of exchange e, f* is $w(f)-w(e)$. So the weight of tree $T-e \cup f$ is the weight of tree T plus the weight of exchange e, f.

A T-exchange can be used to derive one minimum weight spanning tree from another, as shown below.

LEMMA 1. *A spanning tree T has minimum weight if and only if no T-exchange has negative weight.*

Proof. The necessity of this condition is obvious. Sufficiency is proved in [4]. □

THEOREM 2. *Let T be a minimum weight spanning tree of G, and let e be an edge in T. Let e, f be a T-exchange having the smallest weight of all T-exchanges e, f'. Then $T-e \cup f$ is a minimum weight spanning tree of graph $G-e$.*

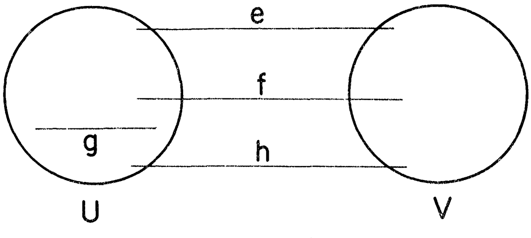


FIG. 1. *Edges in Theorem* 2

Proof. Let $S=T-e\cup f$. Suppose S does not have minimum weight. By Lemma 1, there is an S-exchange g, h having negative weight. We derive a contradiction below.

We first show edges e, f, g, h are situated as in Fig. 1. Let $T-e$ consist of the two trees U, V. Edge e joins U and V. Edge f also joins U and V, since e, f is a T-exchange.

Edge h also joins U and V. For if not, assume without loss of generality that h joins two vertices in U. Since g, h is an S-exchange, g is in tree U. Thus g, h is a T-exchange. But g, h has negative weight, and T is a minimum weight spanning tree. This contradiction proves h joins U and V.

Now we show edge $g \in U\cup V$. Since e, h is a T-exchange, its weight is no smaller than that of e, f. Thus

$$w(f) \leqq w(h) < w(g). \tag{1}$$

So $g \neq f$, and $g \in S-f=U\cup V$.

Assume without loss of generality that $g \in U$. This gives Fig. 1.

Now let $U-g$ consist of the two trees W, X. Edge e is incident to one of these trees, say W. Since $T-e-g\cup f\cup h$ is a spanning tree, either f or h is incident to X. So either g, f or g, h is a T-exchange. But (1) implies both of these exchanges have negative weight, a contradiction. This contradiction shows the original assumption is false. Thus S has minimum weight. □

Now let T be a minimum weight spanning tree. The theorem shows a spanning tree with the next smallest weight is $T-e\cup f$, where e, f is a minimum weight T-exchange. This observation is the basis of the algorithm.

4. Algorithm for *K* spanning trees. The algorithm for generating K spanning trees consists of procedures EX, GEN, and a main procedure GENK. This section describes these procedures in turn.

EX finds a minimum weight T-exchange e, f subject to certain constraints, *IN*, *OUT*. We describe EX, first assuming there are no constraints and then incorporating the constraints.

For each edge $g \in T$, EX finds a minimum weight T-exchange g, h; it sets e, f to the smallest of these exchanges. During the execution of EX, we call an edge $g \in T$ *eligible* if a minimum weight exchange g, h has not been found.

EX works as follows. Originally all edges in T are eligible. A list L contains all edges $h_1, h_2, \cdots, h_E$ of G, sorted in order of increasing weight. EX begins by finding every T-exchange g, h_1. This is a minimum exchange for g, since h_1 has the smallest weight possible. Now all edges g become ineligible.

Edge h_2 is processed similarly. EX finds every T-exchange g, h_2, where g is eligible. This is a minimum exchange for g. The edges g become ineligible.

The procedure is repeated for every edge h_k in L. The smallest exchange found is the minimum T-exchange e, f.

The tree T is represented by a father array F. Vertex 1 is the root; for any vertex $v \neq 1$, $F(v)$ is the father of v. Call a vertex $v \neq 1$ *eligible* if edge $(v, F(v))$ is eligible, i.e., a minimum exchange for $(v, F(v))$ has not been found; call vertex 1 eligible too.

Now suppose EX examines the edge $h_k=(x, y)$, as shown in Fig. 2. Vertex a is the first common ancestor of x and y that is eligible; vertices x_i, for $1 \leqq i \leqq n$, are

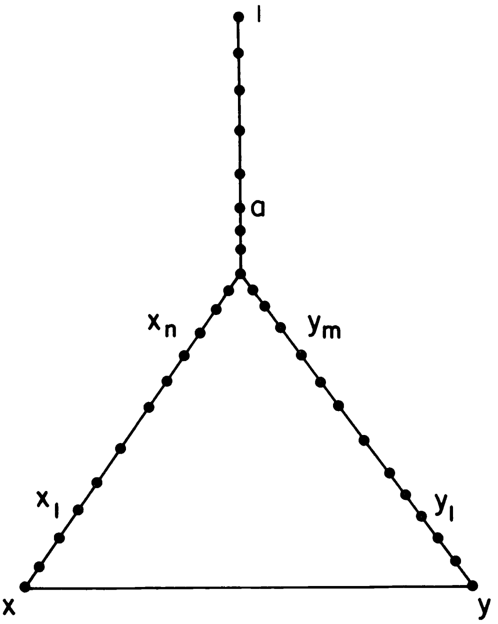


FIG. 2. *Exchanging* (x, y) *into* T

the other eligible vertices on the path from x to a; vertices y_j, for $1 \leqq j \leqq m$, are defined similarly. Edge h_k gives a minimum exchange for each edge $(x_i, F(x_i))$, $(y_j, F(y_j))$, and no others.

EX finds the vertices x_i, y_j, by using set merging techniques. A family of sets partitions the vertices of G. Each set contains a unique eligible vertex, which is used as the name of the set. Vertex w is in the set named v when v is the first eligible ancestor of w. (Since 1 is eligible, v exists for any w.) EX uses two procedures to manipulate these sets: FIND(v) computes the name of the set containing vertex v; UNION (v, w, x) combines sets named v and w into a new set named x (destroying sets v and w).

EX computes the vertices x_i using the equations $x_1 = \text{FIND}(x)$, $x_{i+1} = \text{FIND}(F(x_i))$. Vertices y_j are computed similarly. Vertex a is the first common value, $a = x_{n+1} = y_{m+1}$. In this manner, EX finds all minimum exchanges for edge h_k.

Now we discuss the constraint lists *IN* and *OUT*. These lists prevent trees from being generated twice; their construction is described fully below. *IN* is a list of edges in T that must remain in T; *OUT* is a list of edges out of T that must remain out of T. Thus EX must find a T-exchange e, f, such that $e \in T - IN$, $f \in G - OUT$, and e, f has the smallest weight possible.

To do this, EX begins by making all *IN* edges (x, y) ineligible. This is done by placing x and y in the same set (since they have the same first eligible ancestor). Also, EX marks all *OUT* edges in the list L, so they are not considered for exchanges. These edges are later unmarked, for subsequent calls to EX. In this manner, the constraints are handled.

The final array used is W, which specifies the weight $W(e)$ of each edge e.

The procedure below sets global variables e, f and r, so e, f is a minimum weight exchange subject to constraints IN, OUT, and r is the weight of e, f.

```
                        procedure EX($F$, $IN$, $OUT$); begin
 1. initialize:         $r \leftarrow \infty$; make each vertex $v$ the only element in a set named
                        $v$;
 2. IN edges:           for edge $(x, y)$ in $IN$ do begin wlog assume $F(x) = y$;
 3.                         $y \leftarrow$ FIND $(y)$; UNION $(x, y, y)$;
                            end;
 4. OUT edges:          for edge $(x, y)$ in $OUT$ do mark $(x, y)$ in $L$;
 5. L edges:            for edge $(x, y)$ in $L$ do comment edges in $L$ are in increasing
                        weight order;
 6.                         if $(x, y)$ is marked then unmark $(x, y)$
 7.                         else if $F(x) \neq y$ and $F(y) \neq x$ then begin
 8. find exchanges:     let $a$ be the first eligible common ancestor of $x$ and $y$;
 9.                     for $v1 \leftarrow x, y$ do begin $v \leftarrow$ FIND$(v1)$;
10.                         while $v \neq a$ do begin
11. check exchange:             $r1 \leftarrow W(x, y) - W(v, F(v))$; comment $r1$ is the
                                exchange's weight;
12.                             if $r1 < r$ then begin $r \leftarrow r1$; $e \leftarrow (v, F(v))$; $f \leftarrow (x, y)$; end;
13. advance:                    $u \leftarrow$ FIND$(F(v))$; UNION$(v, u, u)$; $v \leftarrow u$;
                        end end end end EX;
```

Table 1 illustrates EX. Input parameters describe T_1 for the graph of Fig. 3. The exchanges found by EX in step 8 are listed, along with the output values describing T_2.

TABLE 1

					EX *processes* T_1			
$F(2)$	$F(3)$	$F(4)$	IN	OUT		r	e	f
1	2	3	—	—	(2, 1), (3, 1); (3, 2), (3, 1); (4, 3), (4, 1)	0	(3, 2)	(3, 1)
		input			exchanges		output	

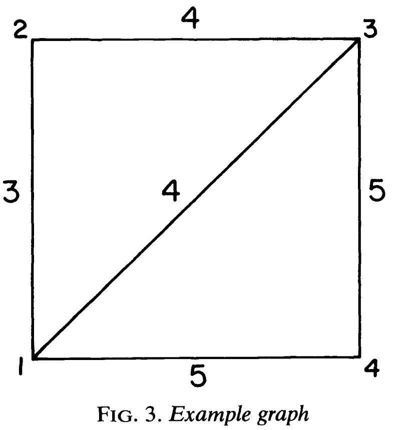


FIG. 3. *Example graph*

LEMMA 3. *Let T be a minimum weight spanning tree subject to constraints $IN \subseteq T \subseteq G - OUT$. Then* EX, *called with a father array for T, sets e, f so $T - e \cup f$ is the next smallest spanning tree subject to the same constraints.*

Proof. Theorem 2 shows $T - e \cup f$ has the desired property if e, f is a minimum weight T-exchange subject to constraints $e \notin IN$, $f \notin OUT$. The remarks preceding EX show EX finds such an exchange. □

LEMMA 4. EX *runs in time $O(E\alpha(E, V))$.*

Proof. First we bound the time for set operations. Lines 3 and 13 together do at most one UNION for each vertex. So they do $O(V)$ UNIONs, and also $O(V)$ FINDs. Line 9 does at most two FINDs for each edge in $G - T$. Thus, a total of $O(V)$ UNIONs and $O(E)$ FINDs are done. If fast set merging algorithms are used, the total time is $O(E\alpha(E, V))$ [10].

Line 4 can be done in time $O(E)$. For OUT contains at most E edges e; e can be marked (in L) in time $O(1)$, if OUT contains a pointer to e in L.

Next consider line 8. Vertex a can actually be computed as exchanges for vertices v are found (lines 9–13). To do this, the paths from x to a and y to a are processed simultaneously. Referring to Fig. 2, suppose x_i and y_j have been found, for some $i \leqq n$, $j \leqq m$; if $x_i \neq y_j$, the vertex with greater depth precedes a, and so is x_{i+1} or y_{j+1}. In this manner, the exchanges for vertices v, and the vertex a, are found.

To do this efficiently, we use an array D, that gives the depth $D(v)$ of vertex v in T. D can be computed from F in time $O(V)$. Thus, line 8 requires only $O(V)$ extra time.

The remaining lines in EX require time $O(E + V)$. This gives the desired time bound. □

Procedures GEN and GENK call EX to generate spanning trees in correct order. A technique resembling branch and bound, described by Lawler [5], is used.

Let T_i, for $1 \leqq i \leqq N$, denote the spanning trees of G, in order of increasing weight. Suppose the algorithm has output the first $j - 1$ trees, $j > 1$. The remaining trees have been partitioned into $j - 1$ disjoint sets of the form

$$P_i^{j-1} = \{T_k | k > j - 1; e_1, e_2, \cdots, e_r \in T_k; e_{r+1}, \cdots, e_s \notin T_k\},$$

for $1 \leqq i < j$ (here r and s vary over sets). Tree T_i satisfies all conditions of this set except the first, $k > j - 1$. The smallest tree in this set is known; it is the result of a T_i-exchange e, f.

Procedure GEN finds the next smallest tree T_j from among the $j - 1$ smallest trees in these sets. Suppose T_j is in the above set P_i^{j-1}, and $T_j = T_i - e \cup f$. GEN computes T_j and outputs it. Then a new partition P_k^j, $1 \leqq k \leqq j$, is formed, by subdividing P_i^{j-1}. For all $k \neq i$ in $1 \leqq k < j$, set $P_k^j = P_k^{j-1}$; further,

$$P_i^j = \{T_k | k > j; e_1, \cdots, e_r, e \in T_k; e_{r+1}, \cdots, e_s \notin T_k\},$$

$$P_j^j = \{T_k | k > j; e_1, \cdots, e_r \in T_k; e_{r+1}, \cdots, e_s, e \notin T_k\}.$$

T_i satisfies all conditions of P_i^j except the first. GEN finds the smallest tree in P_i^j by executing EX on T_i, with edges $e_1, \cdots, e_r$, e in IN, and $e_{r+1}, \cdots, e_s$ in OUT.

(Lemma 3 shows EX works correctly.) P_j^j is processed similarly, using tree T_j. In this way, the partition is updated. Now T_{j+1} can be found.

The algorithm uses a list P to represent the partition. A set P_i^j is represented by a tuple (t, e, f, F, IN, OUT) in P. Here t is the weight of the smallest tree in P_i^j; F is the father array for T_i; the smallest tree in P_i^j results from the T_i-exchange e, f; IN is the list of edges that are in all trees of P_i^j; OUT is the list of edges that are out of all trees of P_i^j.

Procedure GEN outputs the next smallest tree T_j from set P_i^{j-1}, and puts the new sets P_i^j, P_j^j in the partition.

```
                  procedure GEN; begin
1. find $P_i^{j-1}$:   remove the tuple $(t, e, f, F, IN, OUT)$ with smallest weight $t$ from
                  $P$;
2.                if $t = \infty$ then stop; comment all spanning trees have been output,
3. output $T_j$:     $Fj \leftarrow F$; modify $Fj$ so edge $f$ replaces $e$; output $(Fj)$;
4.                $ti \leftarrow t - W(f) + W(e)$; comment $ti$ is the weight of $T_i$;
5.                form list $INi$ by adding $e$ to $IN$; form list $OUTj$ by adding $e$ to
                  $OUT$;
6. form $P_i^j$:     EX$(F, INi, OUT)$; add $(ti + r, e, f, F, INi, OUT)$ to $P$; comment
                  EX sets $e, f, r$ for the minimum weight exchange;
7. form $P_j^j$:     EX$(F_j, IN, OUTj)$; add $(t + r, e, f, Fj, IN, OUTj)$ to $P$;
                  end GEN;
```

Table 2 illustrates GEN, by showing the partition after tree T_2 for Fig. 3 is output.

TABLE 2

Partition P_i^2

	t	e	f	$F(2)$	$F(3)$	$F(4)$	IN	OUT
P_1^2	12	(4, 3)	(4, 1)	1	2	3	(3, 2)	—
P_2^2	12	(4, 3)	(4, 1)	1	1	3	—	(3, 2)

LEMMA 5. *Let* GEN *be called, with P containing tuples for the sets P_k^{j-1}, $1 \leqq k < j$. Then* GEN *outputs T_j, and changes P to tuples for the sets P_k^j, $1 \leqq k \leqq j$.*

Proof. The Lemma follows from the remarks preceding GEN. □

LEMMA 6. GEN *runs in time $O(E\alpha(E, V))$.*

Proof. Lines 1, 6 and 7 of procedure GEN remove and add tuples to P. These operations can be done in time $O(E)$. For this, we use a heap [1]. The tuples in P are numbered, by assigning i to P_i^j. An entry in the heap is a number i; entries are ordered on the weight t of the corresponding tuple. The heap contains at most K entries. So an entry can be removed or added in time $O(\log K)$. Since $K < 2^E$, this is $O(E)$.

Line 3 requires time $O(V)$ to derive Fj. Referring to Fig. 2, let $e = (x_i, F(x_i))$, $f = (x, y)$. To derive Fj, array F need only be changed on the path from x to x_i.

The rest of the time is dominated by the two calls to EX. This gives the desired bound. □

The main procedure GENK finds a minimum spanning tree, and calls GEN for the next $(K-1)$ trees.

```
                     procedure GENK($K$); begin
1.                   make $L$ a list of the edges of $G$, sorted in order of increasing
                     weight;
2. find $T_1$:       make $F$ a father array for a minimum weight spanning tree,
                     having weight $t$; output($F$);
3. form $P_1^1$:     EX ($F, \phi, \phi$); make $(t+r, e, f, F, \phi, \phi)$ the only tuple in $P$;
4. generate $T_j$:   for $j \leftarrow 2$ to $K$ do GEN;
                     end GENK;
```

Table 3 shows the partition after all minimum weight spanning trees for Fig. 3 are output.

TABLE 3

Partition P_i^4

	t	e	f	$F(2)$	$F(3)$	$F(4)$	IN	OUT
P_1^4	13	(2, 1)	(3, 1)	1	2	3	(3, 2), (4, 3)	—
P_2^4	13	(3, 1)	(4, 1)	1	1	3	(4, 3)	(3, 2)
P_3^4	13	(2, 1)	(3, 1)	1	2	1	(3, 2)	(4, 3)
P_4^4	∞	—	—	1	1	1	—	(3, 2), (4, 3)

LEMMA 7. GENK *generates the K smallest weight spanning trees in order, in time* $O(KE\alpha(E, V)+E \log E)$.

Proof. The correctness of GENK follows by induction, using Lemma 5. Now we analyze the time. Line 1 requires time $O(E \log E)$ to sort E edges. Line 2 can be done in time $O(E \log E)$, using Kruskal's algorithm [1]. The rest of the time is dominated by $O(K)$ calls to GEN, requiring time $O(KE\alpha(E, V))$ by Lemma 6. □

When K is small, the term $E \log E$ in the run time is significant. It can be reduced to $E \log\log E$. This is done by using a faster minimum spanning tree algorithm [3], [12], and changing EX so the edges of G need not be sorted. For the latter, the techniques developed by Tarjan [11] are used.

Now we examine the space requirements. The dominating requirement is for P. Since a tuple can contain $O(E)$ words, P can use $O(KE)$ words. Below we modify the data structure to reduce this bound, without changing the run time.

LEMMA 8. GENK *uses* $O(K+E)$ *space.*

Proof. We show the structures F, IN, OUT in a tuple can be replaced by 6 words, and still be computed in time $O(E)$. The extra time is spent in line 1 of GEN, and does not change the algorithm's time bound; P is reduced to $O(K)$ words. So this suffices to prove the Lemma.

Suppose the first k trees T_j, $1 \leqq j \leqq k$, have been output. Make these trees the vertices of an oriented tree τ, as follows. Make T_1 the root of τ. For $j > 1$, make T_j

a son of T_i if GEN derives T_j from a T_i-exchange. Let e_j, f_j denote this T_i-exchange. Arrange the sons T_j of T_i so j increases from left to right.

For a set P_j^k, the structures F, IN, and OUT can be derived from τ, as follows. Let A be the path in τ from T_j to the root T_1. Consider these sets:

$$O_1 = \{e_i | T_i \text{ is in } A \text{ and } i > 1\},$$

$$O_2 = \{f_i | T_i \text{ is in } A \text{ and } i > 1\},$$

$$T = (T_1 \cup O_2) - O_1,$$

$$I_1 = \{e_i | T_i \text{ is a left brother of some } T_l \text{ in } A\},$$

$$I_2 = \{e_i | T_i \text{ is a son of } T_j\}.$$

T contains the edges of T_j; the father array F for T_j is easily derived. $I_1 \cup I_2$ contains the edges of IN, and O_1 contains the edges of OUT.

Now for each set P_j^k, replace F, IN, OUT by the 6 words j, i, ej, fj, s, b. Here j is the index in P_j^k; i is the index of the father T_i of T_j; e_j, fj is the exchange that derives T_j from T_i; s is the index of the rightmost son (if any) of T_j; b is the index of the brother (if any) immediately to the left of T_j.

These 6 words are easily computed in GEN. Let the tuple found in line 1 be $(t, e, f, i, il, ei, fi, s, b)$. Then line 6 adds the tuple $(ti + r, e, f, i, il, ei, fi, j, b)$ to P; line 4 saves e, f as ej, fj, and line 8 adds the tuple $(t + r, e, f, j, i, ej, fj, 0, s)$ to P.

Finally, it is easy to see the 5 sets above, and F, IN, OUT, can be constructed (in line 1 of GEN) in time $O(E)$. □

Table 4 shows how the partition in Table 3 is changed using this scheme.

TABLE 4

Changes in P_i^4 using storage reduction scheme

	j	i	ej	fj	s	b
P_1^4	1	—	—	—	3	—
P_2^4	2	1	(3, 2)	(3, 1)	4	—
P_3^4	3	1	(4, 3)	(4, 1)	—	2
P_4^4	4	2	(4, 3)	(4, 1)	—	—

5. Algorithm for all spanning trees. This section describes a modification of the algorithm that generates all spanning trees of a connected graph, in order of increasing weight. The run time is $O(NE)$, a slight improvement over GENK. The three main changes are described below.

The first change eliminates extra calls to EX, by modifying the tuples for sets P_i^j in P. The tuple's minimum T_i-exchange e, f is replaced by X, a list of T_i-exchanges. For each edge $g \in T_i - IN$, X contains g, h, the smallest exchange with $h \in G - OUT$. So e, f is the smallest exchange in X.

EX is modified to generate X. X is initialized to an empty list (in line 1), and a line is added after 12:

```
12.1.   add exchange $(v, F(v))$, $(x, y)$ to $X$;
```

GEN is modified to use X. In line 1, the minimum exchange e, f is found by examining X. In line 6, the call to EX is eliminated; the tuple for P_i^j is formed by rejmoving e, f from X, and computing the new minimum weight, t, using X.

The second change, in the set merging algorithms, speeds up FIND at the expense of UNION. Sets are represented by an array VSET, where VSET$(w) = v$ when v is the name of the set containing w. With this approach, FIND takes time $O(1)$ and UNION takes time $O(V)$.

EX uses these algorithms to manipulate the sets of vertices. Also, lines 2–3 are modified to avoid calls to UNION:

```
2'. IN edges:  sort the edges $(x, y)$ of $IN$, so $F(x) = y$ and the depth of $x$
               increases;
3'.            for edge $(x, y)$ in $IN$ do VSET$(x) \leftarrow$ VSET$(y)$;
```

When (x, y) is processed in line 3′, VSET(y) already has its final value. So lines 2′–3′ initialize the sets correctly.

The third change involves the main procedure. A new main procedure is used to insure all trees are generated:

```
procedure GENA; begin GENK($\infty$) end,
```

Note the algorithm halts in line 2 of GEN.

Further, GENK is modified to reduce the set-up time in lines 1–2. Line 1 is changed:

```
1'. make $B$ a list of the bridges of $G$; make $L$ a list of the edges of $G$-$B$, sorted in
    order of increasing weight;
```

In line 2, the bridges B are placed in the spanning tree, before the minimum spanning tree algorithm is called.

The remaining changes in the algorithm are typographical.

LEMMA 9. GENA *generates all spanning trees in order, in time* $O(NE)$ *and space* $O(N+E)$, *where N is the number of spanning trees.*

Proof. The correctness of GENA follows from Lemma 7 and the discussion above. Now we analyze the time, beginning with GENK.

Let E' be the number of edges that are not bridges. Then lines 1′–2 of GENK use time $O(E+E' \log E')$. For in line 1′, the bridges B can be found in time $O(E)$ [9]; L can be sorted in time $O(E' \log E')$. Line 2, using Kruskal's algorithm with the bridges already in the tree, uses time $O(E' \log E')$.

Now we show $E' \log E'$ is $O(NE)$. It suffices to show $E' < 2N$. Let T be a spanning tree. We can list T-exchanges e, f, so each edge that is not a bridge occurs in the list. Since each exchange represents a distinct tree, $E' < 2N$. Thus, lines 1′–2 use time $O(NE)$.

The rest of GENK is dominated by $O(N)$ calls to GEN. As before, the heap operations in GEN (lines 1, 6, 7) require time $O(NE)$; the rest of GEN is dominated by $O(N)$ calls to EX (line 7).

In EX, line 2′ can be done once in time $O(V)$. First the depth of each vertex in the tree is computed, using F. Then the edges of IN are sorted on depth, using a bucket sort [1] with one bucket for each depth. Thus, line 2′ uses total time $O(NV)$.

Line 13 of EX, using the modified FIND and UNION procedures, is done once in time $O(V)$. It is executed N times, once for each spanning tree. So the total time is $O(NV)$.

The rest of EX uses time $O(NE)$. This gives the desired time bound for GENA.

Now we examine the space. The X lists in P require a total of $O(N)$ words, since each exchange in an X list represents a distinct spanning tree. Using the modification of Lemma 8, the rest of P uses $O(N)$ words. So $O(N+E)$ space is used. □

Minty [7] and Read and Tarjan [8] give an algorithm that generates all spanning trees of a graph without weights. It requires time $O(NE)$ and space $O(E)$, assuming each tree is output as soon as it is generated. We can generate all spanning trees in a weighted graph, by first applying this algorithm, and then sorting the trees in order of increasing weight. This method requires time $O(NE)$. The space is $O(NV+E)$, since each tree must be stored until the sort is done. GENA has the same time bound but uses less space. Since N can be large ($N=V^{V-2}$ in a complete graph), space is likely to be a limiting factor, so this improvement is significant.

6. Computational experience. A version of GENK has been programmed in PASCAL and run on the CDC 6400. The program stores tuples in a hybrid form (using the notation defined above, a tuple is (t, X, i, fj, F, IN)). The time bound for the program is $O(KE\alpha(E, V)+E\log E)$, and the space is $O(KV+E)$.

Experiments were conducted on random connected graphs with random integer edge weights between 50 and 10000. Tables 5–6 show results. The run time is specified by $\bar{T}$, the average time in milliseconds to generate one spanning tree. (Experiments show for a given graph, the time per spanning tree is approximately constant, as expected.) Table 5 shows $\bar{T}$ for graphs that are almost complete; $\bar{T}$ is computed from $K=15$. Table 6 shows $\bar{T}$ for graphs with 60 vertices; again $K=15$. When $\bar{T}$ is plotted against E, the data is almost linear. Least square fits for the two tables have slopes 0.72 and 0.68, respectively. Almost linear performance is expected from the asymptotic time bound, since the factor $\alpha(E, V)$ is constant in this relatively small range.

TABLE 5
Time for generating one tree

V	10	15	25	35	45	55	60
E	44	104	300	595	985	1476	1762
$\bar{T}$	12	27	74	143	236	362	422

TABLE 6
Time for generating one tree, $V = 60$

E	159	336	513	690	867	1044	1221	1398	1575	1757
$\bar{T}$	48	86	124	162	199	237	279	324	365	413

Acknowledgment. The author thanks Professor Donald Johnson of Pennsylvania State University for his inspiring discussions, and Mr. Robert Palasek for programming GENK. He also thanks the referee for his penetrating comments, in particular for indicating how to improve GENK for small K.

REFERENCES

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

[2] P. M. CAMERINI, L. FRATTA AND F. MAFFIOLI, *The K shortest spanning trees of a graph*, Int. Rep. 73-10, IEE-LCE Politecnico di Milano, Italy, 1974.

[3] D. CHERITON AND R. E. TARJAN, *Finding minimum spanning trees*, this Journal, to appear.

[4] N. DEO, *Graph Theory with Applications to Engineering and Computer Science*, Prentice-Hall, Englewood Cliffs, N.J., 1974.

[5] E. L. LAWLER, *A procedure for computing the K best solutions to discrete optimization problems and its application to the shortest path problem*, Management Sci., 18 (1972), pp. 401–405.

[6] W. MAYEDA AND S. SESHU, *Generation of trees without duplications*, IEEE Trans. Circuit Theory, CT-12 (1965), pp. 181–185.

[7] G. J. MINTY, *A simple algorithm for listing all the trees of a graph*, IEEE Trans. Circuit Theory, CT-12 (1965), p. 120.

[8] R. C. READ AND R. E. TARJAN, *Bounds on backtrack algorithms for listing cycles, paths, and spanning trees*, Networks, 5 (1975), no. 3. pp. 237–252.

[9] R. E. TARJAN, *A note on finding the bridges of a graph*, Information Processing Lett., 2 (1974), pp. 160–161.

[10] ———, *Efficiency of a good but not linear set union algorithm*, J. Assoc. Comput. Mach., 22 (1975), pp. 215–225.

[11] ———, *Applications of path compression on balanced trees*, Tech. Rep. STAN-CS-75-512, Comp. Sci. Dept., Stanford Univ., Stanford, Calif., 1975.

[12] A. C. YAO, *An $O(|E| \log\log |V|)$ algorithm for finding minimum spanning trees*, Information Processing Lett., 4 (1975), pp. 21–23.

SIAM J. COMPUT.
Vol. 6, No. 1, March 1977

A LINEAR TREE PARTITIONING ALGORITHM*

SUKHAMAY KUNDU† AND JAYADEV MISRA‡

Abstract. Given a rooted tree with a positive weight associated with every node, a linear algorithm is presented that will partition the tree into a minimum number of subtrees such that the sum of node weights in no subtree exceed a prespecified value k.

Key words. tree, partition

1. Introduction. Let T be a rooted tree with a positive weight associated with every node. A feasible k-partition C of T is a set of edges such that upon removal of these edges from the tree, each of the resulting component *subtrees* has a total node weight (sum of node weights) at most k. The problem studied in this paper is to find a feasible k-partition of minimum cardinality (an optimal k-partition). Note that an optimal k-partition partitions the tree into minimum number of components each of whose total weight is less than or equal to k. We will use "partition" instead of "k-partition" when the context is understood. The weight of each node of T is assumed to be at most k.

A more general problem is when the edges are also weighted, and it is required to find a k-partition that has minimum sum of edge weights. Lukes [5] has given an $O(k^2n)$ algorithm for this general problem, where n = number of nodes in the tree. We present an $O(n)$ algorithm for the special case of unit edge weights. Our algorithm is also valid for weighted edges if the weights satisfy a certain monotone property, to be defined later. Note that for large k, say, order of $\sqrt{n}$, Lukes' algorithm will be less efficient than a linear algorithm.

The tree partition problem arises in partitioning any hierarchical structure into a minimum number of segments when there is a constraint on the size of a segment. For example, distributing a hierarchical data base into a minimum number of pages fits into this model; k denotes the size of a page. Usually the nodes in the data base would be of different sizes reflected by the corresponding weights. Partitioning of logic modules into a minimum number of blocks (when block sizes are fixed) to minimize the total number of interconnections among blocks also fits into this model assuming that the modules are arranged in the form of a tree.

Problems of partitioning arise in several different contexts. For example [3], [4] consider the problem of optimal segmentation of a program into pages so that the average number of interpage branching is minimized under certain assumed probabilities of branching. The problem of optimally partitioning a tree into disjoint *chains* has been considered in [6].

2. Results. Let T_p denote the subtree rooted at node p, $S(p)$ the set of sons of node p, $w(p)$ the weight of node p, and $W(p)$ the sum of node weights in T_p. The following lemma is fundamental to our algorithm.

* Received by the editors September 24, 1975, and in revised form May 6, 1976.

† Computer Science Department, University of Texas at Austin, Austin, Texas. Now at LOGICON Corporation, San Pedro, California 90733.

‡ Computer Science Department, University of Texas at Austin, Austin, Texas. This research was supported in part by the National Science Foundation under Grant DCR75-09842.

LEMMA 1. *Let p be a node in T such that $W(p)>k$ and $W(r)\leqq k, \forall r\in S(p)$. Then there exists an optimal k-partition containing the edge (p, r_0), where*

$$W(r_0)=\max_{r\in S(p)}\{W(r)\}.$$

Proof. Since $W(p)>k$, any feasible partition C necessarily contains an edge from T_p. Let (u, v) be such an edge in C from T_p. If $u\neq p$, then (u, v) is in the subtree $T_r, r\in S(p)$. Clearly, $C'=C-\{(u, v)\}+\{(p, r)\}$ is a feasible k-partition, and is also optimal. We may thus assume that each edge of C from T_p is of the form (p, r). The lemma follows by replacing one of the edges (p, r) in C by (p, r_0).

r_0 is called a *heaviest* son of p. Note that the sum of the node weights in a subtree (not the weight of the node) determines the heaviest son. $T-T_r$ will denote the rooted tree obtained upon deletion of T_r from T.

LEMMA 2. *Let (p, r) be in some optimal partition of T. If C_1, C_2 are optimal partitions of $T-T_r$ and T_r respectively, then $C=C_1+C_2+\{(p, r)\}$ is an optimal partition for T.*

Proof. C is a feasible partition. Let C' be any optimal partition containing (p, r), and let C_1', C_2' denote the set of edges in C' from $T-T_r$ and T_r respectively. Obviously, $|C_1|\leqq|C_1'|$ and $|C_2|\leqq|C_2'|$. Hence $|C|\leqq|C'|$.

Lemmas 1 and 2 lead to the following algorithm. Find a node p such that $W(p)>k$ and $W(r)\leqq k, \forall r\in S(p)$. Let r_0 be a heaviest son of p. Then construct an optimal k-partition C_1 of $T-T_{r_0}$ and let $C=C_1+\{(p, r_0)\}$. (Since $W(r_0)\leqq k$, the optimal partition of T_{r_0} is null.)

A node p as above can be located by proceeding along the tree level by level, beginning at the highest level (distance from root) and going down to the root level (level 1). At any stage of the algorithm, we have a single tree which is modified by deletion of a subtree. We let $W^*(p)$ denote the weight of subtree rooted at node p in the modified tree.

ALGORITHM FOR OPTIMAL k-PARTITION

```
begin
    $C := \emptyset$; assign $W^*(q) := w(q)$ to all leaf nodes in $T$;
    for $i$ := maximum-level-in-$T$ downto 1 do
    begin process $i$th level:
        while there is an unprocessed node $p$ in level $i$ do
        begin process node $p$:
            remove heaviest sons of $p$ one by one from $S(p)$ until
            $W^*(p)=\sum_{q\in S(p)} W^*(q)+w(p)\leqq k$;
            For every such son $r$ removed, add the edge $(p, r)$ to $C$
        end process node
    end process level
end algorithm;
```

A simple technique to process node p is to rank order all the sons of p based on their $W^*(\cdot)$ value. The heaviest sons can then be removed one by one until the total weight is less than or equal to $k-w(p)$. This step however, requires $O(|S(p)|\log|S(p)|)$ time to process node p and hence the overall running time of the algorithm becomes $O(n\log n)$.

The following technique for processing node p was suggested by an anonymous referee, resulting in a reduced running time for this step to $O(|S(p)|)$ only. "Process node p" step essentially partitions $S(p)$ into two subsets $S_L(p)$ and $S_H(p)$ (light and heavy) such that

$$(1) \qquad q \in S_L(p) \quad \text{and} \quad r \in S_H(p) \Rightarrow W^*(q) \leqq W^*(r),$$

$$(2) \qquad \sum_{q \in S_L(p)} W^*(q) + w(p) \leqq k,$$

$$(3) \qquad \sum_{q \in S_L(p)} W^*(q) + w(p) + W^*(r) > k, \quad \forall r \in S_H(p).$$

This partitioning can be performed by successively splitting $S(p)$ using a linear median finding algorithm [1]. First $S(p)$ is split into two parts $S_L(p)$, $S_H(p)$ satisfying (1) and $|S_H(p)| \leqq S_L(p)| \leqq |S_H(p)| + 1$. Then conditions (2), (3) are checked in $O(|S(p)|)$ time. If both conditions hold, we have located the desired partition. If (2) holds but (3) does not hold, then $S_H(p)$ is split into "lower" and "upper" parts and the algorithm is repeated. If (3) holds but (2) does not hold, then $S_L(p)$ is split.

More formally, the following routine returns the set S_H as value given the inputs S and $k - w(p)$. Medianfind-and-halve(S, S_1) returns the "upper" half S_1 of S.

```
split($S$, $k$):
begin
    if $|S| = 1$ then [split := if $W^*(q)$, $q \in S \leqq k$ then $\emptyset$
                                                   else $S$]
        else begin
            medianfind-and-halve($S$, $S_1$);
            $t := \sum_{q \notin S_1} W^*(q)$;
            case $t$ of
                $t = k$: split := $S_1$;
                $t < k$: split := split($S_1$, $k - t$);
                $t > k$: split := split($S - S_1$, $k$) + $S_1$
            end
end split;
```

Medianfind-and-halve takes linear time. Since we examine a set of approximately half the size in every succeeding step, split($S(p)$, k) requires

$$O\left(|S(p)| + \frac{|S(p)|}{2} + \frac{|S(p)|}{4} + \cdots\right) = O(|S(p)|)$$

time. Hence the running time of the optimal k-partition algorithm is $O(n)$. For small values of k, good running time can be obtained by distributing $S(p)$ into k buckets—node q goes to the ith bucket for $W^*(q) = i$, $q \in S(p)$. Then lighter sons' weights are successively added to obtain a value as close as possible, but not exceeding $k - w(p)$.

Remarks. The algorithm presented above can be applied to a few other similar problems. First, assume that the edges are also weighted and the weights satisfy the following *monotone* property: on each path from the root to a node of the tree, the edges closer to the root have smaller weights than those that are further away, and all edges directed away from a node have equal weights. To obtain a k-partition with minimum sum of edge weights, the partition algorithm can be used successfully since at each iteration step an edge is added to C which is as close to the root as possible.

Next, consider the partition problem where it is required that each rooted subtree of the partition be a simple chain with total weight not exceeding k. We process the nodes p from higher to lower levels as before. In the step "process node p," we remove all but the lightest son of p, i.e., we find $r \in S(p)$, where

$$W^*(r) = \min_{q \in S[p]} \{W^*(q)\}$$

and add every (p, q), $q \neq r$, $q \in S(p)$ to C. If furthermore $w(p) + W^*(r) > k$, then (p, r) is also added to C. The running time of this algorithm is also $O(n)$.

Acknowledgment. The authors are indebted to an anonymous referee who suggested the use of median finding algorithm in processing a node. This has lowered the running time to $O(n)$ from the previous bound of $O(n \log n)$.

REFERENCES

[1] M. BLUM, R. FLOYD, V. PRATT, R. RIVEST AND R. TARJAN, *Time bounds for selection*, J. Comput. Systems Sci., 7 (1973), pp. 448–461.

[2] M. R. GAREY, D. S. JOHNSON AND L. J. STOCKMEYER, *Some simplified NP-complete problems*, 6th ACM Symp. on Theory of Computing, Seattle, WA, 1974, pp. 47–68.

[3] B. W. KERNIGHAN, *Optimal sequential partitions of graphs*, J. Assoc. Comput. Mach., 18 (1971), pp. 34–40.

[4] J. KRAL, *To the problem of segmentation of a program*, Information Processing Machines, (1965), pp. 140–149.

[5] J. A. LUKES, *Efficient algorithm for partitioning of trees*, IBM J. Res. Develop., 18 (1974), no. 3, p. 217.

[6] J. MISRA AND R. E. TARJAN, *Optimal chain partitions of trees*, Information Processing Lett., 4 (1975), pp. 24–26.

[7] W. H. HOSKEN, *Optimum partitions of tree addressing structures*, this Journal, 4 (1975), pp. 341–347.

SIAM J. COMPUT.
Vol. 6, No. 1, March 1977

BOUNDS FOR LPT SCHEDULES ON UNIFORM PROCESSORS*

TEOFILO GONZALEZ†, OSCAR H. IBARRA‡ AND SARTAJ SAHNI‡

Abstract. We study the performance of LPT (largest processing time) schedules with respect to optimal schedules in a nonpreemptive multiprocessor environment. The processors are assumed to have different speeds and the tasks being scheduled are independent.

Key words. LPT schedules, uniform processors, nonpreemptive scheduling, independent tasks

1. Introduction. A uniform processor system [4] is one in which the processors $P_1, \cdots, P_m$ have relative speeds $s_1, \cdots, s_m$ respectively. It is assumed that the speeds have been normalized such that $s_1 = 1$ and $s_i \geqq 1$, $2 \leqq i \leqq m$. The problem of scheduling n independent tasks $(J_1, \cdots, J_n)$ with execution times $(t_1, \cdots, t_n)$ on m uniform processors to obtain a schedule with the optimal (least) finish time is known to be NP-complete [1], [4]. Hence, it appears unlikely that there is any polynomial time bounded algorithm to generate such schedules. For preemptive scheduling, however, optimal finish time algorithms can be obtained in polynomial time [6], [7]. Horowitz and Sahni [4] showed that for any m, polynomial time algorithms exist to obtain schedules with a finish time arbitrarily close to the optimal finish time. The complexity of these algorithms was, however, exponential in m. The purpose of this paper is to study the finish time properties of LPT schedules with respect to the optimal finish time.

DEFINITION. An *LPT* (*largest processing time*) *schedule* is a schedule obtained by assigning tasks to processors in order of nonincreasing processing times. When a task is being considered for assignment to a processor, it is assigned to that processor on which its finishing time will be earliest. Ties are broken by assigning the task to the processor with least index.

One may easily verify that for identical processor systems, this definition is equivalent to that of [2, p. 100]. Graham [3] studied LPT schedules for the special case of identical processors, i.e., $s_i = 1$, $1 \leqq i \leqq m$. If $\hat{f}$ is the finish time of the LPT schedule and f^* the optimal finish time, then Graham's result is that $\hat{f}/f^* \leqq \frac{4}{3} - \frac{1}{3m}$ and that this bound is the best possible bound. In § 2 we extend his work to the general case of uniform processors. While the bound we obtain is best possible for $m = 2$, it appears that it is not so for $m > 2$. In view of this, we turn our attention to another special case of uniform processors, i.e., $s_1 = 1$, $1 \leqq i < m$ and $s_m = s \geqq 1$. This case has previously been studied by J. W. S. Liu and C. L. Liu [5]. Using a priority assignment according to lengths of tasks, they show that $f/f^* \leqq 2(m-1+s)/(s+2)$ for $s \leqq 2$ and $f/f^* \leqq (m-1+s)/2$ for $s \geqq 2$, where f is the finish time of the priority schedule.

* Received by the editors January 24, 1975, and in final revised form March 28, 1976. This work was supported in part by the National Science Foundation under Grants DCR72-03728-A01 and DCR74-10081.

† Department of Computer Science, University of Minnesota, Minneapolis, Minnesota. Now at Department of Computer Science, Pennsylvania State University, University Park, Pennsylvania 16802.

‡ Department of Computer Science, University of Minnesota, Minneapolis, Minnesota 55455.

Similar bounds for list schedules are also obtained by them. We show that for $m \geqq 3$, $\hat{f}/f^* \leqq 3/2 - 1/(2m)$ and that this bound is the best possible for $m = 3$. For $m > 3$ we conjecture that $\hat{f}/f^* \leqq 4/3$.

Before presenting our results we develop the necessary notation and basic results. Throughout the remainder of this paper $\hat{f}$ and f^* will denote the finish times of LPT and optimal schedules respectively. Let S be the set of tasks being scheduled. It will sometimes be necessary to distinguish between finish times of different sets of tasks. To do this, S will appear as a superscript along with $\hat{f}$ or f^* as in $\hat{f}^S$ and f^{*S}. If the number of processors is important, then this number will appear as a subscript as in $\hat{f}_m$, f^{*S}_m etc. We shall refer to the sets of tasks (jobs) by their task execution time. Thus, we speak of a set, S, of tasks $(t_1 \geqq t_2 \geqq \cdots \geqq t_n)$ meaning the execution time of task i is t_i and $t_i \geqq t_{i+1}$, $1 \leqq i < n$. The m processors $P_1, \cdots, P_m$ are assumed ordered such that $s_1 = 1$ and $1 \leqq s_i \leqq s_{i+1}$, $2 \leqq i < m$. The following result from [2, p. 102] is made use of:

LEMMA 1.1. *If for any m, $S = (t_1 \geqq t_2 \geqq \cdots \geqq t_n)$ is the smallest set of tasks for which $\hat{f}/f^* > k$, then t_n determines the finish time $\hat{f}$ (i.e., task n has the latest completion time).*

2. Basic results.

In this section, we prove two important lemmas that are used throughout the paper (Lemmas 2.2 and 2.3). We also derive the bound $2m/(m+1)$ for the ratio $\hat{f}/f^*$ for the general m-processor system. Examples are shown for which $\hat{f}/f^*$ approaches $3/2$ as $m \to \infty$.

We begin with the following lemma. Informally, it states that if either the LPT or optimal schedule of an $(m+1)$-processor system has an idle processor, then the ratio $\hat{f}/f^*$ for this schedule is no worse than $\hat{f}/f^*$ for m processors.

LEMMA 2.1. *For $m \geqq 1$, let $g(m, s_2, \cdots, s_m)$ be such that $\hat{f}_m/f^*_m \leqq g(m, s_2, \cdots, s_m)$. Consider any $(m+1)$-processor system with job set $S = (t_1 \geqq t_2 \geqq \cdots \geqq t_n)$ and processor speeds $1 = s_1 \leqq s_2 \leqq \cdots \leqq s_{m+1}$. If a processor is idle in either the LPT or optimal schedule of S, then $\hat{f}^S_{m+1}/f^{*S}_{m+1} \leqq g(m, s_3/s_2, \cdots, s_{m+1}/s_2)$.*

Proof. Suppose in the LPT schedule of S a processor P_i is idle. Then it must be the case that in the optimal schedule, P_i is also idle. Otherwise, $\hat{f}^S_{m+1} \leqq t_n/s_i$, $f^{*S}_{m+1} \geqq t_n/s_i$ and $\hat{f}^S_{m+1}/f^{*S}_{m+1} = 1$. So we need only consider the case when P_i is idle in the optimal schedule. If P_i is idle then clearly P_1 is also idle or can be made idle without increasing f^* by scheduling the jobs from P_1 onto P_i. Consider the m-processor system with job set S and processor speeds $1 = s_2/s_2 \leqq s_3/s_2 \leqq \cdots \leqq s_{m+1}/s_2$. Then by assumption, for this system, $\hat{f}^S_m/f^{*S}_m \leqq g(m, s_3/s_2, \cdots, s_{m+1}/s_2)$. Moreover, $\hat{f}^S_{m+1} \leqq \hat{f}^S_m/s_2$ and $f^{*S}_{m+1} = f^{*S}_m/s_2$. It follows that $\hat{f}^S_{m+1}/\hat{f}^{*S}_{m+1} \leqq g(m, s_3/s_2, \cdots, s_{m+1}/s_2)$. □

The next lemma gives an estimate of $\hat{f}/f^*$ for the case when $\hat{f}$ is determined by the job with the smallest execution time.

LEMMA 2.2. *Consider an m-processor system with job set $S = (t_1 \geqq t_2 \geqq \cdots \geqq t_n)$ and speeds $s_1, \cdots, s_m$. If in the LPT schedule of S, the finish time $\hat{f}$ is determined by t_n, (i.e., if task n has the latest completion time), then $\hat{f}/f^* \leqq 1 + (m-1)t_n/(Qf^*)$, where $Q = \sum s_i$.*

Proof. Let the LPT schedule be as shown in Fig. 2.1, where P_k determines the finish time. Each T_i is the sum (possibly 0) of task times of jobs scheduled on P_i

prior to t_n's assignment, $T_1+\cdots+T_m=t_1+\cdots+t_{n-1}$.

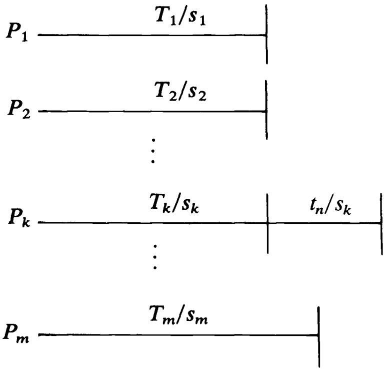


FIG. 2.1

Since task n determines the finish time, $\hat f=(T_k+t_n)/s_k$ and $(T_i+t_n)/s_i \geqq \hat f$ for $i \neq k$. Hence, $\hat f s_i - T_i \leqq t_n$ and so $\hat f \sum_{i\neq k} s_i - \sum_{i\neq k} T_i \leqq (m-1)t_n$. This, together with $\hat f s_k = T_k + t_n$ yields

$$\begin{aligned}\hat f Q &\leqq \sum T_i + m t_n\\ &= \sum t_i + (m-1)t_n.\end{aligned}$$

Since $f^* \geqq \sum t_i/Q$, we get $\hat f/f^* \leqq 1+(m-1)t_n/(f^*Q)$. □

Using Lemmas 2.1 and 2.2, we can now derive a bound for the m processor system.

THEOREM 2.1. *For an m-processor system, $\hat f/f^* \leqq 2m/(m+1)$.*

Proof. For $m=1$, the theorem obviously holds. Now suppose the theorem holds for $1,2,\cdots,m-1$ processors but fails for m-processors. Let $S=(t_1 \geqq t_2 \geqq \cdots \geqq t_n)$ be the smallest set of jobs which gives a bound $\hat f_m/f^*_m > 2m/(m+1)$. Then by Lemma 1.1, t_n determines the finish time. There are two cases to consider. Both lead to a contradiction.

Case 1. $n \geqq m+1$. Then by Lemma 2.2,

$$\begin{aligned}\hat f_m/f^*_m &\leqq 1+\frac{(m-1)t_n}{Qf^*}\\ &\leqq 1+(m-1)t_n/[Q(\sum t_i/Q)]\\ &\leqq 1+\frac{(m-1)t_n}{nt_n}=1+\frac{(m-1)}{n}\leqq 1+\frac{m-1}{m+1}=\frac{2m}{m+1}, \quad \text{a contradiction.}\end{aligned}$$

Case 2. $n \leqq m$. Then in the optimal schedule, either each processor has exactly one job or a processor is idle. In the first case, $\hat f_m/f^*_m=1$, since no processor can be idle in the LPT schedule (see proof of Lemma 2.1). For the second case $\hat f^S_m/f^{*S}_m \leqq \hat f^S_{m-1}/f^{*S}_{m-1} \leqq 2(m-1)/m \leqq 2m/(m+1)$ by Lemma 2.1. Either case leads to a contradiction. □

COROLLARY 2.1. *For an m-processor system, $\hat f/f^* < 2$.*

The bound of Theorem 2.1 is probably not a tight bound. However, we can show that there are examples approaching the bound 1.5 as $m \to \infty$.

THEOREM 2.2. *For every $m \geqq 2$, there is an example of an m-processor system and a set of jobs S for which $\hat{f}^S/f^{S*} = c$, where c is a positive root of the equation $2s^m - s^{m-1} - \cdots - s - 2 = 0$.*

Proof. The example we shall construct has job set $S = (t_1 \geqq t_2 \geqq \cdots \geqq t_m \geqq t_{m+1})$ (where m is the number of processors) and processor speeds $1 = s_1 \leqq \cdots \leqq s_m$. The t_i's and s_i's will satisfy the following properties (see Fig. 2.2):

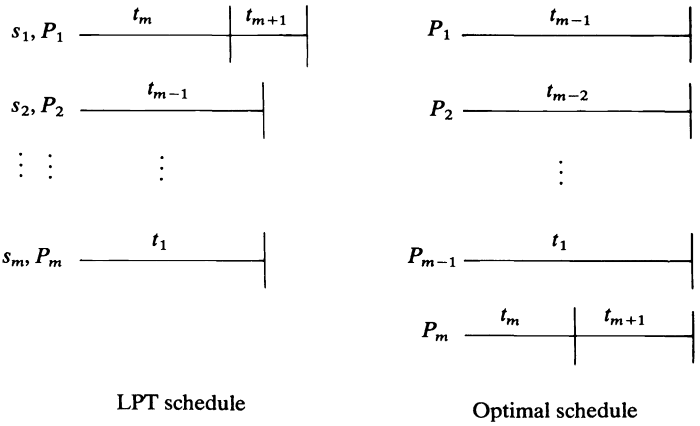


FIG. 2.2

$$\text{(2.1)} \qquad \hat{f} = t_m + t_{m+1} \quad \text{and} \quad f^* = \frac{t_m + t_{m+1}}{s_m},$$

$$\text{(2.2)} \qquad t_m = t_{m+1} = t,$$

$$\text{(2.3)} \qquad t_m + t_{m+1} = 2t = \frac{t_i + t}{s_{m-i+1}} \quad \text{for} \quad 1 \leqq i \leqq m-1,$$

$$\text{(2.4)} \qquad \frac{t_m + t_{m+1}}{s_m} = \frac{2t}{s_m} = \frac{t_i}{s_{m-i}} \quad \text{for} \quad 1 \leqq i \leqq m-1.$$

Then $\hat{f}/f^* = 2t/(2t/s_m) = s_m$. From properties (2.1)–(2.4) we can derive the equation for s_m. From (2.3) we get

$$\text{(2.5)} \qquad t_i = 2ts_{m-i+1} - t = t(2s_{m-i+1} - 1).$$

From (2.4), we have

$$\text{(2.6)} \qquad s_m t_i = 2ts_{m-i}.$$

Equations (2.5) and (2.6) yield

$$\text{(2.7)} \qquad s_{m-i+1} = \frac{2s_{m-i} + s_m}{2s_m} \quad \text{for} \quad 1 \leqq i \leqq m-1.$$

Using (2.7) repeatedly for $i = 1, 2, \cdots, m-1$ we get

$$
\begin{aligned}
s_m &= \frac{2s_{m-1}+s_m}{2s_m} = \frac{2\left(\dfrac{2s_{m-2}+s_m}{2s_m}\right)+s_m}{2s_m} \\
&= \frac{2s_{m-2}+s_m+s_m^2}{2s_m^2} = \frac{2\left(\dfrac{2s_{m-3}+s_m}{2s_m}\right)+s_m+s_m^2}{2s_m^2} \\
&= \frac{2s_{m-3}+s_m+s_m^2+s_m^3}{2s_m^3} \\
&\vdots \\
&= \frac{2s_1+s_m+s_m^2+\cdots+s_m^{m-1}}{2s_m^{m-1}}
\end{aligned}
$$

Hence,

$$s_m = \frac{2+s_m+s_m^2+\cdots+s_m^{m-1}}{2s_m^{m-1}} \quad \text{(since } s_1 = 1\text{)}$$

or

$$2s_m^m - s_m^{m-1} - s_m^{m-2} - \cdots - s_m - 2 = 0. \tag{2.8}$$

The polynomial on the left-hand side of (2.8) has one sign change and so from Descartes' rule of sign it also has one positive real root. This root must clearly be >1 as otherwise the left-hand side is <0.

Let c be this positive real root of equation (2.8). We can construct an example of an m-processor system with $\hat{f}/f^* = c$ by setting $s_m = c$ and computing $s_2, \cdots, s_{m-1}$ in terms of c using (2.7). (Of course, $s_1 = 1$.) Then by letting $t_m = t_{m+1} = t$, we can determine the values of $t_1, \cdots, t_{m-1}$ in terms of t using (2.4). □

COROLLARY 2.2. *There exist uniform processor systems and job sets S for which* $\hat{f}/f^* \approx 1.5$.

Proof. From Theorem 2.2 we know that there are job sets, S, for which $\hat{f}/f^* = c$ where c is a positive root of (2.8). Let s be a root. Rearranging terms, we get

$$
\begin{aligned}
2s^m - 1 &= \sum_{0 \leqq i < m} s^i \\
&= \frac{s^m - 1}{s - 1}
\end{aligned}
$$

or $2s^{m+1} - 3s^m - s + 2 = 0$.

Since $s > 1$, for $m \to \infty$ we have $s \to 3/2$ as a root. □

Example. (a) $m=2$: Then we have $2s_2^2-s_2-s=0$, where we find $s_2=(1+\sqrt{17})/4$. Of course, $s_1=1$. Let $t_2=t_3=1$. From (2.4), we find

$$t_1=\frac{2t}{s_2}\cdot s_1=\frac{8}{1+\sqrt{17}}.$$

One easily verifies that $\hat{f}/f^*=(1+\sqrt{17})/4$.

(b) $m=3$: The equation to use is $2s_3^3-s_3^2-s_3-2=0$. $s_3=1.384$ is an approximate root of this equation. Using (2.7), we find $s_2=s_3(2s_3-1)/2=1.223$; $s_1=1$. Let $t_3=t_4=t=1$. Using (2.4), find $t_2=(2t/s_3)\cdot s_2=1.767$ and $t_1=(2t/s_3)\cdot s_1=1.445$. Again we can check that $\hat{f}/f^*$ is approximately 1.384.

(c) Some other roots of (2.8) are 1.493 for $m=10$ and 1.499 for $m=20$.

3. The case $s_i=1$, $1\leqq i\leqq m$, and $s_m\geqq 1$. In this section we study the special case in which all but one of the $m\geqq 1$ processors has a speed of 1. The mth processor P_m has a speed $s\geqq 1$. The main result of this section is stated below as Theorem 3.1.

THEOREM 3.1. *For $m\geqq 2$ the ratio $\hat{f}/f^*$ has the following bounds*:

(i) $\hat{f}/f^*\leqq(1+\sqrt{17})/4$ *for* $m=2$,

(ii) $\hat{f}/f^*\leqq 3/2-1/(2m)$ *for* $m>2$.

Proof. (i) is proved in Lemma 3.2. (ii) follows from Lemmas 3.1–3.6 and the fact that the bound is a monotone increasing function in m.

Before proving the theorem we derive a general bound for $\hat{f}/f^*$ in terms of m and s.

LEMMA 3.1. *For an m-processor system with $s_i=1$ for $1\leqq i<m$ and $s_m=s$, $\hat{f}/f^*\leqq 2(m-1+s)/(m-1+2s)$.*

Proof. If $m=1$, the lemma is obviously true since $\hat{f}/f^*=1$. Now assume that the lemma holds for $1, 2, \cdots, m-1$ processors but fails for m $(m\geqq 2)$. For this m, let $S=(t_1\geqq t_2\geqq\cdots\geqq t_n)$ be the smallest set of jobs for which $\hat{f}/f^*>2(m-1+s)/(m-1+2s)$. Suppose a processor is idle in either the LPT or optimal schedule of S. Then $\hat{f}_m^S/f_m^{*S}\leqq\hat{f}_{m-1}^S/f_{m-1}^{*S}\leqq 2(m-2+s)/(m-2+2s)\leqq 2(m-1+s)/(m-1+2s)$ by Lemma 2.1.

So we may assume that no processor is idle in either the LPT or optimal schedule of S. We consider two cases, both leading to a contradiction.

Case 1. The LPT schedule is as shown in Fig. 3.1, where each T_i represents the sum of execution times of jobs scheduled on P_i prior to the assignment of t_n, $T_1+\cdots+T_m=t_1+\cdots+t_{n-1}$. By assumption, no processor is idle. Hence $T_i>0$ for $2\leqq i\leqq m$. Since the first $m-1$ processors have speed 1, we may assume that $T_i\geqq T_1$ for $1\leqq i\leqq m-1$. Now if $T_1=0$, then $\hat{f}=t_n$. But $f^*\geqq t_n$ since by assumption no processor is idle in the optimal schedule. Then $\hat{f}/f^*=1$. So we may also assume that $T_1\geqq t_n$.

Thus, $\hat{f}\geqq 2t_n$. From Lemma 2.2 we have $\hat{f}/f^*\leqq 1+(m-1)t_n/(Qf^*)$, where $Q=(m-1)+s$. This implies that $Qf^*\geqq Q\hat{f}-(m-1)t_n\geqq 2Qt_n-(m-1)t_n$. Substituting this inequality back into Lemma 2.2 gives

$$\frac{\hat{f}}{f^*}\leqq 1+\frac{(m-1)t_n}{2Qt_n-(m-1)t_n}=1+\frac{m-1}{2Q-m+1}=\frac{2Q}{2Q-m+1}=\frac{2(m-1+s)}{m-1+2s},$$

a contradiction.

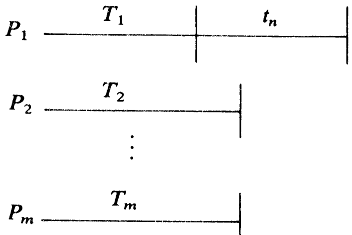


FIG. 3.1

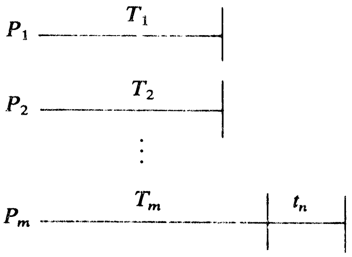


FIG. 3.2

Case 2. Suppose the LPT schedule is as shown in Fig. 3.2, where we again assume that $T_i \geqq T_1 \geqslant t_n$. We may also assume that $T_m > 0$; otherwise $\hat{f}/f^* = 1$ since $\hat{f} = t_n/s$. If $\hat{f} \geqq 2t_n$ then the proof proceeds as in Case 1. Otherwise, $\hat{f} < 2t_n$. Note that $\sum t_i \geqq s\hat{f} + (m-1)t_n$. Therefore,

$$\frac{\hat{f}}{f^*} \leqq \frac{Q\hat{f}}{s\hat{f} + (m-1)t_n} = \frac{Q}{s + (m-1)t_n/\hat{f}}.$$

Since $\hat{f} < 2t_n$ we have

$$\frac{\hat{f}}{f^*} \leqq \frac{Q}{s + (m-1)/2} = \frac{2Q}{m-1+2s} = \frac{2(m-1+s)}{m-1+2s}. \quad \square$$

The bound for $m = 2$ follows from the following lemma.

LEMMA 3.2. *For an m processor system with $s_i = 1$, $1 \leqq i < m$ and $s_m = s$, $/f^* \leqq \frac{1}{4}[(3-m) + \sqrt{(3-m)^2 + 16(m-1)}]$. Moreover, for $m = 2$, the bound is tight.*

Proof. Let $k > 1$ be the desired bound for $\hat{f}/f^*$. Let $Q = \Sigma s_l = m - 1 + s$. First we show that if $s \leqq 2Q(k-1)/(m-1)$, then $\hat{f}/f^* \leqq k$. Suppose not. Let $S = (t_1 \geqq \cdots \geqq t_n)$ be the smallest set of jobs for which $\hat{f}/f^* > k$. Then t_n determines the finish time and by Lemma 2.2, $\hat{f}/f^* \leqq 1 + (m-1)t_n/(Qf^*)$. Hence $f^* < (m-1)t_n/[Q(k-1)]$. It follows that the number of jobs on each processor in the optimal schedule of S is less than $(m-1)s/[Q(k-1)] \leqq 2$. But then in this case, $\hat{f}/f^* = 1$. This contradicts the assumption that S produces a bound $>k$. Thus if $s \leqq 2Q(k-1)/(m-1)$, then $\hat{f}/f^* \leqq k$. This, in turn, implies that if $Q \leqq (m-1) + 2Q(k-1)/(m-1)$, then $\hat{f}/f^* \leqq k$ or that

$$\text{(3.1)} \qquad \text{if} \quad Q \leqq \frac{(m-1)^2}{m-2k+1}, \quad \text{then} \quad \hat{f}/f^* \leqq k.$$

Now by Lemma 3.1, we have

$$\frac{\hat{f}}{f^*} \leqq \frac{2(m-1+s)}{m-1+2s} = \frac{2(m-1+s)}{2(m-1+s)-(m-1)} = \frac{2Q}{2Q-(m-1)}.$$

It follows that if $2Q/(2Q-(m-1)) \leqq k$, then $\hat{f}/f^* \leqq k$ or

$$\text{(3.2)} \qquad \text{if} \quad Q \geqq \frac{(m-1)k}{2(k-1)} \quad \text{then} \quad \hat{f}/f^* \leqq k.$$

To satisfy (3.1) and (3.2) simultaneously, we must have $(m-1)^2/(m-2k+1) \geqq (m-1)k/(2(k-1))$, from which we get $k \geqq \frac{1}{4}[(3-m)+\sqrt{(3-m)^2+16(m-1)}]$. For all such k, $\hat{f}/f^* \leqq k$ for all Q.

For the case $m=2$, we have $k=(1+\sqrt{17})/4$. In § 2 we saw an example with $s_2=(1+\sqrt{17})/4$ for which $\hat{f}/f^*=(1+\sqrt{17})/4$. Hence, this bound is tight for $m=2$. □

In arriving at the proof of the theorem for $m>2$, it is necessary to prove four lemmas. To begin with, we show that if for any set of jobs, S, an optimal schedule has more than one job on any of the processors $P_1, \cdots, P_{m-1}$ then $\hat{f}^S/f^{*S} \leqq 3/2-1/(2m)$.

LEMMA 3.3. *For any set of jobs, S, either* (i) *processors P_1-P_{m-1} have at most one job scheduled on each in every optimal schedule or*

$$\text{(ii)} \qquad \frac{\hat{f}_m^S}{f_m^{*S}} \leqq \frac{3}{2}-\frac{1}{2m}.$$

Proof. Suppose (ii) is not true for some set of jobs. Let $S=(t_1 \geqq t_2 \geqq \cdots \geqq t_n)$ be the smallest set of jobs for which $\hat{f}_m^S/f_m^{*S} > 3/2-1/(2m)$. From Lemma 2.2 we get

$$\frac{\hat{f}_m^S}{f_m^{*S}} \leqq 1+\frac{(m-1)t_n}{(m-1+s)f_m^*} > \frac{3}{2}-\frac{1}{2m}$$

or

$$\frac{(m-1)t_n}{(m-1+s)f_m^*} > \frac{m-1}{2m}$$

or

$$t_n > \frac{m-1+s}{2m} f_m^*$$
$$\geqq \frac{1}{2} f_m^*$$

i.e., $f_m^* < 2t_n$ which, in turn, means that none of the processors P_1-P_{m-1} can have more than one job scheduled on them in an optimal schedule. □

Next, we prove that if $s \geqq m-1$ then $\hat{f}/f^* \leqq 4/3$.

LEMMA 3.4. *If $s \geqq m-1$ then $\hat{f}/f^* \leqq 4/3 \leqq 3/2 - 1/(2m)$ for $m > 2$.*

Proof. Lemma 3.1 gives

$$\hat{f}/f^* \leqq \frac{2(m-1+s)}{m-1+2s}.$$

The right-hand side of the above inequality is a decreasing function of s. Hence, for $s \geqq m-1$ we obtain

$$\begin{aligned}\frac{\hat{f}_m}{f^*_m} &\leqq \frac{4m-4}{3(m-1)}\\ &= 4/3\\ &\leqq 3/2 - 1/(2m), \quad m > 2. \quad \square\end{aligned}$$

As a result of Lemmas 3.3 and 3.4 the only counterexamples to Theorem 3.1 are sets of jobs, S, for which the optimal schedules have at most one job on each of $P_1 - P_{m-1}$ and the speed, s, of P_m is $< m-1$. The next two lemmas show that for this kind of an optimal schedule and $s < m-1$ the bound of Theorem 3.1 cannot be violated.

LEMMA 3.5. *Let $S = (t_1 \geqq t_2 \geqq \cdots \geqq t_n)$ be the smallest set of jobs for which $\hat{f}/f^* > 3/2 - 1/(2m)$. If in the LPT schedule, t_i is the only job scheduled on one of the processors, $P_1, \cdots, P_{m-1}$ and if in an optimal schedule t_j is the only job scheduled on one of the processors, $P_1, \cdots, P_{m-1}$ then, either*

$$\text{(i)} \qquad \hat{f}^S_m/f^{*S}_m \leqq \hat{f}_{m-1}/f^*_{m-1}$$

or

$$\text{(ii)} \qquad t_i < t_j.$$

Proof. From Lemma 1.1 it follows that t_n determines the finish time $\hat{f}^S$. If any one of the processors $P_1, \cdots, P_m$ is idle in an optimal solution (i.e. no jobs have been scheduled on it), then $f^{*S}_m = f^{*S}_{m-1}$. But, $\hat{f}^S_m \leqq \hat{f}^S_{m-1}$ and so $\hat{f}^S_m/f^{*S}_m \leqq \hat{f}^S_{m-1}/f^{*S}_{m-1}$. We may therefore assume that no processor is idle in any optimal solution. Hence, $f^{*S}_m \geqq t_n$. If $i = n$, then $\hat{f}^S_m = t_n$ (as t_i is the only job on some processor $P_1, \cdots, P_{m-1}$) and $\hat{f}^S_m/f^{*S}_m \leqq 1$. Therefore $i \neq n$. Now, we have

$$\begin{aligned}f^{*S}_m &= \max\{t_j, f^{*S-\{t_j\}}_{m-1}\}\\ &\geqq f^{*S-\{t_j\}}_{m-1}\\ &\geqq f^{*S-\{t_i\}}_{m-1} \cdots \quad \text{as} \quad t_i \geqq t_j,\end{aligned}$$

but

$$\hat{f}^S_m = \hat{f}^{S-\{t_i\}}_{m-1} \quad \cdots \quad \text{as} \quad i \neq n.$$

Therefore,

$$\frac{\hat{f}^S_m}{f^{*S}_m} \leqq \frac{\hat{f}^{S-\{t_i\}}_{m-1}}{f^{*S-\{t_i\}}_{m-1}} \leqq \frac{\hat{f}_{m-1}}{f^*_{m-1}}.$$

LEMMA 3.6. *When $s < m-1$ and an optimal schedule for any set of jobs S has at most one job on each of processors $P_1 - P_{m-1}$, then $\hat{f}_m/f^*_m \leqq 3/2 - 1/(2m)$.*

Proof. Let $S = (t_1 \geqq t_2 \geqq \cdots \geqq t_n)$ be the smallest set of jobs and m the least $m > 2$ for which the lemma is not true. From Lemma 3.1 we obtain $\hat{f}/f^* \leqq 1 + (m-1)/(m-1+s)(t_n/f^*)$. By assumption $\hat{f}/f^* > 3/2 - 1/(2m)$. Therefore,

$$1 + \frac{(m-1)}{m-1+s}\frac{t_n}{f^*} > \frac{3}{2} - \frac{1}{2m}$$

or

$$f^* < \frac{2m}{m-1+s}t_n. \tag{3.3}$$

If $\#_m$ is the number of jobs on P_m in an optimal schedule then, $f^* \geqq \#_m t_n/s$. Substituting this inequality into (3.3) yields

$$\#_m < \frac{2sm}{m-1+s}. \tag{3.4}$$

The right-hand side of the inequality (3.4) is an increasing function of s. Since $s < m-1$, (3.4) yields the following bound on $\#_m$:

$$\#_m \leqq \frac{2(m-1)m}{2(m-1)} = m.$$

The optimal schedule has at most one job on each of $P_1 - P_{m-1}$. Hence, $n \leqq 2m-2$.

The remainder of the proof shows that if $n \leqq 2m-2$ then Lemma 3.5 can be used to show that $\hat{f}^S_m/f^{*S}_m \leqq \hat{f}^S_{m-1}/f^{*S}_{m-1}$, thus contradicting the assumption that this was the least m for which the lemma was false. (The contradiction comes about as $3/2 - 1/(2m)$ is monotone increasing in m and the fact that when $m = 3$ this bound is $4/3$ which is greater than the known bound for $m = 2$.) Clearly, we may assume that each processor has at least one job scheduled on it in every optimal schedule.

Let k be the smallest index (i.e. largest job) on any of the processors $P_1 - P_{m-1}$ in an optimal schedule. Then, the schedule obtained by assigning job t_{k+i-1} to processor P_i, $1 \leqq i < m$, and the remaining jobs to processor P_m has a finish time no greater than the optimal finish time f^{*S}_m. Such a schedule shall be denoted by OPT_k. Clearly, $1 \leqq k \leqq n - m + 2$. Since, $n \leqq 2m-2$ at least one of the processors $P_1 - P_{m-1}$ has exactly one job scheduled on it (every processor must have at least one job on it as otherwise, by the definition of LPT, $\hat{f} \leqq t_n$ but $f^* \geqq t_n$). Let the index of this job be i. Then, t_i must be the largest job amongst jobs scheduled on $P_1 - P_{m-1}$ in the LPT schedule (this again follows from the definition of LPT). But, $s < m-1$ implies $t_i \geqq t_{m-1}$ as LPT cannot schedule all of the first $m-1$ jobs on P_m when $s < m-1$. For all $k \geqq 1$, OPT_k has a job with index $j = k + m - 2 \geqq m - 1$ on P_{m-1} and this is the only job on P_{m-1}. By the ordering on the jobs, $t_j \leqq t_{m-1}$. So, $t_i \geqq t_j$. Lemma 3.5 now implies that $\hat{f}^S_m/f^{*S}_m \leqq \hat{f}_{m-1}/f^*_{m-1}$; a contradiction. □

Having shown that $\hat{f}/f^*$ is indeed bounded as in Theorem 3.1, the next question is: How good is the bound. From the previous section we know that the bound for $m = 2$ is tight. Lemma 3.7 shows that the bound is also tight for $m = 3$ and that for all $m > 3$ it is possible to have an $\hat{f}/f^*$ arbitrarily close to 4/3. Lemma 3.8 shows that for $m = 4$ and 5 there is no set of jobs S for which $\hat{f}/f^* > 4/3$. This shows that the bound of $3/2 - 1/(2m)$ is not a tight bound for all values of m and leads us to conjecture that for $m \geqq 3$ the bound is in fact 4/3. Note the closeness of this bound of 4/3 to the bound $4/3 - 1/(3m)$ obtained by Graham [3] for the case of $s = 1$ (i.e., m identical processors).

LEMMA 3.7. *For $m \geqq 3$ and any $\varepsilon > 0$, there is a set of jobs, S, and a speed $s > 1$ for which $\hat{f}/f^* > 4/3 - \varepsilon$.*

Proof. For any $m \geqq 3$ consider the set of jobs $t_1 = 1.5$, $t_2 = 1.5$, $t_j = 1$, $3 \leqq j \leqq m + 2$ and $s = 2 + \varepsilon'$ with ε' very close to zero. The LPT schedule has jobs t_1, t_2 and t_{m+2} on P_m with $\hat{f} = 4/(2 + \varepsilon')$. One optimal schedule is shown in Fig. 3.3. $f^* = 1.5$. Hence, $\hat{f}/f^* = 8/(6 + 3\varepsilon') \to 4/3$ as $\varepsilon' \to 0$.

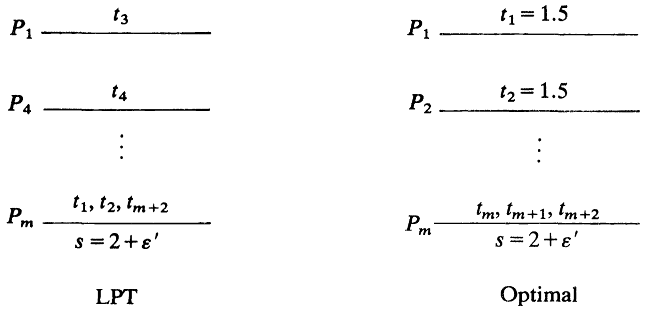


FIG. 3.3. *LPT and optimal schedules for Lemma* 3.7

LEMMA 3.8. *For $m = 4$ and 5, $\hat{f}/f^* \leqq 4/3$.*

Proof. The proof is omitted and may be found in [8].

Conjecture. $\hat{f}/f^* \leqq 4/3$ for $m \geqq 3$ and $s_i = 1$, $1 \leqq i < m$ and $s_m \geqq 1$.

4. Conclusions. We have shown that in the case of uniform processors LPT schedules have a finish time at most twice the optimal finish time. The worst examples we could construct result in LPT schedules with finish times 1.5 times the optimal for $m \to \infty$. For the special case studied in [5] it is shown that $\hat{f}/f^* \leqq 3/2 - 1/(2m)$.

Acknowledgment. We are grateful to the referee for providing a simpler proof of Lemma 3.1.

REFERENCES

[1] J. BRUNO, E. G. COFFMAN, JR. AND R. SETHI, *Scheduling independent tasks to reduce mean finishing-time*, Comm. ACM, 17 (1974), pp. 382–387.

[2] E. G. COFFMAN, JR. AND P. J. DENNING, *Operating Systems Theory*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[3] R. L. GRAHAM, *Bounds on multiprocessing timing anomalies*, SIAM J. Appl. Math., 17 (1969), pp. 416–429.

[4] E. Horowitz and S. Sahni, *Exact and approximate algorithms for scheduling non-identical processors*, J. Assoc. Comput. Mach., 23 (1976), pp. 317–327.

[5] J. W. S. Liu and C. L. Liu, *Bounds on scheduling algorithms for heterogeneous computing systems*, Proc. IFIP, (1974), pp. 349–353.

[6] J. W. S. Liu and A. Yang, *Optimal scheduling of independent tasks on heterogeneous computing systems*, ACM National Conference, 1974, pp. 38–45.

[7] E. Horvath and R. Sethi, *Preemptive schedules for independent tasks*, Computer Science Tech. Rep. 162, Pennsylvania State Univ., College Park, 1975.

[8] T. Gonzalez, O. H. Ibarra and S. Sahni, *Bounds for LPT schedules on uniform processors*, Tech. Rep. 75-1, Univ. of Minnesota, Minneapolis, 1975.

SIAM J. COMPUT.
Vol. 6, No. 1, March 1977

TASK SCHEDULING ON A MULTIPROCESSOR SYSTEM WITH INDEPENDENT MEMORIES*

D. G. KAFURA† AND V. Y. SHEN‡

Abstract. This paper considers a model of a computing system with several independent but identical processors, each with a private memory of limited, and possibly different, storage capacity. The tasks are assumed to have known resource demands expressed as processing times and memory requirements. Several scheduling strategies are evaluated by worst-case performance bounds and simulation results. Both preemptive and nonpreemptive scheduling are considered. An optimal preemptive algorithm is given to find the shortest schedule for a task set with no precedence constraints.

Key words. scheduling, scheduling algorithms, multiprocessor system, deterministic models, worst-case bounds, memory constraints

1. Introduction. This paper analyzes a model of computation with processors and memory as resources. The allocation of memory has the constraint that only fixed (but possibly different) amounts can be allocated during the scheduling process. The analysis differs from previous investigations which consider only the processors as the resource to be scheduled [3], [6], [9]–[11], [22], [23], or which consider the memory resource as a contiguous quantity such that each memory request may be satisfied by allocating the exact amount requested [8], [19], [20]. Since the problem of finding the optimal schedule for most interesting models of computation is "polynomial complete" [4], [17], [25], this paper analyzes heuristic algorithms in terms of their worst-case behavior except in the case that an optimal algorithm is available. The measure of performance in this investigation is the maximum finishing time ([9], [10], etc.) rather than the mean finishing time sometimes used [1].

The model of computation being analyzed consists of m identical, independent, abstract processors $P_1, P_2, \cdots, P_m$. Associated with the jth processor is a private memory with fixed size denoted by $|P_j|$. Each memory is private in the sense that information contained in the jth memory is accessible only by P_j. The memory sizes are fixed since $|P_1|, |P_2|, \cdots, |P_m|$ remain constant throughout the execution of a task set. For convenience, we will index the processors so that $|P_j| \geqq |P_{j+1}|$. The task set, J, consists of n tasks, $\{J_1, J_2, \cdots, J_n\}$, where the ith task, J_i: (m_i, t_i), specifies both a memory requirement, m_i, and a time requirement, t_i. An important scheduling parameter is the number of processors which can execute a given task under the memory constraint. For the ith task the number of such processors will be denoted by n_i. The tasks in J are partially ordered by a precedence relation, $<$. This relation is interpreted to mean that if $J_i < J_k$, then J_k cannot begin in any valid schedule before J_i has completed. In this case J_i is called a predecessor of J_k. If the precedence relation is empty, the tasks are considered to be mutually independent. An example of a task set is shown in Fig. 1.1.

* Received by the editors February 13, 1975, and in final revised form February 24, 1976.

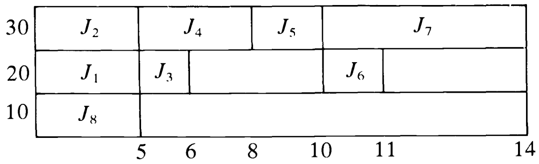
† Computer Science Department, Iowa State University, Ames, Iowa 50011

‡ Computer Science Department, Purdue University, West Lafayette, Indiana 47907

Task set		Precedence relation	
J_1: (20, 5)	J_5: (20, 2)	$J_1 < J_5$	$J_3 < J_6$
J_2: (30, 5)	J_6: (10, 1)	$J_1 < J_3$	$J_5 < J_6$
J_3: (20, 1)	J_7: (10, 4)	$J_4 < J_5$	$J_5 < J_7$
J_4: (20, 3)	J_8: (10, 5)		

FIG. 1.1. *Example of a task set and precedence relation*

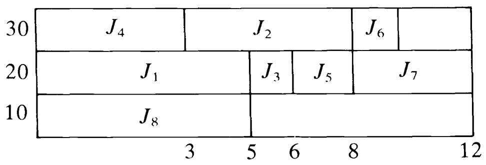
Based on J a task list, L, is formed. L is a permutation (renumbering) of the tasks in J, $J_{i_1}, J_{i_2}, \cdots, J_{i_n}$. The scheduling of task proceeds as follows: whenever P_j becomes idle, it instantaneously scans the tasks list from left to right until it finds the first task, J_i, such that $|P_j| \geqq m_i$ (the task can be contained in P_j's private memory) and all of the predecessors of J_i are completed. J_i is then removed from L and begins executing on P_j. If two or more precessors become available simultaneously we use the "tie-breaking" rule which allows the processor with the smallest index (largest memory) to proceed first.[1] Figure·1.2 shows a task list and the resulting schedule using the task set described in Fig. 1.1. A schedule is pictured as a two-dimensional Gantt chart. The size of each memory will be indicated as in Fig. 1.2.



$L = (J_2, J_1, J_4, J_5, J_3, J_7, J_6, J_8)$

FIG. 1.2. *A sample schedule*

The maximum finishing time (final completion time) used as a performance measure is considered equivalent to a measure to processor utilization. The schedule in Fig. 1.2 has a final completion time of 14. An optimal schedule is one having the minimum final completion time among all possible valid schedules. Figure 1.3 shows the task list which produces an optimal schedule for the task set presented in Fig. 1.1. Since a task list defines the resulting schedule, these two terms will be used interchangeably.



$L = (J_4, J_1, J_2, J_3, J_5, J_6, J_7, J_8)$

FIG. 1.3. *An optimal schedule*

[1] It may be more intuitively appealing to use the reverse rule that allows the processor with the largest index to proceed first. However, the worst case results developed in this paper are independent of the tie-breaking rule.

It should be observed that, for a nonpreemptive scheduling strategy, this model subsumes as a special case the model considered in [1]. Thus, the problem of finding an optimal nonpreemptive strategy for this model is polynomial complete.

The model may be interpreted in three ways. First, we may consider a network of minicomputers (uniprocessors) composed of processors with identical capabilities, but where the private memories associated with the processors may be of different sizes. The idea of a network of small computers collaborating to produce the computing power of a single, large machine has recently gained some attention [5], [13], [26]. The network approach offers advantages in terms of reliability and flexibility over a single machine and also makes possible an incremental acquisition of computing power. The second interpretation of the model is of a partitioned memory operating system such as the IBM 360-OS/MFT [12] or SINGER 10. In these cases, the private memories correspond to fixed memory partitions and the processors to the degree of multiprogramming which is supported. Although such systems may not be in the mainstream of operating system development, there are environments which do not require sophisticated operating system features, and for which a partitioned memory operating system represents adequate support with reduced overhead. The third interpretation is a combinatorial problem related to bin-packing [15]. The problem may be viewed as one of packing a number of two-dimensional objects into a fixed number of unequal-size bins. The questions for various packing rules is: how long must the longest bin be in order to accommodate all the objects.

Section 2 of this paper analyzes several basic scheduling algorithms by deriving worst-case bounds on their performance as compared to the optimal. The bounds are shown to be the best possible by exhibiting general examples that achieve these bounds in the limit. Section 3 considers two more sophisticated strategies for obtaining improved performance over the basic strategies discussed in § 2. Section 4 shows an efficient, optimal preemptive strategy for mutually independent tasks. Finally, in § 5, a summary of simulation studies is presented. These results suggest that the worst-case bounds are good indicators of the average performance of scheduling strategies.

2. Analysis of basic scheduling heuristics. The investigation of this model begins with an analysis of an arbitrary demand scheduling strategy. Such a strategy constructs task lists, hence schedules, from an arbitrary (random) ordering of the task set. The worst-case bound for the arbitrary demand strategy bounds all other strategies since the arbitrary task list ordering can be chosen to be the same ordering produced by any other strategy.

If T_{DS} represents the final completion time of an arbitrary demand schedule, the following theorem establishes a bound on the ratio between T_{DS} and the length of the optimal schedule, T_{MIN}.

THEOREM 2.1.

$$T_{\mathrm{DS}}/T_{\mathrm{MIN}} \leqq m. \tag{2.1}$$

Proof.

$$T_{\mathrm{DS}} \leqq \sum_{i=1}^{n} t_i \leqq m T_{\mathrm{MIN}}. \tag{2.2}$$

The left inequality is true since at least one processor must be active at every time during the schedule. The right inequality holds since the total time available in the optimal schedule must be large enough to contain the total time in J.

The bound of Theorem 2.1 can be reached by the following general task set:

$$\begin{aligned} &|P_1|=2, && |P_i|=1 \quad \text{for } 2\leqq i\leqq m,\\ &J_i^*: (2, \varepsilon), && 1\leqq i\leqq m,\\ &J_{i,1}: (1, 1), && 1\leqq i\leqq m,\\ &J_{i,j}: (1, \varepsilon), && 1\leqq i\leqq m, \quad 2\leqq j\leqq m. \end{aligned}$$

The task execution sequence is constrained by the precedence relation:

$$J_i^* < J_{i,j}, \qquad 1\leqq i\leqq m, \quad 2\leqq j\leqq m.$$

The worst-case schedule is obtained from the task list:

$$L = (J_{1,1}, J_{1,2}, \cdots, J_{1,m}, J_{2,1}, \cdots, J_{m,m}, J_1^*, \cdots, J_m^*).$$

In the worst-case schedule the $2m$ tasks, $J_1^*, \cdots, J_m^*, J_{1,m}, \cdots, J_{m,m}$, are all executed by P_1, finishing at time $m+m\varepsilon$. The optimal schedule results from the task list:

$$L' = (J_1^*, \cdots, J_m^*, J_{1,2}, \cdots, J_{1,m}, J_{2,2}, \cdots, J_{m,m}, J_{1,1}, J_{2,1}, \cdots, J_{m,1}).$$

In this schedule tasks $J_1^*, \cdots, J_m^*$ are executed by P_1 in sequence overlapping the execution of the block of tasks, $J_{1,1}, \cdots, J_{1,m-1}, \cdots, J_{m,m-1}$. Finally, the unit time tasks are executed in parallel finishing at time $1+m\varepsilon$. As $\varepsilon \to 0$ the ratio of these finishing times is m.

The fact that Theorem 2.1 is best possible depends heavily on the use of precedence constraints to delay the scheduling of tasks until a single, assigned resource becomes available. The impact of the precedence constraints is revealed by the next theorem which analyzes an arbitrary demand strategy for mutually independent tasks (the precedence relation is empty).

Because of the scarcity of known results involving precedence constraints (together with arbitrary task times and an arbitrary number of processors) the assumption of mutually independent tasks will be used throughout the remainder of this paper.

THEOREM 2.2. *For mutually independent tasks,*

$$T_{\text{DS}}/T_{\text{MIN}} \leqq n+1+\frac{i}{2^n} \leqq 1+\log_2(m), \tag{2.3}$$

where n and i are chosen so that $m = 2^n + i$ and $i < 2^n$. This bound is the best possible.

Proof. The proof is by contradiction. Assume for a given system there exists a list of tasks whose demand schedule has length $T_{\text{DS}} > (n+1+i/2^n)T_{\text{MIN}}$ where $m = 2^n + i$. We shall analyze the structure of such a schedule and derive several conditions that must be satisfied by the given tasks.

The demand schedule is divided from right-to-left into $n+2$ intervals $\{d_1, d_2, \cdots, d_{n+2}\}$. Each of the first $n+1$ intervals has length T_{MIN}, covering the time period $[T_{\text{DS}}-(n+1)T_{\text{MIN}}, T_{\text{DS}}]$ in the schedule. The ith interval is the half open time period $[T_{\mathrm{DS}}-iT_{\mathrm{MIN}}, T_{\mathrm{DS}}-(i-1)T_{\mathrm{MIN}})$. The final interval, d_{n+2}, covers the time period $[0, T_{\mathrm{DS}}-(n+1)T_{\mathrm{MIN}})$. With the ith task we have associated the label, n_j, which is the number of memories into which the task will fit.

The following notations are used in the proof:

a_i is the number of processors containing no idle time in the interval d_i;

x_i is the largest label of any task beginning in the interval d_i;

T_i is the total task time in the closed interval $[T_{\mathrm{DS}}-iT_{\mathrm{MIN}}, T_{\mathrm{DS}}]$.

Three relations apply to the interval d_1:

(2.4) $$a_1 \geqq 1,$$

(2.5) $$T_1 \geqq T_{\mathrm{MIN}},$$

and

(2.6) $$x_1 \geqq 1.$$

Relation (2.4) is true since every interval must have at least one processor which is continuously active. This also implies (2.5) since the length of d_1 is T_{MIN}. Since no task can have length greater than T_{MIN}, at least one task begins in d_1. Relation (2.6) follows immediately, as no task has label less than 1.

We now derive three recursive formulas defining a_i, T_i, and x_i for $2 \leqq i \leqq n+1$. The first of these is:

(2.7) $$a_i \geqq x_{i-1}.$$

This can be seen since, if there is some task in the interval d_{i-1} which can fit into the x_{i-1} largest memories then the corresponding processors cannot be idle prior to the beginning of the task. In particular, these processors cannot be idle during the interval d_i.

The second formula follows immediately from the definitions of a_i and T_i:

(2.8) $$T_i \geqq T_{i-1} + a_i T_{\mathrm{MIN}}.$$

By expansion,

(2.9) $$T_i \geqq a_i T_{\mathrm{MIN}} + a_{i-1} T_{\mathrm{MIN}} + \cdots + a_1 T_{\mathrm{MIN}},$$

or

(2.10) $$T_i \geqq T_{\mathrm{MIN}} \sum_{j=1}^{i} a_j$$

which is an expression to be used in proving the third formula.

Also needed in proving the third formula is the inequality:

(2.11) $$x_i \geqq x_{i-1}.$$

If this inequality did not hold, there would exist a task in d_{i-1} which could fit into some memory which no task beginning in d_i could fit into. This would imply that the given task could have begun earlier, violating the demand scheduling principle. This shows that the inequality must hold in any valid schedule.

Consider the interval $[T_{\mathrm{DS}}-iT_{\mathrm{MIN}}, T_{\mathrm{DS}}]$. By definition there are T_i units of task time contained in that interval. Inequality (2.11) implies that the largest number of memories which can contain the tasks in this interval is x_i. By the definition of T_{MIN} we must have:

$$T_{\mathrm{MIN}} \geqq T_i/x_i, \tag{2.12}$$

or

$$x_i \geqq T_i/T_{\mathrm{MIN}}. \tag{2.13}$$

Substituting from (2.10), we finally obtain

$$x_i \geqq T_{\mathrm{MIN}} \sum_{j=1}^{i} a_j/T_{\mathrm{MIN}} = \sum_{j=1}^{i} a_j. \tag{2.14}$$

Thus we obtain the third formula:

$$x_i \geqq \sum_{j=1}^{i} a_j. \tag{2.15}$$

We thus have three recursive formulas given by (2.7), (2.8) and (2.15) with the initial conditions given by (2.4)–(2.6). It is easy to show by induction that formulas (2.7), (2.8) and (2.15) yield the following:

$$a_i \geqq 2^{i-2}, \tag{2.16}$$

$$x_i \geqq 2^{i-1}, \tag{2.17}$$

and

$$T_i \geqq 2^{i-1} T_{\mathrm{MIN}}. \tag{2.18}$$

Now, in the large interval $[T_{\mathrm{DS}} - (n+1)T_{\mathrm{MIN}}, T_{\mathrm{DS}}]$, T_{N+1} total task time has been used. By the induction proof above, $T_{n+1} \geqq 2^n T_{\mathrm{MIN}}$. This leaves at most iT_{MIN} task time to be used in d_{n+2}, since the total task time cannot exceed $mT_{\mathrm{MIN}} = (2^n + i)T_{\mathrm{MIN}}$. In the interval d_{n+1} there begins a task with label at least 2^n, since $x_{n+1} \geqq 2^n$. Thus, in the interval d_{n+2} at least 2^n processors contain no idle time. This implies that the length of the interval d_{n+2} is at most $iT_{\mathrm{MIN}}/2^n$. Since each of the other $n+1$ intervals is at length T_{MIN} we obtain

$$T_{\mathrm{DS}} \leqq (n+1)T_{\mathrm{MIN}} + iT_{\mathrm{MIN}}/2^n. \tag{2.19}$$

This contradicts our original assumption that the length of the demand schedule exceeded the length $(n+1+i/2^n)T_{\mathrm{MIN}}$. Hence, the original assumption is incorrect and the first part of the theorem is established.

The second inequality,

$$n+1+\frac{i}{2^n} \leqq 1 + \log_2(m), \tag{2.20}$$

may be shown directly by mathematical analysis. It is of interest to know that the function $n+1+i/2^n$ is a piecewise linear function lying below the curve $1 + \log_2(m)$ and touching it at those points where $m = 2^n$ for some integer n.

As with the preceding theorem we would like to demonstrate that the bound of Theorem 2.2 is best possible by exhibiting a method for constructing task sets which achieve this bound at arbitrary values of m. The construction of this family of examples is directly related to the analysis of the scheduling strategies considered in Theorem 2.3. Therefore, the examples reaching the bound of Theorem 2.2 are contained in the proof of Theorem 2.3.

A comparison of Theorems 2.1 and 2.2 reveals that while the assumption of mutual independence improves the worst-case performance, both bounds increase without limit as m increases. This means that the guaranteed performance level continues to decline as the number of processors increases. Even for a fixed number of processors these bounds compare unfavorably with the bounds derived for related models [9], [19].

One natural method to obtain improved performance over the arbitrary demand strategy is to adopt a heuristic rule for ordering the task list. Such a heuristic attempts to place the most critical tasks at the beginning of the task list (assign them higher priority). We will analyze four basic heuristics. They are:

—smallest time first (STF)
—smallest memory first (SMF)
—largest time first (LTF)
—largest memory first (LMF)

In previous work, the smallest-first strategies usually lead to desirable mean-finishing time properties while the largest-first strategies possess better final completion time properties. The simplicity of these heuristics guarantees that they are easy to implement, possible to analyze and modest in the time required to produce a task list (schedule).

The next theorem analyzes the smallest-first strategies and also contains the examples illustrating the reachability of Theorem 2.2.

THEOREM 2.3.

$$\left.\begin{matrix} T_{\mathrm{SMF}}/T_{\mathrm{MIN}} \\ T_{\mathrm{STF}}/T_{\mathrm{MIN}} \end{matrix}\right\} \leqq n+1+i/2^n \leqq 1+\log_2(m), \tag{2.21}$$

where $m = 2^n + i$ and $i < 2^n$ for integers n and i, and this is the best possible bound.

Proof. The two functions have already been shown to be upper bounds by Theorem 2.2. The analysis to show how these bounds can be achieved is divided into three cases. Case 1 demonstrates that the bound $1+\log_2(m)$ can be reached when $m = 2^n$ and $i = 0$. This case establishes the important characteristic of the smallest-first strategies. Case 2 demonstrates that the bound of $n+1+i/2^n$ is achieved when $m = 2^n$ and $i = 2^k$ for some integer k. Finally, Case 3 demonstrates that when $i > 0$ but not a power of 2, a lower bound "close" to $n+1+i/2^n$ is at least obtainable.

Only the first of these cases is presented here. The other two cases may be found in [16]. In each of the following examples it is assumed that the memory sizes are distinct (i.e., $|P_1| > |P_2| > \cdots > |P_m|$).[2]

Case 1. $m = 2^n$ and $i = 0$. Consider the task set $\{J_1, J_2, \cdots, J_m\}$ where the resource requirement of J_i is $(|P_{m-i+1}|, 1)$, for $1 \leqq i \leqq m$. The ordering $L = (J_1, J_2, \cdots, J_m)$ is easily seen to be both an SMF and an STF ordering. At the

[2] The example presented here was suggested by Garey [7], and relies on the tie-breaking rule stated in § 1. A more elaborate example using the reverse tie-breaking rule was presented in [16].

beginning of the schedule P_1 selects J_1 for execution, P_2 selects J_2, and, in general, P_k selects J_k. When $k = m/2$, P_k can execute J_k since:

$$m_k = |P_{m-k+1}| = |P_{m/2+1}| < |P_{m/2}| = |P_k|. \tag{2.22}$$

However, when $k \geqq m/2+1$, we find that:

$$m_k = |P_{m-k+1}| \geqq |P_{m-(m/2+1)-1}| \geqq |P_{m/2}| > |P_{m/2+1}| \geqq |P_k|. \tag{2.23}$$

This means that the $m/2$ processors $P_{m/2+1}, \cdots, P_m$ remain idle. The interval $[0, 1)$, then, only contains $m/2 = 2^{n-1}$ active processors.

The ordering of tasks in L forces the processors with the largest memory capacity to select from the remaining tasks those with the smallest memory requirement. This causes a progressive halving of the number of active processors as shown in Fig. 2.1. Thus, the interval $[k-1, k)$ contains only 2^{n-k} processors which are not idle. When $k = n$, there is a single active processor. At the end of this interval the total number of completed tasks is:

$$\sum_{k=1}^{n} 2^{n-k} = \sum_{k=0}^{n-1} 2^k = 2^n - 1 = m - 1. \tag{2.24}$$

The single remaining task, J_m, has a memory requirement of $|P_1|$. This task executes alone during the interval $[n, n+1)$. The final completion time of the SMF-STF schedule is then $n+1$.

An optimal schedule of length 1 can be formed by executing all m tasks in parallel, task J_k scheduled on processor P_{m-k+1}. The ratio of these completion times is $n+1 = 1+\log_2(m)$.

Cases 2 *and* 3. See [16].

The next heuristic to be considered is the largest-time-first strategy (LTF). Although a tight bound is not yet known, the following theorem shows that it must contain a logarithmic factor that increases as m increases.

THEOREM 2.4. *A lower limit on the worst-case bound of* LTF *is*[3]

$$\max\left[\frac{T_{\text{LTF}}}{T_{\text{MIN}}}\right] \geqq \sum_{i=1}^{m} \frac{1}{i} \approx \ln(m). \tag{2.25}$$

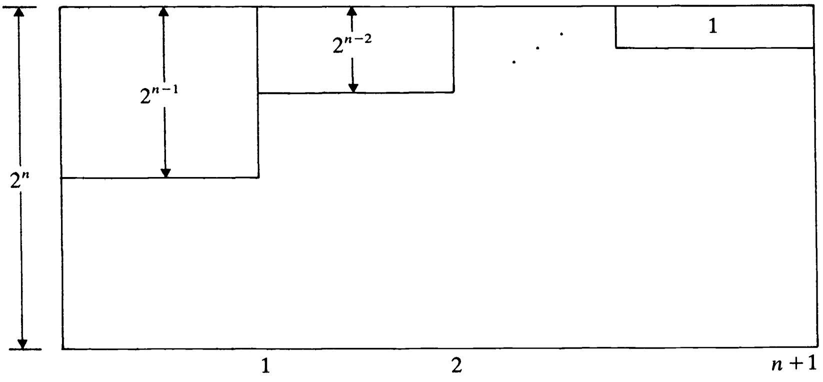


FIG. 2.1. *General Case* 1 *schedule*

[3] Slightly worse examples are known for this strategy [7]. We chose this example for simplicity.

Proof. Because a lower limit is being proven, a method will be given which achieves the bound for arbitrary m.

The general example consists of m groups of tasks. All tasks within a group are identical. In the LTF task list (and in the resulting schedule) all tasks in group $k+1$ preceed those in group k. The total time required for group k will be 1 and the memory requirement of each task will be $|P_k|$. Thus, each task in group k can only fit into k memories. The number of tasks in group k is contrived so that the tasks in group k contributes a solid block to the schedule of length $1/k$. The final schedule length will then be $\sum_{i=1}^{m} 1/i$. Finally, it will be shown that an optimal schedule of length 1 can be formed. The example, then, reaches the bound given in the theorem statement. In general, the group $m-k$, $0 \leqq k \leqq m-1$, consists of $2^k(m-k)$ tasks each with a memory requirement of $|P_k|$ and a time requirement of $1/(2^k(m-k))$. The m tasks in group m all terminate at the same time. In an inductive-like manner let us assume that the final $k+1$ tasks in group $k+1$ terminate at the same time. Hence, the k processors capable of executing the tasks in group k begin executing these tasks simultaneously. All tasks in group k are of the same length and by definition their number is divisible by k. Thus, each of the k active processors will execute the same number of tasks and finish at the same time. Group k then contributes $1/k$ to the schedule length. The appearance of the general schedule for these tasks is given in Fig. 2.2.

The total schedule length is then $\sum_{k=1}^{m} 1/k$. By construction, if processor P_k executed only tasks in group k, then all processors would terminate at time 1. This is the optimal schedule. The bound produced by this example is the bound given in the theorem.

The final heuristic to be considered is the largest-memory-first strategy (LMF). The next theorem establishes a well-behaved bound for this strategy.

THEOREM 2.5.

$$T_{\text{LMF}}/T_{\text{MIN}} \leqq 2-1/m. \tag{2.26}$$

Proof. The proof is by contradiction. Assume that there exists a counterexample to the theorem. Let J_k be the longest task in the counterexample schedule

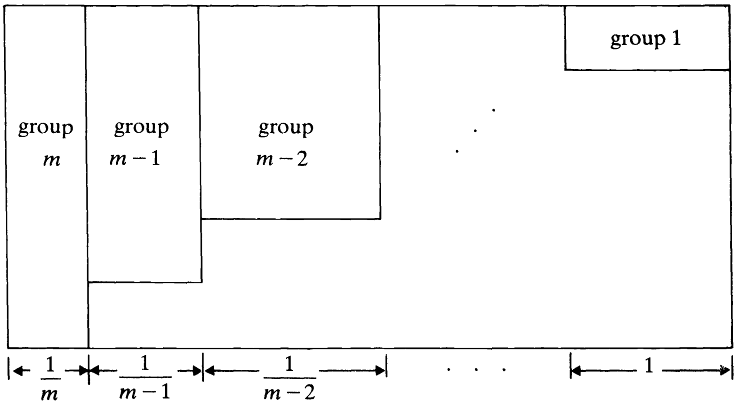


FIG. 2.2. *General* LTF *schedule*

to finish at time T_{LMF}. Let r be the number of memories which can contain J_k. Due to the LMF ordering the tasks J_i of $1 \leqq i \leqq k-1$ all have memory requirements at least as large as J_k. These tasks must be scheduled in the r largest memories prior to J_k such that none of $P_1, \cdots, P_r$ become idle prior to the beginning of J_k. This implies:

$$T_{\mathrm{LMF}} \leqq \frac{\sum_{i=1}^{k-1} t_i}{r} + t_k. \tag{2.27}$$

The k tasks with largest-memory requirements are constrained to be scheduled only on the r largest memories in any schedule. The optimal schedule for the entire task set is, then, at least as long as the optimal schedule for the k tasks using only the largest r memories. So

$$T_{\mathrm{MIN}} \geqq \frac{\sum_{i=1}^{k} t_i}{r}. \tag{2.28}$$

Combining (2.27) and (2.28) with the contradictory hypothesis and rearranging, we have

$$2 - \frac{1}{m} < \frac{\frac{\sum_{i=1}^{k} t_i}{r} + t_k(1-1/r)}{T_{\mathrm{MIN}}}, \tag{2.29}$$

which simplifies to

$$T_{\mathrm{MIN}}(1-1/m) < t_k(1-1/r). \tag{2.30}$$

Since $r \leqq m$, this last line implies that $T_{\mathrm{MIN}} < t_k$, which is a contradiction. Theorem 2.5 represents the best possible bound on the LMF strategy as shown by the following general task set:

$$J_{i,j}\colon (|P_m|, 1), \qquad 1 \leqq i \leqq m-1, \quad 1 \leqq j \leqq m,$$
$$J^*\colon (|P_m| - \varepsilon, m).$$

The LMF task list is given by:

$$L = (J_{1,1}, J_{1,2}, \cdots, J_{1,m}, J_{2,1}, J_{2,2}, \cdots, J_{2,m}, \cdots, J_{m-1,m}, J^*).$$

In the LMF schedules the first $m(m-1)$ tasks form a solid block ending at time $m-1$. Task J^*, of length m, then completes at time $T_{\mathrm{LMF}} = 2m-1$. In the optimal schedule task J^* begins at time 0 and the remaining $m(m-1)$ tasks are distributed among the other $m-1$ processors. All processors finish at time $T_{\mathrm{MIN}} = m$. So $T_{\mathrm{LMF}}/T_{\mathrm{MIN}} = (2m-1)/m = 2 - 1/m$.

3. Analysis of improved scheduling strategies. This section presents the analysis of two scheduling algorithms which attempt to improve on the LMF bound by expending more time in constructing the task ordering.

Consider two tasks, J_i and J_k with $n_i = n_k$. Given a choice between these two tasks, past results in similar models [9], [19] suggest that the task with the larger time requirement should be scheduled earlier in order to reduce the final completion time.

An LMTF-ordered task list is defined by the following principle: task J_i preceeds task J_k if: (i) $n_i < n_k$ or (ii) $n_i = n_k$ and $t_i \geqq t_k$.

The analysis of the LMTF strategy is presented in two parts. The first part develops the bound for $m \geqq 3$.

THEOREM 3.1. *For* $m \geqq 3$,

$$T_{\text{LMTF}}/T_{\text{MIN}} \leqq 2 - 1/(m-1). \tag{3.1}$$

Proof. Renumber the tasks in J so that $(J_1, J_2, \cdots, J_n)$ is an LMTF ordering. Let J_k be the longest task to finish at time T_{LMTF}. Define $r = n_k$. Consider the truncated task list $(J_1, J_2, \cdots, J_k)$. Let T'_{LMTF} denote the completion time of this truncated list on $P_1, P_2, \cdots, P_r$, and let T'_{MIN} denote the corresponding optimal completion time. Clearly, $T'_{\text{LMTF}} = T_{\text{LMTF}}$. By the definition of an LMTF order, $n_i \geqq n_k$, for $i < k$. Thus, $J_1, \cdots, J_k$ can only be executed by $P_1, \cdots, P_r$. This implies that $T'_{\text{MIN}} \leqq T_{\text{MIN}}$. The conclusion of this argument is that

$$T_{\text{LMTF}}/T_{\text{MIN}} \leqq T'_{\text{LMTF}}/T'_{\text{MIN}}. \tag{3.2}$$

The analaysis, then, need only be concerned with the truncated task list and $P_1, \cdots, P_r$.

Now let y be the time in the LMTF schedule at which J_k begins. Since $m_k \leqq |P_r|$, no processor can become idle prior to the beginning of J_k. In addition, the scheduling algorithm requires that all tasks executing on P_r have at least as long a time requirement as J_k. Thus, the total task time on P_r is at least t_k. Figure 3.1 illustrates the form of the LMTF schedule using these facts.

By examining the total task time we find

$$(r-1)y + 2t_k \leqq \sum_{i=1}^{k} t_i \leqq rT'_{\text{MIN}}. \tag{3.3}$$

Rearranging terms yields the following definition for y:

$$y \leqq \frac{rT'_{\text{MIN}} - 2t_k}{r-1}. \tag{3.4}$$

By the definition of y it is also true that:

$$T'_{\text{LMTF}} \leqq y + t_k. \tag{3.5}$$

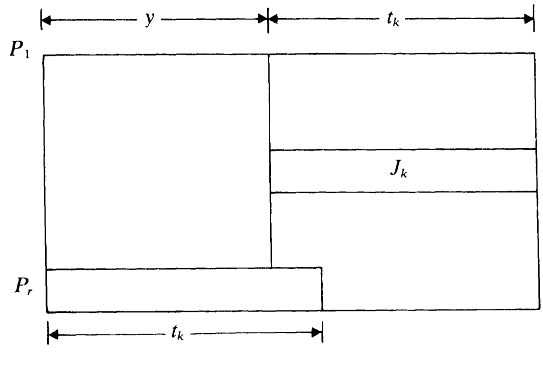


FIG. 3.1. *General form of the* LMTF *schedule*

Substituting for y produces:

$$T'_{\text{LMTF}} \leqq \frac{rT'_{\text{MIN}}-2t_k}{r-1}+t_k=\frac{rT'_{\text{MIN}}+(r-3)t_k}{r-1}. \tag{3.6}$$

The case $r=2$ is covered in Theorem 3.2. Assume $r \geqq 3$. Since $t_k \leqq T'_{\text{MIN}}$ we obtain

$$T'_{\text{LMTF}} \leqq \frac{rT'_{\text{MIN}}+(r-3)T'_{\text{MIN}}}{r-1} \leqq \left(2-\frac{1}{r-1}\right)T'_{\text{MIN}}. \tag{3.7}$$

Finally, since $m \geqq r$ this last line becomes

$$T'_{\text{LMTF}}/T'_{\text{MIN}} \leqq 2-1/(m-1). \tag{3.8}$$

The general task set which illustrates that the bound of Theorem 3.1 is the best possible is defined by:

$$\begin{aligned} &J_i\colon \left(|P_{m-1}|, \frac{m-2}{m-1}\right), && 1 \leqq i \leqq m-2,\\ &J_i\colon \left(|P_{m-1}|, \frac{1}{(m-1)}\right), && m-1 \leqq i \leqq 2(m-2),\\ &J_i\colon (|P_m|, 1), && 2(m-2)+1 \leqq i \leqq 2(m-2)+2. \end{aligned}$$

Clearly, the task list $(J_1, J_2, \cdots, J_{2m-2})$ is an LMTF ordering. The schedule resulting from this task list is shown in Fig. 3.2. The corresponding optimal schedule is given in Fig. 3.3. The ratio of the completion times of these two schedules is the value predicted by Theorem 3.1.

The situation for the LMTF strategy is slightly different when there are only two processors ($m=2$). This also covers the case of $r=2$ for the previous theorem.

THEOREM 3.2. *For* $m=2$,

$$T_{\text{LMTF}}/T_{\text{MIN}} \leqq 5/4 \tag{3.9}$$

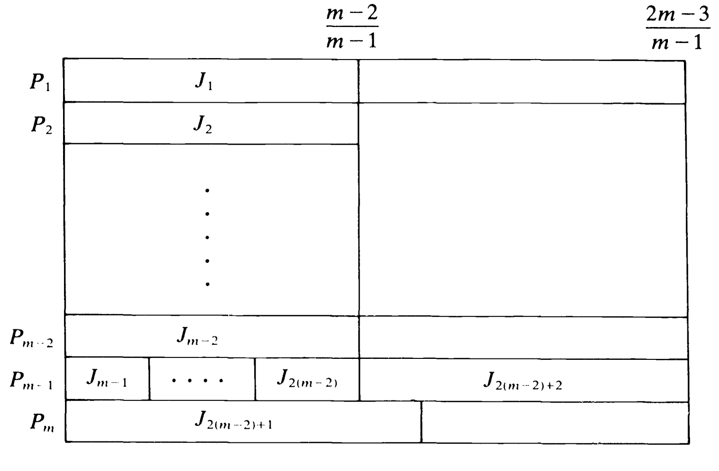


FIG. 3.2. *Worst-case schedule illustrating Theorem* 3.1

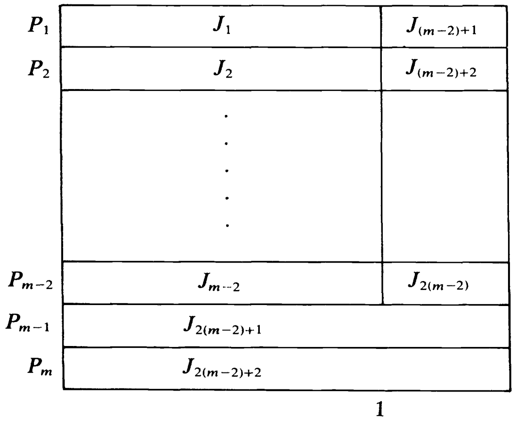


FIG. 3.3. *Optimal schedule illustrating Theorem* 3.1

and there exist task sets which realize this bound.

Proof. See [16].

The LMTF performance bound is not significantly better than that of LMF itself. Another possible technique for achieving such improved performance, called a two-dimensional strategy, is suggested by previous research results [14], [19], [20]. In general, such a strategy iteratively selects a final completion time and, by means of a placement policy, attempts to construct a schedule completing at the chosen time. Improved performance can be achieved if the placement policy is effective and the iteration scheme tends toward minimal-length schedules.

In the two-dimensional strategy considered below the placement policy will be used which selects tasks from an LMF ordering and assigns the next task to the processor with the largest available memory such that the task's finishing time does not exceed the current schedule completion time. More formally, assuming that the tasks are in LMF order and the current schedule length is T, the placement policy is defined as:

Step 1. Set $i = 1$; set $T_1 = T_2 = \cdots, = T_m = 0$.

Step 2. Let j be the smallest integer such that $T_j + t_i \leqq T$ and $m_i \leqq |P_j|$. If such a j exists then proceed to Step 3; otherwise, indicate failure.

Step 3. Set $T_j = T_j + t_i$; set $i = i + 1$;

Step 4. If $i \leqq n$, then return to Step 2; otherwise indicate success.

The iteration scheme to be analyzed is an incremental method (INC). In this method, u is defined as the greatest common divisor of all tasks in the system. Beginning with a lower bound, L_0, on the optimal schedule length, the target final completion time is successively incremented by u until the placement policy is successful in creating a schedule.

An example of this strategy is seen by examining the task set and system described in Fig. 3.4. If the incremental method chooses $L_0 = 9(= \sum t_i/m)$ and $u = 1$, then it will successively try schedules of lengths 9 and 10 before succeeding at $T_{\text{INC}} = 10$. This compares to $T_{\text{MIN}} = 9$ and $T_{\text{LMF}} = 12$.

Task set
J_1: (30, 1) J_4: (10, 5)
J_2: (10, 4) J_5: (10, 5)
J_3: (10,4) J_6: (5, 8)
Processing system: $m=3$, $|P_1|=30$, $|P_2|=20$, $|P_3|=10$.

FIG. 3.4. *A task set and system*

The final schedules produced by the two-dimensional algorithm may violate the demand principle since tasks are assigned to each processor regardless of the earlier availability of another processor, so long as the total schedule length is not exceeded. However, the demand principle may be satisfied by allowing some tasks to start earlier in previously idle periods. The result of this rearrangement is a valid schedule with length no longer than the final schedule produced by the algorithm.

Formally, the INC algorithm is stated as:

Step 1. Order the task list on a largest-memory-first basis. Set u equal to the maximum divisor of the t_i's.

Step 2. Select an optimistic value for the final completion time

$$T_{\mathrm{INC}}=\left[\max\left(\max(t_i),\ \sum_{i=1}^{n} t_i/m,\ \sum_{i=1}^{n} m_it_i/M\right)\right]$$

where M is the total available memory.

Step 3. Apply the placement policy to a schedule of length T_{INC}. If this succeeds then stop, otherwise set $T_{\mathrm{INC}}=T_{\mathrm{INC}}+u$ and repeat Step 2.

The optimistic value chosen as the initial value for T_{INC} is actually a lower bound on the optimal schedule length. This lower bound is the smallest integer greater than the maximum of three quantities. These three quantities correspond to the length of the longest single task, the completion time if all processors are busy all the time, and the completion time if all the memory is used all the time. The optimal schedule cannot be shorter than the maximum of these three quantities. Analysis of this algorithm shows that it also produces only a slight improvement over the LMF strategy.

THEOREM 3.3.

$$T_{\mathrm{INC}}/T_{\mathrm{MIN}}\leqq 2-2/(m+1). \tag{3.10}$$

Proof. For task set J suppose the INC algorithm stops with final completion time T_{INC}. If the algorithm succeeded in placing all the tasks on its first attempt, then the schedule is optimal. However, if several iterations were required to produce the final schedule, then consider the partial schedule, S, formed by the algorithm when attempting a completion time of $T_{\mathrm{INC}}-u$. Let $J_1, J_2, \cdots, J_{k-1}$ be successfully placed in S and J_k be the first task which cannot be placed using this estimate for the schedule length. Define r as the number of memories which can contain J_k. Since an LMF task list is being used, tasks $J_1, \cdots, J_k$ can only be placed on $P_1, \cdots, P_r$ in any schedule. Thus, if T'_{MIN} represents the optimal schedule for $J_1, \cdots, J_k$ on $P_1, \cdots, P_r$, then it follows that

$$T_{\mathrm{INC}}/T_{\mathrm{MIN}}\leqq T_{\mathrm{INC}}/T'_{\mathrm{MIN}}. \tag{3.11}$$

The analysis, then, will only consider the case of the first k tasks and the first r processors.

Define T_i, $1 \leqq i \leqq r$, as the time in S when P_i becomes idle. The operation of the algorithm requires that for $1 \leqq i, j \leqq r$ and $i \neq j$:

$$T_i + T_j > T_{\text{INC}} - u. \tag{3.12}$$

If this were not true then for some i and j the algorithm would have successfully scheduled all tasks contributing to T_j on P_i, which is a contradiction. In addition, since J_k cannot be placed in S, it follows that for $1 \leqq i \leqq r$:

$$T_i + t_k > T_{\text{INC}} - u. \tag{3.13}$$

By the definition of u, these last two conditions are equivalent to:

$$T_i + T_j \geqq T_{\text{INC}} \tag{3.14}$$

and

$$T_i + t_k \geqq T_{\text{INC}}, \tag{3.15}$$

for $1 \leqq i$, $j \leqq r$ and $i \neq j$. Equation (3.14) implies that there is at most one T_x, $1 \leqq x \leqq r$, such that

$$T_x < T_{\text{INC}}/2. \tag{3.16}$$

Hence, for $1 \leqq i \leqq r$ and $i \neq x$,

$$T_i \geqq T_{\text{INC}}/2, \tag{3.17}$$

or

$$\sum_{i=1}^{r} T_i \geqq \frac{(r-1)T_{\text{INC}}}{2}. \tag{3.18}$$

By (3.15) it is apparent that

$$T_x + t_k \geqq T_{\text{INC}}. \tag{3.19}$$

Examining the total task times reveals

$$\sum_{i=1}^{k} t_i \geqq \sum_{\substack{i=1 \\ i \neq x}}^{r} T_i + (T_x + t_k). \tag{3.20}$$

Using (3.18) and (3.19) in (3.20) we conclude that

$$\sum_{i=1}^{k} t_i \geqq (r-1)T_{\text{INC}}/2 + T_{\text{INC}} \geqq (r+1)T_{\text{INC}}/2. \tag{3.21}$$

By definition $T'_{\text{MIN}} \geqq \sum_{i=1}^{k} t_i/r$. Thus,

$$T'_{\text{MIN}} \geqq (r+1)T_{\text{INC}}/(2r). \tag{3.22}$$

Rearranging terms yields

$$T_{\text{INC}}/T'_{\text{MIN}} \leqq 2r/(r+1) = 2 - 2/(r+1). \tag{3.23}$$

Since $r \leqq m$, we finally obtain

$$T_{\mathrm{INC}}/T'_{\mathrm{MIN}} \leqq 2-2/(m+1). \tag{3.24}$$

This bound can be reached for arbitrary m by the following general task set:

$$J_i\colon (|P_i|, 1), \qquad 1 \leqq i \leqq m,$$
$$J_{m+1}\colon (|P_m|, m), \quad 1 \leqq i \leqq m.$$

The final completion time of the schedule produced by INC is $2m$. The optimal schedule finishes at time $m+1$. The ratio of these times is $2-2/(m+1)$.

The bounds derived in this section indicate that two more sophisticated scheduling techniques cannot perform significantly better than the LMF strategy. It may be conjectured that the inherent complexity of the model precludes such a possibility.

4. Preemptive scheduling. The results of the previous section indicate that significantly improved scheduling performance may only be possible by enhancing the basic capability of the model. One standard improvement is the addition of a preempt-resume feature which allows an executing task to be suspended and removed from the memory of the processor to which it was assigned. The processor is then assigned to execute another task from the task list or to resume some previously preempted task. At some later time the preempted task resumes its execution on a possibly different processor constrained only by the task's memory requirement. It is assumed here, as in [20], [22] and [23], that the preempt and resume operations are instantaneous and do not affect the computation performed by the task.

There are several conditions which must be observed in constructing valid preemptive schedules. First, the total execution time of a task, over all the processors to which it has been assigned, must satisfy the task's time requirement. Second, no two processors can ever simultaneously execute parts of the same task. Lastly, the demand principle must still be observed. No processor is allowed to remain idle if there is a task in the task list or in a preempted state which it could execute. A preemptive, two-dimensional algorithm P2D, is presented for constructing such schedules. This algorithm is an extension of [24].

The preemptive capability allows the optimal final completion time for a given task set to be calculated exactly. This computation requires the following notation. Let F_i be the set of all tasks which, because of their memory requirements, can only be scheduled on $P_1, P_2, \cdots, P_i$. Thus, $F_1 \subseteq F_2 \subseteq \cdots \subseteq F_m = J$. Let X_i be the sum of all the task times in F_i (if F_i is empty, then $X_i = 0$).

Using this notation, the P2D algorithm is formally defined by:

Step 1. Order the task list on an LMF basis.

Step 2. Compute the values of F_i and X_i, for $1 \leqq i \leqq m$, and set $T_X = \max_i (X_i/i)$, $t_{\max} = \max_i (t_i)$, and $T_{\mathrm{P2D}} = \max(t_{\max}, T_X)$. Set $j = 1$.

Step 3. (a) If all tasks have been scheduled, then stop. Otherwise, select the next task, J_k, from the task list and proceed to Step 3(b).

(b) If the placement of J_k on P_j does not cause the completion time of the task to exceed T_{P2D}, then schedule J_k on P_j and return to Step 3(a). If T_{P2D} is exceeded, proceed to Step 3(c).

(c) Suppose the placement of J_k on P_j causes t_a (of J_k's total time requirement, t_k) to occur before T_{P2D} and t_b after T_{P2D}. Schedule t_a of J_k on P_j and t_b on P_{j+1}. Increment j by 1 and return to Step 3(a).

An example of the P2D strategy is shown in Fig. 4.1.

Task set

J_1: (30, 4)

J_2: (20, 4)

J_3: (20, 5)

J_4: (20, 4)

J_5: (20, 1)

J_6: (20, 2)

J_7: (10, 4)

$F_1=\{J_1\}$ $\quad X_1=4$

$F_2=\{J_1, J_2, \cdots, J_6\}$ $\quad X_2=20$

$F_3=\{J_1, J_2, \cdots, J_7\}$ $\quad X_3=24$

$T_X=\max_i\{X_i/i\}=\max\{4, 10, 8\}=10$

$t_{\max}=\max_i\{t_i\}=5$

$T_{\text{P2D}}=\max\{t_{\max}, T_X\}=\max\{5, 10\}=10$

Schedule

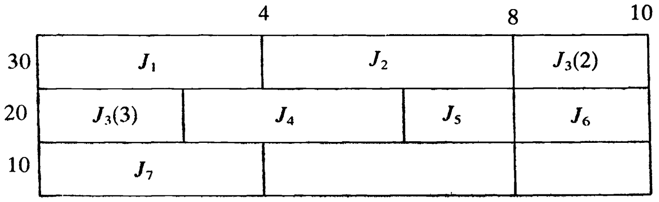


FIG. 4.1. *Example of the* P2D *strategy*

To show that the P2D strategy generates valid schedules we need only examine the placement of tasks as determined by Step 3(c) of the algorithm. Each task, J_k, is preempted at most once into a portion of length t_a and one of length t_b. Since $T_{\text{P2D}} \geqq t_{\max} \geqq t_k = t_a + t_b$, we are assured that sufficient processor time has been allocated to the task and that its two parts do not overlap. The strategy also guarantees that if some part of J_k is scheduled on P_j, then $m_k \leqq P_j$. If this were not true, then the tasks being in an LMF order, we would have the following contradiction:

$$X_k/k > T_{\text{P2D}} \geqq T_X = \max_i (X_i/i).$$

Since T_{P2D} is clearly a lower bound on T_{MIN}, we have the following:

THEOREM 4.1. *The* P2D *strategy is optimal.*

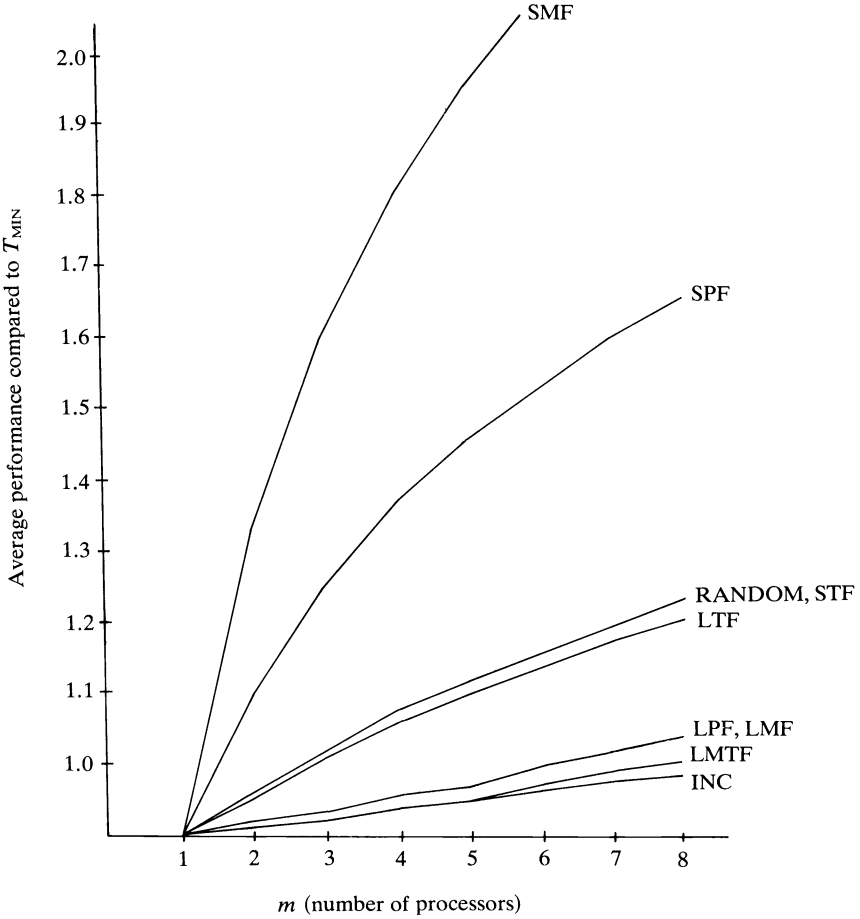


FIG. 5.1. *A sample result*

The P2D algorithm is computationally efficient. Step 1 (sorting) can be performed in the order of $n \log n$ steps, where n is the number of tasks (see, for example, [18]). Once the tasks are sorted in LMF order the computation of the F_i's and the X_i's can be performed in the order of n steps. The determination of the processor assignments can also be carried out in the order of n steps. Thus, $O(n \log n)$ steps are required to produce the final schedule.

This algorithm also requires few preemptions to achieve the optimal schedule. In fact, it never requires more than $m-1$ preemptions, since preemptions occur only at the end of the assignments for the first $m-1$ processors.

5. Simulation results. The scheduling strategies considered in §§ 2 and 3 were evaluated by bounding their worst-case performance. This method of analysis is applicable to systems where a guaranteed level of performance must be provided. Such systems arise when dealing with critical, real-time events. A contrasting type of analysis evaluates the performance of a scheduling strategy by its expected (average, mean) behavior. In systems where there are no critical deadlines, the expected performance of a strategy may be a more meaningful measure than the worst-case bound. Aside from the obvious fact that the worst-case bound limits the expected performance, the literature contains little evidence relating these two measures of scheduling performance.

To examine the relationship between expected and worst-case behavior, a computer program was written that simulated the model of computation and evaluated the various scheduling strategies by computing the maximum finishing times for different task sets to determine the expected performance. Results on the correlation (or lack of) between the two measures should be of value in the continuing effort to understand the underlying behavior of multiprocessor systems.

A number of parameters are used to define the simulation experiments. These parameters describe the processing system (the number of processors, the size of the private memories) and the characteristics of a randomly generated task set (number of tasks in the task set, distribution functions governing the selection of time and memory requirements). Once these parameters are determined a number of trials (samples) are made. Each trial constructs a random task set according to the parameters of the experiment. This task set, together with the processing system defined by the experiment, is presented to each of the seven scheduling strategies (RANDOM, SMF, STF, LMF, LTF, LMTF and INC) and the final completion time of each strategy is calculated. In addition, the SPF and LPF strategies[4] are also included for comparison. Since it is impractical to determine the optimal completion time for an arbtirary task set, an estimate of the optimal will be used. This estimate is the length of the optimal preemptıve schedule which is efficiently derived by the P2D strategy presented in § 4 (see Theorem 4.1). At the end of each trial, statistics are collected on the ratio of the completion time of each strategy as compared to the estimated optimal completion time. As successive trials are performed the mean of each statistic should converge to the expected completion time ratio of the corresponding strategy for the given probability distribution on tasks. In order to determine the number of trials which must be made to obtain meaningful results a 95 percent confidence interval is calculated for each statistic according to the method described in [21]. In statistical terms this means that the probability of the true means of the statistic (i.e., the expected performance of the strategy) lying within the calculated confidence interval is 0.95. Two strategies will be ranked according to their computed mean performance only when their respective 95 percent confidence intervals are nonoverlapping. In case the confidence intervals do overlap the strategies will be considered identical.

A sample result of the simulation experiments is shown in Fig. 5.1. The tasks in this experiment were generated by selecting the time requirement from a uniform distribution in the range [0, 100] and the memory requirements from a uniform distribution in the range $[25, 25+5(m-1)]$. The processor memory size for processor P_i was set at $25+5(m-i)$. In each trial the task set consisted of 50

[4] Smallest-product- or largest-product-first strategies use the product of time and memory requirements to order the task list for additional information. It has been shown [16] that the bounds of the LPF and SPF strategies also approximate that of Theorem 2.2.

tasks. Figure 5.1 shows the average ratio between the final completion time and the estimated optimal value as m, the number of processors, is increased. The results of this experiment are typical in that other experiments with different distributions and parameters were not essentially different.

Four conclusions can be drawn from this and other experiments. First, strategies which have identical worst-case bounds may have significantly different expected performance. This is illustrated by the four strategies SMF, STF, SPF, and RANDOM. Second, strategies with distinct, though close, bounds may possess expected performances which are indistinguishable by simulation techniques. Such is the case of the LMTF and INC strategies. It is not until $m=6$ that their confidence intervals become separated. At $m = 5$ the differences in their worst-case bound is less than 4 percent. So, it is not surprising that their expected performance should be so close as to be beyond the practical resolution of simple simulation methods. Third, the improved strategies presented in § 3 (LMTF and INC) display very good performance characteristics. For the experiments shown in Fig. 5.1 the average difference between these strategies and the estimated optimal is approximately 6 percent. It should be remembered that the estimated optimal is a lower bound on the true optimal. Thus, the strategies may actually perform better than indicated. Fourth, and finally, for the parameter used in these experiments, there is perfect agreement between the ranking by expected performance and worst-case performance.

The last conclusion may be due only to a fortunate choice of parameters defining the experiment. Different parameters are used in other tests, some with task sets whose resource requirements are not uniformly distributed. The results are reported in detail in [16].

Three observations are made as a result of these experiments. First, the worst-case rankings are not exact indicators of rankings by expected performance under all circumstances. The LPF strategy, in particular, has consistently shown that it can perform better in an expected performance sense than would be predicted by its worst-case bound. Second, the rankings show general stability under a variety of assumptions, consistency with the results of simpler models and considerable agreement with the worst-case rankings. This evidence supports the conjecture that the ranking of strategies by worst-case performance is highly correlated to the ranking by expected performance. Third, two strategies (INC and LMTF) display very desirable expected performance figures in all of the cases considered.

REFERENCES

[1] J. Bruno, E. G. Coffman, Jr. and R. Sethi, *Scheduling independent tasks to reduce mean finishing time*, Comm. ACM, 17 (1974), pp. 382–387.

[2] E. G. Coffman, Jr., *A survey of mathematical results in flow-time scheduling for computer systems*, Lecture Notes in Computer Science, Vol. 1, A. Goos and J. Hartmanis, eds., Springer-Verlag, New York, pp. 25–46.

[3] E. G. Coffman, Jr. and R. L. Graham, *Optimal scheduling for two processor systems*, Acta Informat., 1 (1972), pp. 200–213.

[4] S. A. Cook, *The complexity of theorem proving procedures*, 3rd ACM Conf. on Theory of Computing, Shaker Heights, Ohio, May 1970, pp. 151–158.

[5] D. J. FARBER, *Networks: Introduction*, Datamation, 18 (1972), No. 4, pp. 36–39.

[6] M. FUJII, T. KASAMI AND K. NINOMIYA, *Optimal sequencing of two equivalent processors*, SIAM J. Appl. Math., 17 (1969), pp. 784–789.

[7] M. R. GAREY, Private communication, Nov. 1974.

[8] M. R. GAREY AND R. L. GRAHAM, *Bounds on scheduling with limited resources*, 4th Symp. on Operating System Principles, Yorktown Heights, N.Y., Oct. 1973, pp. 104–111.

[9] R. L. GRAHAM, *Bounds on multiprocessing timing anomalies*, SIAM J. Appl. Math., 17 (1969), pp. 416–429.

[10] ———, *Bounds on multiprocessing anomalies and related packing algorithms*, Spring Joint Computer Conf., Atlantic City, N.J., 1972, pp. 205–217.

[11] T. C. HU, *Parallel sequencing and assembly line problems*, Operations Res., 9 (1961), pp. 841–848.

[12] IBM, *System/360 Operating System MFT Guide*, International Business Machines, Poughkeepsie, N.Y. GC27-6939-10, 1972.

[13] E. C. JOSEPH, *Innovations in heterogeneous and homogeneous distributed-function architectures*, Computer, March 1974, pp. 17–24.

[14] D. S. JOHNSON, *Fast allocation algorithms*, 13th Ann. Symp. on Switching and Automata Theory, College Park, Md., 1972, pp. 144–154.

[15] D. S. JOHNSON, A. DEMERS, J. D. ULLMAN, M. R. GAREY AND R. L. GRAHAM, *Worst-case performance bounds for simple one-dimensional packing algorithms*, this Journal, 3 (1974), pp. 299–325.

[16] D. G. KAFURA, *Analysis of scheduling algorithms for a model of a multiprocessing computer system*, Ph.D. thesis, Purdue Univ., W. Lafayette, Ind., 1974.

[17] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972.

[18] D. E. KNUTH, *Searching and Sorting*, Addison-Wesley, Reading, Mass., 1972.

[19] K. L. KRAUSE, *Analysis of computer scheduling with memory constraints*, Ph.D. thesis, Purdue Univ., W. Lafayette, Ind., 1973.

[20] K. L. KRAUSE, V. Y. SHEN AND H. D. SCHWETMAN, *Analysis of several task-scheduling algorithms for a model of multiprogramming computer systems*, J. Assoc. Comput. Mach, 22 (1975), pp. 522–550.

[21] E. KREYZIG, *Introductory Mathematical Statistics*, John Wiley, New York, 1970.

[22] R. R. MUNTZ AND E. G. COFFMAN, JR., *Optimal preemptive scheduling on two-processor systems*, IEEE Trans. Computers, C-18 (1969), pp. 1014–1020.

[23] ———, *Preemptive scheduling of real time tasks on multiprocessor systems*, J. Assoc. Comput. Mach., 17 (1970), pp. 324–328.

[24] R. H. ROTHKOPF, *Scheduling independent tasks on parallel processors*, Management Sci., 12 (1966), pp. 437–447.

[25] J. D. ULLMAN, *Polynomial Complete Scheduling Problems*, 4th Symp. on Operating System Principles, Yorktown Heights, N.Y., October 1973, pp. 96–101.

[26] W. A. WULF AND C. G. BELL, *C. mmp—A multi-mini-processor*, Proc. Fall Joint Computer Conf., Anaheim, Calif., 1972, pp. 766–778.

SIAM J. COMPUT.
Vol. 6, No. 1, March 1977

ADDITION REQUIREMENTS FOR RATIONAL FUNCTIONS*

DAVID G. KIRKPATRICK† AND ZVI M. KEDEM‡

Abstract. A notion of rank or independence for arbitrary sets of rational functions is developed, which bounds from below the number of additions and subtractions required of all straight-line algorithms which compute those functions. This permits a uniform derivation of the best lower bounds known for a number of familiar sets of rational functions.

The result is proved without the use of substitution arguments. This not only provides an interesting contrast to standard approaches for arithmetic lower bounds, but also allows the algebraic setting to be somewhat generalized.

Key words. additions, algorithms, analysis of algorithms, arithmetic complexity, computational complexity, dimensionality, lower bounds, matrix multiplication, optimality, polynomials, rational functions

1. Introduction. A central problem in arithmetic complexity is to take some set of rational functions and determine a lower bound on the number of arithmetic operations which is sufficient to compute the functions. It is a symptom of our lack of understanding of the interaction between multiplicative operations (i.e. multiplication and division) and additive operations (i.e. addition and subtraction), that this problem is rarely treated in its full generality. Indeed, most research in arithmetic complexity has focused on one, most often the multiplicative, operation type.

While there is evidence that multiplication is inherently more difficult than addition, this does not justify the relative lack of attention paid to additive complexity. This lack is perhaps most effectively illustrated by the fact that, prior to the work presented in this paper, there did not exist any general framework for directly proving a nontrivial lower bound on the additive complexity of the simplest of expressions, $a_1+a_2+\cdots+a_n$. While it is this lack which motivates our study, we emphasize that both our techniques and our results tend to complement as well as supplement previous work in arithmetic complexity.

We shall first present some basic definitions and a survey of related work. Section 3 describes the algebraic setting for our work, and introduces our notion of independence. This notion is developed in the context of multivariate polynomials in § 4, and extended to general rational functions in § 5. Section 6 contains applications of our central result to a number of common arithmetic expressions. Finally, in § 7, we mention a few open questions related to our work.

* Received by the editors August 4, 1975, and in revised form April 20, 1976.

† Computing Science Program, Simon Fraser University, Burnaby, British Columbia, Canada V5A 1S6. This work was done in part at the University of Toronto, with the support of the National Research Council of Canada. It was completed at Cornell University, Ithaca, New York.

‡ Mathematical Sciences Program, University of Texas at Dallas, Richardson, Texas 75080. This work was done in part at the Massachusetts Institute of Technology. It was supported in part by the National Science Foundation Grant GP 22796.

2. Related work. If F is any field and $\mathbf{a} \overset{\circ}{=} a_1, \cdots, a_n$ is a sequence of distinct indeterminates over F, then elements of $F(\mathbf{a})$[1] are called rational functions in $a_1, \cdots, a_n$ over F.

Following Winograd [11], we say that $\mathscr{A}$ is a (*rational*) *algorithm* over $(F(\mathbf{a}), G)$, computing $\psi \subseteq F(\mathbf{a})$ given G, if:

1. $G \subseteq F(\mathbf{a})$;
2. $\mathscr{A}$ is a finite sequence of pairs $(\alpha_1, \beta_1), \cdots, (\alpha_t, \beta_t)$ where either
   (a) $\alpha_i = (y)$ and $\beta_i = y$, where $y \in G$ or
   (b) $\alpha_i = (\circ, j, k)$ where $\circ \in \{+, -, \times, /\}$, $j, k < i$, and $\beta_i = \beta_j \circ \beta_k$. Furthermore, if $\circ = /$ then $\beta_k \neq 0$; and
3. $\psi \subseteq \{\beta_1, \cdots, \beta_t\}$.

If we restrict 2(b) so that $\circ \in \{+, -, \times\}$, then we say that $\mathscr{A}$ is a *polynomial algorithm* over $(F(\mathbf{a}), G)$.

We will denote by $\nu(\mathscr{A})$ the number of additions and subtractions in $\mathscr{A}$.

In pioneering the area of arithmetic complexity, Ostrowski [10], considered the problem of determining both additive and multiplicative operation requirements for computing a general nth degree polynomial. Using straightforward substitution techniques, he showed that for polynomial algorithms, n additive operations are necessary.

Techniques which apply to more general classes of functions are presented by both Belaga [1] and Winograd [11]. Belaga employs the notion of degrees of freedom for rational functions in a single indeterminate. The degree of freedom of such an expression corresponds to the number of algebraically independent coefficients.

THEOREM A (Belaga [1]). *Any algorithm which contains p additions, computes a rational expression with at most* $p+1$ *degrees of freedom.*

In a sense, Belaga's result is restricted by a dual theorem due to Motzkin [9] which relates the same degrees of freedom to multiplicative requirements. Combining the two, we find that if a function can be computed using k multiplication/divisions then the best lower bound on additions that can be obtained by degrees of freedom arguments is about $2k$. Degrees of freedom arguments provide tight bounds for polynomials whose terms are all algebraically independent, but they can easily fail to do so if this condition is relaxed. For example, the polynomial $a^2x^2 + ab^3x + b^5$ requires two additions despite having algebraically dependent coefficients.

Winograd [11] deals with the computation of functions which are linear in the indeterminates $x_1, \cdots, x_n$. A set of such functions can be expressed as a matrix-vector multiplication, $\Phi\mathbf{x}$, where Φ is a $t \times n$ matrix whose elements are drawn from some field F, and $\mathbf{x}$ denotes the vector $(x_1, \cdots, x_n)$. Winograd's theorem can be stated as

THEOREM B (Winograd [11]). *Any algorithm over* $(F(\mathbf{x}), F \cup \mathbf{x})$ *which computes the product* $\Phi\mathbf{x}$, *requires at least* $N(\Phi) - t$ *addition/substractions,* (*where* $N(\Phi)$ *is the column rank of* Φ *with respect to a rational subfield* $G \subset F$).

Actually, this can be strengthened in the case that $F = G(y_1, \cdots, y_t)$ and G is a subfield of the complex numbers. In this case, Theorem B holds for all

[1] $F(\mathbf{a})$ denotes the field extension of F by the indeterminates $a_1, \cdots, a_n$.

algorithms over $(F(\mathbf{x}), F \cup G(\mathbf{x}))$, which means that preconditioning of the set $\mathbf{x}$ is not charged. Winograd also has the following unpublished result.

THEOREM C (Winograd [12]). *Any polynomial algorithm over* $(F(\mathbf{x}), F \cup \mathbf{x})$ *which computes the product* $\Phi\mathbf{x}$, *requires at least* $N^*(\Phi) - t$ *addition/subtractions,* (*where* $N^*(\Phi)$ *is the number of nonzero columns in* Φ).

Since $N^*(\Phi) \geqq N(\Phi)$, this gives a uniformly stronger bound than Theorem B, at the cost of restricting the class of algorithms.

The principal advantage of Winograd's framework is that it allows a straightforward analysis of problems which concern the computation of a family of expressions. The obvious drawback is that many problems cannot be advantageously expressed in this framework. For example, it appears that Theorem B would give a lower bound of zero for both the expression $x_1 + \cdots + x_n$ and the pair of expressions, $ax_1 - bx_2$, $bx_1 + ax_2$ (complex product).[2]

Morgenstern [7] also deals with the computation of functions which are linear in the indeterminates $x_1, \cdots, x_n$. Restricting his attention to linear algorithms, (all multiplications are scalar multiplications), he is able to provide an interesting characterization of the minimum number of additions necessary to compute a given set of linear forms.

In this paper we present a new complexity measure which has its roots in linear independence. Unlike earlier measures, our measure applies to arbitrary multivariate rational functions. The results apply to all rational algorithms. Furthermore, straightforward applications of our central result generalize both Theorems B and C.

The measure is a refinement of the notion "rational independence" defined by Kirkpatrick [6]. The present formulation was influenced by the observations of Kedem [4] and Morgenstern [8].

3. Algebraic preliminaries. Let D be any integral domain[3] and let F be the quotient field of D. Let $\mathbf{a} \triangleq a_1, \cdots, a_n$ and $\mathbf{b} \triangleq b_1, \cdots, b_m$ be sequences of distinct indeterminates over F. We are interested in finding lower bounds on the number of additions and subtractions required to compute finite subsets of $F(\mathbf{a})$ using algorithms over $(F(\mathbf{a}), D \cup \mathbf{a})$.

Let N and Q denote the natural and rational numbers respectively. Let $N\langle \mathbf{a}, \mathbf{b}\rangle \triangleq \{a_1^{\lambda_1} \cdots a_n^{\lambda_n} b_1^{\delta_1} \cdots b_m^{\delta_m} | \lambda_i, \delta_i \in N\}$. We make use of the injection $\Theta: N\langle \mathbf{a}, \mathbf{b}\rangle \to Q^{n+m}$ defined by

$$\Theta(a_1^{\lambda_1} \cdots a_n^{\lambda_n} b_1^{\delta_1} \cdots b_m^{\delta_m}) \triangleq (\lambda_1, \cdots, \lambda_n, \delta_1, \cdots, \delta_m).$$

Θ extends to subsets $X \subseteq N\langle \mathbf{a}, \mathbf{b}\rangle$ by $\Theta(X) \triangleq \{\Theta(X_i) | X_i \in X\}$. Thus Θ maps a set of monomials onto a set of vectors in Q^{n+m}. So, if we let $\rho_V(S)$ denote the vector-rank[4] of any finite subset $S \subset Q^{n+m}$, then we can define the *monomial-rank* (denoted ρ_M) of any subset $X \subset N\langle \mathbf{a}, \mathbf{b}\rangle$ as $\rho_M(X) \triangleq \rho_V(\Theta(X))$.

[2] The only straightforward adaptations are

$$(1, \cdots, 1)\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \text{ and } \begin{pmatrix} a & -b \\ b & a \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}.$$

[3] An integral domain is a ring D for which $uv = 0 \Rightarrow u = 0$ or $v = 0$, for all $u, v \in D$.

[4] The vector-rank of S is just the dimension of the subspace $\{\sum \lambda_i s_i | \lambda_i \in Q \text{ and } s_i \in S\} \subseteq Q^{n+m}$.

Two important properties of monomial-rank are summarized in the following lemma.

LEMMA 1. *Let* $Y \subseteq X \subseteq N\langle \mathbf{a}, b_1, \cdots, b_k\rangle$ *and* $e \in N\langle \mathbf{a}\rangle$. *Then,*

(a) $\rho_M(Y) \leqq \rho_M(X) \leqq \rho_M(Y) + |X| - |Y|$[5] *and*

(b) $\rho_M(X \cup \{eb_{k+1}\}) = \rho_M(X) + 1$.

Proof. These follow by straightforward applications of the definitions. □

4. Computing multivariate polynomials. We start by restricting our attention to the computation of arbitrary elements of $D[\mathbf{a}]$,[6] using polynomial algorithms over $(F(\mathbf{a}))$, $D \cup \{\mathbf{a}\}$). If R is any ring, let R^+ denote the set $R - \{0\}$.

Let $E \in D[\mathbf{a}, \mathbf{b}]^+$. We define the *term set* of E, $\bar{E}$, by

$$\bar{E} \overset{\circ}{=} \left\{ e \;\middle|\; \begin{array}{l} e \in N\langle \mathbf{a}, \mathbf{b}\rangle \text{ and } e \text{ appears with} \\ \text{a nonzero coefficient in } E \end{array} \right\}.$$

We can now define the *expression-rank* (denoted ρ_E) of a set of expressions $\{E_1, \cdots, E_t\} \subset D[\mathbf{a}]$ as

$$\rho_E(E_1, \cdots, E_t) \overset{\circ}{=} \rho_M(\overline{E_1b_1 + \cdots + E_tb_t}) - t.$$

That is, in order to find the expression-rank of a set of expressions one must first combine the expressions, using new indeterminates, and then find the monomial-rank of the term set of the resulting expression.

The following simple properties of the expression-rank of a set of expressions should indicate its potential as a measure of arithmetic complexity.

LEMMA 2. *Let* $E_1, \cdots, E_{k+1} \in D[\mathbf{a}]^+$ *and let* $H \in D^+ \cup \{\mathbf{a}\}$. *Then, for all* $1 \leqq i, j \leqq k$,

(a) $\rho_E(E_1, \cdots, E_k, E_{k+1}) \geqq \rho_E(E_1, \cdots, E_k)$,

(b) $\rho_E(E_1, \cdots, E_k, H) = \rho_E(E_1, \cdots, E_k)$,

(c) $\rho_E(E_1, \cdots, E_k, E_i \times E_j) = \rho_E(E_1, \cdots, E_k)$ *and*

(d) $\rho_E(E_1, \cdots, E_k, E_i \pm E_j) \leqq \rho_E(E_1, \cdots, E_k) + 1$.

Remark. These statements assert that, (a) expression-rank is not decreased by the addition of new expressions, (b), (c) the addition of "free" expressions or products of earlier expressions does not alter the expression-rank, and (d) the addition of an expression formed by addition/subtraction of earlier expressions can increase the expression-rank by at most one.

Proof. Let e_1^0 and e_i denote a fixed and an arbitrary element of $\bar{E}_i$, respectively. Then,

(a)
$$\begin{aligned} &\rho_E(E_1, \cdots, E_k, E_{k+1}) \\ &\overset{\circ}{=} \rho_M(\overline{E_1b_1 + \cdots + E_{k+1}b_{k+1}}) - (k+1) \\ &= \rho_M(\overline{E_1b_1 + \cdots + E_kb_k} \cup \overline{E_{k+1}b_{k+1}}) - (k+1) \\ &\geqq \rho_M(\overline{E_1b_1 + \cdots + E_kb_k} \cup \{e_{k+1}^0 b_{k+1}\}) - (k+1) && \text{Lemma 1(a)} \\ &= \rho_M(\overline{E_1b_1 + \cdots + E_kb_k}) - k && \text{Lemma 1(b)} \\ &\overset{\circ}{=} \rho_E(E_1, \cdots, E_k). \end{aligned}$$

[5] $|X|$ denotes the cardinality of the set X.

[6] $D[\mathbf{a}]$ denotes the ring extension of D by the indeterminates $a_1, \cdots, a_n$.

(b) Since $H \in D^+ \cup \{\mathbf{a}\}$, $\bar{H}$ contains exactly one element which we denote by h. Hence,

$$\begin{aligned}
&\rho_E(E_1, \cdots, E_k, H) \\
&\quad \overset{\circ}{=} \rho_M(\overline{E_1b_1 + \cdots + E_kb_k + Hb_{k+1}}) - (k+1) \\
&\quad = \rho_M(\overline{E_1b_1 + \cdots + E_kb_k} \cup \{hb_{k+1}\}) - (k+1) \\
&\quad = \rho_M(\overline{E_1b_1 + \cdots + E_kb_k}) - k \qquad \text{Lemma 1(b)} \\
&\quad \overset{\circ}{=} \rho_E(E_1, \cdots, E_k).
\end{aligned}$$

(c) Since $e_ie_jb_{k+1} = (e_i^0e_j^0b_{k+1})(e_ib_i)(e_i^0b_i)^{-1}(e_jb_j)(e_j^0b_j)^{-1}$ it follows that $\Theta(\overline{E_ib_i} \cup \overline{E_jb_j} \cup \{e_i^0e_j^0b_{k+1}\})$ generates all of $\Theta(\overline{(E_i \times E_j)b_{k+1}})$. Hence,

$$\begin{aligned}
&\rho_M(\overline{E_1b_1 + \cdots + E_kb_k} \cup \overline{(E_i \times E_j)b_{k+1}}) \\
&\quad = \rho_M(\overline{E_1b_1 + \cdots + E_kb_k} \cup \{e_i^0e_j^0b_{k+1}\}).
\end{aligned}$$

Consequently,

$$\begin{aligned}
&\rho_E(E_1, \cdots, E_k, E_i \times E_j) \\
&\quad \overset{\circ}{=} \rho_M(\overline{E_1b_1 + \cdots + E_kb_k + (E_i \times E_j)b_{k+1}}) - (k+1) \\
&\quad = \rho_M(\overline{E_1b_1 + \cdots + E_kb_k} \cup \{e_i^0e_j^0b_{k+1}\}) - (k+1) \\
&\quad = \rho_M(\overline{E_1b_1 + \cdots + E_kb_k}) - k \qquad \text{Lemma 1(b)} \\
&\quad = \rho_E(E_1, \cdots, E_k).
\end{aligned}$$

(d) Since $e_ib_{k+1} = (e_i^0b_{k+1})(e_ib_i)(e_i^0b_i)^{-1}$ and $e_jb_{k+1} = (e_j^0b_{k+1})(e_jb_j)(e_j^0b_j)^{-1}$ it follows that $\Theta(\overline{E_ib_i} \cup \overline{E_jb_j} \cup \{e_i^0b_{k+1}, e_j^0b_{k+1}\})$ generates all of $\Theta(\overline{(E_i \pm E_j)b_{k+1}})$. Hence,

$$\begin{aligned}
&\rho_M(\overline{E_1b_1 + \cdots + E_kb_k} \cup \overline{(E_i \pm E_j)b_{k+1}}) \\
&\quad \leqq \rho_M(\overline{E_1b_1 + \cdots + E_kb_k} \cup \{e_i^0b_{k+1}, e_j^0b_{k+1}\}).
\end{aligned}$$

Consequently,

$$\begin{aligned}
&\rho_E(E_1, \cdots, E_k, E_i \pm E_j) \\
&\quad \overset{\circ}{=} \rho_M(\overline{E_1b_1 + \cdots + E_kb_k + (E_i \pm E_j)b_{k+1}}) - (k+1) \\
&\quad \leqq \rho_M(\overline{E_1b_1 + \cdots + E_kb_k} \cup \{e_i^0b_{k+1}, e_j^0b_{k+1}\}) - (k+1) \\
&\quad \leqq \rho_M(\overline{E_1b_1 + \cdots + E_kb_k}) - k + 1 \qquad \text{Lemma 1(a)} \\
&\quad \overset{\circ}{=} \rho_M(E_1, \cdots, E_k) + 1. \qquad \square
\end{aligned}$$

It is now possible to give a straightforward proof of the following:

THEOREM 1. *Let $\mathscr{A} = (\alpha_1, \beta_1), \cdots, (\alpha_t, \beta_t)$ be any polynomial algorithm over $(F(\mathbf{a}), D \cup \{\mathbf{a}\})$. Then, $\nu(\mathscr{A}) \geqq \rho_E(\beta_1, \cdots, \beta_t)$.*

Remark. This asserts that the number of addition/subtractions in a polynomial algorithm $\mathscr{A}$ is at least the expression-rank of the set of expressions computed by $\mathscr{A}$.

Proof (by induction on t).

($t=1$) In this case $\beta_1 \in D^+ \cup \{\mathbf{a}\}$, and hence $\rho_E(\beta_1)=0$. Thus the theorem holds trivially.

($t \leqq s$) Assume that the theorem holds for all $t \leqq s$.

($t=s+1$) Let $\mathscr{A}'=(\alpha_1,\beta_1),\cdots,(\alpha_s,\beta_s)$. It follows from the induction hypothesis that $\nu(\mathscr{A}') \geqq \rho_E(\beta_1,\cdots,\beta_s)$.

There are three cases to consider:

(i) The tth step introduces a new input. That is,

$$\alpha_t=(H) \quad \text{and} \quad \beta_t=H \in D^+ \cup \{\mathbf{a}\}.$$

Then,

$$\begin{aligned}\rho_E(\beta_1,\cdots,\beta_t)&=\rho_E(\beta_1,\cdots,\beta_s) && \text{Lemma 2(b)}\\ &\leqq \nu(\mathscr{A}')=\nu(\mathscr{A}).\end{aligned}$$

(ii) The tth step is a multiplication. That is,

$$\alpha_t=(\times,i,j) \quad \text{and} \quad \beta_t=\beta_i \times \beta_j.$$

Then,

$$\begin{aligned}\rho_E(\beta_1,\cdots,\beta_t)&=\rho_E(\beta_1,\cdots,\beta_s) && \text{Lemma 2(c)}\\ &\leqq \nu(\mathscr{A}')=\nu(\mathscr{A}).\end{aligned}$$

(iii) The tth step is an addition/subtraction. That is,

$$\alpha_t=(\pm,i,j) \quad \text{and} \quad \beta_t=\beta_i \pm \beta_j.$$

Then,

$$\begin{aligned}\rho_E(\beta_1,\cdots,\beta_t)&\leqq\rho_E(\beta_1,\cdots,\beta_s)+1 && \text{Lemma 2(d)}\\ &\leqq \nu(\mathscr{A}')+1=\nu(\mathscr{A}).\end{aligned}$$

Hence, the hypothesis holds for $t=s+1$, and by induction the theorem is true for all $t \geqq 1$. □

COROLLARY 1. *If $\mathscr{A}$ is any polynomial algorithm over $(F(\mathbf{a}), D \cup \{\mathbf{a}\})$ which computes the expressions $E_1,\cdots,E_k \in D[\mathbf{a}]^+$, then $\nu(\mathscr{A}) \geqq \rho_E(E_1,\cdots,E_k)$.*

Proof. If $\mathscr{A}=(\alpha_1,\beta_1),\cdots,(\alpha_t,\beta_t)$, then by the theorem $\nu(\mathscr{A}) \geqq \rho_E(\beta_1,\cdots,\beta_t)$. But, by definition $\{E_1,\cdots,E_k\} \subseteq \{\beta_1,\cdots,\beta_t\}$, so by Lemma 2(a),

$$\rho_E(\beta_1,\cdots,\beta_t) \geqq \rho_E(E_1,\cdots,E_k). \qquad \square$$

5. Computing rational functions. We now remove our earlier restrictions and consider the computation of arbitrary finite subsets of $F(\mathbf{a})^+$, using general algorithms over $(F(\mathbf{a}), D \cup \{\mathbf{a}\})$.

Given any rational algorithm, we can construct a polynomial algorithm which simulates the first by keeping track of the numerator and denominator of every intermediate expression. The following lemma shows that this can be done without increasing the number of addition/subtractions.

LEMMA 3. *Let $\mathscr{A}=(\alpha_1,\beta_1),\cdots,(\alpha_t,\beta_t)$ be any rational algorithm over $(F(\mathbf{a}),D\cup\{\mathbf{a}\})$. Then, there exists a polynomial algorithm, $\mathscr{A}'=(\alpha'_1,\beta'_1),\cdots,(\alpha'_{2t},\beta'_{2t})$, where $\nu(\mathscr{A}')=\nu(\mathscr{A})$ and for all i satisfying $1\leqq i\leqq t$, $(\beta'_{2i-1}/\beta'_{2i})=\beta_i$.*

Proof (by induction on t). We shall first make two assumptions concerning the form of $\mathscr{A}$, neither of which affects the generality of our arguments:

(i) We assume that $(\alpha_1,\beta_1)=((1),1)$. That is, the first step of $\mathscr{A}$ introduces the constant 1.

(ii) We assume that all addition/subtraction steps are of the form (α_i,β_i) where $\alpha_i=(\pm,j,1)$ and $\beta_i=\beta_j\pm 1$, for some $j<i$.

Given the identity, $\beta_j\pm\beta_k=((\beta_j/\beta_k)\pm 1)\times\beta_k$, it follows that both of these assumptions can be ensured without modifying the number of addition/subtractions in an algorithm.

$(t=1)$ Since $(\alpha_1,\beta_1)=((1),1)$, it suffices to make $\alpha'_1=\alpha'_2=(1)$ and $\beta'_1=\beta'_2=1$.

$(t\leqq s)$ Assume that the theorem holds for all $t\leqq s$.

$(t=s+1)$ Let $\mathscr{B}=(\alpha_1,\beta_1),\cdots,(\alpha_s,\beta_s)$ and let $\mathscr{B}'=(\alpha'_1,\beta'_1),\cdots,(\alpha'_{2s},\beta'_{2s})$ satisfy the induction hypothesis. There are four cases to consider:

(i) The tth step introduces a constant. That is,

$$(\alpha_t,\beta_t)=((H),H)\quad\text{where}\quad H\in D^+\cup\{\mathbf{a}\}.$$

Then, let $(\alpha'_{2t-1},\beta'_{2t-1})=((H),H)$ and $(\alpha'_{2t},\beta'_{2t})=((1),1)$.

(ii) The tth step is a multiplication. That is,

$$(\alpha_t,\beta_t)=((\times,i,j),\beta_i\times\beta_j).$$

Then, let $(\alpha'_{2t-1},\beta'_{2t-1})=((\times,2i-1,2j-1),\beta'_{2i-1}\times\beta'_{2j-1})$ and $(\alpha'_{2t},\beta'_{2t})=((\times,2i,2j),\beta'_{2i}\times\beta'_{2j})$.

(iii) The tth step is a division. That is,

$$(\alpha_t,\beta_t)=((/,i,j),\beta_i/\beta_j).$$

Then, let $(\alpha'_{2t-1},\beta'_{2t-1})=((\times,2i-1,2j),\beta'_{2i-1}\times\beta'_{2j})$ and $(\alpha'_{2t},\beta'_{2t})=((\times,2i,2j-1),\beta'_{2i}\times\beta'_{2j-1})$.

(iv) The tth step is an addition/subtraction. That is,

$$(\alpha_t,\beta_t)=((\pm,i,1),\beta_i\pm 1).$$

Then, let $(\alpha'_{2t-1},\beta'_{2t-1})=((\pm,2i-1,2i),\beta'_{2i-1}\pm\beta'_{2i})$ and $(\alpha'_{2t},\beta'_{2t})=((\times,2i,1),\beta'_{2i}\times 1)$.

In all cases let $\mathscr{A}'=\mathscr{B}',(\alpha'_{2t-1},\beta'_{2t-1}),(\alpha'_{2t},\beta'_{2t})$. It follows from our construction that $\nu(\mathscr{A}')=\nu(\mathscr{A})$, and $(\beta'_{2t-1}/\beta'_{2t})=\beta_t$. Hence, the hypothesis holds for $t=s+1$, and by induction the lemma holds for all $t\geqq 1$. □

COROLLARY 2. *Let $\mathscr{A}=(\alpha_1,\beta_1),\cdots,(\alpha_t,\beta_t)$ be any rational algorithm over $(F(\mathbf{a}),D\cup\{\mathbf{a}\})$, and suppose $\beta_i=A_i/B_i$, where $A_i,B_i\in D[\mathbf{a}]^+$ are relatively prime. Then, there exist polynomials $C_1,\cdots,C_t\in D[\mathbf{a}]^+$ and a polynomial algorithm $\mathscr{A}'$, over $(F(\mathbf{a}),D\cup\{\mathbf{a}\})$, such that $\nu(\mathscr{A}')=\nu(\mathscr{A})$ and $\mathscr{A}'$ computes the polynomials $A_1C_1,B_1C_1,\cdots,A_tC_t,B_tC_t$.*

Proof. Let $\mathscr{A}'$ be constructed as in Lemma 3. Then $\beta'_{2i-1}/\beta'_{2i}=A_i/B_i$. But, A_i and B_i relatively prime implies that, for some $C_i\in D[\mathbf{a}]^+$, $\beta'_{2i-1}=A_iC_i$ and $\beta'_{2i}=B_iC_i$. □

As a result of Corollary 2, we are led to ask whether it is possible for the expression-rank of a sequence of expressions to be decreased through multiplication by nonzero polynomials. The following two lemmas provide the desired answer.

Suppose X, $Y \in D[\mathbf{a}]^+$. We denote by $\bar{X} \cdot \bar{Y}$ the set $\{xy | x \in \bar{X}, y \in \bar{Y}\}$. Note, $\overline{XY}$ is contained in but not necessarily equal to $\bar{X} \cdot \bar{Y}$, since some terms may cancel in the product XY.

If $W \subset N\langle \mathbf{a}, \mathbf{b} \rangle$, then we define the convex interior of W,

$$I(W) \overset{\circ}{=} \{\textstyle\prod w_j^{\lambda_j} | w_j \in W, 0 \leqq \lambda_j < 1 \quad \text{and} \quad \sum \lambda_j = 1\}.$$

We call the set $V(W) \overset{\circ}{=} \{w \in W | w \notin I(W)\}$, the set of vertices of W. An elementary result in the study of convex regions (see for example [3]) is that $I(W) = I(V(W))$.

Lemma 4. *If* X, $Y \in D[\mathbf{a}]^+$, *then* $V(\bar{X} \cdot \bar{Y}) \subseteq \overline{XY}$.

Proof. By definition, $V(\bar{X} \cdot \bar{Y}) \subseteq \bar{X} \cdot \bar{Y}$. Assume $\bar{X} \cdot \bar{Y} - \overline{XY} \neq \varnothing$, (otherwise there is nothing to prove). Let $e \in \bar{X} \cdot \bar{Y} - \overline{XY}$, i.e., e is canceled in the product XY. Since D is an integral domain, there must exist distinct x, $x' \in \bar{X}$, and distinct y, $y' \in \bar{Y}$, such that $e = xy = x'y'$. Hence,

$$e = (xy')^{1/2}(x'y)^{1/2} \in I(\bar{X} \cdot \bar{Y}).$$

Thus, $e \notin V(\bar{X} \cdot \bar{Y})$, and in general $V(\bar{X} \cdot \bar{Y}) \subseteq \overline{XY}$. □

This technical lemma allows us to give a very simple proof the following:

Lemma 5. *Let* $X_1, \cdots, X_t$, $Y_1, \cdots, Y_t \in D[\mathbf{a}]^+$. *Then,*

$$\rho_E(X_1 Y_1, \cdots, X_t Y_t) \geqq \rho_E(X_1, \cdots, X_t).$$

Proof. Let $W_i = \bar{X}_1 \cdot \overline{Y_i b_i}$, and let $V_i \overset{\circ}{=} V(W_i)$. Since $V_i \subseteq W_i \subseteq I(W_i) \cup V(W_i) = I(V_i) \cup V_i$, it follows that $\Theta(V_i)$ generates all of $\Theta(W_i)$, and hence

$$\rho_M(V_1 \cup \cdots \cup V_t) = \rho_M(W_1 \cup \cdots \cup W_t).$$

But, by Lemma 4, we know that $V_i \subseteq \overline{X_i Y_i b_i}$, and so, by Lemma 1(a), $\rho_M(V_1 \cup \cdots \cup V_t) \leqq \rho_M(\overline{X_1 Y_1 b_1 + \cdots + X_t Y_t b_t})$.

Thus, if y_i denotes an arbitrary element of $\bar{Y}_i$, we have

$$\begin{aligned}
&\rho_E(X_1 Y_1, \cdots, X_t Y_t) \\
&\quad \overset{\circ}{=} \rho_M(\overline{X_1 Y_1 b_1 + \cdots + X_t Y_t b_t}) - t \\
&\quad \geqq \rho_M(V_1 \cup \cdots \cup V_t) - t \\
&\quad = \rho_M(W_1 \cup \cdots \cup W_t) - t \\
&\quad \overset{\circ}{=} \rho_M(\overline{X_1} \cdot \overline{Y_1 b_1} \cup \cdots \cup \overline{X_t} \cdot \overline{Y_t b_t}) - t \\
&\quad \geqq \rho_M(\overline{X_1 y_1 b_1} \cup \cdots \cup \overline{X_t y_t b_t}) - t && \text{Lemma 1(a)} \\
&\quad = \rho_M(\overline{X_1 b_1' + \cdots + X_t b_t'}) - t \\
&\quad \overset{\circ}{=} \rho_E(X_1, \cdots, X_t).
\end{aligned}$$

□

Finally, we have

THEOREM 2. *Let $\mathscr{A}$ be any rational algorithm, over $(F(\mathbf{a}), D \cup \{\mathbf{a}\})$, which computes the rational functions $A_1/B_1, \cdots, A_k/B_k$, where $A_i, B_i \in D[\mathbf{a}]^+$ are relatively prime. Then,*

$$\nu(\mathscr{A}) \geqq \rho_E(A_1, B_1, \cdots, A_k, B_k).$$

Proof. By Corollary 2, we know that there exist polynomials $C_1, \cdots, C_k \in D[\mathbf{a}]^+$, and a polynomial algorithm $\mathscr{A}'$, with $\nu(\mathscr{A}') = \nu(\mathscr{A})$, such that $\mathscr{A}'$ computes $A_1C_1, B_1C_1, \cdots, A_kC_k, B_kC_k$. But,

$$\nu(\mathscr{A}') \geqq \rho_E(A_1C_1, B_1C_1, \cdots, A_kC_k, B_kC_k) \qquad \text{Corollary 1}$$

$$\geqq \rho_E(A_1, B_1, \cdots, A_k, B_k) \qquad \text{Lemma 5}$$

Hence,

$$\nu(\mathscr{A}) \geqq \rho_E(A_1, B_1, \cdots, A_k, B_k). \qquad \square$$

COROLLARY 3. *Let $\mathscr{A}$ be any rational algorithm over $(F(\mathbf{a}), D \cup \{\mathbf{a}\})$, which computes the rational functions, $A_1/B_1, \cdots, A_k/B_k$, where $A_i, B_i \in D[\mathbf{a}]^+$ are relatively prime. Then,*

$$\nu(\mathscr{A}) \geqq \rho_V(\overline{\Theta(A_1b_1 + B_1b_2 + \cdots + A_kb_{2k-1} + B_kb_{2k})}) - 2k.$$

In the case that the functions to be computed are all multivariate polynomials, the following corollary provides the same bound as Corollary 3, while being somewhat less cumbersome to apply.

COROLLARY 4. *Let $\mathscr{A}$ be any rational algorithm over $(F(\mathbf{a}), D \cup \{\mathbf{a}\})$, which computes the polynomials, $A_1, \cdots, A_k \in D[\mathbf{a}]^+$. Then,*

$$\nu(\mathscr{A}) \geqq \rho_V(\overline{\Theta(A_1b_1 + \cdots + A_kb_k)}) - k.$$

Proof. This follows directly from Theorem 2 and the definitions, given the equality,

$$\rho_E(A_1, 1, A_2, 1, \cdots, A_k, 1) = \rho_E(A_1, \cdots, A_k)$$

which was established by Lemma 2(b). $\square$

6. Applications. Corollaries 3 and 4 provide straightforward procedures for reducing the problem of determining addition/subtraction requirements for arbitrary sets of rational functions to the problem of determining the rank of a set of vectors in Q^t. In this way, we can generate the best lower bounds known, in a number of cases optimal bounds, for a large number of familiar arithmetic expressions.[7]

As before, we let D denote an arbitrary integral domain, and F the quotient field of D. For the sake of uniformity, let I denote the set of symbols formed from $\{a, x, y\}$ by the possible addition of primes or subscripts. I can be thought of as an arbitrarily large pool of distinct indeterminates, and will take the place of the set $\{\mathbf{a}\}$ of the preceding development.

Let $\mathscr{A}$ be any rational algorithm over $(F(I), D \cup I)$. As before, $\nu(\mathscr{A})$ denotes the number of addition/subtraction steps in $\mathscr{A}$.

A1. *If $\mathscr{A}$ computes the expression $a_1 + a_2 + \cdots + a_n$, then $\nu(\mathscr{A}) \geqq n - 1$.*

[7] Most of these first appeared in [5] or [6]. Others were given in [4].

Proof. We know, by Corollary 4, that

$$\nu(\mathscr{A}) \geqq \rho_V(\Theta(\overline{(a_1+\cdots+a_n)b_1})) - 1.$$

But, it is easy to verify that the vectors in $\Theta(\overline{(a_1+\cdots+a_n)b_1})$ are all independent, over Q, and hence $\rho_V(\Theta((a_1+\cdots+a_n)b_1)) = n$. □

More generally, we have

A2. *If $\mathscr{A}$ computes the expression*

$$\frac{a_1+a_2+\cdots+a_n}{a_{n+1}+a_{n+2}+\cdots+a_{n+m}},$$

then $\omega(\mathscr{A}) \geqq n+m-2$.

A3. *If $\mathscr{A}$ computes the pair of expressions, $a_1a_3-a_2a_4$ and $a_1a_4+a_2a_3$, (the real and imaginary parts of the complex product $(a_1+a_2i)(a_3+a_4i)$), then, $\nu(\mathscr{A}) \geqq 2$.*

Proof. By Corollary 4, it suffices to verify that

$$\rho_V(\Theta(\overline{(a_1a_3+a_2a_4)b_1+(a_1a_4+a_2a_3)b_2})) = 4. \qquad □$$

A4. *If $\mathscr{A}$ computes the general rational function,*

$$\frac{a_nx^n+a_{n-1}x^{n-1}+\cdots+a_0}{a'_mx^m+a'_{m-1}x^{m-1}+\cdots+a'_0},$$

then $\nu(\mathscr{A}) \geqq n+m$.

The following result generalizes both Theorems B and C.

A5. *Let Φ be an $t \times n$ matrix over D, and let $N^*(\Phi)$ denote the number of columns of Φ which are not identically zero. If $\mathscr{A}$ computes the matrix-vector product,*

$$\Phi \cdot \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}, \quad \textit{then} \quad \nu(\mathscr{A}) \geqq N^*(\Phi) - t.$$

Proof. Let $k = N^*(\Phi)$, and assume, without loss of generality, that the first k columns of Φ are not identically zero. Let $\Phi = \begin{pmatrix} \phi_{11} & \cdots & \phi_{1n} \\ \vdots & & \\ \phi_{t1} & \cdots & \phi_{tn} \end{pmatrix}$ and for $1 \leqq i \leqq k$, define $[i] = \min\{j \mid \phi_{ji} \neq 0\}$. By Corollary 4, it suffices to show that $\rho_V(\Theta(E)) \geqq k$, where

$$\begin{aligned} E = (\phi_{11}a_1 + \cdots + \phi_{1n}a_n)b_1 \\ + \\ \cdot \\ \cdot \\ \cdot \\ + (\phi_{t1}a_1 + \cdots + \phi_{tn}a_n)b_t. \end{aligned}$$

But, by the definition of $[i]$, we know that

$$\{a_1b_{[1]}, a_2b_{[2]}, \cdots, a_kb_{[k]}\} \subseteq \bar{E}.$$

Hence, $\rho_V(\Theta(\bar{E})) \geqq \rho_V(\Theta(\{a_1b_{[1]}, \cdots, a_kb_{[k]}\})) = k$. □

A6. *Let $A \doteq (a_{ij})$ and $X \doteq (x_{ij})$ be $m \times n$ and $n \times p$ matrices, respectively. If $\mathcal{A}$ computes the matrix product $A \cdot X$, then $\nu(\mathcal{A}) \geqq (m+p-1)(n-1)$.*

In particular, this gives addition/subtraction lower bounds of $m(n-1)$, for the product of an $m \times n$ matrix with an n-vector, and $2n^2-3n+1$, for the product of two $n \times n$ matrices.

A7. *Let*

$$X \doteq \begin{pmatrix} x_1^n & x_1^{n-1} & \cdots & x_1 & 1 \\ \vdots & \vdots & & \vdots & \vdots \\ x_m^n & x_m^{n-1} & \cdots & x_m & 1 \end{pmatrix}.$$

If $\mathcal{A}$ computes the matrix-vector product

$$X \cdot \begin{pmatrix} a_n \\ \vdots \\ a_1 \\ a_0 \end{pmatrix}, \quad \textit{then} \quad \nu(\mathcal{A}) \geqq n+m-1.$$

The obvious interpretation of the above is that it requires at least $n+m-1$ addition/subtractions to evaluate an nth degree polynomial at m arbitrary points.

A8. *Let $P_i = \sum_{j=0}^{n(i)} a_{ij}x^j$, for $i = 1, \cdots, t$. If $\mathcal{A}$ computes the set $P_1, \cdots, P_t$, then $\nu(\mathcal{A}) \geqq \sum_{i=1}^{t} n(i)$.*

A9. *If $\mathcal{A}$ computes the expression $\sum_{i=0}^{n} \sum_{j=0}^{n} a_{ij}x^iy^j$, then $\nu(\mathcal{A}) \geqq (n+1)^2-1$.*

7. Open questions. The results and techniques of this paper leave unanswered a number of interesting questions. Some of these, including both a related development using substitution techniques and observations on the structure of algorithms (if they exist) which achieve the lower bounds given by our independence measure, will be considered in future papers.

Of major importance is the problem of developing techniques for lower bounds on additions, which are nonlinear in the number of indeterminants. We should mention the initial success of Borodin and Cook [2] in this direction. It is hoped that, perhaps through consideration of the geometrical interpretations employed in Lemma 4, our techniques might be modified to provide this kind of bound.

Acknowledgment. We would like to thank both A. B. Borodin and S. Winograd for their encouragement and numerous helpful comments.

REFERENCES

[1] E. C. BELAGA, *On computing polynomials in one variable with initial preconditioning of coefficients*, Problemy Kibernet., 5 (1961), pp. 7–15.

[2] A. B. BORODIN AND S. A. COOK, *On the number of additions to compute specific polynomials*, Proc. 6th Annual ACM Symp. on Theory of Computing, Association for Computing Machinery, New York, 1974, pp. 342–347.

[3] B. GRÜNBAUM, *Convex Polytopes*, John Wiley, London, 1967.

[4] Z. M. KEDEM, *Studies in algebraic computational complexity*, Doctor of Science thesis, Israel Institute of Technology, December, 1973.

[5] D. G. KIRKPATRICK, *On the additions necessary to compute certain functions*, Proc. 4th Annual ACM Symp. on Theory of Computing, Association for Computing Machinery, New York, 1972, pp. 94–101.

[6] ———, an expanded version of [5], Tech. Rep. 39, University of Toronto, Canada, 1972.

[7] J. MORGENSTERN, *The linear complexity of computation*, J. Assoc. Comput. Mach., 22 (1975), pp. 184–194.

[8] ———, private communication, 1972.

[9] T. S. MOTZKIN, *Evaluation of polynomials and evaluation of rational functions*, Bull. Amer. Math. Soc., 61 (1955), p. 163.

[10] A. M. OSTROWSKI, *On two problems in abstract algebra connected with Horner's rule*, Studies Presented to R. von Mises, Academic Press, New York, 1954.

[11] S. WINOGRAD, *On the algebraic complexity of functions*, Actes, Congres Internat. Math., 3 (1970), pp. 283–288.

[12] ———, private communication, 1971.

SIAM J. COMPUT.
Vol. 6, No. 2, June 1977

OPTIMUM SEQUENCE TREES*

MASAHIRO MIYAKAWA, TOSHITSUGU YUBA, YOSHIO SUGITO
AND MAMORU HOSHI†

Abstract. The construction problems of optimum sequence trees (or digital search trees) are considered in the following frameworks: 1. construction of optimum trees from a set of keys, 2. transformation of an arbitrary tree into an optimum one, 3. optimum insertions of keys into an optimum tree, 4. optimum deletions of keys from an optimum tree.

Algorithms are shown. The number of operations needed for the algorithm in the framework 1 is at most $O(N^2L)$, and in the framework 2 it is at most $O(N^3L)$ both with $O(N)$ storage locations, where N and L are the number of keys and the length of coded keys respectively.

Necessary and sufficient conditions for the optimality of sequence trees are also given.

Key words. algorithm, digital search trees, optimization, searching, sequence trees

CONTENTS

List of notations.

a, b, c, d	codes in general (with suffixes).
ADDR(p)	the address of a node p.
C	the set of given N codes.
CODE$_T(p)$	the code placed at p in T.

* Received by the editors August 2, 1974, and in revised form April 15, 1976.

† Electrotechnical Laboratory, Tokyo, 100, Japan.

h, i, j, k, l, m, n	suffixes representing natural numbers.
L	the code length (given).
$\mathrm{lev}(p)$	the level of p ($\mathrm{lev}(R)=0$).
ℓ_i	$\mathrm{lev}(p_i)$ or $\mathrm{lev}(q_i)$.
ℓ_p	$\mathrm{lev}(p)$.
mas	minimum ascending shift.
mds	minimum descending shift.
N	the number of codes (given).
p, q, r, s	nodes in general (with suffixes).
$p(a)$	the node p containing a code a.
p^*	the support node of p.
$[p, q]$	the path from p to q.
R	the root of a tree.
$\mathrm{SON}_a(p)$	the left or right son of p ($a=0$ or 1, respectively).
t	a leaf in general.
T	a binary tree, or a sequence tree in general.
$T[r]$	the subtree of T whose root is r.
$\|T\|$	the weighted number of comparisons or the cost of the tree T.
v	an external node (i.e., a son of a leaf).
$\mathrm{val}_T(\pi)$	the value of π in T.
$\mathrm{wt}_T(p_i)$	the weight of the code of p_i in T.
w_i	a weight in general or $\mathrm{wt}_T(p_i)$.
α	the number of primitive operations needed for an A-test.
β	the number of primitive operations needed for a C-test.
δ	a transposition of codes.
$\Delta_T[p, r]$	an mas originating from p and terminating at r in T.
$\Delta_T[*, r]$	an mas terminating at r in T.
$\nabla_T[p, r]$	an mds originating from p and terminating at r in T.
$\nabla_T[p, *]$	an mds originating from p in T.
ε	the identity permutation.
$\eta_T[p, q]$	the exchangeable sequence originating from p and terminating at q in T.
λ	the address of the root R, or null code, or empty sequence.
$\mu_T[p, q]$	a minimum shift originating from p and terminating at q in T.
$\mu_T[*, p]$	a minimum shift terminating at p in T.
$\mu_T[p, *]$	a minimum shift originating from p in T.
$\pi_T[p, q]$	a permutation sequence (p-sequence) originating from p and terminating at q in T.
π^+	the first part of π.
π^-	the second part of π.
$\leftrightarrow$	$a_i \leftrightarrow a_j$ (or $p_i \leftrightarrow p_j$) means that these codes are exchangeable for each other.
$\succ$	$p \succ q$ means that p is higher (larger) than q.

1. Introduction. The digital tree search is originally due to Coffman and Eve [1]. In their original paper digital search trees are called sequence (hash) trees. Much discussion about *digital tree search* method is to be found in the book by Knuth (see [5, pp. 481–505]). We will slightly extend the original definition of sequence (hash) trees still calling them sequence trees. This is partly because we hope that sequence trees may have other applications in addition to searching. Therefore, in our description, we focus on the general structural characteristics of the trees. However, the motivations of our problems to be considered here are from computer file systems, in which the reduction of search time is one of the major concerns.

In this paper we solve the optimization problems of sequence trees in the following frameworks:

1. Given a set of keys, construct an optimum tree.[1]
2. Given an arbitrary tree, transform it into an optimum one.
3. Given an optimum tree, insert a new key into it in such a way that the resulting tree becomes optimum (optimum insertion).
4. Given an optimum tree, delete a key from it in such a way that the resulting tree becomes optimum (optimum deletion).

These four problems are mutually dependent. If we construct a tree, for example, the prefix tree [1] from the key set, then problem 1 is reduced to problem 2. The solution to problem 3 is also a solution to problem 1, since we can construct an optimum tree by successive insertions. After inserting (deleting) a key by any method into (from) an optimum tree, we can optimize the resulting tree by using the solution of problem 2. However, in problem 3 or 4 we require some algorithm to decide how a key should be inserted or deleted respectively.

In practical file systems, insertion and deletion of keys play an important role. Hence optimum insertion and deletion algorithms are very important, since we can always keep the file structure optimum if we use them.

Next, let us consider situations of on-line file systems. In these cases, access frequency of each file changes momentarily, so we need to reconstruct the file from time to time in order to make its structure optimum. This self-optimizing file system is our motivation of framework 2.

In framework 2 our optimization algorithm needs at most $O(N^3L)$ operations with $O(N)$ storage locations, where N and L are the number of keys and the length of coded keys respectively.

The case of uniform weights is considered in [8], [9] and [10] (each paper considers the problem in frameworks 1, 2, 3 and 4 respectively).

In this paper § 2 gives definitions and basic operations on sequence trees. In § 3 properties of regular trees, a subset of sequence trees, are investigated. Necessary and sufficient conditions for the optimality of sequence trees are given. Section 4 describes transformations into optimum trees. In § 5 descriptions of algorithms are given in detail. Section 6 is devoted to the evaluation of the number of operations for the bottom-up optimization. In the concluding discussion of § 7 some related problems are given.

[1] Recently we found that the same problem of constructing optimum digital search trees is listed as a research problem in [5, p. 504, problem 39].

2. Basic operations on sequence trees. In this section we give definitions of terms used in this paper. Search, insertion, and permutation sequences on sequence trees are explained. Unless stated explicitly to the contrary, all definitions and notations are the same as those used in the book by Knuth [3, pp. 205–405].

We consider binary trees and say simply trees for binary trees. (The formulation in this paper is also valid for M-ary trees.) The *level* of a node p (denoted by $\mathrm{lev}(p)$) is the number of edges from the root R to the node p (thus $\mathrm{lev}(R)=0$). The *path* of a node is the set of nodes which are on a chain of edges from the root to the node. If nodes p and q are on the same path, then the *path from p to q* (denoted by $[p, q]$) is the set of nodes which are on a chain of edges from p to q (we put $[p, p]=\{p\}$). The number of the edges of a path is called its *path length.*

If p and q are on a path and $\mathrm{lev}(p) \leqq \mathrm{lev}(q)$, then we write $p \succeq q$ (p is *higher than or equal to* q). Thus we consider a tree as a set of nodes partially ordered by the relation $\succeq$. The root is the maximal (or highest) node of the tree.

For our purpose we consider a tree T with a maximum level L and with 2^i nodes at each level i for $0 \leqq i \leqq L$, where L is a nonnegative integer. Thus every node of T has two sons if its level is less than L and each node at level L has no son (T has $2^{L+1}-1$ nodes). With each node p we associate an address (a bit sequence denoted by $\mathrm{ADDR}(p)$) as follows:

$$\mathrm{ADDR}(R)=\lambda,$$

$$\mathrm{ADDR}(\mathrm{SON}_0(p))=\mathrm{ADDR}(p)\cdot 0,$$

$$\mathrm{ADDR}(\mathrm{SON}_1(p))=\mathrm{ADDR}(p)\cdot 1,$$

where $\mathrm{SON}_0(p)$ and $\mathrm{SON}_1(p)$ denote the left and right sons of p, $\cdot$ and λ denote concatenation and the empty sequence respectively.

We represent a node by its address. Figure 2.1 shows an addressed binary tree with a maximum level $L=2$.

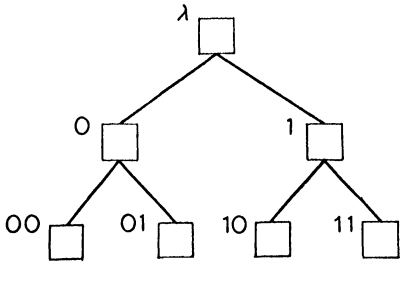


FIG. 2.1. *An addressed binary tree*

To represent the relations of nodes by using their addresses, we define the prefixes of a bit sequence. Let $b=b_1b_2\cdots b_n$ be a bit sequence ($b_i=0$ or 1), then the set $\{\lambda, b_1, b_1b_2, \cdots, b_1b_2\cdots b_n\}$ is the set of prefixes of b. Then we note the following relations:

$$p \succeq q \Leftrightarrow \mathrm{ADDR}(p) \text{ is a prefix of } \mathrm{ADDR}(q).$$

The symbol $\succeq$ is used for bit sequences as well. Thus $a \succeq b$ means that a is a prefix of b, where a and b are bit sequences.

Next, we define a set of weighted codes. Suppose that a set on N records should be stored for use of searching. In general, each record includes a special field called its *key* to identify the record. Here we map keys into bit sequences and call them *codes*. A code set is denoted by $C=\{c_i|i=1,\cdots,N\}$ $(N\leqq 2^L)$, where each code c_i is a bit sequence with the length L (L is the fixed code length and N is the number of codes in C). We assume that codes are distinct from each other. With each code we associate its *weight*, a nonnegative real number. Each weight may correspond to an access frequency of the record. We can construct a tree where search, insertion and deletion of codes are conveniently performed [1].

Example 2.1 (Insertion algorithm). We illustrate an algorithm which inserts a code into a tree (see Fig. 2.2). Let the code set be $C=\{a=00, b=01, c=10\}$. We insert c, a and b in this order. Initially the tree consists of all empty nodes.

First we will insert c. Since the root is empty, c is placed there. Next we will insert a. The algorithm compares a with the code at R (now it contains c). Since they do not match, the algorithm checks the first bit of a. Since it is 0, the algorithm traverses the left edge of the node (thus the code "guides" in the tree). Since the newly current node 0 is empty, a is placed there. Finally we will insert b. The algorithm accordingly arrives at the empty node 01 and places the code b there, since the first and second bits of b are 0 and 1. Thus we have constructed the tree T_1 in Fig. 2.2 (we associate the *null code* λ with each empty node). Almost the same algorithm can be used for searching. Different insertion sequences lead in general to different trees.

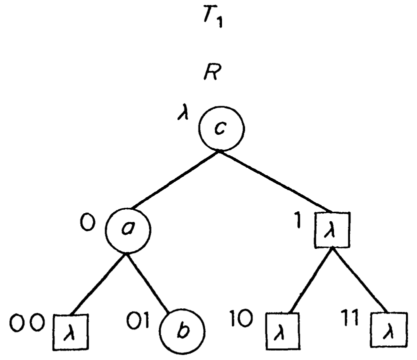


FIG. 2.2. *Insertion of codes c, a and b*

Thus we consider that with each node of a tree T an address and a code are associated. Therefore, a node can be represented by $p(a)$, where p is the address of the node and a is the code placed at p. Symbols p, q, r, s and a, b, c, d possibly with suffixes denote nodes (addresses) and codes respectively. The code placed at p in T is denoted by $\mathrm{CODE}_T(p)$. The weight of the code of p in T is denoted by $\mathrm{wt}_T(p)$. The weight of the null code λ is defined to be zero.

The insertion algorithm suggests that each code specifies a path on which that code should be placed. For each node p we define its admissible codes as follows.

If the address p is a prefix of a code a (i.e., $p \geq a$), then a is *admissible* to p. Besides, the null code λ is admissible to any node, by convention. Each node at level L has at most one admissible code except λ. In Example 2.1 admissible codes of the node 0 are $\{\lambda, a, b\}$, and of the node 01 are $\{\lambda, b\}$.

A tree T is a *sequence tree* if every node of T has an admissible code. That is, T is a sequence tree if for every nonempty node $p(a)$, the address p is a prefix of the code a. We may simply write trees for sequence trees. A *leaf* of T is a nonempty node whose descendants are all empty. Thus leaves are the minimal nonempty nodes of T. An *external node* is a son of a leaf. Thus every leaf has two external nodes if its level is not L. A subtree of T is denoted by $T[p]$, where p is the root of the subtree. Two examples of sequence trees are shown. The first one is a *packed tree*,[2] where $p(a) \in T$ and $a \neq \lambda$ imply that q is nonempty for any $q \geq p$. The insertion algorithm shows that packed trees can be constructed by successively inserting codes from the root. Our second example is a *prefix tree*,[3] where each code is placed at the node whose address is identical to the code (see Fig. 2.3). Thus in a prefix tree all nonempty nodes are at the level L and every nonempty node is a leaf. The prefix tree is uniquely determined by a code set.

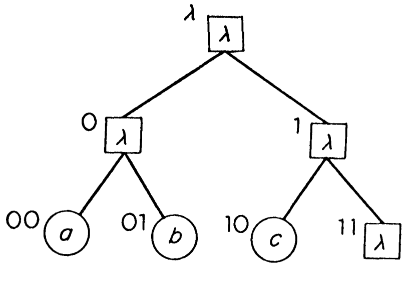


FIG. 2.3. *A prefix tree*

A code a is admissible to any node of the path $[R, p]$, if the address p is identical to a (the path is called the admissible path of a). Hence a code placed at q is admissible to either $\mathrm{SON}_0(q)$ or $\mathrm{SON}_1(q)$. This is an interesting characteristic of sequence trees.

Consider searching the tree T_1 of Example 2.1. We need two, three, and one comparisons to search a, b, and c respectively. Let the weight (access frequency) of codes be $w_a = 4$, $w_b = 3$, and $w_c = 5$. Then we need on the average $(4 \cdot 2 + 3 \cdot 3 + 5 \cdot 1)/(4+3+5) = 22/12$ comparisons for a successful search. Thus the relative amount of work to search a tree T can be measured by the *weighted number of comparisons*, or the *cost* $|T|$ of the tree defined by

$$|T| = \sum_{p \in T} \mathrm{wt}(p)(\mathrm{lev}(p)+1)/\sum \mathrm{wt}(p),$$

where the summation can be restricted to the nonempty nodes of T, since empty

[2] A packed tree is called a *sequence hash tree* in [1], and a *binary digital search tree* in [5]; however the former has no code at the root.

[3] This definition differs slightly from that in [1].

nodes carry no weight. We omit the normalization factor $\sum \mathrm{wt}(p)$, because we compare different trees for a fixed set of codes and weights.

The set of all sequence trees over a code set is finite. A tree is *optimum* if its cost is a minimum in the set of all sequence trees over the code set. A subtree is an *optimum subtree* if it is optimum for the codes which are contained in the subtree. For any nonpacked tree there is a packed one whose cost is not greater than the former's.

Example 2.2. Let $C=\{a=00, b=01, c=10\}$ as in Example 2.1. Figure 2.3 shows the prefix tree of C. Figure 2.4 shows all the packed trees over C (empty nodes lower than leaves are usually omitted). Figure 2.5 shows the ranges of w_a, w_b and w_c (the weights of a, b and c respectively, with $w_a+w_b+w_c=1$) for which each tree is optimum. (This diagram is due to Knuth [5, p. 434].)

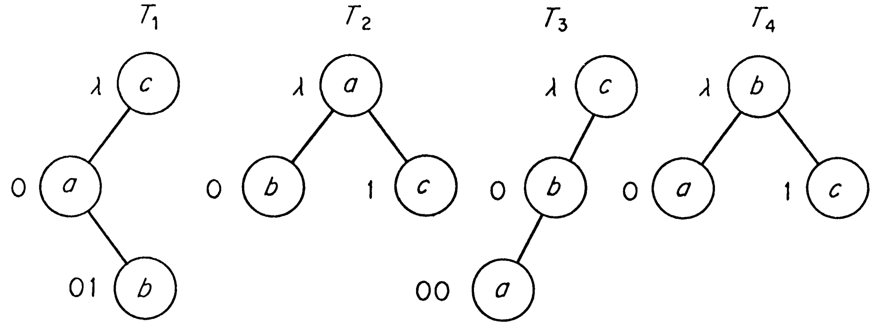


FIG. 2.4. *All packed trees over* $C=\{a, b, c\}$

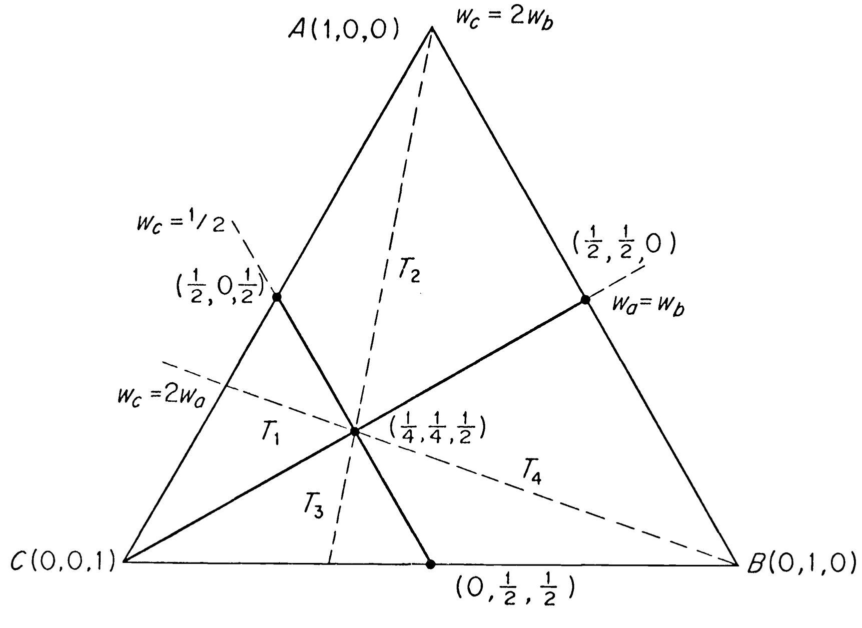

$A(1,0,0)$
$w_c = 2w_b$
$w_c = 1/2$
$(\frac{1}{2},0,\frac{1}{2})$
$(\frac{1}{2},\frac{1}{2},0)$
$w_a = w_b$
T_2
$w_c = 2w_a$
T_1
$(\frac{1}{4},\frac{1}{4},\frac{1}{2})$
T_4
T_3
$C(0,0,1)$
$B(0,1,0)$
$(0,\frac{1}{2},\frac{1}{2})$

FIG. 2.5. *Optimum domain of* T_i $(w_a+w_b+w_c=1)$

In an attempt to construct optimum trees, a fairly reasonable strategy is to repeatedly insert codes in decreasing order of their weights. However this idea does not always work:

Example 2.3. Consider the same code set C as in Example 2.1. Let the weights be $w_a=4$, $w_b=3$, and $w_c=5$. Then we have $|T_1|=22$ and $|T_2|=20$. T_2 is optimum (cf. Fig. 2.5). This example also shows that in optimum trees the weight of the root is not always greater than the weights of its descendants. We can transform T_1 into T_2 by a permutation of codes; namely $01(b)\rightarrow 0(a)\rightarrow\lambda(c)\rightarrow 1(\lambda)$ (see Fig. 2.6). However moving the code b from the node 01 to the node 1, which would obviously result in a "better" tree, is forbidden, because b is not admissible to the node 1.

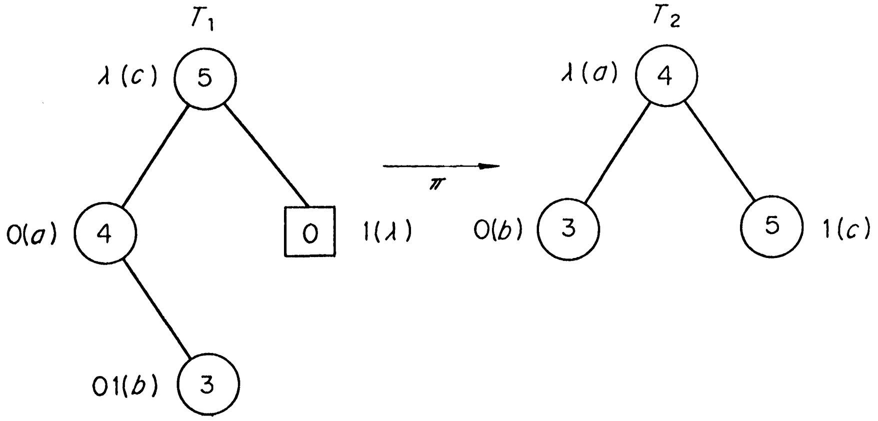


FIG. 2.6. *A transformation by the* p*-sequence* π

This observation leads us to define the following transformations.

Let $\pi: p_0(a_0), p_1(a_1), \cdots, p_n(a_n)$ be an ordered sequence of nodes of T satisfying:

1. $p_0, \cdots, p_n$ are distinct,
2. $a_i \neq \lambda$ $(i=0, 1, \cdots, n-1$, note that a_n may be $\lambda)$,
3. a_i is admissible to both p_i and p_{i+1} $(i=0, 1, \cdots, n)$, where $p_{n+1}=p_0$.

We transform the tree T by replacing the code a_{i+1} with a_i for $i=0, 1, \cdots, n$ (we put $a_{n+1}=a_0$, thus the code a_0 is replaced by a_n). The resulting tree is also a sequence tree from condition 3. Note that this condition implies the comparability of p_i and p_{i+1} by the relation $\succeq$ for $0\leqq i<n$ (also, of p_n and p_0 when $a_n\neq\lambda$). We call the sequence π a *permutation sequence* (abbreviated to p-*sequence*). The term p-sequence is used to represent the transformation by π as well. In the p-sequence, if $a_n=\lambda$, then π is a *shift*, else π is a *cycle*. The nodes p_0 and p_n are the *initial node* and the *terminal node* of π respectively (the p-sequence π *originates* from p_0 and *terminates* at p_n). The number n is called the *length of the* p-*sequence*. To represent the initial node and the terminal node of π explicitly, we use the notation $\pi_T[p_0, p_n]$. However we may omit $[p_0, p_n]$ as well as T.

We can transform a given tree to an arbitrary tree by successively applying p-sequences. The tree resulting from successive applications (*composition*) of p-sequences $\pi_1, \pi_2, \cdots, \pi_m$ to T is written as $\pi_m \cdot \pi_{m-1} \cdots \pi_1 T$ (dots may be omitted).

An essential property of p-sequences is their ability to change the cost of a tree. We define the value of the p-sequence π on the tree T by:

$$\mathrm{val}_T(\pi) = |\pi T| - |T| = \sum_{i=0}^{n} (\ell_{i+1} - \ell_i) \cdot \mathrm{wt}(p_i),$$

where ℓ_i is the level of p_i. Note that this value is negative when π maps the tree T into a "better" one (with smaller cost). For optimization purposes, interest lies primarily in minimum p-sequences.

Two p-sequences $\pi_1[p, q]$ and $\pi_2[p, q]$ are equivalent (in symbol $\pi_1 \cong \pi_2$) iff $\mathrm{val}(\pi_1) = \mathrm{val}(\pi_2)$. A p-sequence π with $\mathrm{val}(\pi) = 0$ is called a *dummy* p-*sequence.* An important special case of a dummy p-sequence is the identity permutation ε, such that $\varepsilon T = T$ for all trees T. Note that any p-sequence of length 0, $\pi : p_0(a_0)$, is just another representation of the identity permutation.

A fundamental property of p-sequences, namely that they form a "complete" set of transformations on sequence trees, is expressed in the following lemma:

LEMMA 2.1 (decomposition lemma). *Let T_0 and T_1 be two sequence trees over the same code set. Then T_0 can be transformed into T_1 by the composition of* independent p-*sequences. In other words, there exist* p-*sequences $\pi_1, \pi_2, \cdots, \pi_m$ such that $T_1 = \pi_m \cdot \pi_{m-1} \cdots \pi_1 \cdot T_0$, and if $i \neq j$ then π_i and π_j have no node in common.*

Proof. This is an extension of the usual decomposition lemma of permutations. We omit the details here. □

It should be noted that the order of $\pi_1, \pi_2, \cdots, \pi_m$ is arbitrary, since they are independent of each other. Further if a node p_i is empty in T_0 and if it is nonempty in T_1, then the lemma implies that there is a p-sequence (in fact a shift) which terminates at $p_i$in T_0.

A cycle with length 1 is called a *transposition.* We can prove the following lemma:

LEMMA 2.2 (cycle decomposition lemma). *A cycle is decomposed into the composition of transpositions.*

Proof. The proof is not obvious, since arbitrary transpositions are not always possible on a sequence tree. An induction on the length of the cycle is used. □

An efficient optimization procedure can be designed in terms of the repeated application of "minimum" p-sequences.

3. Regular trees. In this section we define a subset of sequence trees which we call regular trees. Properties of regular trees are investigated. With each empty node a real number called the "potential" of the node is associated. Necessary and sufficient conditions of optimality are presented in terms of the potentials of empty nodes.

Let $p(a)$ and $q(b)$ be nodes of a tree T. If a is admissible to q and if b is admissible to p, then we say that a and b or, for convenience, p and q are *exchangeable* in T (in symbol $a \leftrightarrow b$, $p \leftrightarrow q$ or $p(a) \leftrightarrow q(b)$ in T). We say that p is admissible to q if the code of p is admissible to q. Here we note a few immediate consequences of the definitions. If $p \succ q$, then $p \leftrightarrow q$ iff p is admissible to q, since q is always admissible to p. If $p \leftrightarrow q$, then $p \leftrightarrow r$ holds for any node r such that $p \succ r \succ q$ (note that $q \leftrightarrow r$ does not necessarily hold). If p and q are exchangeable and not both empty, then they are comparable by $\succeq$. We use the notational conventions w_i and w_p for $\mathrm{wt}_T(p_i)$ and $\mathrm{wt}_T(p)$ respectively. A tree T is *regular* if for any two nonempty exchangeable nodes p and q in T, $p \succeq q$ implies $w_p \geqq w_q$. In other words, T is regular if no transposition can decrease its cost. Optimum trees are regular. Note that any tree over uniformly weighted codes (the weights of all codes are equal) is also regular from this definition.

Let p_0 be a node of T and $\pi: p_n, \cdots, p_0$ be a p-sequence of distinct nodes satisfying for $i = 0, 1, \cdots, n-1$:

1. $p_i \succ p_{i+1}$,
2. $p_i \leftrightarrow p_{i+1}$,
3. p_{i+1} is minimal that is admissible to p_i.

We say that π is an *exchangeable sequence* of p_0. Note that p_{i+1} is uniquely determined from p_i as the nonempty ancestor of p_i exchangeable with p_i which is the closest to p_i. To represent an exchangeable sequence we use the notation $\eta_T[p_n, p_0]$ instead of $\pi_T[p_n, p_0]$. If there is no exchangeable sequence originating from p_n and terminating at p_0, then $\eta_T[p_n, p_0]$ is defined to be the identity permutation ε.

Let $\eta_T[p_n, p_0]$ be the exchangeable sequence of p_0 of maximum length, i.e., there exists no node p_{n+1} that is admissible to p_n and $p_{n+1} \succ p_n$. Then the node p_n is called the *support node* (or simply *support*) of p_0 in T.

The support of p is denoted by p^*. Unless stated explicitly to the contrary we usually understand the exchangeable sequence of p as one that originates from p^* (i.e., $\eta_T[p^*, p]$, see Fig. 3.1). Note that a nonempty node $p_i \succ p$ admissible to p is included in the exchangeable sequence of p by definition. If there exists no nonempty node $p_i \succ p$ admissible to an empty node p, then we put $p^* = p$ (an empty node can be a support only of itself). In case that the support p^* coincides with p, the exchangeable sequence η is the identity permutation ε. The supports of empty nodes play an important role in the definition of the potential of the nodes.

For an empty node $r(\lambda)$ and a nonempty node $p(a)$ of a tree T, a *minimum shift originating from p and terminating at r* denoted by $\mu_T[p, r]$ is a shift whose value is a minimum in all these shifts. Further, we denote by $\mu_T[*, r]$ a *minimum shift terminating at* $r(\lambda)$ (or *minimum shift of* $r(\lambda)$) whose value is a minimum in all shifts terminating at $r(\lambda)$, where $*$ denotes a certain nonempty node of T. Analogously, a *minimum shift originating from* $p(a)$ (or *minimum shift of* $p(a)$) is denoted by $\mu_T[p, *]$, where $*$ denotes a certain empty node of T.

We call a shift $\pi: p_0, p_1, \cdots, p_n$ *ascending* if $p_i \prec p_{i+1}$ holds for $i = 0, 1, \cdots, n-1$. If $p_i \succ p_{i+1}$ holds for $i = 0, 1, \cdots, n-1$, then π is *descending*.

Let $r(\lambda)$ and $p(a)$ be nodes of T. If $p(a) \prec r(\lambda)$, then a *minimum ascending shift* (abbreviated to mas) *originating from p and terminating at r* is a shift whose value is a minimum in all these ascending shifts (we denote it by $\Delta_T[p, r]$). If $p(a) \succ r(\lambda)$, then a *minimum descending shift* (abbreviated to mds) *originating from p and terminating at r* is a shift whose value is a minimum in all these descending shifts (we denote it by $\nabla_T[p, r]$).

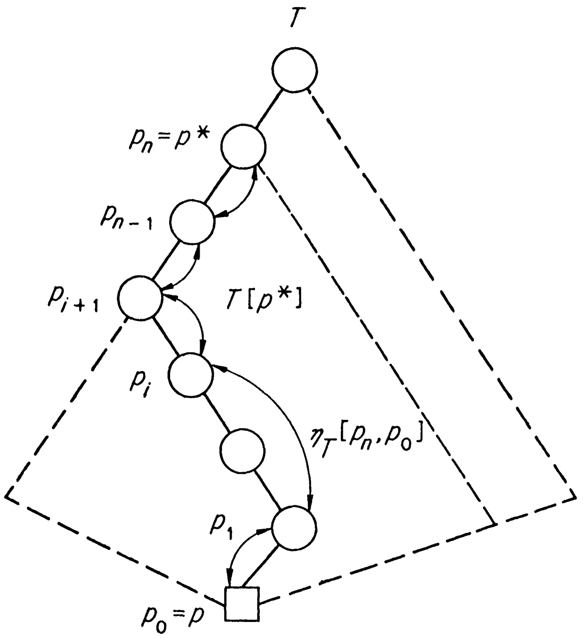


FIG. 3.1. *An exchangeable sequence and a support*

For $r(\lambda) \in T$, an mas *terminating at* $r(\lambda)$ (or mas *of* $r(\lambda)$) denoted by $\Delta_T[*, r]$ is an mas whose value is a minimum in all mas terminating at $r(\lambda)$. Analogously, for $p(a) \in T$, an mds *originating from* $p(a)$ (or mds *of* $p(a)$) denoted by $\nabla_T[p, *]$ is an mds whose value is a minimum in all mds originating from $p(a)$. Note that mas and mds are not necessarily unique (there may be many equivalent mas and mds, and equivalent mas or mds are identified), and in case that there are no such shifts, they are defined to be the identity permutation ε. Parameters $[p, r]$ may be omitted as well as suffix T in mas Δ, mds ∇ and minimum shifts μ in our notations.

Let $\pi : p_0, p_1, \cdots, p_n$ be a p-sequence in T. The maximal (highest) node of π is called the *top* of π (the top is uniquely determined). Minimal nodes of π are called *bottoms* of π. When π is a cycle, we adopt the convention that the initial node p_0 always represents a bottom. Let p_i be the top of π. Then the sequence $p_0, p_1, \cdots, p_i$ is called the *first part* of π (denote it by π^-). The sequence $p_i, p_{i+1}, \cdots, p_n$ is called the *second part* of π (denote it by π^+).

If π is a shift, then π^+ also represents a shift in T and π^- represents a shift in π^+T (note that p_i becomes empty in π^+T and hence the sequence π^- satisfies the condition of p-sequence in π^+T). Thus a shift π is represented by the composition of two shifts as $\pi = \pi^- \cdot \pi^+$.

A p-sequence π is called *backtrack-free* if π^+ is descending and π^- is ascending (see Fig. 3.2). Thus a backtrack-free cycle has a unique bottom and a backtrack-free shift has at most two bottoms (initial node and terminal node). In a p-sequence either $p_j > p_{j+1}$ or $p_j < p_{j+1}$ should hold for any two successive nodes. Therefore in π^+ (π^-) any pair of nodes p_j and p_{j+1} such that $p_j < p_{j+1}$ ($p_j > p_{j+1}$) is called a *backtrack* of π (π *has a backtrack at* p_j).

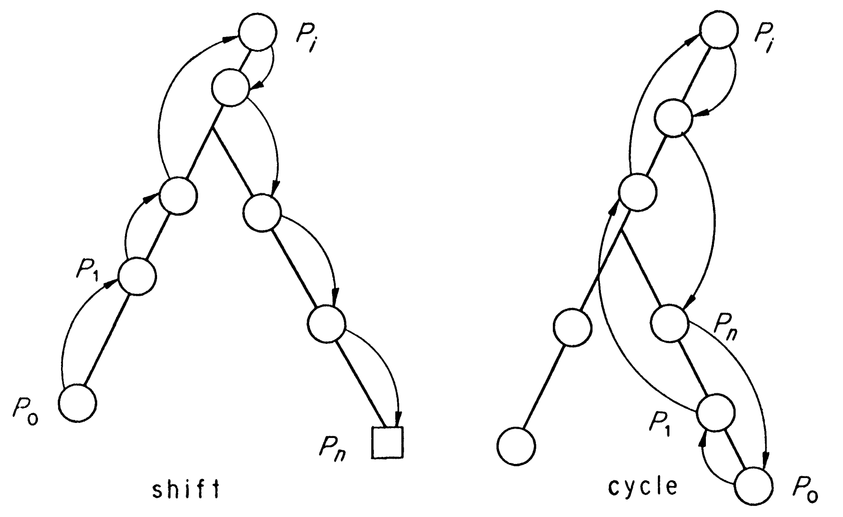


FIG. 3.2. *Backtrack-free* p-*sequences*

The following property holds for the mds and mas in a regular tree. We use the notational conventions ℓ_i and ℓ_p for lev(p_i) and lev(p) respectively.

LEMMA 3.1. *Suppose T is regular. Let an* mds $\nabla[p_0, p_n]$ (mas $\Delta[p_0, p_n]$) *be* $\pi: p_0, p_1, \cdots, p_n$ *in* T; *then* $p_{i-1} > q > p_i$ $(p_{i-1} < q < p_i)$ *and* $q \leftrightarrow p_i$ $(p_{i-1} \leftrightarrow q)$ *imply* $w_{i-1} = w_q$ $(w_i = w_q)$ *for each* $i = 1, \cdots, n$.

Proof. (i) *In case of* mds. Assume that there exists a node q satisfying the conditions of the lemma. Then we have $p_{i-1} \leftrightarrow q$, so $w_{i-1} \geqq w_q$, since T is regular. Assume $w_{i-1} > w_q$. Then a sequence $\pi': p_0, p_1, \cdots, p_{i-1}, q, p_i, \cdots, p_n$ is a descending shift of p_0 (note that $q \leftrightarrow p_i$). And $\text{val}(\pi) - \text{val}(\pi') = (\ell_i - \ell_{i-1})w_{i-1} - ((\ell_q - \ell_{i-1})w_{i-1} + (\ell_i - \ell_q)w_q) = (\ell_i - \ell_q)(w_{i-1} - w_q) > 0$. This contradicts the minimality of π.

(ii) *In case of* mas. The proof is analogous to (i). □

The next lemma asserts that the regularity of trees is preserved under transformations by mas and mds.

LEMMA 3.2. *Let T be regular and π be any* mds (mas) *in T. Then πT is a regular tree.*

Proof. (i) *In case of* mds. Let $\pi: p_0, \cdots, p_n$ be an mds in T and $T_1 = \pi T$. Assume T_1 is not regular; then there exist nonempty nodes p and q in T_1 such that $p > q$, $p \leftrightarrow q$, and $\text{wt}_{T_1}(p) < \text{wt}_{T_1}(q)$. Since T is regular, either p or q must belong to π (note that p and q cannot both belong to π). These two cases are considered separately. Let w_i be the weight of the code of p_i in T, and w_p and w_q be the weights of p and q in T.

Case a. Assume $p \in \pi$ (let $p = p_i$, then $i \neq 0$, since p is nonempty in T_1). From $\text{CODE}_{T_1}(p_i) = \text{CODE}_T(p_{i-1})$ we have $\text{wt}_{T_1}(p) = w_{i-1}$, and from $p_i \leftrightarrow q$ in T_1, $\text{CODE}_T(p_{i-1})$ should be admissible to q in T (see Fig. 3.3). This means $p_{i-1} \leftrightarrow q$ in T. But our assumption that $\text{wt}_{T_1}(p) = w_{i-1} < \text{wt}_{T_1}(q) = w_q$ then contradicts the regularity of T.

Case b. Assume $q \in \pi$ (then $q = p_i$ with $i \neq 0$). The node $p > q$, admissible to $q = p_i$, has weight $w_p < w_q = w_{i-1}$. In fact, p is also an ancestor of p_{i-1} (if this were

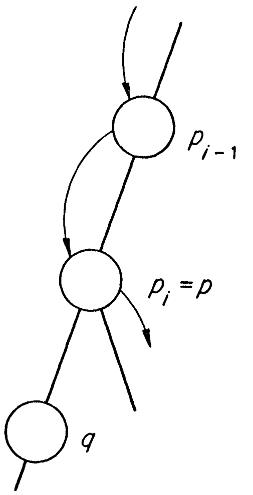


FIG. 3.3

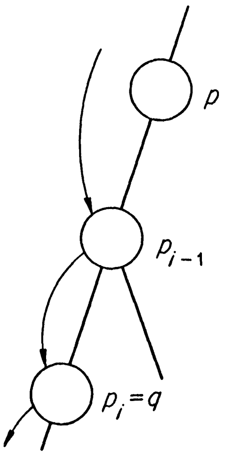


FIG. 3.4

not the case, Lemma 3.1 would imply $w_p = w_{i-1}$). Thus $p > p_{i-1} > q = p_i$ (see Fig. 3.4). Since $p \leftrightarrow p_i$ in T_1, we have $p \leftrightarrow p_{i-1}$ in T, and again our assumption that $\mathrm{wt}_{T_1}(p) = w_p < \mathrm{wt}_{T_1}(q) = w_{i-1}$ contradicts the regularity of T.

(ii) *In case of* mas. The proof is analogous to (i). □

The next two lemmas will show that in a regular tree a minimum shift has no backtrack.

LEMMA 3.3. *Let* $T[r(\lambda)]$ *be a regular tree with an empty root* $r(\lambda)$. *Then an* mas *of* $r(\lambda)$ *is a minimum shift terminating at* $r(\lambda)$, *i.e.*, $\Delta_T[*, r]$ *coincides with* $\mu_T[*, r]$.

Proof. It suffices to prove that if a shift π has a backtrack, then there is a backtrack-free shift whose value is not greater than π. Let $\pi : p_0, p_1, \cdots, p_n = r(\lambda)$ be a shift. Assume that the pair p_i and p_{i+1} is a backtrack and that there is no backtrack in $p_{i+1}, \cdots, p_n = r(\lambda)$ (i.e., $p_i > p_{i+1} < p_{i+2} < \cdots < r$). CODE($p_i$) is admissible to p_{i+1}, and since $p_{i+2} > p_{i+1}$, it is also admissible to p_{i+2} (see Fig. 3.5). Thus, by deleting p_{i+1} from π we can construct another shift π': $p_1, \cdots, p_i, p_{i+2}, \cdots, p_n$. Then $\mathrm{val}_T(\pi') - \mathrm{val}_T(\pi) = (\ell_{i+2} - \ell_i)w_i - ((\ell_{i+1} - \ell_i)w_i + (\ell_{i+2} - \ell_{i+1})w_{i+1}) = (\ell_{i+2} - \ell_{i+1})(w_i - w_{i+1})$. Since T is regular we have $w_i \geqq w_{i+1}$, and from the assumption, $\ell_{i+2} - \ell_{i+1} < 0$. Hence $\mathrm{val}(\pi') - \mathrm{val}(\pi) \leqq 0$. By repeating this backtrack elimination process we have a backtrack-free shift whose value is not greater than π. Note that the initial and the terminal nodes do not change by these backtrack eliminations. Thus an mas of $r(\lambda)$ is a minimum shift terminating at $r(\lambda)$. □

LEMMA 3.4. *Let* $T[r(a)]$ *be a regular tree with a nonempty root* $r(a)$. *Then an* mds *of* $r(a)$ *is a minimum shift originating from* $r(a)$, *i.e.*, $\nabla_T[r, *]$ *coincides with* $\mu_T[r, *]$.

Proof. The proof is analogous to that of Lemma 3.3. □

From Lemma 3.3 and Lemma 3.4 we have the following theorem.

THEOREM 3.1. *In a regular tree, for any* p-*sequence* π *there is a backtrack-free* p-*sequence* π' *such that* $\mathrm{val}(\pi') \leqq \mathrm{val}(\pi)$ *and the tops, the initial and the terminal nodes of* π *and* π' *are identical.*

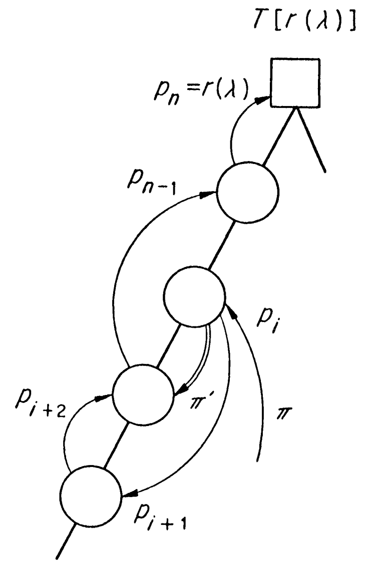


FIG. 3.5

Proof. First we note that the backtrack elimination process is applicable to a cycle as well as a shift. Hence we apply the process of Lemma 3.3 to the first part of π and eliminate all backtracks. Analogously the process of Lemma 3.4 is applied to the second part of π. The resulting p-sequence is π'. Note that the tops, the initial and the terminal nodes are unchanged by the backtrack eliminations. □

Immediately we have:

COROLLARY 3.1. *In a regular tree a minimum shift has no backtrack (more precisely, there is a backtrack-free minimum shift).*

Let p_i be the top of $\mu[p, q]$, then $\mu[p, q] = \mu^- \cdot \mu^+ = \Delta[p, p_i] \cdot \nabla[p_i, q]$ from Corollary 3.1, Lemma 3.3 and Lemma 3.4. Hence from Lemma 3.2 we can say that regularity is preserved under transformations by minimum shifts. (However this assertion does not hold for minimum cycles.)

The next theorem asserts that $\mathrm{val}(\pi) \geqq 0$ for any cycle π in a regular tree.

THEOREM 3.2. *If* $\mathrm{val}(\pi) < 0$, *then* π *is a shift in a regular tree T.*

Proof. Assume that $\pi : p_0(a_0), p_1(a_1), \cdots, p_n(a_n)$ is a cycle. We prove that $\mathrm{val}(\pi) \geqq 0$ by using the induction on the length n of the cycle. If $n = 1$, then π is a transposition. Hence we have $\mathrm{val}(\pi) \geqq 0$ because T is regular. Now assume that all cycles of length $n - 1$ have nonnegative values. From Theorem 3.1 we may assume π backtrack-free. Since p_0 is the bottom of the cycle, it is easy to see that p_n is admissible to p_1. Hence π can be decomposed into $\pi = \delta \cdot \pi'$, where $\pi' : p_1(a_1), \cdots, p_n(a_n)$ is a cycle with the length $n - 1$ in T and $\delta : p_0(a_0), p_1(a_n)$ is a transposition in $\pi' T$. From the induction hypothesis we have $\mathrm{val}(\pi') \geqq 0$. Since T is regular and $p_n > p_0$, we have $w_{a_n} \geqq w_{a_0}$. Hence $\mathrm{val}(\delta) \geqq 0$. Thus we have $\mathrm{val}(\pi) \geqq 0$. □

In the remaining part of this section, let $r(\lambda)$ and $p(a)$ denote any empty and nonempty nodes respectively. We consider the decomposition of $\mu[*, r(\lambda)]$ and

$\mu[p(a), *]$ in terms of ∇ and Δ in a regular tree T and give necessary and sufficient conditions of optimum trees. The following lemmas on properties of shifts are preparations for these purposes.

LEMMA 3.5. *Let $\pi : p_0, p_1, \cdots, p_i, \cdots, p_n = r(\lambda)$ be a shift which terminates at $r(\lambda)$ and p_i be the top of π in T. Then p_i belongs to the exchangeable sequence of r, namely $\eta[r^1, r]$.*

Proof. If $p_i = r$ the lemma is true. Assuming $p_i \neq r$, we have in fact $p_i > r$, since any p-sequence with the top p_i terminates in $T[p_i]$. Let j be the largest integer such that $i \leqq j < n$ and $p_i > r$. Since p_j is admissible to p_{j+1} and $p_{j+1} \preceq r$, p_j is admissible to r. Then p_j is an element of $\eta[r^1, r]$ from the property of exchangeable sequences. If $p_j = p_i$ we are done. Otherwise, consider the sequence $p_0, p_1, \cdots, p_i, \cdots, p_j$. The conditions of this lemma hold for the case where r is replaced by p_j. Thus repeating this process we conclude that p_i is an element of $\eta[r^*, r]$. □

LEMMA 3.6. *Let π be a shift which terminates at $r(\lambda)$ in T. Then π is decomposed into the composition of two shifts $\pi = \pi_2 \cdot \pi_1$, where π_1 originates from r^* in T and π_2 terminates at r^* in $T_1 = \pi_1 T$.*

Proof. Let p_i be the top of $\pi : p_0, \cdots, p_n = r(\lambda)$. Then $p_i \in [r, r^*]$ from Lemma 3.5. Let the exchangeable sequence originating from r^* and terminating at p_i be $q_m = r^*, \cdots, q_{h+1}, q_h = p_i$. We put $\pi_1 : q_m, \cdots, q_h = p_i, p_{i+1}, \cdots, p_n$ and $\pi_2 : p_0, \cdots, p_i = q_h, q_{h+1}, \cdots, q_m$. Then obviously we have $\pi = \pi_2 \cdot \pi_1$ (see Fig. 3.6). □

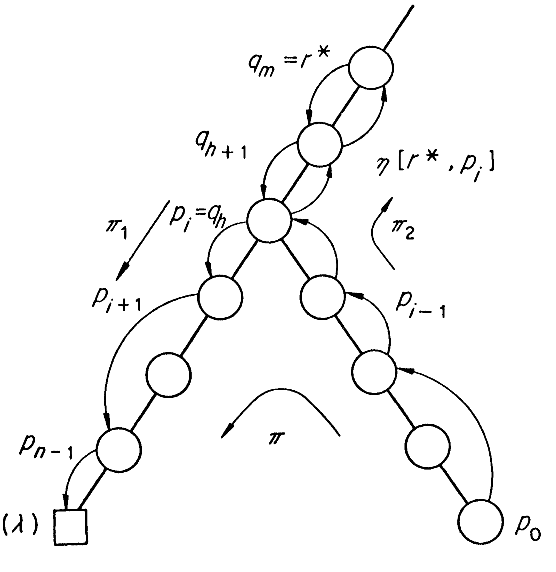


FIG. 3.6. *Decomposition of $\pi = \pi_2 \cdot \pi_1$*

LEMMA 3.7. *Let r be an empty node of a regular tree T and $\eta[p, r]$ be the exchangeable sequence of r originating from p. Then the exchangeable sequence $\eta_T[p, r]$ is a* mds $\nabla_T[p, r]$.

Proof. This lemma will be restated later as Lemma 5.2 and will be proved there. □

LEMMA 3.8. *Let T be regular and π_1 be an arbitrary descending shift. Put $T_1 = \pi_1 T$. Then for any ascending shift π_2 that is well-defined in T and T_1, we have*:

$$\mathrm{val}_{T_1}(\pi_2) \leqq \mathrm{val}_T(\pi_2).$$

Proof. If π_1 and π_2 have no interaction, then the codes of π_2 do not change in T and T_1. Hence we have $\mathrm{val}_{T_1}(\pi_2) = \mathrm{val}_T(\pi_2)$. Consider the case that π_1 and π_2 interact. Let $p \in \pi_1 \cap \pi_2$. Let the codes at p in T and T_1 be a and a_1 respectively. Then we have $w_a \leqq w_{a_1}$, since π_1 is descending and T is regular. This implies $\mathrm{val}_{T_1}(\pi_2) \leqq \mathrm{val}_T(\pi_2)$. □

We extend the definition of equivalence of p-sequences. If $\mathrm{val}(\pi[p, q]) = \mathrm{val}(\pi_2[p, r] \cdot \pi_1[r, q])$, then π is *equivalent* to $\pi_2 \cdot \pi_1$ (in symbol $\pi \cong \pi_2 \cdot \pi_1$).

The next theorem asserts that $\mu[*, r(\lambda)]$ is decomposed into the equivalent $\Delta \cdot \nabla$ in a regular tree.

THEOREM 3.3. *Let T be regular and $r(\lambda)$ be any empty node of T. Then we have*:

$$\mu_T[*, r(\lambda)] \cong \Delta_{T_1}[*, r^*(\lambda)] \cdot \nabla_T[r^*, r(\lambda)],$$

where r^* is the support of r in T and $T_1 = \nabla_T[r^*, r(\lambda)]T$.

Proof. We show that $\mathrm{val}_T(\Delta \cdot \nabla) \leqq \mathrm{val}_T(\pi)$ for any shift π terminating at $r(\lambda)$. From Theorem 3.1 we may assume that $\pi : p_0, p_1, \cdots, p_n = r(\lambda)$ is backtrack-free. Let the exchangeable sequence of $r(\lambda)$ be $\eta : q_m = r^*, q_{m-1}, \cdots, q_0 = r(\lambda)$. Then from Lemma 3.7 the exchangeable sequence $\eta[r^*, r]$ coincides with $\nabla[r^*, r]$. Since the top p_i of π is included in η from Lemma 3.5, we put $p_i = q_h$. Let $\pi = \pi_2 \cdot \pi_1$ as in Lemma 3.6. (Note that Lemma 3.6 is also valid in case that π is ascending.) Since we constructed π_1 and π_2 by using the sequence $\eta[r^*, q_h]$, these three sequences ∇, π_1 and π_2 (considered as the sets of addresses) coincide with each other on the path $[r^*, q_h]$. From the definition of $\nabla = \nabla_T[r^*, r]$ we have:

$$\mathrm{val}_T(\nabla) \leqq \mathrm{val}_T(\pi_1). \tag{3.1}$$

Let $T_1 = \nabla T$ and $T_2 = \pi_1 T$ and $\Delta = \Delta_{T_1}[*, r^*]$.

We will prove that:

$$\mathrm{val}_{T_1}(\Delta) \leqq \mathrm{val}_{T_2}(\pi_2). \tag{3.2}$$

We note that π_2 is well-defined in T_1 as well as Δ, because r^* is empty and all the other nodes of π_2 are not empty in T_1. However, the codes placed at the nodes of π_2 in T_1 may differ from those of π_2 in T_2, since ∇ and π_2 may interact. Hence $\mathrm{val}(\pi_2)$ in T_1 differs in general from $\mathrm{val}(\pi_2)$ in T_2 (see Fig. 3.7). However, from the definition of Δ, we have:

$$\mathrm{val}_{T_1}(\Delta) \leqq \mathrm{val}_{T_1}(\pi_2). \tag{3.3}$$

We will prove that:

$$\mathrm{val}_{T_1}(\pi_2) \leqq \mathrm{val}_{T_2}(\pi_2). \tag{3.4}$$

We note that π_2 in T_1 and T_2 on the path $[q_h, r^*]$ are the same. Hence it suffices to consider the subshift $\pi_2' : p_0, p_1, \cdots, p_{i-1}, p_i$ in T_1 and T_2 (we may assume that p_i is

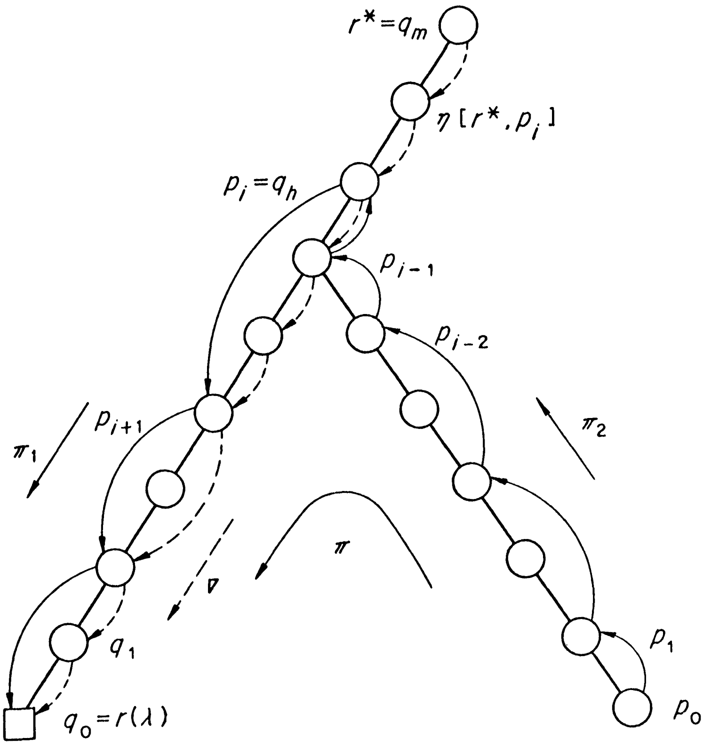


FIG. 3.7. ∇ *and* π_2 *interact at* p_{i-1}

empty at this moment). Also note that $\mathrm{val}_{T_2}(\pi'_2) = \mathrm{val}_T(\pi'_2)$. Applying Lemma 3.8 for subtree $T[p_i]$ and $T_1[p_i]$, we have

$$\mathrm{val}_{T_1}(\pi'_2) \leqq \mathrm{val}_T(\pi'_2) = \mathrm{val}_{T_2}(\pi'_2).$$

Hence we have (3.4).

Thus from (3.4) and (3.3) we have (3.2). Then from (3.1) and (3.2) we have $\mathrm{val}_T(\Delta \cdot \nabla) = \mathrm{val}_{T_1}(\Delta) + \mathrm{val}_T(\nabla) \leqq \mathrm{val}_{T_2}(\pi_2) + \mathrm{val}_T(\pi_1) = \mathrm{val}_T(\pi_2 \cdot \pi_1) = \mathrm{val}_T(\pi)$. □

Now we define the *potential* of an empty node $r(\lambda)$ in a regular tree by

$$\mathrm{potential}(r) = \mathrm{val}(\Delta[*, r^*] \cdot \nabla[r^*, r(\lambda)]).$$

Then we have the following:

THEOREM 3.4 (optimality theorem). *A regular tree T is optimum iff potential$(r) = 0$ for all empty nodes $r(\lambda)$ of T.*

Proof. From Theorem 3.2 and the decomposition lemma it is obvious that T is optimum iff $\mathrm{val}(\mu[*, r(\lambda)]) = 0$ for all $r(\lambda)$. From Theorem 3.3 we have $\mathrm{potential}(r) = \mathrm{val}(\mu[*, r(\lambda)])$. □

This is a generalization of the optimality theorem stated in [9], where the weights of the codes are uniform.

Concerning minimum shifts $\mu[p, *]$ originating from a nonempty node p in a regular tree T, we have a decomposition analogous to the one obtained for $\mu[*, r(\lambda)]$:

THEOREM 3.5. *In a regular tree T,*

$$\mu[p, *] \cong \min_{p \preceq q_j \preceq R} \Delta[p, q_j] \cdot \nabla[q_j, *],$$

where R is the root of T.

Proof. First we note $\mu[p, *] = \Delta[p, p_i] : \nabla[p_i, *]$, where p_i is the top of μ ($p \preceq p_i \preceq R$). Assume q be a node such that $\text{val}(\Delta[p, q] \cdot \nabla[q, *]) \leqq \text{val}(\Delta[p, q_j] \cdot \nabla[q_j, *])$ for all $p \preceq q_j \preceq R$. We prove $\mu[p, *] \cong \Delta[p, q] \cdot \nabla[q, *]$. Put $\nabla_0 = \Delta[q, *]$ and $\Delta_1 = \Delta[p, q]$. Then we have $\text{val}(\mu) \geqq \text{val}(\Delta_1 \cdot \nabla_0)$ from the definition of q. Assume that ∇_0 and Δ_1 interact at m nodes. If $m = 1$, then $q = p_i$ and we are done. Assume $m \geqq 2$. Let the interaction nodes be $r_1, r_2, \cdots, r_m$, where $r_1 = q > r_2 > \cdots > r_m$. Then from the decomposition lemma $\Delta_1 \cdot \nabla_0$ is the composition $\pi_1 \pi_2 \cdots \pi_m$, where π_h $(1 \leqq h \leqq m-1)$ is a cycle with the top r_h and π_m is a shift originating from p and terminating at the $*$ with the top r_m (see Fig. 3.8). Obviously π_m can be written as $\pi_m = \Delta[p, r_m] \cdot \nabla[r_m, *]$. Hence $\text{val}(\pi_m) \geqq \text{val}(\mu)$ from the definition of μ. From Theorem 3.2 we have $\text{val}(\pi_h) \geqq 0$ for $1 \leqq h \leqq m-1$. Hence $\text{val}(\Delta_1 \cdot \nabla_0) \geqq \text{val}(\pi_m)$. Thus $\text{val}(\mu) \geqq \text{val}(\Delta_1 \cdot \nabla_0) \geqq \text{val}(\pi_m) \geqq \text{val}(\mu)$. Hence $\text{val}(\Delta_1 \cdot \nabla_0) = \text{val}(\pi_m) = \text{val}(\mu[p, *])$. □

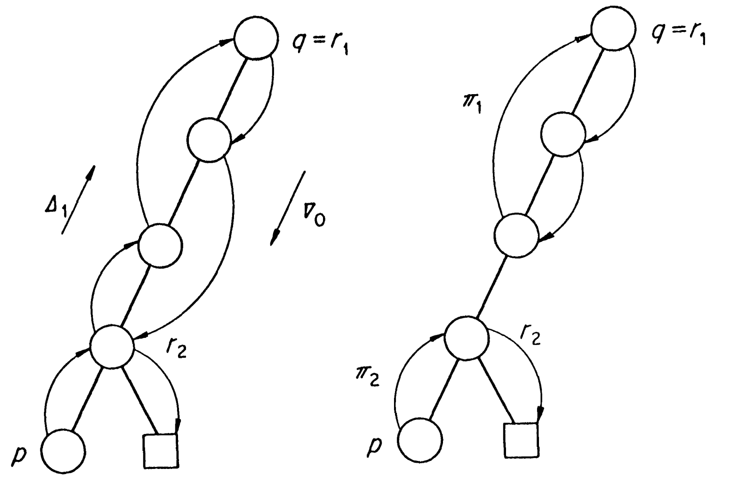


FIG. 3.8. *Decomposition of* $\Delta_1 \cdot \nabla_0$

4. Transformations into optimum trees. In this section we consider transformations of trees into optimum trees. The transformations are presented in terms of minimum shifts μ, mas Δ, mds ∇, and compositions of these shifts. The detailed descriptions of algorithms to find μ, Δ and ∇ are given in the following section.

A consideration of a bottom-up optimization leads us to the definition of two types of trees. A tree $T[r]$ is *preoptimum* if the root r is empty and its two subtrees $T[\text{SON}_0(r)]$ and $T[\text{SON}_1(r)]$ are optimum. A tree $T[r]$ is *quasioptimum* if the root r is not empty and its two subtrees $T[\text{SON}_0(r)]$ and $T[\text{SON}_1(r)]$ are optimum. Note that a preoptimum tree is regular but a quasioptimum tree is not necessarily regular. Further, note that a quasioptimum tree $T[r]$ is regular iff the weight of the code of the root is greater than or equal to the weights of its all admissible nodes.

THEOREM 4.1. *A preoptimum tree $T[r]$ is transformed into an optimum tree by a minimum ascending shift $\Delta_T[*, r]$.*

Proof. Since a preoptimum tree $T[r]$ is regular, an mas $\Delta[*, r]$ coincides with a minimum shift $\mu = \mu[*, r]$. Let $T_1 = \mu T$, and let T_0 be an arbitrary tree. We show that $|T_0| \geqq |T_1|$. From the decomposition lemma, T can be transformed into T_0 by a composition of independent p-sequences, $\pi_1 \cdot \pi_2 \cdots \pi_m$. Since the root r is empty in T and it is not empty in T_0, there is a shift terminating at $r(\lambda)$ among these p-sequences. Let this shift be π_1. Then we have $\text{val}(\mu) \leqq \text{val}(\pi_1)$, since μ is a minimum shift. The remaining p-sequences (including the identity permutation ε when $T_0 = \pi_1 T$) are ones in the two optimum subtrees of r; therefore they have nonnegative values. We have thus:

$$\begin{aligned}|T_0| &= \text{val}(\pi_1) + \text{val}(\pi_2) + \cdots + \text{val}(\pi_m) + |T|\\ &\geqq \text{val}(\mu) + |T| = |\mu T| = |T_1|.\end{aligned}$$

Since T_0 is arbitrary, this shows that T_1 is optimum. □

THEOREM 4.2. *A quasioptimum tree $T[r]$ is transformed into a preoptimum tree by a minimum shift $\mu_T[r, *]$.*

Proof. Put $\mu = \mu[r, *]$ and $T_1 = \mu T$. To show that T_1 is preoptimum, we show that $|\pi T_1| \geqq |T_1|$, for any p-sequence π such that πT_1 has an empty root. Let us decompose $\pi \cdot \mu$ into independent p-sequences $\pi_1, \cdots, \pi_m$. Then there is a p-sequence originating from r among these p-sequences, since r is not empty in T and it is empty in T_1. Let it be π_1. Since the remaining p-sequences (including the identity permutation ε when $\pi T_1 = \pi_1 T_1$) are ones in optimum subtrees $T[\text{SON}_0(r)]$ and $T[\text{SON}_1(r)]$, we have $\text{val}(\pi_i) \geqq 0$ for $i = 2, \cdots, m$.

From the definition of a minimum shift, we have $\text{val}(\mu) \leqq \text{val}(\pi_1)$. Thus

$$\begin{aligned}|\pi T_1| = |\pi \cdot \mu T| &= |\pi_1 \cdot \pi_2 \cdots \pi_m T|\\ &= \text{val}(\pi_1) + \sum_{i=2}^{m} \text{val}(\pi_i) + |T|\\ &\geqq \text{val}(\mu) + |T| = |\mu T| = |T|.\end{aligned}$$

This shows that T_1 is preoptimum. □

Since a quasioptimum tree $T[r]$ is not always regular, a minimum shift $\mu[r, *]$ does not coincide with an mds in general. Suppose the code of r is shifted to a node p_i by $\mu[r, *]$. Then the subsequence originating from the node p_i of the shift $\mu[r, *]$ should coincide with a minimum shift $\mu[p_i, *]$. Since $\mu[p_i, *]$ is a shift in the optimum subtree, it can be chosen with no backtrack. Thus there is a minimum shift $\mu[r, *]$ with at most one backtrack at the node p_i, and it can be represented as a shift $r, \mu[p_i, *]$. We leave the precise algorithms to find $\Delta[*, r]$ and $\mu[r, *]$ to the following section.

From Theorem 4.1 and Theorem 4.2 we can transform a quasioptimum tree into an optimum tree by successively applying μ and Δ (see Fig. 4.1). Thus, given an arbitrary tree, we can transform it into an optimum one by applying the above procedure to the all subtrees from the leaves to the root. A bottom-up optimization algorithm can be stated simply as follows.

ALGORITHM 4.1 (bottom-up optimization algorithm). Given an arbitrary tree, we can transform it into an optimum tree by applying the following procedure:

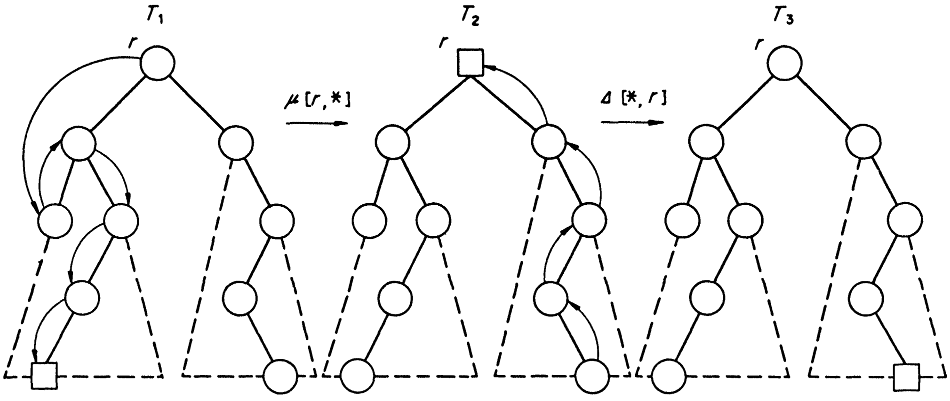
1. [Loop on LEV.] Do step 2 for all LEV $= L, L-1, \cdots, 0$ (L is the code length);

2. [Loop on r.] Do step 3 and step 4 for all subtrees $T[r]$ at the level LEV;

3. [Apply μ.] If r is nonempty, apply a minimum shift $\mu[r, *]$ to the quasioptimum subtree $T[r]$;

4. [Apply Δ.] Apply a minimum ascending shift $\Delta[*, r]$ to the resulting preoptimum tree $T[r]$;

Note that we can skip the processing of empty nodes lower than leaves and that the nodes can be processed in any convenient order, as long as every node is processed after its descendants.



A quasioptimum tree A preoptimum tree An optimum tree

FIG. 4.1. *Optimization of a quasioptimum tree*

Example 4.1. We can construct an optimum tree by applying Algorithm 4.1 to a prefix tree. Note that step 3 can always be omitted in this case, since we always have preoptimum trees in each stage. In Fig. 4.2, T_1 is a prefix tree and T_4^1 and T_4^2 are optimum trees. Note that if there are more than one mas Δ, then each choice of a mas Δ leads to another optimum tree.

Next we show that the optimum insertion problem can be solved as a corollary of Theorem 4.2.

THEOREM 4.3 (optimum insertion). *Let T be an optimum tree and a be a new code to be inserted into T. Let us consider a tree T' whose root r' has a code a and whose only subtree (either left or right) is T (r' is a dummy root and the code a is admissible to a path of T). Then T' is a quasioptimum tree; hence by a minimum shift $\mu_{T'}[r', *]$ of the root $r'(a)$, T (as a subtree of T') is transformed into a required optimum tree.*

Proof. The proof is immediate from Theorem 4.2 (see Fig. 4.3). □

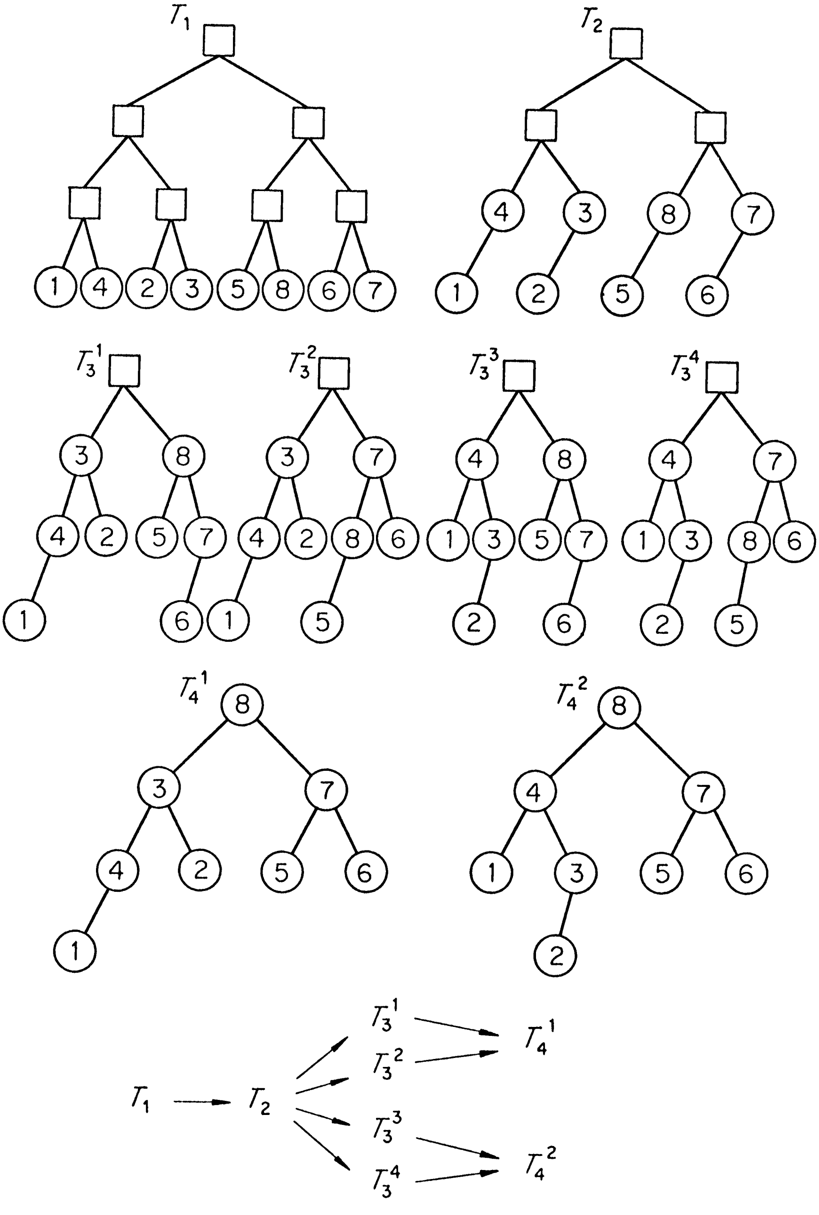


FIG. 4.2. *Optimum tree construction from a prefix tree*

The optimum deletion algorithm is a little different from the optimum insertion algorithm; however they are similar in the sense that a minimum shift plays an important role.

THEOREM 4.4 (optimum deletion). *Let T be an optimum tree and a be the code to be deleted from T. Delete the code a from a node $p(a)$. Let this tree be T_1 (i.e., the node $p(a)$ becomes $p(\lambda)$ in T_1). Perform a minimum shift $\mu = \mu_{T_1}[*, p(\lambda)]$ in T_1; then the resulting tree μT_1 is the required optimum tree.*

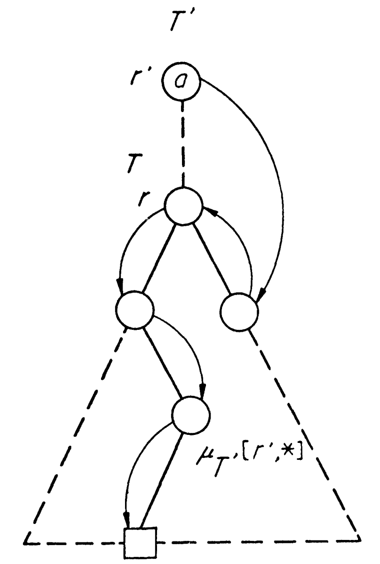


FIG. 4.3. *Optimum insertion*

Proof. Let $T_2 = \mu T_1$. We show $|T_2| \leqq |\pi T_2|$ for any p-sequence π. Let $T_3 = \pi T_2 = \pi \cdot \mu T_1$. Applying the decomposition lemma, we have $T_3 = \pi_2 \cdot \pi_1 T_1$, where π_1 is a shift terminating at p (we include the identity permutation ε as π_1), and π_2 is the composition of p-sequences not including p (cf. the discussion in the proof of Theorem 4.2). Then we have $\text{val}(\mu) \leqq \text{val}(\pi_1)$ from the definition of μ. Since T is optimum we have $\text{val}(\pi_2) \geqq 0$. Thus $|T_3| = \text{val}(\pi_1) + \text{val}(\pi_2) + |T_1|$, while $|T_2| = \text{val}(\mu) + |T_1|$; therefore we have $|T_3| \geqq |T_2|$. Since π is arbitrary, T_2 is optimum. □

Now from Theorem 3.3 (note that T_1 is regular) we have:

$$\mu_{T_1}[*, p(\lambda)] \cong \Delta_{T_2}[*, p^*(\lambda)] \cdot \nabla_{T_1}[p^*, p(\lambda)],$$

where p^* is the support of p and $T_2 = \nabla_{T_1}[p^*, p(\lambda)]T_1$ (see Fig. 4.4).

More generally, it results from our previous discussions that all our optimization problems reduce to those of finding:

1. an mas $\Delta_T[p(a), r(\lambda)]$ in a regular (or preoptimum) tree $T[r(\lambda)]$;
2. an mds $\nabla_T[r(a), p(\lambda)]$ in a regular (or optimum) tree $T[r(a)]$;
3. a minimum shift $\mu_T[r(a), *]$ in a quasioptimum tree $T[r(a)]$.

5. Algorithms to find minimum shifts. This section is composed of two parts. In the first part we describe algorithms to find an mds $\nabla_T[r(a), p(\lambda)]$ and an mas $\Delta_T[p(a), r(\lambda)]$ in a regular tree T. In the second part we consider an algorithm of optimum insertion, that is, to find a minimum shift $\mu_T[r(a), *]$ in a quasi-optimum tree $T[r(a)]$.

5.1. Algorithms to find $\nabla_T[r, p]$ and $\Delta_T[p, r]$. The root r of the tree $T[r]$ is assumed to be nonempty when we consider $\nabla[r, p]$, while it is assumed empty

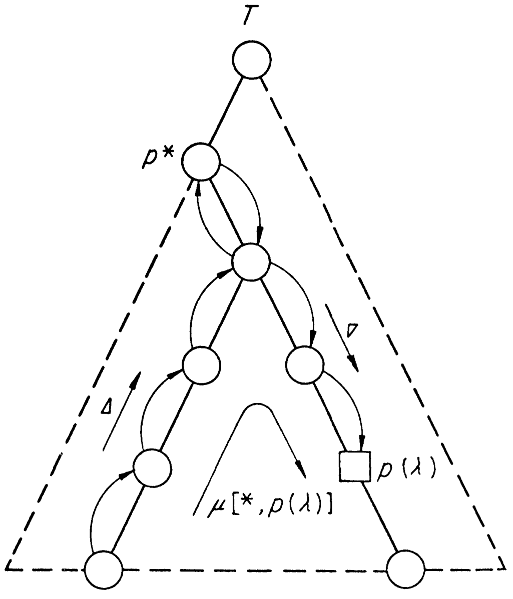


FIG. 4.4. *Optimum deletion*

when we consider $\Delta[p, r]$. We note that from the definition of the symbol $*$ we have:

$$\nabla_T[r, *] = \min_{p(\lambda)\in T} \nabla_T[r, p(\lambda)] \quad \text{and}$$

$$\Delta_T[*, r] = \min_{q(a)\in T} \Delta_T[q(a), r].$$

We consider algorithms to find $\nabla[r, v]$ and $\Delta[t, r]$ for an external node v and a leaf t (the same algorithms are valid when we replace v and t by any $p(\lambda)$ and $q(a)$ respectively).

Before showing lemmas we need some preparations. We associate a function with each shift π. The discussion is done separately for the descending and ascending case.

Let $\pi: p_0 = r, p_1, \cdots, p_m = v(\lambda)$ be a descending shift on a path $[r, v]: q_0 = r, q_1, \cdots, q_n = v(\lambda)$. Then there exists a unique p_j for each q_i $(i \neq 0)$ such that $p_j > q_i \geq p_{j+1}$. The node p_j is the *cover of* q_i (p_j *covers* q_i) with respect to π. The *canonical value function* f_π *associated with* π is defined by

$$f_\pi(q_i) = \text{wt}(p_j) \quad \text{for } i = 1, \cdots, n.$$

Note that $f_\pi(q_0)$ is not defined. Since p_j covers $\ell_{j+1} - \ell_j$ nodes between p_j and p_{j+1} (including p_{j+1}), we have the following equation:

$$\text{val}(\pi) = \sum_{i=1}^{n} f_\pi(q_i). \tag{5.1}$$

LEMMA 5.1. *Let $\eta[r, v]$ be an exchangeable sequence and $\pi[r, v]$ be an arbitrary descending shift on the path $[r, v]$. Then the cover of q_i with respect to π is greater than or equal to the cover of q_i with respect to η for any $q_i \in [r, v]$.*

Proof. From the property of the exchangeable sequence it is obvious that η is a refinement of π. From this the lemma results immediately. □

LEMMA 5.2. *Let $[r, v]$ be a path of a regular tree. Then the exchangeable sequence $\eta[r, v]$ coincides with an* mds $\nabla[r, v]$.

Proof. When $\eta = \varepsilon$ (i.e., there exists no exchangeable sequence originating from r and terminating at v), ∇ coincides with ε (i.e., there exists no shift originating from r and terminating at v). Let $\eta[r, v]$ be $s_l = r, s_{l-1}, \cdots, s_0 = v$ and $\pi: p_m = r, p_{m-1}, \cdots, p_0 = v$ be an arbitrary descending shift originating from r and terminating at v. Let f_η and f_π be the canonical value functions associated with η and π. We show that:

$$f_\eta(q_i) \leqq f_\pi(q_i) \quad \text{for any } q_i \in [r, v].$$

Suppose the covers of q_i with respect to η and π be s_j and p_h respectively; then we have $q_i \prec s_j \preceq p_h$ from Lemma 5.1. Since p_h is admissible to s_j and T is regular, we have $w_{s_j} \leqq w_{p_h}$. Hence $f_\eta(q_i) \leqq f_\pi(q_i)$, and from (5.1) we have:

$$\mathrm{val}(\eta) = \sum_{i=1}^{n} f_\eta(q_i) \leqq \sum_{i=1}^{n} f_\pi(q_i) = \mathrm{val}(\pi).$$

Since π is arbitrary, η is an mds ∇. □

The exchangeable sequence of a node is easily computed by the following algorithm.

ALGORITHM 5.1. The following algorithm enumerates the exchangeable sequence $\eta[r, v]: s_l = r, s_{l-1}, \cdots, s_0 = v$ in reverse order (if $l = 0$ is returned, $\eta[r, v]$ is ε).

```
1. [Initialize.] $l := 0$; $s_0 := v$; $q := v$;
2. [Advance.] $q :=$ FATHER($q$);
3. [Admissible?] if $q$ is nonempty and admissible to $s_l$
   then $l := l+1$ and $s_l := q$;
4. [End of path?] if $q \neq r$ then go to 2,
   else {if $q \neq s_l$ then $l := 0$}; terminate; ($l$ is the length of the exchangeable
   sequence, $\eta: s_l, s_{l-1}, \cdots, s_0$).
```

Now we consider a minimum ascending shift $\Delta[t, r]$. In case that π is ascending, p_j covers q_i if $p_j \prec q_i \preceq p_{j+1}$, and analogously to (5.1) we have:

$$\mathrm{val}(\pi) = -\sum_{i=1}^{n} f_\pi(q_i). \tag{5.2}$$

LEMMA 5.3. *Let $[t, r]: q_0 = t, q_1, \cdots, q_n = r(\lambda)$ be an ascending path of a tree. With each node q_i, for $1 \leqq i \leqq n$, we associate a "super-weight" W_i by:*

$$W_i = \max_{0 \leqq j \leqq i-1} w_j \qquad (W_0 \textit{ is not defined}).$$

Then $\mathrm{val}(\Delta[t, r]) = -\sum_{i=1}^{n} W_i$.

Proof. Let $\pi : p_0 = t, p_1, \cdots, p_m = r(\lambda)$ be an arbitrary ascending shift and f_π be the canonical value function associated with π. We prove $f_\pi(q_i) \leqq W_i$ for each q_i.

Suppose p_j is the cover of q_i with respect to π. Then we have $p_j < q_i \preceq p_{j+1}$ and $f_\pi(q_i) = w_{p_j}$. On the other hand W_i is the maximum weight of the codes which are placed at the nodes strictly lower than q_i. So we have $f_\pi(q_i) \leqq W_i$. Thus from (5.2) we have $\mathrm{val}(\pi) = -\sum_{i=1}^{n} f_\pi(q_i) \geqq -\sum_{i=1}^{n} W_i$. Since π is arbitrary, this shows that $\mathrm{val}(\Delta[t, r]) \geqq -\sum_{i=1}^{n} W_i$. But this lower bound can be achieved with Algorithm 5.2 below. □

ALGORITHM 5.2. The next algorithm enumerates an mas $\Delta[t, r]: s_0 = t, s_1, \cdots, s_l = r$ on a path $[t, r]$:

```
1. [Initialize.] MAX := $w_t$; $l$ := 1; $s_0$ := $t$; $q$ := $t$;
2. [Advance.] $q$ := FATHER($q$);
3. [Compare.] if wt($q$) > MAX then
            {MAX := wt($q$); $s_l$ := $q$; $l$ := $l+1$};
4. [Done?] if $q = r$ then $s_l$ := $q$ and terminate, else go to 2;
```

5.2. An optimum insertion algorithm. A quasioptimum tree $T[r, (a)]$ is transformed into a preoptimum tree by a minimum shift $\mu[r(a), *]$ from Theorem 4.2. The code a is shifted to one of its admissible nodes by μ. For simplicity of notations, we denote the admissible nodes of a by numbers $n, n-1, \cdots, 1, 0$ from the root r to the external node successively (n denotes the root r and 0 denotes the external admissible node of a). Suppose the code a is shifted to a node i by $\mu[r, *]$. Then $\mu[r, *]$ can be represented as a shift $r, \mu[i, *]$. Since $\mu[i, *]$ is a shift in the optimum subtree $T[n-1]$, $\mu[i, *]$ can be represented as $\mu[i, *] \cong \min \Delta[i, y] \cdot \nabla[y, *]$ from Theorem 3.5, where $i \preceq y \preceq n-1$. Generally $\Delta[i, y] \cdot \nabla[y, *]$ involves dummy cycles. However, if we select j as the smallest (lowest) node on the path $[i, n-1]$ which gives $\min \Delta[i, y] \cdot \nabla[y, *]$, then $\Delta[i, j] \cdot \nabla[j, *]$ involves no dummy cycle (cf. the proof of Theorem 3.5). Hence it coincides with $\mu[i, *]$.

In a quasioptimum tree $T[r(a)]$ the node i to which the code a is shifted by $\mu[r, *]$ is determined as a node $x (0 \preceq x \preceq n-1)$ which minimizes $(\ell_x - \ell_r) w_a + \mathrm{val}(\mu[x, *])$.

The above discussions can be summarized by the following algorithm.

ALGORITHM 5.3. This algorithm finds a minimum shift $\mu[r, *]$ in a quasioptimum tree $T[r(a)]$. The admissible path of a is denoted by $[0, n]$. The nodes x and y denote a node on the path $[0, n-1]$ and the top of $\mu[x, *]$ respectively.

```
1. [Determine ∇.] Determine $\nabla[y, *]$ for $y < r$ on the admissible path of $a$,
   i.e., for $y \in [0, n-1]$;
2. [Loop on x.] Initialize MIN := $nw_a$ and $\mu$ := $n, 0$; then do step 3 for
   $x \in [1, n-1]$;
3. [Loop on y.] Do step 4 for $y \in [x, n-1]$;
4. [Update μ.] Determine $\Delta[x, y]$ in $\nabla[y, *]T$; and if $(\ell_x - \ell_r) w_a +$
   val($\Delta \cdot \nabla$) < MIN then update MIN and $\mu$ accordingly;
```

Further, the following two remarks can be proved on the algorithm.

(i) "Backtrackable" node $x \in [1, n-1]$ can be restricted to a monotone sequence of nodes $i_1 > i_2 > \cdots > i_k$ such that $w_{i_1} \geqq w_{i_2} \geqq \cdots \geqq w_{i_k}$ $(> w_a)$. (If $w_j > w_i$ for $j < i$, then i can not be backtrackable.)

(ii) The minimum shifts $\Delta[x, y]$ can be determined actually in T and not in ∇T.

Example 5.1. Consider a quasioptimum tree $T[r(a)]$ that has T_4^1 of Example 4.1 as its only subtree (Fig. 5.1). The admissible nodes of the code a are numbered as 0, 1, 2, 3, 4 along a broken line which indicates the admissible path of a (w_a, the weight of the code a, is simply denoted by w in this example). The node p_8 has no son. This is because we restricted the code length to $L = 3$. (Cf. Example 4.1.) Table 5.2 shows possible combinations of x and y ($x = 0$ and $1 \leqq x \leqq y \leqq 3$) and the values of shifts corresponding to them. The corresponding trees are shown in Fig. 5.2. Figure 5.3 indicates the optimum domains of the resulting trees. Note that T_c coincides with T_b, and $|T_b| = |T_c| < |T_d|$, $|T_e| < |T_f|$. Also note that each of the cases d, f and g of Table 5.1 has an equivalent tree (cf. T_3^1 and T_3^2 of Fig. 4.2). When $w \geqq 8$, the tree $T[r]$ becomes regular, and hence from Lemma 5.2 a minimum shift $\mu[r, *]$ coincides with one of the exchangeable sequences of Table 5.1. (Table 5.1 shows all exchangeable sequences originating from r and terminating at an external node). If $8 \leqq w \leqq 16$, η_2 becomes a minimum, while if $16 \leqq w$, then η_4 or η_6 is a minimum. This observation is in accordance with Fig. 5.3.

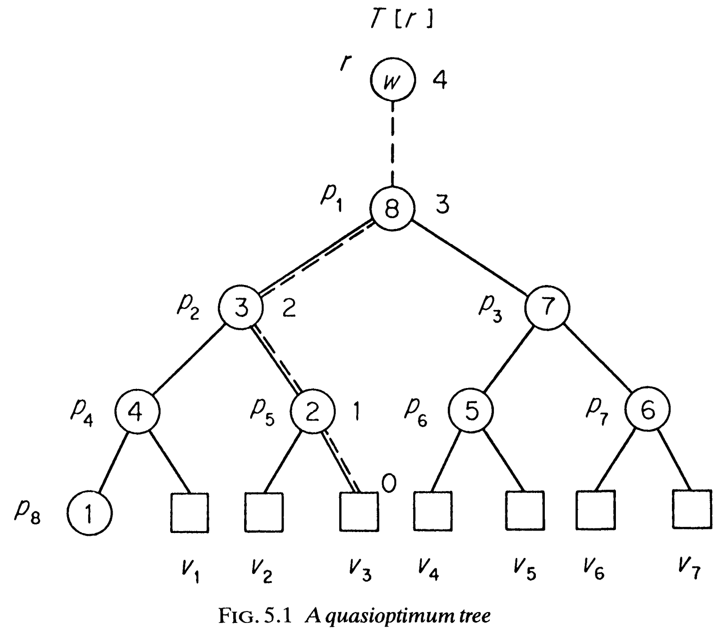


FIG. 5.1 *A quasioptimum tree*

6. Implementations of the algorithms. In this section an upper bound on the number of operations needed to optimize a tree according to Algorithm 4.1 (bottom-up optimization) is evaluated very roughly. First we evaluate the number of operations to find $\nabla[r, *]$ and $\Delta[*, r]$. Then we consider $\mu[r, *]$ of a quasioptimum tree. According to the bottom-up optimization algorithm the number of

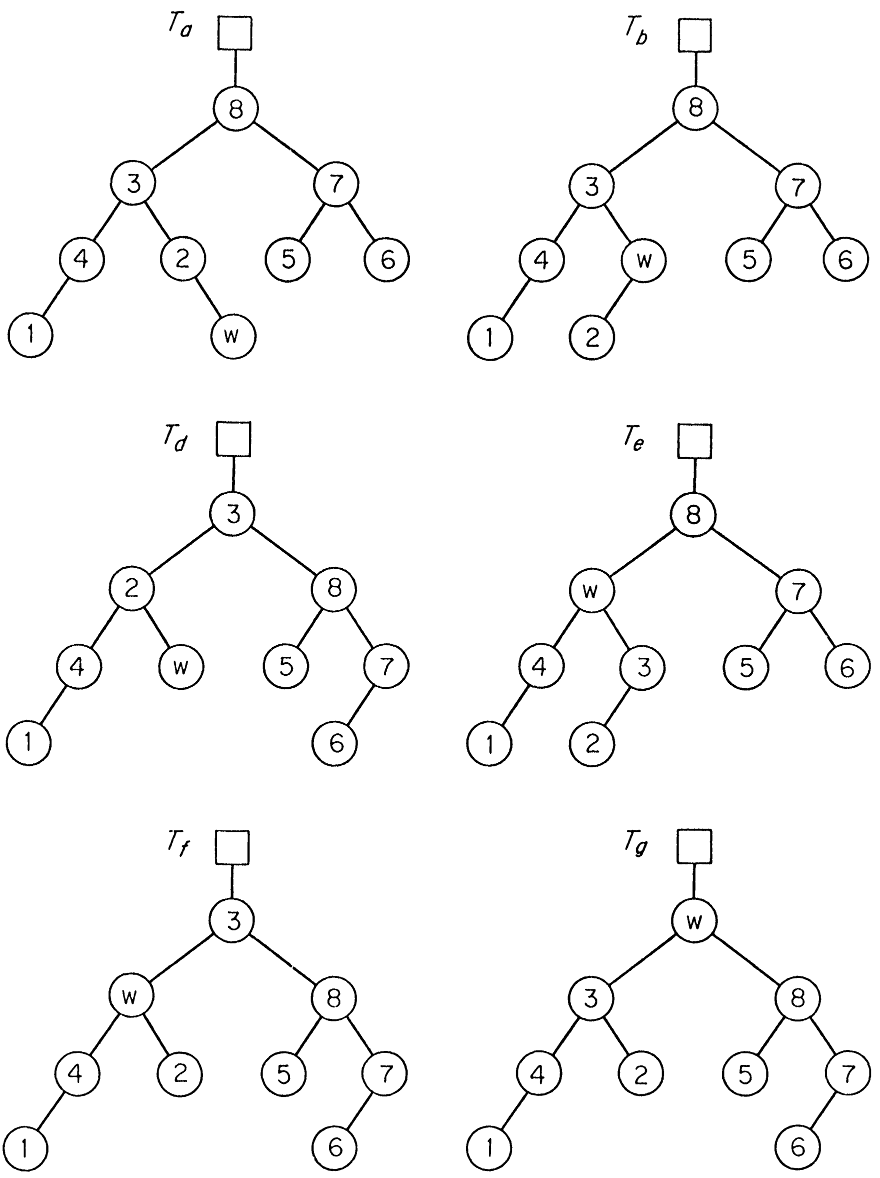


FIG. 5.2. *Transformed trees*

operations to optimize a tree is the sum of operations to optimize all subtrees of the tree. The number of operations to optimize a quasioptimum subtree $T[r(a)]$ is the sum of operations to find and perform $\mu[r, *]$ of the quasioptimum subtree and $\Delta[*, r]$ of the resulting preoptimum subtree.

6.1. Computations of $\nabla[r, *]$ and $\Delta[*, r]$. To find a minimum descending shift $\nabla[r, v]$ on the path $[r, v]$, Algorithm 5.1 needs $\ell_v - \ell_r$ *admissibility tests* (or in short, A-*tests*).

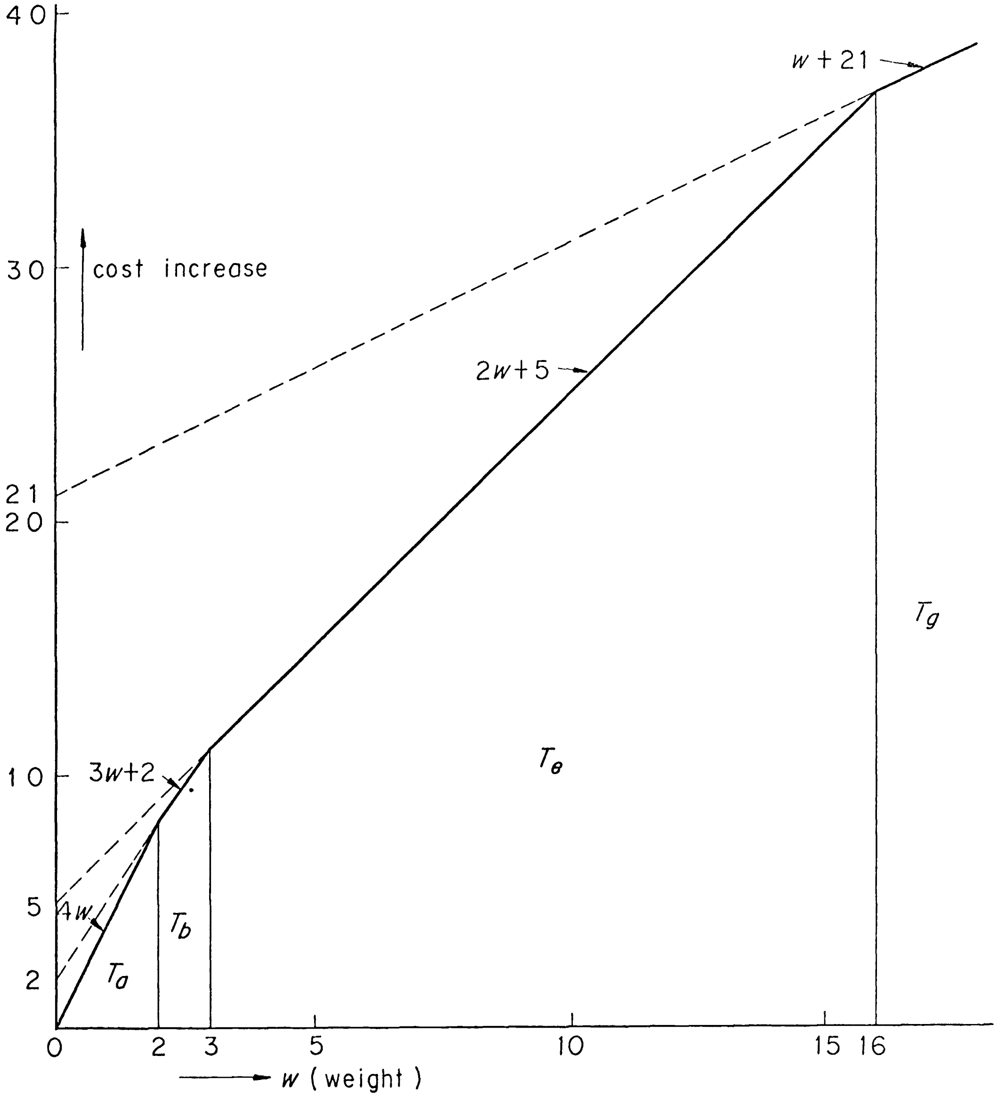


FIG. 5.3. *Optimum domains of the transformed trees*

Similarly, to find a minimum ascending shift $\Delta[t, r]$ on the path $[t, r]$, Algorithm 5.2 needs $\ell_t - \ell_r$ *weight comparisons* (in short, C-*tests*).

Now consider the number of tests needed to find $\nabla[r, *]$ in an quasioptimum tree $T[r]$. We must minimize the value of $\nabla[r, v]$ over the set of external nodes v of T. The corresponding number of A-tests is thus bounded by:

$$\sum_{v \in T} (\ell_v - \ell_r). \tag{6.1}$$

This bound is not tight (observe that $\eta[r, v]$ is ε, when v and CODE(r) are not on the same side of the root r).

TABLE 5.1

Exchangeable sequence $\eta_i[r, v_i]$

η_i	Sequence	Value
η_2	r, p_2, p_5, v_2	$2w+5$
η_3	r, p_2, v_3	$2w+6$
η_4	r, p_1, p_6, v_4	$w+21$
η_5	r, p_1, v_5	$w+24$
η_6	r, p_1, p_3, p_7, v_6	$w+21$
η_7	r, p_1, p_3, v_7	$w+22$

TABLE 5.2

Computation of a minimum shift

Case	x	y	$(\ell_x-\ell_r)\cdot w$	$\Delta[x, y]$	$\nabla[y, *]$	Total
a	0	0	$4w$	0	0	$4w$
b	1	1	$3w$	0	2	$3w+2$
c	1	2	$3w$	-3	$3+2$	$3w+2$
d	1	3	$3w$	$-3-2$	$8+7+6$	$3w+16$
e	2	2	$2w$	0	$3+2$	$2w+5$
f	2	3	$2w$	-3	$8+7+6$	$2w+18$
g	3	3	w	0	$8+7+6$	$w+21$

In the same way, to find $\Delta[*, r]$ in a preoptimum tree $T[r]$, we must minimize the value of $\Delta[t, r]$ over the set of leaves t of T. The corresponding number of C-tests is thus bounded by:

$$\sum_{t\in T} (\ell_t-\ell_r). \tag{6.2}$$

Note that this evaluation is tight in the sense that this is necessary to determine $\Delta[*, r]$. In addition to these operations we need as many "value comparisons" as the number of external nodes and leaves to determine the minimum ∇ and Δ respectively. These value comparisons, identified as weight comparisons, can be neglected.

As for storage we must always keep in memory a minimum shift up to the v or t that is processed at that time. The amount of storage needed for this shift is bounded by the height, the length of the longest path of the leaves, of $T[r]$. In the following discussions we focus our attention only on the numbers of A-tests and C-tests, because they are characteristic to this optimization procedure. The number of operations needed to perform a shift is always less than (or equal to) the number of operations needed to find the shift (within a constant factor). Therefore, we can neglect it, and other auxiliary operations such as tree traversal can be neglected too.

6.2. Computations of $\mu[r, *]$. The computation of $\mu[r, *]$ of a quasioptimum tree in Algorithm 5.3 can be considered to consist of mainly the following two parts. Let $[0, n]: 0, 1, \cdots, n = r$ be the admissible path of the code placed at r.

1. Computation of $\nabla[y, *]$ for all $1 \leq y \leq n-1$.
2. Computation of $\Delta[x, y]$ for all $1 \leq x \leq y \leq n-1$, and value comparisons (C-tests) for updating μ.

The numbers of operations for these computations are considered separately. (We excluded the trivial case of $x = 0$ or $y = 0$ from consideration.)

From (6.1) the number of operations for the computation of ∇ over $T[r]$ is bounded by:

$$\sum_{y=1}^{n-1} \Bigl(\sum_{v \in T[y]} (\ell_v - \ell_y) \Bigr). \tag{6.3}$$

Let us denote the height and the number of nodes of $T[y]$ by H_y and N_y respectively. Then (6.3) is bounded by:

$$\sum_{y=1}^{n-1} (N_y + 1)(H_y + 1) \leqq (n-1)(N_n + 1)(H_n + 1). \tag{6.4}$$

According to the two remarks on Algorithm 5.3, we can immediately determine $\Delta[x, y]$ for any $1 \leq x \leq y \leq n-1$, if we previously determine all the backtrackable nodes. The backtrackable nodes are determined by scanning the admissible path $[0, n]$ with n C-tests. Thus the computation for all $\Delta[x, y]$ is bounded by n C-tests. Necessary C-tests for updating μ are bounded by $\sum_{x=1}^{n-1} (n-x) = n(n-1)/2$. Thus the total number of C-tests is bounded by:

$$n(n+1)/2. \tag{6.5}$$

As for storage we need to memorize $\nabla[y, *]$ for each y $(1 \leq y \leq n-1)$. However, linked list structure can save the memory, because $\nabla[y_1, *]$ is strictly a subsequence of $\nabla[y, *]$ if $y_1 \in \nabla[y, *]$. Hence the total storage for all $\nabla[y, *]$ is bounded by N_n (the storage for the links is neglected). The storage locations used for the current best $\Delta[x, y]$ is bounded by n. Hence the total storage locations needed to compute $\mu[r, *]$ are bounded by:

$$N_n + n. \tag{6.6}$$

6.3. An upper bound of the computation of Algorithm 4.1. To transform a quasioptimum subtree into an optimum subtree, $\mu[r(a), *]$ and $\Delta[*, r(\lambda)]$ should be performed successively. From (6.2) the number of operations to find $\Delta[*, r(\lambda)]$ is bounded by:

$$(6.7) \qquad \sum_{t\in T[r]} (\ell_t - \ell_r) \leqq H_n N_n.$$

Thus from (6.4), (6.5) and (6.7) the number of operations needed to optimize a given quasioptimum subtree $T[r(a)]$ is bounded by:

$$(6.8) \qquad (n-1)(N_n+1)(H_n+1)\alpha + (n(n+1)/2 + H_n N_n)\beta,$$

where α and β denote the number of primitive operations (say, in terms of memory cycles) needed for an admissibility test (A-test) and a weight comparison (C-test) respectively. As for storage we must add the storage locations for $\Delta[*, r(\lambda)]$ to (6.6). Hence we have an upper bound of:

$$(6.9) \qquad N_n + H_n + n.$$

Suppose a given tree T has N codes and its height is M. We perform the optimization of T according to Algorithm 4.1. The shapes of subtrees change in general by the optimization. Since the height of a subtree increases by at most one by the subtree optimization, the height of a subtree does not exceed $2M$ at any moment of the optimization. The numbers of leaves and external nodes are bounded by N and $N+1$ respectively. Thus we have $n \leqq 2M$, $N_n \leqq N$ and $H_n < N$ for all n. Hence by (6.8) the computation of a subtree optimization is bounded by about $2MN^2\alpha + (2M^2 + N^2)\beta$. Thus summing this over N subtrees we have an upper bound $2MN^3\alpha + (2M^2N + N^3)\beta$ to optimize a tree. Since α and β are both $O(1)$ in practical situations, and further, from $\log N \leqq M \leqq L$ and M is at most $O(N)$, this upper bound can be expressed as $O(N^3L)$. From (6.9) the storage is bounded by $O(N)$.

In case that T is a prefix tree with the height M, we have only to compute and perform $\Delta[*, r]$ for each $r \in T$, and the number of C-tests is bounded by:

$$\sum_{r\in T} \sum_{r\in T[r]} (\ell_t - \ell_r) \leqq \sum_{r\in T} NM \leqq N^2M.$$

Since we have $M \leqq L$, the upper bound becomes $O(N^2L)$. The storage is $O(N)$. In particular, when T is a full prefix tree with $N = 2^L$, the order becomes $O(N^2 \log N)$.

7. Conclusions. We have described properties of sequence trees. Necessary and sufficient conditions of optimum sequence trees are given. The following algorithms are shown: 1. construction of an optimum tree from a given code set, 2. optimization of a given tree, 3. optimum insertion and 4. optimum deletion. The number of operations needed to optimize a tree using the bottom-up optimization algorithm is shown to be bounded by $O(N^3L)$, where N and L are the number of keys and the length of coded keys.

Another optimization problem, very similar to optimum sequence trees, is one about optimum search trees. Much research has been devoted to optimum search trees (see [2], [4] and [5]); however considerations in these papers are limited to the construction of optimum trees. The transformations of search trees are considered in [6] only with the case of uniform weights. Investigations of optimum search trees from the standpoint of optimization algorithms seem unsatisfactory at the present time.

We have discussed binary trees; however it is obvious that every notion such as admissibility, p-sequence and regularity can be extended to general (M-ary) trees without essential changes. Thus the results of this paper are valid also for M-ary trees.

As for unsuccessful searches [5], the "unsuccessful" notion of sequence trees does not correspond to that of search trees; the weights associated with external nodes do not mean the betweenness frequencies in the sense of search trees. However our results can be extended to unsuccessful searches if we slightly modify the formulation. In a sequence tree we may consider that unsuccessful searches are guided by "unsuccessful" codes. The empty nodes at the bottom of a prefix tree correspond to these codes. We call them *unsuccessful codes* (in short u-*codes*, while the members of a given code set C are called s-*codes*). We assume that every code (including u-code) has its weight. Since an internal node with a u-code has no significance in searching, we consider trees where with every external node a set of admissible u-codes is associated and every internal node has an s-code (we call these trees *extended packed trees*). Thus the unsuccessful searches are taken into consideration, the cost of a tree T can be defined by:

$$|T| = \sum_{p:internal} \mathrm{wt}(p)(\mathrm{lev}(p)+1) + \sum_{v:external} \mathrm{wt}(v)(\mathrm{lev}(v)),$$

where the weight of an external node v is defined by the sum of all weights of admissible u-codes of v. A shift operation $\pi : p_0, p_1, \cdots, p_n$ should be restricted so that either it originates from a leaf or it terminates at an external node and it should be modified. The s-codes at p_i for $i = 1, \cdots, n-1$ are moved as before, and furthermore the following two operations are performed as shown in Fig. 7.1:

1. *join*: The set of u-codes associated with p_0 in πT is the join of the two sets of u-code associated with v_1 and v_2 in T.

2. *fork*: The set of u-codes associated with p_n in T is divided into two sets accordingly in πT.

Now for the transformations of extended packed trees, it suffices to consider the set of cycles and the shifts described above.

The effect of optimization for a code set can be measured by the "compression" index κ defined by the ratio of the cost of an optimum tree to the average cost of random trees. According to our preliminary simulation with $L = 10$ and $N = 200$, the results are: with the uniform weights we have $\kappa \doteqdot 0.90$, while with the "80–20" distribution rule [5] we have $\kappa \doteqdot 0.50$.

Some related open problems are as follows:

1. Changing the definition of the cost to the weighted leaf path length defined by

$$|T| = \sum_t \mathrm{wt}(t)(\mathrm{lev}(t)+1),$$

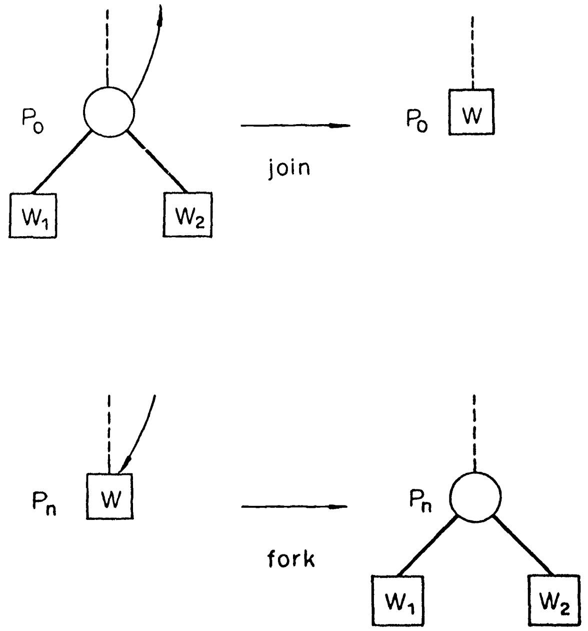


FIG. 7.1. *Fork and join* $W = W_1 + W_2$

where t is a leaf of a packed sequence tree T, which yields another optimization problem;

2. Theoretical and experimental evaluations of optimization procedures and effectiveness of the optimization;

3. Finding another optimization scheme, not the bottom-up one described here, having less operations;

4. Construction of nearly optimum trees with fewer operations.

Acknowledgment. The authors are grateful to the referees for a truly extraordinary amount of help which has led to improvements of the manuscript. The authors thank for the valuable discussions Dr. Koji Torii and Mr. Akito Sakurai. They also thank Drs. Hiroji Nishino, Kazuo Kurokawa, Osamu Ishii and Ryota Suekane of the Electrotechnical Laboratory for providing an opportunity to do this research.

REFERENCES

[1] E. G. COFFMAN, JR. AND J. EVE, *File structure using hashing functions*, Comm. ACM, 13 (1970), pp. 427–432, 436.

[2] T. C. HU AND A. C. TUCKER, *Optimal computer search trees and variable-length alphabetical codes*, SIAM J. Appl. Math., 21 (1971), pp. 514–532.

[3] D. E. KNUTH, *The Art of Computer Programming*, vol. 1, Addison-Wesley, Reading, MA, 1968.

[4] ———, *Optimum binary search trees*, Acta Informat., 1 (1971), pp. 14–25.

[5] ———, *The Art of Computer Programming*, vol. 3, Addison-Wesley, Reading, MA, 1973.

[6] W. A. MARTIN AND D. N. NESS, *Optimizing binary trees grown with a sorting algorithm*, Comm. ACM, 15 (1972), pp. 88–93.

[7] M. Miyakawa, T. Yuba and M. Hoshi, *Construction of optimum sequence trees with weighted codes*, Res. Rep. Inst. Electronics and Communication Engrs. of Japan, EC72-35 (1972).

[8] Y. Sugito, T. Yuba and K. Torii, *Optimization of sequence hash trees using prefix hash trees*, Proc. Joint Convention Rec. of Four Electrical and Electronics Institutes of Japan, No. 1063, 1971, Inst. Electronics and Communication Engrs. of Japan, Tokyo, 1971.

[9] T. Yuba and M. Miyakawa, *Optimum sequence hash trees*, Trans. Inst. Electronics and Communication Engrs. of Japan, 56-D (1973), pp. 9–16; Systems-Computers-Controls, 4 (1973), pp. 10–17.

[10] T. Yuba, M. Miyakawa and M. Hoshi, *Optimization problems of sequence trees*, Trans. Inst. Electronics and Communication Engrs. of Japan, 57-D (1974), pp. 238–239.

SIAM J. COMPUT.
Vol. 6, No. 2, June 1977

A BEST POSSIBLE BOUND FOR THE WEIGHTED PATH LENGTH OF BINARY SEARCH TREES*

KURT MEHLHORN†

Abstract. The weighted path length of optimum binary search trees is bounded above by $\sum \beta_i + 2\sum \alpha_j + H$ where H is the entropy of the frequency distribution, $\sum \beta_i$ is the total weight of the internal nodes, and $\sum \alpha_j$ is the total weight of the leaves. This bound is best possible. A linear time algorithm for constructing nearly optimal trees is described.

Key words. binary search tree, complexity, average search time, entropy

One of the popular methods for retrieving information by its "name" is to store the names in a binary tree. We are given n names $B_1, B_2, \cdots, B_n$ and $2n+1$ frequencies $\beta_1, \cdots, \beta_n, \alpha_0, \cdots, \alpha_n$ with $\sum \beta_i + \sum \alpha_j = 1$. Here β_i is the frequency of encountering name B_i, and α_j is the frequency of encountering a name which lies between B_j and B_{j+1}, α_0 and α_n have obvious interpretations [4].

A binary search tree T for the names $B_1, B_2, \cdots, B_n$ is a tree with n interior nodes (nodes having two sons), which we denote by circles, and $n+1$ leaves, which we denote by squares. The interior nodes are labeled with the B_i in increasing order from left to right and the leaves are labeled with the intervals (B_j, B_{j+1}) in increasing order from left to right. Let b_i be the distance of interior node B_i from the root and let a_j be the distance of leaf (B_j, B_{j+1}) from the root. To retrieve a name X, b_i+1 comparisons are needed if $X = B_i$ and a_j comparisons are required if $B_j < X < B_{j+1}$. Therefore we define the weighted path length of tree T as:

$$P = \sum_{i=1}^{n} \beta_i(b_i+1) + \sum_{j=0}^{n} \alpha_j a_j.$$

It is equal to the expected number of comparisons needed to retrieve a name.

In [4] D. E. Knuth gives an algorithm for constructing an optimum binary search tree, i.e., a tree with minimal weighted path length. His algorithm operates in $O(n^2)$ units of time and $O(n^2)$ units of space. In [6] we discuss the following "rule of thumb" for constructing nearly optimal binary search trees: choose the root so as to equalize the total weight of the left and right subtree as much as possible, then proceed recursively. The weighted path length of a tree constructed according to this rule is bounded above by $2 + 1.44 \cdot H$, where $H = \sum \beta_i \log(1/\beta_i) + \sum \alpha_j \log(1/\alpha_j)$ is the entropy of the frequency distribution. This bound was recently improved by P. J. Bayer [1] to $2+H$. Here we discuss a different rule of thumb suggested by [3] and prove the upper bound $1 + \sum \alpha_j + H$ for the weighted path length. This bound is best possible.

The rule presented here as well as the rules described in [6] can be implemented to work in linear time and space ([2]).

* Received by the editors September 24, 1975, and in revised form June 1, 1976.

† Universität des Saarlandes, 66 Saarbrücken, West Germany

We describe and analyze an approximation algorithm. The algorithm constructs binary search trees in a top-down fashion. It uses bisection on the set

$$\left\{s_i;\, s_i=\sum_{p=0}^{i-1}(\alpha_p+\beta_p)+\beta_i+\frac{\alpha_i}{2} \quad\text{and}\quad 0\leqq i\leqq n\right\},$$

i.e., the root $\textcircled{k}$ is determined such that $s_{k-1}\leqq\frac{1}{2}$ and $s_k\geqq\frac{1}{2}$. It then proceeds recursively on the subsets $\{s_i;\, i\leqq k-1\}$ and $\{s_i;\, i\geqq k\}$. In the definition of the s_i's we assumed $\beta_0=0$ for ease of writing. The main program

```
          begin
               let $s_i \leftarrow \sum_{p=0}^{i-1}(\alpha_p+\beta_p)+\beta_i+\alpha_i/2$ for $0\leqq i\leqq n$;
               construct-tree $(0, n, 0, 1)$

          end
```

uses the recursive procedure construct-tree:

```
procedure construct-tree $(i, j, cut, l)$;
comment we assume that the actual parameters of any call of construct-tree satisfy
the following conditions.
(1) $i$ and $j$ are integers with $0\leqq i<j\leqq n$,
(2) $l$ is an integer with $l\geqq 1$,
(3) $cut=\sum_{p=1}^{l-1} x_p 2^{-P}$ with $x_p\in\{0, 1\}$ for all $p$,
(4) $cut\leqq s_i\leqq s_j\leqq cut+2^{-l+1}$.
A call construct-tree $(i, j, —, —,)$ will construct a binary search tree for the nodes
$\textcircled{i+1}, \cdots, \textcircled{j}$ and the leaves $\boxed{i}, \cdots, \boxed{j}$;
begin
if $i+1=j$ (Case A)
then return the tree shown in Fig. 1.
else comment we determine the root so as to bisect the interval
     $(cut, cut+2^{-l+1})$
     begin
     determine $k$ such that
     (5) $i<k\leqq j$
     (6) $k=i+1$ or $s_{k-1}\leqq cut+2^{-l}$
     (7) $k=j$ or $s_k\geqq cut+2^{-l}$
     comment $k$ exists because the actual parameters are supposed to satisfy
     condition (4);
     if $k=i+1$ (Case B)
     then return the tree shown in Fig. 2;
     if $k=j$ (Case C)
     then return the tree shown in Fig. 3;
     if $i+1<k<j$ (Case D)
     then return the tree shown in Fig. 4;
     end
end procedure construct-tree;
```

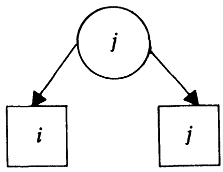


FIG. 1

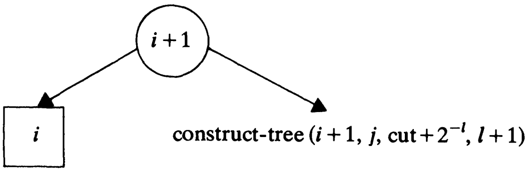


FIG. 2

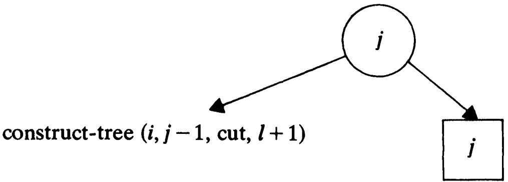


FIG. 3

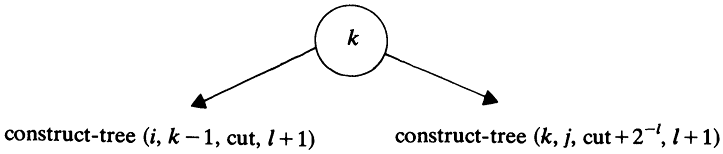

k
construct-tree (i, k−1, cut, l+1)
construct-tree (k, j, cut+2^{-l}, l+1)

FIG. 4

LEMMA. *The approximation algorithm constructs a binary search tree whose weighted path length P_{approx} is bounded above by* $1+\sum \alpha_j + H$.

Proof. We state several simple facts.

FACT 1. *If the actual parameters of a call* construct-tree (i, j, cut, l) *satisfy conditions* (1) *to* (4) *and* $i+1 \neq j$, *then a* k *satisfying conditions* (5) *to* (7) *exists and the actual parameters of the recursive calls of* construct-tree *initiated by this call again satisfy conditions* (1) *to* (4).

Proof. Assume that the parameters satisfy conditions (1) to (4) and that $i+1 \neq j$. In particular, $cut \leqq s_j \leqq cut + 2^{-l+1}$. Suppose, that there is no k, $i < k \leqq j$, with $s_{k-1} \leqq cut + 2^{-l}$ and $s_k \geqq cut + 2^{-l}$. Then either for all k, $i < k \leqq j$, $s_k < cut + 2^{-l}$ or for all k, $i < k \leqq j$, $s_k > cut + 2^{-l}$. In the first case $k = j$ satisfies (6) and (7), in the second case $k=i+1$ satisfies (6) and (7). This shows that k always exists. It remains to show that the parameters of the recursive calls satisfy again (1) and (4). This follows immediately from the fact that k satisfies (5) to (7) and that $i+1 \neq j$ and hence $s_k \geqq cut + 2^{-l}$ in Case B and $s_{k-1} \leqq cut + 2^{-l}$ in Case C. Q.E.D.

FACT 2. *The actual parameters of every call of* construct-tree *satisfy conditions* (1) *to* (4) (*if the arguments of the top-level call do*).

Proof. The proof is by induction, Fact 1 and the observation that the actual parameters of the top-level call construct-tree $(0, n, 0, 1)$ satisfy (1) to (4). Q.E.D.

We say that node $\textcircled{h}$ (leaf $\boxed{h}$ resp.) is constructed by the call construct-tree (i, j, cut, l) if $h=j$ ($h=i$ or $h=j$) and Case A is taken or if $h=i+1$ ($h=i$) and Case B is taken or if $h=j$ ($h=j$) and Case C is taken or if $h=k$ and Case D is taken. Let b_i be the depth of node $\textcircled{i}$ and let a_j be the depth of leaf $\boxed{j}$ in the tree returned by the call construct-tree $(0, n, 0, 1)$.

FACT 3. *If node* $\textcircled{h}$ (*leaf* $\boxed{h}$ *is constructed by the call* construct-tree (i, j, cut, l), *then* $b_h+1=l$ ($a_h=l$).

Proof. The proof is by induction on l.

FACT 4. *If node* $\textcircled{h}$ (*leaf* $\boxed{h}$) *is constructed by the call* construct-tree (i, j, cut, l), *then* $\beta_h \leqq 2^{-l+1}$ ($\alpha_h \leqq 2^{-l+2}$).

Proof. The actual parameters of the call satisfy condition (4) by Fact 2. Thus

$$2^{-l+1} \geqq s_j - s_i = (\alpha_i + \alpha_j)/2 + \beta_{i+1} + \alpha_{i+1} + \cdots + \beta_j$$
$$\geqq \beta_h \text{ (resp. } \alpha_h/s). \qquad \text{Q.E.D.}$$

FACT 5. *The weighted path length P_{approx} of the tree constructed by the approximation algorithm is bounded above by $\sum \beta_j + 2\sum \alpha_j + H$.*

Proof.

$$\begin{aligned} P_{\text{approx}} &= \sum \beta_i(b_i+1) + \sum \alpha_j a_j \\ &\leqq \sum \beta_i(\log(1/\beta_i)+1) + \sum \alpha_j(\log(1/\alpha_j)+2) \\ &\leqq \sum \beta_j + 2\cdot\sum \alpha_j + H. \qquad \text{Q.E.D.} \end{aligned}$$

THEOREM. *Let $\alpha_0, \beta_1, \alpha_1, \cdots, \beta_n, \alpha_n$ be any frequency distribution, let P_{opt} be the weighted path length of the optimum binary search tree for this distribution, let P_{approx} be the weighted path length of the tree constructed by the approximation algorithm, and let $H = -\sum \beta_i \log \beta_i - \sum \alpha_j \log \alpha_j$ be the entropy of the frequency distribution. Then*

$$P_{\text{opt}} \leqq P_{\text{approx}} \leqq \sum \beta_j + 2\cdot\sum \alpha_j + H.$$

Furthermore, this upper bound is the best possible in the following sense: if $c_1 \sum \beta_i + c_2 \sum \alpha_j + c_3 \cdot H$ is an upper bound for P_{opt}, then $c_1 \geqq 1$, $c_2 \geqq 2$, and $c_3 \geqq 1$.

Proof. The first part of the theorem follows from the preceding lemma. The second part is proven by exhibiting suitable frequency distributions:

$c_1 \geqq 1$: Take $n=1$, $\alpha_0=\alpha_1=0$ and $\beta_1=1$.
$c_2 \geqq 2$: Take $n=2$, $\alpha_0=\alpha_2=\beta_1=\beta_2=0$, $\alpha_1=1$.
$c_3 \geqq 1$: Take $n=2^k-1$, $\beta_1=0$ for all i and $\alpha_j=2^{-k}$ for all j.

It is easy to see that the complete binary tree is the optimal binary search tree for this distribution. Thus

$$H = \log(n+1) = k = \sum_{\text{leaves}} (1/2^k) \cdot k = P_{\text{opt}}. \qquad \text{Q.E.D.}$$

REFERENCES

[1] P. J. BAYER, *Improved bounds on the cost of optimal and balanced binary search trees*, M.Sc. thesis, Mass. Inst. of Tech., Cambridge, MA, 1975.

[2] M. L. FREDMAN, *Two applications of a probabilistic search technique: Sorting $X + Y$ and building balanced search trees*, 7th Symp. on Theory of Computing, Albuquerque, NM., 1975.

[3] E. N. GILBERT AND E. F. MOORE, *Variable-length binary encodings*, Bell System Tech. J., 38 (1959), pp. 933–968.

[4] D. E. KNUTH, *Optimum binary search trees*, Acta Informatica, 1 (1971), pp. 14–25.

[5] ———, *The Art of Computer Programming*, vol. 3, Addison-Wesley, Reading, MA., 1973.

[6] K. MEHLHORN, *Nearly optimal binary search trees*, Acta Informatica, 5 (1975), pp. 287–295.

SIAM J. COMPUT.
Vol. 6, No. 2, June 1977

QUICKSORT WITH EQUAL KEYS*

ROBERT SEDGEWICK†

Abstract. This paper considers the problem of implementing and analyzing a Quicksort program when equal keys are likely to be present in the file to be sorted. Upper and lower bounds are derived on the average number of comparisons needed by any Quicksort program when equal keys are present. It is shown that, of the three strategies which have been suggested for dealing with equal keys, the method of always stopping the scanning pointers on keys equal to the partitioning element performs best.

Key words. analysis of algorithms, equal keys, Quicksort, sorting

Introduction. The Quicksort algorithm, which was introduced by C. A. R. Hoare in 1960 [6], [7], has gained wide acceptance as the most efficient general-purpose sorting method suitable for use on computers. The algorithm has a rich history: many modifications have been suggested to improve its performance, and exact formulas have been derived describing the time and space requirements of the most important variants [7], [9], [14].

Although most files to be sorted contain at least some equal keys and sorting programs must always deal with them properly, it is generally considered reasonable to assume in the analysis that the keys are distinct. This assumption is fundamental to the analysis of nearly all sorting programs, and it is very often realistic. In any situation where the number of possible key values far exceeds the number of keys to be sorted, the probability that equal keys are present will be very small. However, if the number of possible key values is not large, or if there is some other information about the file which indicates that equal keys are likely to be present, then the performance of many sorting programs, including Quicksort, has not been carefully studied.

The purpose of this paper, then, is to investigate the performance of Quicksort when equal keys are present. The following section describes the algorithm and its analysis for distinct keys. Next, lower and upper bounds are derived for the average number of comparisons taken when equal keys are present. Following that, we shall consider, from a practical standpoint, the problem of implementing a version of Quicksort to handle equal keys. Finally we shall compare the various methods and discover which is the most useful in practical sorting applications.

1. Distinct keys.

Suppose that an array of keys $A[1], \cdots, A[N]$ is to be rearranged to make

$$A[1]<A[2]<\cdots<A[N],$$

where the order relation $<$ is any transitive relation whatever defined on all the keys.

* Received by the editors September 2, 1975, and in revised form May 3, 1976.

† Division of Applied Mathematics, Brown University, Providence, Rhode Island 02912. This work was supported in part by the National Science Foundation under Grants GJ-28074 and MCS75-23738, and in part by the Fannie and John Hertz Foundation.

Quicksort is a "divide and conquer" approach to this problem. For some key with value v, the file is rearranged so that $A[j]=v$ for some j, $1 \leqq j \leqq N$, all of the keys to the left of $A[j]$ are $<v$ and all of the keys to the right of $A[j]$ are $>v$. This process is called *partitioning*, and it turns out that it can be performed efficiently. After partitioning, the key $A[j]$ is in its final position in the sorted file and need not be considered further. If the same procedure is applied recursively to the subfiles $A[1], \cdots, A[j-1]$ and $A[j+1], \cdots, A[N]$, then the whole file becomes sorted. The following program is an implementation of the method, and the partitioning process is spelled out explicitly.

PROGRAM 1.

```
procedure quicksort (integer value $l, r$);
    comment The array $A$ is declared to be $A[1:N+1]$; with $A[N+1]=\infty$;
    if $r>l$ then
        $i := l$; $j := r+1$; $v := A[l]$
        loop:
            loop: $i := i+1$; while $A[i]<v$ repeat;
            loop: $j := j-1$; while $A[j]>v$ repeat;
        until $j<i$:
            $A[i] :=: A[j]$;
        repeat;
        $A[l] :=: A[j]$;
        quicksort($l, j-1$);
        quicksort($i, r$);
    endif;
```

(This program uses an exchange operator :=: , and the control constructs **loop** · · · **repeat** and **if** · · · **then** · · · **endif**, which are like those described by D. E. Knuth in [10].) The leftmost element is chosen as the partitioning element, and then the rest of the array is partitioned on that value. This is done by scanning from the left to find an element $>v$, scanning from the right to find an element $<v$, exchanging them, and continuing the process until the scanning pointers cross. The loop always terminates with $j+1=i$, and it is known at that point that $A[l+1], \cdots, A[j]$ are $<v$ and $A[j+1], \cdots, A[r]$ are $>v$, so that the exchange $A[l] :=: A[j]$ completes the job of partitioning $A[l], \cdots, A[r]$. The procedure call "quicksort(1, N)" will therefore sort $A[1], \cdots, A[N]$. Figure 1 shows the operation of the program on the first 9 distinct digits of π.

```
3 1 4 5 9 2 6 8 7
2 1 ③ 5 9 4 6 8 7
1 ②
      4 ⑤ 9 6 8 7
          7 6 8 ⑨
          6 ⑦ 8
1 2 3 4 5 6 7 8 9
```

FIG. 1

A number of different partitioning methods have been suggested for the implementation of Quicksort, and the particular method described above is motivated fully in [14]. There are, however, some facets of the implementation which should be noted here.

The loops which implement the pointer scans are the "inner loops" of the program—most of the execution time is spent there. (This fact comes from the analysis, which is discussed more fully below.) Some efficiency is achieved in the inner loops by introducing two redundant comparisons to avoid the necessity for checking if the pointers have crossed each time a pointer is changed. The last comparison in each of the loops is redundant: the last $i := i+1$ makes $i = j$ and it is known that $A[j] > v$ at that point (provided that $A[N+1]$ is greater than all the other keys—this is the meaning of the notation $A[N+1] = \infty$); the last $j := j-1$ makes $j = i-1$ and it is known that $A[i-1] \leqq v$ at that point. Program 1 uses $N+1$ comparisons on the first partitioning stage when only $N-1$ are absolutely necessary, but its inner loop is much more efficient as a result.

Although the above program gives a very efficient implementation of partitioning, there are a number of ways that the program as a whole can be made more efficient. It turns out that efficiency can be gained by choosing the partitioning element based on a small sample from the file; by removing the recursion and always sorting the smaller of the two subfiles first; and especially by handling small subfiles differently. All of these improvements are documented in [9] and [14], and they apply uniformly to all of the programs that we will consider.

We can derive exact formulas for the total average running time of Quicksort by solving recurrence relations which describe the average number of times the various statements in the program are executed. In this paper, we shall be concerned chiefly with the average number of comparisons: the average number of times the tests "$A[i] < v$" and "$A[j] > v$" are performed during the execution of Program 1 on a randomly ordered input file. If we denote this quantity by C_N, we find that it is described by the recurrence

$$\begin{aligned} C_N &= N+1+\frac{1}{N}\sum_{1\leqq k\leqq N}(C_{k-1}+C_{N-k}), \qquad N\geqq 2 \\ &= N+1+\frac{2}{N}\sum_{1\leqq k\leqq N}C_{k-1} \end{aligned}$$

with $C_0 = C_1 = 0$. The $N+1$ term represents the number of comparisons used on the first partitioning stage, and the other term represents the average number of comparisons used for the subfiles. By writing this recurrence, we have made the important assumption that the partitioning process preserves randomness in the subfiles: if the original file is a random permutation of its elements, then the left and right subfiles will also have this property. It is easy to prove that the partitioning method in Program 1 preserves randomness, but there are partitioning methods which do not (see [10], [14]).

To solve the recurrence, we first multiply by N and then eliminate the summation by differencing (subtracting the same equation for $N-1$). After rearranging terms, we get

$$NC_N = (N+1)C_{N-1}+2N, \qquad N\geqq 3,$$

which, after we divide by $N(N+1)$, telescopes to the solution

$$\frac{C_N}{N+1}=\frac{C_2}{3}+\sum_{3\leqq k\leqq N}\frac{2}{k+1}$$

or

$$C_N=2(N+1)(H_{N+1}-\tfrac{4}{3}),\qquad N\geqq 2.$$

This shows that Quicksort achieves the theoretical minimum of $O(N\log N)$ comparisons on the average. In a similar fashion, the average number of times each of the other statements in Program 1 is executed can be calculated exactly. If the time taken to execute each statement is also known, then we can get an exact formula for the total expected running time of Program 1 (see [9] and [14]).

In studying the performance of Quicksort with equal keys, we will deal chiefly with the average number of comparisons, not with the total running time. Analysis shows that the comparisons do dominate (though there is about one exchange for every three comparisons) and the calculations become tedious when dealing with the total running time. We will be comparing the relative performance of a number of very similar variants of Quicksort, and we can be fairly certain that conclusions that we draw based on the number of comparisons will carry through to the total running time. It is nearly always the case that if one version of Quicksort uses a lower number of comparisons than another, then the frequencies of execution of all of the other instructions are also lower.

It is intuitive that Quicksort performs best when the partition happens to be near the middle of the file at each stage, and worst when the partition falls near the ends. (The exact analysis of the best and worst case for practical versions of the program is interesting and complex, but not particularly relevant to the study of Quicksort with equal keys.) For Program 1, it turns out that the worst case for the number of comparisons (and for the whole algorithm) occurs when the file is already in order, and partitioning does nothing but take one key from the left end of the file at each stage. The total number of comparisons in this case is $\sum_{2\leqq k\leqq N}(k+1)=\frac{1}{2}(N+4)(N-1)$. This $O(N^2)$ worst case is often viewed as Quicksort's weakness, but there are many ways to avoid it in practical situations. It will be a concern when we begin to consider equal keys. On the other hand, the best case occurs when the file is split exactly in half at each stage, and the total number of comparisons taken turns out to be less than $N\lg N$ ($\lg\equiv\log_2$). A complete discussion and derivations of exact upper and lower bounds for Quicksort may be found in [14].

In summary, the operation of Quicksort on files of distinct keys is very completely understood. Unfortunately, few of the results carry over to the case when equal keys are present. Before examining the question of actually implementing a program to handle equal keys properly, let us look more carefully into the analysis, so that we may get some idea of how well we might expect to do.

2. Basic assumptions. The first problem that we face in trying to analyze any sorting method with equal keys is the formulation of an appropriate model describing the input file. Suppose that the N keys to be sorted have n distinct values. Since we only use the relative values of the keys in sorting, we may as well assume that the values are $1, 2, \cdots, n$. If we also know that there are x_1 ones, x_2 twos, etc., then we might consider each of the $N!$ permutations of the multiset $\{x_1 \cdot 1, x_2 \cdot 2, \cdots, x_n \cdot n\}$ (where $x_1 + \cdots + x_n = N$) to be equally likely as input files. If this additional information is not available, a second possibility would be to assume that each of the n^N ways of making an input file of length N from n distinct values is equally likely. (Notice that many of the possible input files have less than n distinct values in this model.) While one or the other of these models might be appropriate for some particular sorting applications, neither is entirely satisfactory as a general model. We shall work to some degree with both. When we speak of sorting a *random permutation from a multiset*, we will be referring to the first model; when we refer to a *random n-ary file*, we will be working with the second.

Now, if we wish to use Quicksort to sort a file containing equal keys, we must decide how to treat keys equal to the partitioning element during the partitioning process. Ideally, we would like to get all of them into position in the file, with all the keys with a smaller value to their left, and all the keys with a larger value to their right. Unfortunately, no efficient method for doing so has yet been devised, so we shall have some keys equal to the partitioning element in the left subfile and some in the right. We shall use the term *Quicksort program* to describe any program which sorts by recursively subdividing files of more than one element into three subfiles: a (nonempty) middle subfile whose elements are all equal to some value j; a left subfile with no elements $>j$; and a right subfile with no elements $<j$. The only further restrictions are that the value s must be chosen by examining one element from the file, and that if the input file is randomly ordered, so must be the subfiles. With these assumptions, we can write down a recurrence for the average number of comparisons to sort a random permutation from the multiset $\{x_1 \cdot 1, \cdots, x_n \cdot n\}$ (with $x_1 + \cdots + x_n = N$) for any Quicksort program:

$$C(x_1, \cdots, x_n) = N + 1 + \frac{1}{N!} \sum_{\substack{\text{all permutations} \\ \text{of } \{x_1 \cdot 1, \cdots, x_n \cdot n\}}} (C(x_1, \cdots, x_{j-1}, \alpha) + C(\beta, x_{j+1}, \cdots, x_n)).$$

(The notation $C(\beta, x_{j+1}, \cdots, x_n)$ is defined to mean $C(\beta)$ when $j = n$.) This formula assumes that $N+1$ comparisons are used in the first partitioning stage. The keys equal to the partitioning element are distributed among the subfiles in some way, as described by the parameters α and β, which are functions of the partitioning method and the particular permutation being sorted. (By assuming that at least one element is put in position, we are assuming that $\alpha + \beta < x_j$.) We will use various initial conditions in the derivations below to complete this recurrence.

3. Lower bounds. From this formula we can begin to derive a lower bound on the number of comparisons, for, as we have already noted, the best that we can do with any partitioning method is to get all of the keys equal to the partitioning element into position at each partitioning stage. If the partitioning element is chosen randomly, then each of the $x_j(N-1)!$ permutations for which j is the partitioning element will be divided into a left subfile which is a random permutation of $\{x_1 \cdot 1, \cdots, x_{j-1} \cdot (j-1)\}$ and a right subfile which is a random permutation of $\{x_{j+1}\cdot(j+1),\cdots,x_n\cdot n\}$. This means that a lower bound on the average number of comparisons used is certainly described by the recurrence

$$C(x_1,\cdots,x_n)=N-1+\frac{1}{N}\sum_{1\leqq j\leqq n}x_j(C(x_1,\cdots,x_{j-1})+C(x_{j+1},\cdots,x_n)),$$

for N and $n\geqq 1$, with $C(0)=C(1)=0$. (As remarked above, only $N-1$ comparisons are absolutely necessary for the first stage.) To solve this recurrence, we will try to eliminate the summation by differencing, as we did above. If we multiply both sides by N, and then subtract the same equation for $\{x_2\cdot 1,\cdots,x_n\cdot(n-1)\}$, we get

$$\begin{aligned}\Big(\sum_{1\leqq j\leqq n}x_j\Big)C(x_1,\cdots,x_n)&-\Big(\sum_{2\leqq j\leqq n}x_j\Big)C(x_2,\cdots,x_n)\\&=x_1^2-x_1+2x_1\sum_{2\leqq j\leqq n}x_j+x_1C(x_2,\cdots,x_n)\\&\quad+\sum_{2\leqq j\leqq n}x_j(C(x_1,\cdots,x_{j-1})-C(x_2,\cdots,x_{j-1}))\quad\text{for } n\geqq 1.\end{aligned}$$

After rearranging terms and defining $G(x_1,\cdots,x_n)=C(x_1,\cdots,x_n)-C(x_2,\cdots,x_n)$, this equation becomes

$$\begin{aligned}\Big(\sum_{1\leqq j\leqq n}x_j\Big)G(x_1,\cdots,x_n)&=x_1^2-x_1+2x_1\sum_{2\leqq j\leqq n}x_j\\&\quad+\sum_{2\leqq j\leqq n}x_jG(x_1,\cdots,x_{j-1})\quad\text{for } n\geqq 1.\end{aligned}$$

Now we difference again, except this time we subtract the same equation for $\{x_1\cdot 1,x_2\cdot 2,\cdots,x_{n-1}\cdot(n-1)\}$ to yield

$$\begin{aligned}\Big(\sum_{1\leqq j\leqq n}x_j\Big)G(x_1,\cdots,x_n)&-\Big(\sum_{1\leqq j\leqq n-1}x_j\Big)G(x_1,\cdots,x_{n-1})\\&=2x_1x_n+x_nG(x_1,\cdots,x_{n-1})\quad\text{for } n\geqq 2,\end{aligned}$$

or

$$G(x_1,\cdots,x_n)=G(x_1,\cdots,x_{n-1})+\frac{2x_1x_n}{x_1+\cdots+x_n}.$$

This equation telescopes to

$$G(x_1,\cdots,x_n)=G(x_1)+\sum_{2\leqq j\leqq n}\frac{2x_1x_j}{x_1+\cdots+x_j}.$$

(This formula assumes that $x_1x_j/(x_1+\cdots+x_j)=0$ if $x_1=x_j=0$, even if $x_2,\cdots,x_{j-1}$ are also 0. We shall adopt this convention throughout this paper.)

After substituting for G, we get another telescoping recurrence,

$$C(x_1,\cdots,x_n)=C(x_2,\cdots,x_n)+C(x_1)+2\sum_{2\leqq j\leqq n}\frac{x_1x_j}{x_1+\cdots+x_j},$$

which leads to the result

$$C(x_1,\cdots,x_n)=\sum_{1\leqq j\leqq n} C(x_j)+2\sum_{1\leqq k<j\leqq n}\frac{x_kx_j}{x_k+\cdots+x_j}.$$

This derivation was suggested by the analysis given by Burge [3] for a similar problem which we will discuss below. The formula is surprisingly simple, and it can tell us exactly how well we can expect to do in a variety of situations. For example, if $x_j=x$ for $1\leqq j\leqq n$, then we have

$$\begin{aligned}C(x,\cdots,x)&=N-n+2\sum_{1\leqq k<j\leqq n}\frac{x}{j-k+1}\\&=N-n+2\frac{N}{n}\sum_{1<j\leqq n}\sum_{1<k\leqq j}\frac{1}{k}\\&=N-n+2\frac{N}{n}\sum_{1<k\leqq n}\frac{n-k+1}{k}\\&=2(1+1/n)NH_n-3N-n.\end{aligned}$$

If we take $x=1$ (and therefore $n=N$), then we have analyzed Program 1 with distinct keys, and this result differs from the answer in the previous section only because we used the lower bound of $N-1$ comparisons for the first partitioning stage.

We can proceed further, and use the general result for a random permutation of a multiset to derive a lower bound for a random n-ary file. If C_{Nn} is defined to be the average number of comparisons taken by a Quicksort program on random n-ary files of length N, then we have

$$C_{Nn}=\frac{1}{n^N}\sum_{x_1+\cdots+x_n=N}\binom{N}{x_1,\cdots,x_n}C(x_1,\cdots,x_n).$$

This is true because the probability that a given input is a permutation of a particular multiset $\{x_1\cdot 1,\cdots,x_n\cdot n\}$ is

$$\frac{1}{n^N}\binom{N}{x_1,\cdots,x_n}.$$

Therefore, our lower bound is given by

$$\frac{1}{n^N}\sum_{x_1+\cdots+x_n=N}\binom{N}{x_1,\cdots,x_n}\Bigl(\sum_{1\leqq k\leqq n}C(x_k)+2\sum_{1\leqq k<j\leqq n}\frac{x_kx_j}{x_k+\cdots+x_j}\Bigr).$$

The first term is easy to evaluate, since $C(x_k)=x_k-1$ for $x_k>0$ and $C(0)=0$. We have

$$\begin{aligned}\frac{1}{n^N}\sum_{x_1+\cdots+x_n=N}\binom{N}{x_1,\cdots,x_n}\sum_{1\leqq k\leqq n}C(x_k)\\
&=\frac{1}{n^N}\sum_{x_1+\cdots+x_n=N}\binom{N}{x_1,\cdots,x_n}\sum_{1\leqq k\leqq n}(x_k-1+\delta_{x_k0})\\
&=N-n+\frac{1}{n^N}\sum_{1\leqq k\leqq n}\sum_{\substack{x_1+\cdots+x_n=N\\x_k=0}}\binom{N}{x_1,\cdots,x_n}\\
&=N-n+(n-1)^N/n^{N-1}\\
&=N-n+n(1-1/n)^N.\end{aligned}$$

The second term is more difficult, but it can also be simplified through the use of the multinomial theorem. After interchanging the order of summation, we have

$$\frac{2}{n^N}\sum_{1\leqq k<j\leqq n}\sum_{x_1+\cdots+x_n=N}\binom{N}{x_1,\cdots,x_n}\frac{x_kx_j}{x_k+\cdots+x_j}.$$

The first step is to split the sum and the multinomial coefficient in two parts:

$$\frac{2}{n^N}\sum_{1\leqq k<j\leqq n}\sum_{x_1+\cdots+x_{k-1}+i+x_{j+1}+\cdots+x_n=N}\binom{N}{x_1,\cdots,x_{k-1},i,x_{j+1},\cdots,x_n}$$
$$\sum_{\substack{x_k+\cdots+x_j=i\\x_k,x_j\geqq 1}}\binom{i}{x_k,\cdots,x_j}\frac{x_kx_j}{i}.$$

(Here we have also taken note of the.fact that all of the terms with x_k or $x_j=0$ vanish.) Now,

$$\binom{i}{x_k,\cdots,x_j}\frac{x_kx_j}{i}=(i-1)\binom{i-2}{x_k-1,x_{k+1},\cdots,x_{j-1},x_j-1},$$

so we can apply the multinomial theorem to the innermost sum, which leaves us with

$$\frac{2}{n^N}\sum_{1\leqq k<j\leqq n}\sum_{\substack{x_1+\cdots+x_{k-1}+i+x_{j+1}+\cdots+x_n=N\\i\geqq 2}}(i-1)\binom{N}{x_1,\cdots,x_{k-1},i,x_{j+1},\cdots,x_n}$$
$$\cdot(j-k+1)^{i-2}.$$

The inner sum now reduces to three terms, one for the case $i=0$ and two more resulting from splitting the first factor, all of which can be evaluated with the multinomial theorem. We have

$$\sum_{\substack{x_1+\cdots+x_{k-1}+i+x_{j+1}+\cdots+x_n=N\\ i\geqq 1}} i\binom{N}{x_1, \cdots, x_{k-1}, i, x_{j+1}, \cdots, x_n}(j-k+1)^{i-2}$$

$$=N\sum\binom{N-1}{x_1, \cdots, i-1, x_{j+1}, \cdots, x_n}(j-k+1)^{i-2}$$

$$=\frac{N}{j-k+1}n^{N-1}$$

and

$$\sum_{x_1+\cdots+x_{k-1}+i+x_{j+1}+\cdots+x_n=N}\binom{N}{x_1, \cdots, x_{k-1}, i, x_{j+1}, \cdots, x_n}(j-k+1)^{i-2}$$

$$=\frac{1}{(j-k+1)^2}n^N$$

and finally

$$\frac{1}{(j-k+1)^2}\sum_{x_1+\cdots+x_{k-1}+x_{j+1}+\cdots+x_n=N}\binom{N}{x_1, \cdots, x_{k-1}, x_{j+1}, \cdots, x_n}$$

$$=\frac{1}{(j-k+1)^2}(n-j+k-1)^N.$$

Substituting all of these into our expression for the lower bound, we have simplified it to

$$N-n+n\left(1-\frac{1}{n}\right)^N+2\sum_{1\leqq k<j\leqq n}\left(\frac{N}{n}\frac{1}{j-k+1}-\frac{1}{(j-k+1)^2}\right.$$
$$\left.+\frac{1}{(j-k+1)^2}\left(1-\frac{j-k+1}{n}\right)^N\right).$$

As we saw when we evaluated $C(x, \cdots, x)$, we know that

$$\sum_{1\leqq k<j\leqq n} f(j-k+1)=\sum_{1<j\leqq n}\sum_{1<k\leqq j} f(k)=\sum_{1<k\leqq n}(n-k+1)f(k),$$

so we now have

$$N-n+n\left(1-\frac{1}{n}\right)^N+2\sum_{1<k\leqq n}(n-k+1)\left(\frac{N}{n}\frac{1}{k}-\frac{1}{k^2}+\frac{1}{k^2}\left(1-\frac{k}{n}\right)^N\right).$$

This sum appears difficult to evaluate explicitly, mainly because of the last term. However, we may use this expression to prove:

THEOREM 1. *Any Quicksort program must require, on the average, at least*

$$N-n+2\sum_{1\leqq k<j\leqq n}\frac{x_k x_j}{x_k+\cdots+x_j}$$

comparisons to sort a random permutation of the multiset $\{x_1\cdot 1, \cdots, x_n\cdot n\}$ (*where*

$x_1+\cdots+x_n=N$) *and at least*

$$2N(1+1/n)H_n-3(N+n)$$

or, for large n, at least

$$2(N+1)H_N-4N+2(N/n)(H_N-1)+O(N^3/n^2)$$

comparisons to sort a random n-ary file of length N.

Proof. The result for multisets is proved in the discussion above, except that the theorem avoids some complications by using the fact that $\sum_{1\leqq k\leqq n} C(x_i)=\sum_{1\leqq k\leqq n}(x_k+1+\delta_{x_k 0})>N-n$.

To prove the results for n-ary files, we follow the discussion above and start with the expression

$$N-n+n\left(1-\frac{1}{n}\right)^N+2\sum_{1<k\leqq n}(n-k+1)\left(\frac{N}{n}\frac{1}{k}+\frac{1}{k^2}+\frac{1}{k^2}\left(1-\frac{k}{n}\right)^N\right).$$

The first sum obviously evaluates to $2N(1+1/n)(H_n-1)-2N(1-1/n)$, as above; the second sum can be bounded by noticing that

$$\sum_{1<k\leqq n}\frac{n-k+1}{k^2}=(n+1)\sum_{1\leqq k\leqq n}\frac{1}{k^2}-n-H_n<n,$$

since $\sum_{1\leqq k\leqq n}(1/k^2)<\sum_{k\geqq 1}(1/k^2)=\pi^2/6$; and the third sum is even smaller in absolute value than the second, so it won't weaken our bound much to ignore it. If these expressions are all substituted in, and the $n(1-1/n)^N$ term is also ignored, we get the expression $2N(1+1/N)H_n-3(N+n)$, as desired.

For large values of n, we can get a somewhat better bound, if we are content with an asymptotic answer. For example, the binomial theorem tells us that

$$N-n+n\left(1-\frac{1}{n}\right)^N=N-n+n\sum_{0\leqq i\leqq N}\binom{N}{i}\left(-\frac{1}{n}\right)^i=\sum_{1<i\leqq N}\binom{N}{i}\frac{(-1)^i}{n^{i-1}}$$
$$=\frac{1}{n}\binom{N}{2}+O\left(\frac{N^3}{n^2}\right).$$

The notation $O(N^3/n^2)$ can be assigned a precise meaning, but here it will suffice to say that the terms represented by this notation can be ignored for $n \gg N$. Now, to evaluate the sum, we begin in the same way, applying the binomial theorem to get

$$-\frac{1}{k^2}+\frac{N}{n}\frac{1}{k}+\frac{1}{k^2}\left(1-\frac{k}{n}\right)^N=\sum_{1<i\leqq N}\binom{N}{i}\left(-\frac{1}{n}\right)^i k^{i-2}.$$

After rearranging terms slightly, our lower bound becomes

$$2\sum_{1\leqq k\leqq n}(n-k+1)\sum_{1<i\leqq N}\binom{N}{i}\frac{k^{i-2}(-1)^i}{n^i}-\frac{1}{n}\binom{N}{2}+O\left(\frac{N^3}{n^2}\right).$$

Now, we know from Euler's summation formula that

$$\sum_{1\leqq k\leqq n}k^{i-2}=\frac{n^{i-1}}{i-1}+\sum_{1\leqq j\leqq i-2}\frac{B_j}{j}\binom{i-2}{j-1}n^{i-j-1},$$

where B_j are the Bernoulli numbers. Therefore,

$$2(n+1)\sum_{1\leqq k\leqq n}\sum_{1<i\leqq N}\binom{N}{i}\frac{k^{i-2}(-1)^i}{n^i}$$

$$=2(n+1)\sum_{1\leqq j\leqq N-2}\frac{B_j}{jn^{j+1}}\sum_{j+2\leqq i\leqq N}\binom{N}{i}\binom{i-2}{j-1}(-1)^i$$

$$+2\left(1+\frac{1}{n}\right)\sum_{1<i\leqq N}\binom{N}{i}\frac{(-1)^i}{i-1}$$

$$=2(n+1)\sum_{1\leqq j\leqq N-2}\frac{B_j}{jn^{j+1}}(-1)^{j+1}\left(N-j-\binom{N}{j+1}\right)$$

$$+2\left(1+\frac{1}{n}\right)\sum_{1<i\leqq N}\binom{N}{i}\frac{(-1)^i}{i-1}$$

$$=2\left(1+\frac{1}{n}\right)\sum_{1<i\leqq N}\binom{N}{i}\frac{(-1)^i}{i-1}-\frac{1}{n}\left(N-1-\binom{N}{2}\right)+O\left(\frac{N^3}{n^2}\right).$$

(The inner sum evaluated on the second line of this derivation is tricky, but involves only elementary identities from Knuth [8, § 1.2.6].) Similarly, we find that

$$2\sum_{1\leqq k\leqq n}\sum_{1<i\leqq N}\binom{N}{i}\frac{k^{i-1}(-1)^i}{n^i}=2\sum_{1<i\leqq N}\binom{N}{i}\frac{(-1)^i}{i}-\frac{N-1}{n}+O\left(\frac{N^2}{n^2}\right).$$

Putting these results together gives a rather simple expression for our lower bound:

$$2\left(1+\frac{1}{n}\right)\sum_{1<i\leqq N}\binom{N}{i}\frac{(-1)^i}{i-1}+2\sum_{1<i\leqq N}\binom{N}{i}\frac{(-1)^i}{i}+O\left(\frac{N^3}{n^2}\right).$$

Finally, we can evaluate these sums by applying an identity given by Knuth [8, §1.2.7, Ex. 13].

$$\sum_{1\leqq i\leqq n}\frac{x^i}{i}=H_n+\sum_{1\leqq k\leqq n}\binom{n}{k}\frac{(x-1)^k}{k}.$$

If we take $x=0$ in this formula, we get an identity for evaluating our first sum; if we integrate the equation from 0 to t, we get

$$\sum_{1\leqq i\leqq n}\frac{t^{i+1}}{i(i+1)}=H_nt+\sum_{1\leqq k\leqq n}\binom{n}{k}\frac{(t-1)^{k+1}}{k(k+1)}-\sum_{1\leqq k\leqq n}\binom{n}{k}\frac{(-1)^{k+1}}{k(k+1)},$$

which, evaluated at $t=1$, gives an identity for evaluating our second sum. The stated result follows immediately. □

The lower bounds given in Theorem 1 are particularly weak for small values of n. For example, when $n=2$, they grow linearly with N. As we will see, many practical implementations of Quicksort do not do so well for binary files. In fact, many implementations use $O(N^2)$ comparisons for binary files, and we can raise our lower bound for such programs.

COROLLARY. *If a Quicksort program requires, on the average, more than $\alpha\binom{N}{2}$ comparisons for unary or binary files of length N, then it will require at least*

$$2N\left(n+\frac{1}{n}\right)H_n-4N-3n+\left(1-\frac{1}{n}\right)\frac{\alpha}{n}\binom{N}{2}$$

comparisons on the average, for n-ary files of length N.

Proof. The result for unary files follows directly by not evaluating $C(x_k)$ immediately in the derivation of Theorem 1: the bound is just

$$C_{Nn}>\frac{1}{n^N}\sum_{1\leqq k\leqq n}\sum_{x_1+\cdots+x_n=N}\binom{N}{x_1,\cdots,x_n}C(x_k)+2N\left(1+\frac{1}{n}\right)H_n-4N-2n.$$

To bound this term, we shall use the general identity

$$\frac{1}{n^N}\sum_{x_1+\cdots+x_n=N}\binom{N}{x_1,\cdots,x_n}\binom{x_k,\cdots,x_{k+m-1}}{t}$$

$$=\frac{1}{n^N}\sum_{x_1+\cdots+x_{k+1}+i+x_{k+m}+\cdots+x_n=N}\binom{i}{t}\binom{N}{x_1,\cdots,x_{k-1},i,x_{k+m},\cdots,x_n}$$

$$\sum_{x_k+\cdots+x_{k+m-1}=i}\binom{i}{x_k,\cdots,x_{k+m-1}}$$

$$=\frac{1}{n^N}\binom{N}{t}\sum_{x_1+\cdots+x_{k-1}+i+x_{k+m}+\cdots+x_n=N}\binom{N-t}{x_1,\cdots,x_{k-1},i-t,x_{k+m},\cdots,x_n}m^i$$

$$=\frac{m^t}{n^N}\binom{N}{t}n^{N-t}=\frac{m^t}{n^t}\binom{N}{t}.$$

If $C(x_k)>\alpha\binom{x_k}{2}$, then we may take $m=1$, $t=2$ to get the result

$$C_{Nn}>2N\left(1+\frac{1}{n}\right)H_n-4N-2n+\frac{\alpha}{n}\binom{N}{2},$$

which implies the stated bound.

For binary files, we follow exactly the derivation of Theorem 1 except that the telescoping recurrences for C and G can each be stopped one step sooner to yield

$$C(x_1,\cdots,x_n)=\sum_{1\leqq k\leqq n-1}C(x_k,x_{k+1})-\sum_{1\leqq k\leqq n-2}C(x_{k+1})$$
$$+2\sum_{3\leqq k+2\leqq j\leqq n}\frac{x_kx_j}{x_k+\cdots+x_j}.$$

When we average over all multisets on N elements, the calculations are similar to those in the proof of Theorem 1. If $C(x_{k+1})>\alpha\binom{x_{k+1}}{2}$, then the proof for unary files proves the corollary. Therefore we may assume that $C(x_{k+1})\leqq\alpha\binom{x_{k+1}}{2}$ and

$C(x_k, x_{k+1}) > \alpha\binom{x_k + x_{k+1}}{2}$. The identity above then tells us that

$$\frac{1}{n^N}\sum_{1\leqq k\leqq n-1}\ \sum_{x_1+\cdots+x_n=N}\binom{N}{x_1,\cdots,x_n}C(x_k, x_{k+1}) > \left(1-\frac{1}{n}\right)\frac{4\alpha}{n}\binom{N}{2} \qquad (m=2, t=2)$$

and

$$\frac{1}{n^N}\sum_{1\leqq k\leqq n-2}\ \sum_{x_1+\cdots+x_n=N}\binom{N}{x_1,\cdots,x_n}C(x_{k+1}) \leqq \left(1-\frac{1}{n}\right)\frac{\alpha}{n}\binom{N}{2} \qquad (m=1, t=2).$$

Evaluating the third term exactly as for Theorem 1, we find that

$$C_{Nn} > 2N\left(1+\frac{1}{n}\right)H_n - \frac{17}{6}n + \frac{5}{6}\frac{N}{n} + \left(1-\frac{1}{n}\right)\frac{3\alpha}{n}\binom{N}{2},$$

which implies the stated bound.

From the corollary, we conclude that if a Quicksort program is quadratic for binary files, then it is quadratic for all n-ary files when n is small. This type of effect arises often in the study of Quicksort, since all files are eventually partitioned to yield degenerate ones. It will be even more prominent in the next section, when we deal with upper bounds.

4. Upper bounds. The derivation of a general upper bound on the number of comparisons needed by any Quicksort program proceeds in much the same manner as for the lower bound. However, some extra care will be necessary for two reasons. First, a bound is needed for the number of comparisons used to partition N elements. We will be content to use $N+1$, since we shall later see some programs that use exactly that many. Other programs might use more, but if the number of comparisons that they use grows linearly with N, we can still apply our results by multiplying through by a constant. Another problem arises because, as we have seen, some partitioning methods can perform badly with files that only have a few distinct values. For the present, it will be convenient to restrict these problems to a single term by defining

$$I(x_1, \cdots, x_n) = \sum_{1\leqq k\leqq n-2} C(x_k, x_{k+1}, x_{k+2}) - \sum_{1\leqq k\leqq n-3} C(x_{k+1}, x_{k+2})$$

where $C(x_1, \cdots, x_n)$ is the maximum number of comparisons needed, on the average, to sort a permutation of the multiset $\{x_1 \cdot 1, \cdots, x_n \cdot n\}$. We will look at assumptions about our programs to help bound $I(x_1, \cdots, x_n)$ after we have proved

THEOREM 2. *Any Quicksort program which partitions N elements with $N+1$ comparisons will require, on the average, no more than*

$$2\sum_{4\leqq k+3\leqq j\leqq n}\frac{x_k x_j}{1+x_{k+1}+\cdots+x_{j-1}} + I(x_1, \cdots, x_n)$$

comparisons to sort a random permutation of the multiset $\{x_1, \cdots, x_n\}$ and no more *than*

$$2N\left(1-\frac{1}{n}\right)H_n-3N+2\frac{N}{n}-9\left(\frac{N}{n}\right)^2-7\frac{N}{n^2}$$

or, for large values of n, no more than

$$2N(H_N+1)-2+O(N^2/n)$$

comparisons to sort a random n-ary file of length N.

Proof. Arguing the same way as in the derivation for the lower bound, we start by noticing that an upper bound is certainly described by the recurrence

$$C(x_1,\cdots,x_n)=N+1+\frac{1}{N}\sum_{1\leqq j\leqq n} x_j(C(x_1,\cdots,x_j-1)+C(x_j,\cdots,x_n)),\qquad n>1.$$

Proceeding exactly as before, we difference twice: first subtract the same equation for $\{x_2\cdot 1,\cdots,x_n\cdot(n-1)\}$; then define $G(x_1,\cdots,x_n)=C(x_1,\cdots,x_n)-C(x_2,\cdots,x_n)$ and subtract the same equation for $\{x_1\cdot 1,\cdots,x_{n-1}\cdot(n-1)\}$. This leaves

$$\left(\sum_{2\leqq j\leqq n} x_j\right)G(x_1,\cdots,x_n)-\left(\sum_{2\leqq j\leqq n-1} x_j\right)G(x_1,\cdots,x_{n-1})$$
$$=2x_1x_n+x_nG(x_1,\cdots,x_{n-1},x_n-1),\qquad n>3$$

(Notice that this equation does not hold for $n=2$ or 3 as was the case for the lower bound.) In order to get this equation to telescope, we multiply both sides by

$$\frac{(x_2+\cdots+x_n-1)!}{x_n!(x_2+\cdots+x_{n-1})!},$$

which gives

$$\binom{x_2+\cdots+x_n}{x_n}G(x_1,\cdots,x_n)=\binom{x_2+\cdots+x_n-1}{x_n-1}(G(x_1,\cdots,x_n-1)+2x_1)$$
$$+\binom{x_2+\cdots+x_n-1}{x_n}G(x_1,\cdots,x_{n-1}).$$

Now every place that x_n appears on the left side, x_n-1 appears in the first term on the right, so this equation telescopes to yield

$$\binom{x_2+\cdots+x_n}{x_n}G(x_1,\cdots,x_n)=\binom{x_2+\cdots+x_n}{x_n}G(x_1,\cdots,x_{n-1})$$
$$+2x_1\binom{x_2+\cdots+x_n}{x_n-1}$$

or

$$G(x_1,\cdots,x_n)=G(x_1,\cdots,x_{n-1})+\frac{2x_1x_n}{1+x_2+\cdots+x_{n-1}},\qquad n>3.$$

Substituting $G(x_1,\cdots,x_n)=C(x_1,\cdots,x_n)-C(x_2,\cdots,x_n)$ and telescoping once more leads to the desired result for permutations of multisets.

To complete the proof, for random n-ary files of length N, we will first evaluate

$$\frac{2}{n^N}\sum_{x_1+\cdots+x_n=N}\binom{N}{x_1,\cdots,x_n}\sum_{4\leqq k+3\leqq j\leqq n}\frac{x_kx_j}{1+x_{k+1}+\cdots+x_{j-1}}.$$

Continuing as before, we interchange the order of summation and then split the inner sum and the multinomial coefficient to get

$$\frac{2}{n^N}\sum_{4\leqq k+3\leqq j\leqq n}\ \sum_{x_1+\cdots+x_k+i+x_j+\cdots+x_n=N}\binom{N}{x_1,\cdots,x_k,i,x_j,\cdots,x_n}$$
$$\frac{x_kx_j}{i+1}\sum_{x_{k+1}+\cdots+x_{j-1}=i}\binom{i}{x_{k+1},\cdots,x_{j-1}}.$$

The multinomial theorem applies to the inner sum, and the remaining multinomial coefficient simplifies, leaving

$$\frac{2N}{n^N}\sum_{4\leqq k+3\leqq j\leqq n}\ \sum_{x_1+\cdots+x_k+i+x_j+\cdots+x_n=N}\binom{N-1}{x_1,\cdots,x_k-1,i+1,x_j-1,\cdots,x_n}$$
$$(j-k-1)^i.$$

After replacing i by $i-1$, including a term for $i=0$, and applying the multinomial theorem twice, we are left with

$$2\frac{N}{n}\sum_{4\leqq k+3\leqq j\leqq n}\left(\frac{1}{j-k-1}-\frac{1}{j-k-1}\left(1-\frac{j-k-1}{n}\right)^{N-1}\right),$$

or

$$2\frac{N}{n}\sum_{2\leqq k\leqq n-2}(n-k-1)\left(\frac{1}{k}-\frac{1}{k}\left(1-\frac{k}{n}\right)^{N-1}\right).$$

If n is not large we will not weaken our bound much by ignoring the second term, so the result

$$2N(1-1/n)H_{n-1}-4N+6N/n$$

follows immediately. If n is large, then we expand $(1-k/n)^{N-1}$ by the binomial theorem to get

$$-2\frac{N}{n}\sum_{2\leqq k\leqq n-2}\frac{n-k-1}{k}\sum_{1\leqq i\leqq N-1}\binom{N-1}{i}\left(-\frac{k}{n}\right)^i,$$

which reduces, after evaluating the sum on k with Euler's summation formula just as above, to

$$-2N\sum_{1\leqq i\leqq N-1}\binom{N-1}{i}\frac{(-1)^i}{i}-2N\sum_{1\leqq i\leqq N-1}\binom{N-1}{i}\frac{(-1)^i}{i+1}+O(N^2/n).$$

The second term turns out to be $2(N-1)$, and the first is just $2NH_N$, so we have

$$2N(H_N+1)-2+O(N^2/n).$$

Finally, we must average $I(x_1,\cdots,x_n)$ over all multisets of length N. First, since a Quicksort program uses $N+1$ comparisons and gets at least one element in place on each partitioning stage, a trivial upper bound is

$$C(x_1, \cdots, x_n) \leqq \sum_{2 \leqq k \leqq N} (k+1) = \tfrac{1}{2}(N+4)(N-1) < \binom{N}{2} + 2N.$$

Therefore, we must certainly have

$$I(x_1, \cdots, x_n) < \sum_{1 \leqq k \leqq n-2} \left(\binom{x_k + x_{k+1} + x_{k+2}}{2} + 2(x_k + x_{k+1} + x_{k+2}) \right)$$

and, applying the identity given in the proof to Corollary 1 of Theorem 1, we then find that

$$\frac{1}{n^N} \sum_{x_1 + \cdots + x_n = N} \binom{N}{x_1, \cdots, x_n} I(x_1, \cdots, x_n) < \left(1 - \frac{2}{n}\right)\left(\frac{9}{n}\binom{N}{2} + N\right),$$

and, in particular, for large n, the right-hand side is $O(N^2/n)$.

The theorem now follows immediately from the results in the preceding two paragraphs. □

Notice that if n is $O(1)$, then the bound becomes $O(N^2)$. Again, this is a result of the recursive structure of Quicksort, and it is due to the fact that some Quicksort programs are inefficient for files with a small number of key values. On the other hand, not all Quicksort programs have this problem, and if the method works well for binary files, we can eliminate the quadratic term.

COROLLARY. *If a Quicksort program can sort a random binary file of N elements with less than $2NH_N$ comparisons, on the average, then it will require no more than*

$$2N\left(1 - \frac{1}{n}\right)H_{n-1} - 4N + 6\frac{N}{n}H_N + \frac{39}{2}\frac{N}{n},$$

comparisons to sort a random n-ary file of N elements.

Proof. The expression given is a crude approximation intended only to show that the bound is not quadratic, so our estimates will be somewhat rough. First, we can remove the term describing ternary files by noticing that, from our most general recurrence, we certainly must have

$$\begin{aligned}(x_1 + x_2 + x_3)C(x_1, x_2, x_3) \leqq 2\binom{x_1 + x_2 + x_3 + 1}{2} + x_1 C(x_1, x_2, x_3) \\ + x_2 C(x_1, x_2) + x_2 C(x_2, x_3) + x_3 C(x_1, x_2, x_3).\end{aligned}$$

From this it follows that

$$C(x_1, x_2, x_3) < \frac{3}{x_2 + 1}\binom{x_1 + x_2 + x_3 + 1}{2} + C(x_1, x_2) + C(x_2, x_3),$$

and, therefore,

$$I(x_1, \cdots, x_n) < 3 \sum_{1 \leqq k \leqq n-2} \frac{1}{x_{k+1}+1} \binom{x_k + x_{k+1} + x_{k+2} + 1}{2} + \sum_{1 \leqq k \leqq n-1} C(x_k, x_{k+1}).$$

By hypothesis, we certainly have $C(x_k, x_{k+1}) < 2(x_k + x_{k+1})H_N$. The calculations involved in averaging this over all multisets of N elements are similar to those we have seen many times before. It turns out that the first term is less than $\frac{27}{2}(N/n)$, and the second is equal to $6(N/n)H_n$, so the desired result follows directly. □

The upper and lower bounds that we have derived for the number of comparisons give some indication of how well we can expect to do when implementing a Quicksort for files with equal keys. If n is very, very large, then the bounds differ by only $6N - 2H_N - 2$, so we have verified the traditional argument that equal keys are unlikely to occur in this case and their effect can be ignored. If $n = O(N)$, then the upper and lower bounds differ only slightly, and we should not expect one method for dealing with equal keys to differ substantially from another. And if n is small, then the bounds tell us that we should take care to ensure that our method operates efficiently for binary files.

5. Implementations. Although it is tempting to contemplate sophisticated algorithms for dealing with equal keys during the partitioning process, we shall be content to study three methods which require virtually no overhead for their implementation. We shall see that one of these performs very well, and it is unlikely that it would be worthwhile to incur any extra overhead in Quicksort to deal with equal keys.

The first method that we shall consider is of course Program 1 as it stands, which sorts properly and efficiently when equal keys are present. If we replace "<" by "≦" and ">" by "≧" in the discussion following the program, we find that it applies as well when equal keys are present, except for one subtle point. It is possible for the condition $i = j$ to occur outside the inner loops, so that the pointer scans ultimately terminate with $i = j + 2$, and the two keys $A[j]$ and $A[j+1]$ are put in place by partitioning. (Although we do an extraneous exchange $A[i] :=: A[j]$ when $j = i$, it is much less efficient to exit the loop when $j = i$ because not only is the chance to get two elements in place missed, but also when the left subfile is later partitioned, the partitioning element chosen will be the largest in that subfile.) It is important to notice such anomalies if the analysis is to be correct. In any case, although Program 1 clearly works, it is reasonable to ask if there is a more efficient method of distributing the keys equal to the partitioning element into the subfiles.

Another possibility is to change the $<$ and $>$ signs in the inner loops of Program 1 to $\leqq$ and $\geqq$. If a key smaller than all the others is chosen as the partitioning element, the j pointer will scan past the left end of the file, so we need to protect against this case by setting $A[0] := -\infty$. Even worse, it might be possible for the pointers to access elements far outside the array bounds $A[l], \cdots, A[r]$ during intermediate partitioning stages. This situation could be avoided by putting tests in the inner loops to check the pointers, but it is more efficient to put $-\infty$ and ∞ in $A[l-1]$ and $A[r+1]$ before partitioning and restore them afterwards. This leaves us with

PROGRAM 2.

```
procedure quicksort (integer value $l, r$);
    comment The array $A$ is declared to be $A[0:N+1]$ with $A[0]=-\infty$ and
            $A[N+1]=\infty$;
    if $r>l$ then
        $A[l-1]:=:A[0]; A[r+1]:=:A[N+1]$
        $i:=l; j:=r+1; v:=A[l]$;
        loop:
            loop: $i:=i+1$; while $A[i]\leqq v$ repeat;
            loop: $j:=j-1$; while $A[j]\geqq v$ repeat;
        until $j<i$:
            $A[i]:=:A[j]$;
        repeat;
        if $j>l$ then $A[l]:=:A[j]; j:=j-1$; endif;
        $A[l-1]:=:A[0]; A[r+1]:=:A[N+1]$;
        quicksort $(l, j)$;
        quicksort $(i, r)$;
    endif;
```

Notice that after partitioning we have $A[l], \cdots, A[j-1]\leqq A[j]=A[j+1]=\cdots=A[i-1]\leqq A[i], \cdots, A[r]$, with $1\leqq i, j\leqq N+1$, so that a number of keys can be put into position on one partitioning stage.

Finally, we might consider allowing equality in only one of the inner loops of Progam 1, and leave the other inequality strict. The two possibilities are basically symmetric, and we will consider

PROGRAM 3.

```
procedure quicksort (integer value $l, r$):
    comment The array $A$ is declared to be $A[1:N+1]$ with $A[N+1]=\infty$;
    if $r>l$ then
        $A[r+1]:=:A[N+1]$;
        $i:=l; j:=r+1; v:=A[l]$;
        loop:
            loop: $i:=i+1$; while $A[i]\leqq v$ repeat;
            loop: $j:=j-1$; while $A[j]>v$ repeat;
        until $j<i$:
            $A[i]:=:A[j]$;
        repeat;
        $A[l]:=:A[j]$;
        $A[r+1]:=:A[N+1]$;
        quicksort $(l, j-1)$;
        quicksort $(j+1, r)$;
    endif;
```

This program always puts exactly one partitioning element into position.

In summary, Program 1 stops the pointers on keys equal to the partitioning element, Program 2 scans over equal keys, and Program 3 puts them into the left subfile. Clearly there is a version symmetric to Program 3 which puts them into the right subfile. Versions of all of these approaches have appeared at one time or another in the literature. Hoare's original program scanned over equal keys [6], [7], and several authors then adopted that approach [2], [5], [11], [13]. (However, the later authors "improved" Hoare's program to test if the pointers cross each time they are changed. The reader will soon appreciate how unfortunate this strategy is when, for example, all the keys are equal.) R. C. Singleton was the first to suggest stopping the pointers on keys equal to the partitioning element [15], and the idea was accepted by others [4], [9], though no analytic justification was given. The idea of putting all the keys equal to the partitioning element in one subfile or the other appears in some versions of Quicksort [1], [3], though no one has given any particular reason for doing so.

It is not at all clear *a priori* which of the programs should be recommended, for there are situations in which each performs better than the others. Figure 2 illustrates this by showing the operation of the programs on three different files, along with the number of comparisons used. (The differences between the progams are most apparent in the second example, which shows all three "sorting" seven equal keys.) Program 1 expends a few extra exchanges to get balanced partitions, Program 2 can get more than one key into place on one partitioning stage, and Program 3, due to its asymmetrical nature, can produce unbalanced partitions. In the following sections, we shall attempt to quantify these remarks by looking at the analysis of the programs. We shall see exactly how many comparisons they use, on the average, for unary and binary files, and then we shall

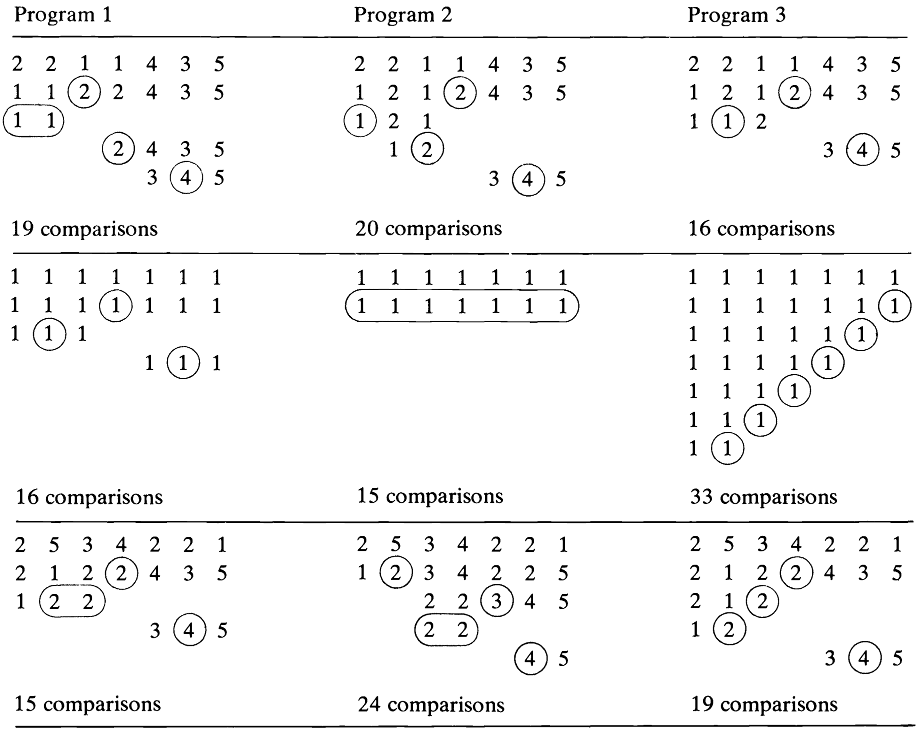


FIG. 2

prove that Program 1 must be preferred because it tends to produce partitions closer to the center.

There are a few more issues relating to the practical implementation of Quicksort on equal keys which we shall treat before moving on to the analysis. The first is a property called *stability* which is often of concern in practical sorting programs. A sorting program is *stable* if it preserves the relative order among equal keys. Unfortunately, our programs are not stable, because no matter how we treat keys equal to the partitioning element, the relative order of other keys might be disturbed. The easiest way to provide stability, if there is extra space available, is to append each key's index to itself before sorting. For example, if we are sorting small integers, this can be done by the statement

$$\textbf{loop for } 1 \leqq i \leqq N: A[i] := A[i]*N+i-1 \textbf{ repeat};$$

This transformation makes all the keys distinct and preserves their relative order. We have $A[i]<A[j]$ before the transformation only if $A[i]<A[j]$ after the transformation; and if $i<j$ and $A[i]=A[j]$ before, then $A[i]<A[j]$ after. We can now achieve a stable method by sorting the file and then transforming back to our original keys:

$$\textbf{loop for } 1 \leqq i \leqq N: A[i] := A[i]/N \textbf{ repeat};$$

Of course, since this method is costly in terms of both time and space (each key must be a little bigger), it should not be used unless stability is important. If the extra space is not available, then Rivest has shown that a stable Quicksort involving $O(N(\log N)^2)$ comparisons can be devised [12]. This method is of limited practical utility, but it is an important theoretical result.

The programs we have defined gain efficiency by using sentinel keys, $-\infty$ and ∞, to stop the scanning pointers from going outside the array bounds. For Program 3 it is necessary for ∞ to be strictly greater than all of the other keys, and for Program 2 it is also necessary for $-\infty$ to be strictly less than the others. It may be difficult to define such keys in some practical situations. For example, if the keys to be sorted can take on any value which can be represented in one word in a computer, then by definition we cannot represent a key larger than all possible values in one word. This is not a problem for Program 1, since it only requires that ∞ be greater than or equal to all the other keys.

6. Unary files. Our derivation of the upper and lower bound suggests that we should know how our programs perform in the degenerate case when only a few distinct key values are present, so let us examine first what happens when all the keys are equal.

Program 1 comes as close as possible to dividing the file exactly in half at each stage. The number of comparisons used is described by the recurrence

$$C_N = N+1+2C_{(N-1)/2}, \qquad N>1,$$

with $C_1=0$. If N is of the form 2^k-1, then this reduces to

$$C_{2^k-1} = 2^k + 2C_{2^{k-1}-1}, \qquad k>1.$$

Dividing both sides by 2^k, this immediately telescopes to the solution

$$C_N = (N+1)\lg((N+1)/2), \quad \text{when } N = 2^k - 1, \quad k > 1.$$

The solution for general N is somewhat complicated because it depends on the binary representation of N, but it is easily shown by induction that $C_N \leqq (N+1)\lg((N+1)/2)$ for all N, so that Program 1 performs acceptably on unary files.

Program 2 is much easier to analyze, for it "sorts" all unary files in only one partitioning stage. Each pointer scans all the way across the file, the left and right subfiles are both empty, and a total of only $2N+1$ comparisons are used.

On the other hand, unary files represent the worst case for Program 3. Each partitioning stage only removes one element from the right end of the file to be sorted, so

$$\sum_{2 \leqq j \leqq N} (j+1) = \tfrac{1}{2}(N-1)(N+4)$$

comparisons are used.

It is interesting that these three programs, which seem to be so similar, perform so differently when the keys are all equal. One uses $O(N \log N)$ comparisons, the second is linear, and the third is quadratic!

7. Binary files. Now let us consider the less degenerate case when binary files are to be sorted. The analysis is more complex, but it does give us some more insight into the relative performance of the programs.

The easiest of the three to analyze is Program 2. We wish to find C_N, the average number of comparisons to sort a binary file of length N, given that all 2^N such files are equally likely. Suppose that the two values are 0 and 1, and define $C_N^{(0)}$ and $C_N^{(1)}$ to be the averages for files that start with 0 and 1, respectively, so that $C_N = \frac{1}{2}(C_N^{(0)} + C_N^{(1)})$. First, we will find a recurrence for $C_N^{(0)}$ by noticing that the situation after the first partitioning stage is as follows ("x" denotes keys which may be 0 or 1):

$$\underset{j}{-\infty}\ \overbrace{0\ \ 0\ \ 0\cdots 0}^{k}\ \overbrace{\underset{i}{1}\ \ x\ \ x\ \ x\cdots x}^{N-k}\ \ \infty.$$

Partitioning required $N+k+1$ comparisons, and all that is left to be sorted is a file of size $N-k$, random, except for its first key, which is 1. This leads us to the recurrence

$$C_N^{(0)} = \frac{2N+1}{2^{N-1}} + \sum_{1 \leqq k \leqq N-1} \frac{1}{2^k}(N+k+1+C_{N-k}^{(1)}).$$

By a similar argument, we can show that

$$C_N^{(1)} = \frac{2N+1}{2^{N-1}} + \sum_{1 \leqq k \leqq N-1} \frac{1}{2^k}(N+k+C_{N-k}^{(0)}),$$

and therefore C_N satisfies

$$C_N=\frac{2N+1}{2^{N-1}}+\sum_{1\leqq k\leqq N-1}\frac{1}{2^k}(N+k+\tfrac{1}{2}+C_{N-k}),\qquad N>0.$$

Multiplying by 2^N and replacing k by $N-k$ in the sum, we get

$$2^N C_N=4N+2+\sum_{1\leqq k\leqq N-1}2^k(2N-k+\tfrac{1}{2}+C_k).$$

Subtracting the same equation for $N-1$, we get the recurrence

$$C_N=C_{N-1}+\frac{N}{2}+\frac{7}{4}\quad \text{for } N\geqq 2,$$

with the initial condition $C_1=3$, which telescopes to the solution

$$\tfrac{1}{4}(N^2+8N+3).$$

We might have expected that this average number of comparisons would be proportional to N^2 if we had noticed that two successive partitioning stages simply exchange the leftmost 1 with the rightmost 0.

Program 3 may be analyzed in a similar fashion, but the calculations are somewhat more complex. Alternatively, we can analyze Program 3 in much the same way as we developed our upper and lower bounds. (Unfortunately, the other programs don't lend themselves as easily to this kind of analysis.) The number of comparisons required by Program 3 to sort to random permutation of the multiset $\{x_1\cdot 1,\cdots,x_n\cdot n\}$ is described by the recurrence

$$C(x_1,\cdots,x_n)=N+1+\frac{1}{N}\sum_{1\leqq j\leqq n}x_j(C(x_1,\cdots,x_j-1)+C(x_{j+1},\cdots,x_n))$$

where $N=\sum_{1\leqq j\leqq n}x_j$, and the notational conventions are the same as above. Proceeding in exactly the same manner as for the derivation of the upper bound, we find that

$$C(x_1,\cdots,x_n)=\sum_{1\leqq k\leqq n}\binom{x_k+2}{2}-n+2\sum_{1\leqq k<j\leqq n}\frac{x_kx_j}{1+x_k+\cdots+x_{j-1}}.$$

This formula is due to Burge [3], although he develops a slightly different version. For binary files, we get

$$C(x,N-x)=\binom{x+2}{2}+\binom{N-x+2}{2}-2+2\frac{x(N-x)}{1+x}.$$

The average is

$$\sum_{0\leqq x\leqq N}\frac{\binom{N}{x}}{2^N}C(x,N-x)=\frac{1}{2^{N-1}}\sum_{0\leqq x\leqq N}\binom{x+2}{2}\binom{N}{x}$$

$$-2+\frac{1}{2^{N-1}}\sum_{0\leqq x\leqq N}\binom{N}{x}\frac{x(N-x)}{1+x}.$$

After application of the identities

$$\binom{x+2}{2}\binom{N}{x}=\binom{N}{2}\binom{N-2}{x-2}+2N\binom{N-1}{x-1}+\binom{N}{x}$$

and

$$\binom{N}{x}\frac{x(N-x)}{1+x}=N\binom{N-1}{x}-\binom{N}{x+1},$$

this reduces to the solution

$$C_N=\tfrac{1}{4}(N^2+11N-8)+\frac{1}{2^{N-1}},$$

so Program 3 is also quadratic for binary files.

Fortunately, we can show that Program 1 does not perform so badly on binary files, even though an exact formula for the average appears difficult to derive. What we can do is derive an upper bound on the number of comparisons taken by Program 1 on any binary file. This will of course also be a bound on the average. The proof is based on a different method of counting comparisons than we have been using. We know that each partitioning stage contributes one comparison to the total for each element involved plus one extra comparison when the pointers cross. But we can also count comparisons by counting how many partitioning stages each element is involved in, then adding N for the pointer crossing overhead (there can be no more than N partitioning stages). Notice that each partition in Program 1 results in one subfile with all keys equal and another "unsorted" subfile. The subfiles with all keys equal are clearly processed in a logarithmic number of stages, since they are always split in the middle. Now consider the unsorted subfile. After each partitioning stage, at least half of the keys equal to the partitioning element must be removed. Therefore the unsorted part of the file cannot last through more than $2 \lg N$ partitions, and every element in the whole file is involved in at most $2 \lg N$ partitioning stages. Therefore the total number of comparisons must be less than $2N \lg N+N$ (and this is not a particularly tight bound). This is substantially better than the quadratic performance of Programs 2 and 3.

These results for binary files, coupled with the upper and lower bounds developed above, represent strong evidence that Program 1 is the method of choice when n is small. The corollary to Theorem 1 says that Programs 2 and 3 will be quadratic; and the corollary to Theorem 2 says that Program 1 will still require only $O(N \log N)$ comparisons, on the average, for small n. Of course, it must be noted that if it is known that n will always be small, a special-purpose sorting program written to take that fact into account might be more appropriate than the general-purpose programs that we have been studying. For example, the best way to sort a binary file is to effectively "partition" the file on the value $\frac{1}{2}$: scan from the left to find a 1, scan from the right to find a 0, exchange them, and continue until the pointers cross. The whole file can be sorted with $N+1$ comparisons. Similarly, if a file is known to be ternary (consisting of 0's, 1's and 2's), it can be sorted with $2(N+1)$ comparisons by first partitioning on the value 1, then treating the binary subfiles as above. In the same manner, a file with 2^t+1 distinct values can always be sorted with $(t+1)(N+1)$ comparisons, if the values are all known. Another example, which is most useful when keys to be sorted fall into a small range, is the idea of *distribution counting* (see [9, § 5.2]), where the file is sorted in two passes: one to count the number of occurrences of each key, and a second to move the keys into place according to the counts. Such special-purpose programs may be made to outperform Program 1 under some conditions for small n; but we have shown that Program 1 does perform acceptably, and it can be expected to perform better than other general-purpose sorting programs when many equal keys are present.

8. The general case. In the general case, the exact analysis of Programs 1 and 2 appears to become intractable, so we shall adopt a more indirect approach to compare the programs. The idea is to notice that Quicksort performs best when the partitions at each stage tend to be near the center. Consequently, we would like to discover which of our algorithms produces partitions closest to the center, on the average.

When Singleton first proposed stopping the pointers on keys equal to the partitioning element [15], he claimed that it produces a "better split" than Hoare's original method of scanning over equal keys. However, he gave only empirical justification, and it is not at all obvious that this is so. For example, given the input file

$$2\ \ 2\ \ 2\ \ 2\ \ 1\ \ 1\ \ 1\ \ 2\ \ 2\ \ 3\ \ 3\ \ 3\ \ 3\ \ 3\ \ 3,$$

the first stage of Program 2 will produce the partition

$$1\ \ 2\ \ 2\ \ 2\ \ 1\ \ 1\ \ \boxed{2\ \ 2\ \ 2}\ \ 3\ \ 3\ \ 3\ \ 3\ \ 3\ \ 3,$$

while Program 1 results in the less balanced partition

$$1\ \ 2\ \ 2\ \ 1\ \ 1\ \ \textcircled{2}\ \ 2\ \ 2\ \ 2\ \ 3\ \ 3\ \ 3\ \ 3\ \ 3\ \ 3.$$

On the other hand, Program 2 performs worse for the input file

$$2\ \ 3\ \ 3\ \ 3\ \ 3\ \ 3\ \ 3\ \ 2\ \ 2\ \ 1\ \ 1\ \ 1\ \ 2\ \ 2\ \ 2,$$

since it produces the partition

$$1\ \ 1\ \ 1\ \ \textcircled{2}\ \ 3\ \ 3\ \ 3\ \ 2\ \ 2\ \ 3\ \ 3\ \ 3\ \ 2\ \ 2\ \ 2,$$

while Program 1 partitions the file perfectly:

$$2\ \ 2\ \ 2\ \ 2\ \ 1\ \ 1\ \ 1\ \ \textcircled{2}\ \ 2\ \ 3\ \ 3\ \ 3\ \ 3\ \ 3\ \ 3.$$

Although examples like these would seem to make comparing the algorithms difficult, it turns out that no matter how well Program 2 performs on an input file, there is another file for which Program 1 does at least as well.

THEOREM 3. *When Program* 2 *operates on a file* $A[1], \cdots, A[N]$, *it produces a partition no closer to the center than Program* 1 *operating on the file* $A[1], A[N], A[N-1], \cdots, A[2]$.

Proof. Specifically, let j and i define the position of the partition after Program 2 is used on $A[1], \cdots, A[N]$, so that after partitioning we have

$A[1], A[2], \cdots, A[j] \leqq A[j+1] = A[j+2] = \cdots = A[i-1] \leqq A[i], \cdots, A[N]$; and let $j'+\delta$ define the position of the partition after Program 1 is used on $A[1], A[N], A[N-1], \cdots, A[2]$, so that after partitioning we have $A[1], A[2], \cdots, A[j'-1] \leqq A[j'] = A[j'+\delta] \leqq A[j'+\delta+1], \cdots, A[N]$, where δ is either 0 or 1 depending on whether the condition $i = j$ occurs outside the inner loops of Program 1. In both programs, the file is partitioned on the value of $A[1]$. Call that value v and let s be the number of keys in the file which are $<v$. Our goal will be to show that the inequality

$$|j'+\delta-(N-1)/2| \leqq |s+k-(N+1)/2|,$$

holds for $j-s<k<i-s$.

First we notice that since Program 2 does not move keys which are $=v$, we can have $j = s+k$ only if exactly k of the keys $A[2], \cdots, A[s+k]$ were originally $=v$. But we also know that $A[j+1]$ was originally $<v$, and that $A[j+2], \cdots, A[i-1]$ were originally $=v$, since they were not moved by partitioning. In short, we can deduce that partitioning must have had the following effect:

before:	v	k keys $=v$		$<v$	v	$\cdots$	v		
after:	$<v$	k keys $=v$		v	v	$\cdots$	v		
			↑ j					↑ i	

In the original file, exactly $k-1$ of the keys $A[2], \cdots, A[s+k]$ were originally $=v$ for all k in the range $j-s<k<i-s$. Similarly, since Program 1 always moves keys which are $=v$, then $j' = s+k'$ for some fixed k' only if there were exactly $k'-1+\delta$ keys $=v$ in the last $N-j'+1$ positions of the reverse file: $A[N-s-k'+2], \cdots, A[2]$. The effect of partitioning when $\delta=1$ is

before:	v		$<v$	v	$k'-1$ keys $=v$
after:	$<v$	$k'-1$ keys $=v$	v	v	
			↑ j'		

and the diagram for $\delta=0$ is similar.

To complete the proof it is necessary to consider three cases depending on the relative values of k and $k'+\delta$. If $k=k'+\delta$, then the inequality $|j'+\delta-(N+1)/2| \leqq |s+k-(N+1)/2|$ obviously holds. If $k>k'+\delta$, then the discussion above says that $A[2], \cdots, A[s+k]$ has more keys $=v$ than $A[N-s-k'+2], \cdots, A[2]$. This can only be true if $s+k>N-s-k'+2$, or $j'+\delta \geqq N-s-k+1$. Now, if $j'+\delta-(N+1)/2$ is $\geqq 0$, then $k'+\delta<k$ implies that $0 \leqq j'+\delta-(N+1)/2 \leqq s+k-(N+1)/2$; and if $(N+1)/2-j'-\delta$ is $\geqq 0$, then $j'+\delta \geqq N-s-k+1$ implies that $0 \leqq (N+1)/2-j'-\delta \leqq s+k-(N+1)/2$. In either case, taking absolute values gives the desired result, $|j'+\delta-(N+1)/2| \leqq$

$|s+k-(N+1)/2|$. If $k<k'+\delta$, then $A[N-s-k'+2], \cdots, A[2]$ has more keys $=v$ than $A[2], \cdots, A[s+k]$. But also we know that $A[N-s-k'+2]<v$, so we must have $N-s-k'+1>s+k$, or $j'+\delta \leqq N-s-k+1$. An argument symmetric to the above shows that $|j'+\delta-(N+1)/2| \leqq |s+k-(N+1)/2|$, and we have shown that this inequality holds for all k in the range $j-s<k<i-s$.

The theorem follows immediately from this inequality. If the first partition is to the left of center $(i-1<(N+1)/2)$, then the second is at least as close $(|(N+1)/2-j'-\delta|<|(N+1)/2-i|)$; and the symmetric argument holds for the right. If the first partition straddles the center, or $j+1 \leqq (N+1)/2 \leqq i-1$, then $|s+k-(N+1)/2| \leqq \frac{1}{2}$ for some k, and therefore $|j'-\delta-(N+1)/2| \leqq \frac{1}{2}$, or the second partition must also be at the center. □

A direct consequence of Theorem 3 is that if the files $A[1], \cdots, A[N]$ and $A[1], A[N], \cdots, A[2]$ always appear with equal probability as input files (for example, if a random permutation of a multiset or a random n-ary file is being sorted), then Program 2 will produce a partition no closer to the center, on the average, than Program 1.

We cannot make quite the same statement when comparing Program 3 with Program 1. For example, when sorting a random permutation of the multiset $\{5 \cdot 1, 1 \cdot 2, 1 \cdot 3, 1 \cdot 4, 1 \cdot 5\}$, the programs will produce the same first partition when 2, 3, 4 or 5 is the partitioning element, but Program 3 will always partition in the center when 1 is the partitioning element, while Program 1 will not. Of course, any advantage gained in this case will be lost because Program 3 will be left with a large unary file for which it requires $O(N^2)$ comparisons. In addition, we can prove the following analogue to Theorem 3.

THEOREM 4. *Consider two files* $A[1], \cdots, A[N]$ *and* $A'[1], \cdots, A'[N]$ *satisfying* $1 \leqq A[i], A'[i] \leqq n$ *and* $A'[i]=n+1-A[i]$ *for* $1 \leqq i \leqq N$. *The average position of the partition when Program* 3 *operates on these files is no closer to the center than the average position of the partition when Program* 1 *operates on them.*

Proof. Suppose that in $A[1], \cdots, A[N]$ there are s keys$<A[1]$, t keys$=A[1]$, and u keys $>A[1]$, so $s+t+u=N$. Then we also know that in $A'[i], \cdots, A'[N]$ there are u keys$<$A'[1], t keys $=A'[1]$, and s keys$>A'[1]$. Since Program 3 puts keys equal to the partitioning element in the left subfile, it partitions $A[1], \cdots, A[N]$ at $s+t$ and it partitions $A'[1], \cdots, A'[N]$ at $u+t$. On the other hand, Program 1 puts $A[s+j_1]$ into place when partitioning $A[1], \cdots, A[N]$ and $A'[u+j_2]$ into place when partitioning $A'[1], \cdots, A'[N]$, where j_1 and j_2 are fixed between 1 and t.

We wish to show that

$$\left|s+t-\frac{N+1}{2}\right|+\left|u+t-\frac{N+1}{2}\right| \geqq \left|s+j_1-\frac{N+1}{2}\right|+\left|u+j_2-\frac{N+1}{2}\right|.$$

If $s+t<(N+1)/2$ and $u+t<(N+1)/2$, then, since $s+t+u=N$, we must have $t<1$ which is impossible. If $s+t$ and $u+t$ are both $\geqq(N+1)/2$, then we can remove the absolute value signs to get $2t \geqq j_1+j_2$, which clearly holds. If $s+t \geqq (N+1)/2$ and $u+t<(N+1)/2$, then the proof is more complex (and the case $s+t<(N+1)/2$ and $u+t \geqq (N+1)/2$ is clearly symmetric). If we also have $s+j_1<(N+1)/2$, then (since $u+t<(N+1)/2$ implies that $u+j_2<(N+1)/2$) we can remove absolute value signs in the inequality to get

$$s+t-\frac{N+1}{2}+\frac{N+1}{2}-u-t \geqq \frac{N+1}{2}-s-j_1+\frac{N+1}{2}-u-j_2$$

or

$$s-u>t-j_1-j_2.$$

But this inequality holds because $u+t<(N+1)/2$ and $u+t+s=N$ implies that $s+1>(N+1)/2$, so we have $u+t<(N+1)/2<s+1$, or $s-u \geqq t$. Finally, we must consider the case where $s+t \geqq (N+1)/2$, $u+t<(N+1)/2$, and $s+j_1 \geqq (N+1)/2$. Removing absolute values, our inequality reduces to

$$s-u \geqq s-u+j_1-j_2,$$

which holds unless $j_1>j_2$. Following the logic in the proof of Theorem 3, if we were to have $j_1>j_2$, this would imply that the number of keys equal to $A[1]$ in $A[s+j_1], \cdots, A[N]$ must exceed the number of keys equal to $A'[1]$ in $A'[u+j_2], \cdots, A'[N]$. Since our transformation between A and A' preserves equality among keys, this can only be true if $s+j_1<u+j_2$. But we know that $s>u$, since we have $s+t \geqq (N+1)/2>u+t$, so this implies that $j_1<j_2$, a contradiction. □

As above, we know from this theorem that if the files $A[1], \cdots, A[N]$ and $A'[1], \cdots, A'[N]$ appear with equal probability as input files (for example, if a random n-ary file or a random symmetrically distributed multiset is being sorted) then Program 3 will produce a partition no closer to the center, on the average, than Program 1.

These theorems, of course, do not represent complete evidence that the total average running time of Program 1 will always be lower than the total average running time of Programs 2 and 3. We have already seen anomalous cases where Program 1 may be slightly slower. Also, we should note that although Program 1 may use extra exchanges to get the partition close to the center, this is more than compensated for by the effects of having the partition more balanced. When the partition is closer to the center, all aspects of total running time are improved, and Theorems 3 and 4 are strong general results.

9. Conclusion. The evidence in favor of stopping the scanning pointers on keys equal to the partitioning element in Quicksort is overwhelming. Theorems 1 and 2 and our analyses of the operation of the programs on unary and binary files show that this method will always require $O(N \log N)$ comparisons on the average, when other methods can be quadratic. Theorems 3 and 4 indicate that it will produce more balanced partitions, on the average, than other methods for most reasonable input distributions. Furthermore, it is easier to implement.

Before it can be recommended for use in a practical situation, three major improvements (fully described in [9] and [14]) must be applied to Program 1. First, the recursion should be removed, and the smaller subfile sorted first. This removes some overhead, and ensures that only limited extra space will be necessary to implement the recursive stack. It applies to all our programs, and has little effect on our results. Second, the partitioning element should be chosen by taking the median of the first, middle, and last element of the file. This not only tends to balance the partitions and so reduce the running time, but also it makes the worst case less likely to occur in a real file, an important practical consideration. This is another advantage of Program 1 over Programs 2 and 3 because even with equal keys present, it is unlikely that a file for which Program 1 will perform really badly will arise in practice, while the others run badly for *any* file with a small number of distinct values. Third, small subfiles should be ignored during partitioning and a single insertion sort applied to the entire file afterwards. This eliminates a considerable amount of overhead, since Quicksort is inefficient on files of about ten elements or less. Its effect on our analyses is to reduce the significance of the differences between the programs, since the anomalies created by small files are removed.

The utility of a general-purpose sorting program may be measured by the range of input files over which it performs efficiently. Quicksort has been shown to be more efficient than most other sorting algorithms for files with distinct keys, but few sorting algorithms have been studied in the case when equal keys are present. The results of this paper demonstrate that the range of files over which a properly implemented Quicksort can run efficiently may be extended to include files with equal keys.

REFERENCES

[1] A. AHO, J. HOPCROFT AND J. ULLMAN, *The Design and Analysis of Computing Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] J. BOOTHROYD, *Sort of a section of the elements of an array by determining the rank of each element* (*Algorithm* 25), Computer J., 10 (1967), pp. 308–310. (See notes by R. Scowen, Computer J., 12 (1969), pp. 408–409, and by A. Woodall, Computer J., 13 (1970), pp. 295–296.)

[3] J. DONNER, *Tree acceptors and some of their applications*, J. Comput. System Sci., 4 (1970), J. Assoc. Comput. Mach., 23(1976), pp. 451–454.

[4] E. DIJKSTRA, *EWD*316: *A short introduction to the art of programming*, Technical University Eindhoven, The Netherlands, 1971.

[5] T. HIBBARD, *Some combinatorial properties of certain trees with applications to searching and sorting*, J. Assoc. Comput. Mach., 9 (1962), pp. 13–18.

[6] C. A. R. HOARE, *Partition* (*Algorithm* 63), *Quicksort* (*Algorithm* 64), *and Find* (*Algorithm* 65), Comm. ACM, 4 (1961), pp. 321–322. (See also certification by J. Hillmore, Comm. ACM, 5 (1962), p. 439, and by B. Randell and L. Russell, Comm. ACM, 6 (1963), p. 446.)

[7] ———, *Quicksort*, Computer J., 5 (1962), pp. 10–15.

[8] D. KNUTH, *Fundamental Algorithms*, The Art of Computer Programming 1, Addison-Wesley, Reading, MA, 1968.

[9] ———, *Sorting and Searching*, The Art of Computer Programming 3, Addison-Wesley, Reading, MA, 1972.

[10] ———, *Structured programming with go to statements*, Comput. Surveys, 6 (1974), pp. 261–301.

[11] R. RICH, *Internal Sorting Methods Illustrated with PL/I Programs*, Prentice-Hall, Englewood Cliffs, NJ, 1972.

[12] R. RIVEST, *A fast stable minimum-storage sorting algorithm*, Institut de Recherche d'Informatique et d'Automatique Rapport 43, 1973.

[13] R. SCOWEN, *Quickersort* (*Algorithm* 271), Comm. ACM, 8 (1965), pp. 669–670. (See also certification by C. Blair, Comm. ACM, 9 (1966), p. 354.)

[14] R. SEDGEWICK, *Quicksort*, Ph.D. thesis, Stanford Univ., Stanford, CA, 1975. (Also Stanford Computer Science Rep. STAN-CS-75-492.)

[15] R. SINGLETON, *An efficient algorithm for sorting with minimal storage* (*Algorithm* 347), Comm. ACM, 12 (1969), pp. 185–187. (See also remarks by R. Griffin and K. Redish, Comm. ACM, 13 (1970), p. 54, and by R. Peto, Comm. ACM, 13 (1970), p. 624.)

SIAM J. COMPUT.
Vol. 6, No. 2, June 1977

SOME REMARKS ON p-WAY MERGING*

JANET FABRI†

Abstract. A p-way merge of m items is usually implemented with sentinel keys of ∞ appended to the end of each input list, resulting in $m(p-1)$ total comparisons. In this note, we analyze the number of comparisons performed when sentinel keys are not used. It is shown that the improvement in this case is very small.

Key words. merging, multi-way merging, p-way merging, sorting methods, algorithmic analysis, multinomial coefficients, random variables

1. Introduction. A p-way merge of m items is usually implemented with sentinel keys of ∞ appended to the end of each input list, resulting in $m(p-1)$ comparisons. In this note, we present an algorithm which uses an indirection mechanism to address lists that have not yet been exhausted. The kth list is eliminated from the comparison set as soon as it is exhausted, thus gradually reducing the number of comparisons to be performed on each pass of the merge. The number of comparisons performed by this algorithm will be analyzed,[1] under the assumption that the implementation overhead of the indirection mechanism is negligible.

2. The algorithm. Suppose that we are given p files of records:

$$F_1: R_1(1), R_1(2), \cdots, R_1(m_1),$$
$$\vdots$$
$$F_k: R_k(1), R_k(2), \cdots, R_k(m_k),$$
$$\vdots$$
$$F_p: R_p(1), R_p(2), \cdots, R_p(m_p),$$

where m_k is the number of records in the kth file. To each record $R_i(j)$ associate a *key* $K_i(j)$, and assume that the p files have been sorted on these keys:

$$K_i(j) \leqq K_i(j+1), \quad 1 \leqq j < m_i, \quad 1 \leqq i \leqq p.$$

We shall merge these p files into an output file $Z = (z_1, z_2, \cdots, z_m)$, where $m = m_1 + m_2 + \cdots + m_p$, using a list $T = (t_1, t_2, \cdots, t_L)$ to keep track of the indices of those files F_i that have not yet been exhausted in the merging process. L is the current length of T, $1 \leqq L \leqq p$.

```
ALGORITHM.
A1:  $K \leftarrow 1$
     $T \leftarrow (1, 2, \cdots, p)$
     $J_i \leftarrow 1$, for $i = 1, \cdots, p$
     $L \leftarrow p$
```

* Received by the editors March 29, 1976, and in revised form July, 1976.

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

[1] This analysis may be regarded as a generalization of Exercise 5.2.4-2 in [2, p. 168].

```
A2:  $s \leftarrow 1$
     $V \leftarrow K_{T_1}(J_{T_1})$
     $i \leftarrow 2$
A3:  **Compare** $K_{T_i}(J_{T_i})$ **with** $V$:
         **If** $K_{T_i}(J_{T_i}) < V$
             **then** $s \leftarrow i$; $V \leftarrow K_{T_i}(J_{T_i})$
A4:  $i \leftarrow i+1$
     **If** $i \leqq L$ **then goto** A3
A5:  $Z_k \leftarrow R_{T_s}(J_{T_s})$
     $k \leftarrow k+1$
     $J_{T_s} \leftarrow J_{T_s}+1$
     **If** $J_{T_s} \leqq m_{T_s}$ **then goto** A2
A6:  $T \leftarrow (T_1, \cdots, T_{s-1}, T_{s+1}, \cdots, T_L)$
     $L \leftarrow L-1$
     **If** $L > 1$ **then goto** A2
A7:  $Z \leftarrow (Z_1, \cdots, Z_{k-1}, R_{T_1}(J_{T_1}), \cdots, R_{T_1}(m_{T_1}))$
     END
```

3. Analysis. We will assume that the keys $\{K_i(j)\}$ are distinct and $1 \leqq K_i(j) \leqq m$, since only their relative order is relevant. We shall also assume that all ways of dividing a collection of m records into p files of sizes $m_1, \cdots, m_p$ are equally likely. The number of such distinct subdivisions is given by the multinomial coefficient

$$C(m:m_1, \cdots, m_p) = m!/(m_1!m_2! \cdots m_p!).$$

We shall assume that each of the $C(m:m_1, \cdots, m_p)$ arrays

$$\{K_i(j): 1 \leqq j \leqq m_i,\ 1 \leqq i \leqq p\}$$

is equally likely.

We shall say that the files are *exhausted in the order* $\pi = (\pi_1, \cdots, \pi_p)$, where π is a permutation of $(1, \cdots, p)$, if

$$K_{\pi_i}(m_{\pi_i}) < K_{\pi_{i+1}}(m_{\pi_{i+1}}), \quad \text{for } 1 \leqq i < p$$

Let

$$\underset{\sim}{K} = \{K_i(j): 1 \leqq j \leqq m_i,\ 1 \leqq i \leqq p\}$$

be the random variable, taking on values in the set of $C(m:m_1, \cdots, m_p)$ partitions of the set $\{1, 2, \cdots, m\}$ into p subsets of sizes $m_1, \cdots, m_p$. Let Pr be the uniform probability measure over this set of partitions. Each value of $\underset{\sim}{K}$ determines a permutation π, establishing the order in which the p files are exhausted in the merging process. Now, define the $(p-1)$ random variables L_i by $L_i = K_{\pi_i}(m_{\pi_i})$ for $1 \leqq i < p$. The value of L_i is the length of the output file Z when the last record $R_{\pi_i}(m_{\pi_i})$ of file F_{π_i} has been appended to Z, thereby exhausting F_{π_i}. Observe that on the kth pass, when the record to output next is being determined, the number of comparisons required is $(p-i)$, where $L_{i-1} < k \leqq L_i$. Thus, the total number of comparisons made in step A3, of the algorithm is $L_1 + \cdots + L_{p-1}$.

Let

$$\psi(u_1, \cdots, u_{p-1}: m_{\pi_1}, \cdots, m_{\pi_p}) = \Pr\{L_1 = u_1, \cdots, L_{p-1} = u_{p-1} \textit{ and}$$
$$\text{the files are exhausted in the order } \pi\}.$$

In what follows we will take π to be the identity permutation. Observe that

$$\psi(u_1, \cdots, u_{p-1}\colon m_1, \cdots, m_p)$$

$$= \prod_{1 \leqq i < p} C(u_i - 1 - m_1 - \cdots - m_{i-1} : m_i - 1)/C(m : m_1, \cdots, m_p).$$

The variables $\{u_i\}$ satisfy the conditions

$$M_i \leqq u_i < u_{i+1} \qquad \text{for } 1 \leqq i < p-1,$$

$$M_{p-1} \leqq u_{p-1} < m$$

where $M_i = m_1 + \cdots + m_i$.

We will make use of the following identities:

$$\text{(1)} \qquad \sum_{n \leqq k < N} C(k-1 : n-1) = C(N-1 : n),$$

$$\text{(2)} \qquad \sum_{n \leqq k < N} kC(k-1 : n-1) = (nN/(n+1))C(N-1 : n).$$

From (1) and (2) it follows that

$$\text{(3)} \qquad \sum_{m_1 \leqq u_1 \leqq u_2 - 1} \psi(u_1, \cdots, u_{p-1} : m_1, \cdots, m_p)$$

$$= (m_2/(m_1 + m_2))\psi(u_2, \cdots, u_{p-1} : M_2, m_3, \cdots, m_p),$$

$$\text{(4)} \qquad \sum_{m_1 \leqq u_1 \leqq u_2 - 1} u_1\psi(u_1, \cdots, u_{p-1} : m_1, \cdots, m_p)$$

$$= (u_2 m_1 m_2/M_2(m_1 + 1))\psi(u_2, \cdots, u_{p-1} : M_2, m_3, \cdots, m_p).$$

From equation (3), we obtain, inductively

$$\text{(5)} \qquad \sum_{M_j \leqq u_j \leqq u_{j+1} - 1} \cdots \sum_{M_1 \leqq u_1 \leqq u_2 - 1} \psi(u_1, \cdots, u_{p-1} : m_1, \cdots, m_p)$$

$$= \left[\prod_{1 \leqq i \leqq j+1} (m_i/M_i) \right] \psi(u_{j+1}, \cdots, u_{p-1}\colon M_{j+1}, m_{j+2}, \cdots, m_p),$$

$$\text{(6)} \qquad \sum_{M_{p-1} \leqq u_{p-1} \leqq m-1} \cdots \sum_{M_1 \leqq u_1 \leqq u_{2-1}} \psi(u_1, \cdots, u_{p-1} : m_1, \cdots, m_p) = \prod_{1 \leqq i \leqq p} (m_i/M_i).$$

Next, from equations (4) and (5) we obtain

$$\text{(7)} \qquad \sum_{M_j \leqq u_j \leqq u_{j+1} - 1} \cdots \sum_{M_1 \leqq u_1 \leqq u_2 - 1} u_j\psi(u_1, \cdots, u_{p-1} : m_1, \cdots, m_p)$$

$$= (M_j/M_j + 1)\left[\prod_{1 \leqq i \leqq j+1} m_i/M_i \right] u_{j+1}\psi(u_{j+1}, \cdots, u_{p-1} : M_{j+1}, m_{j+2}, \cdots, m_p),$$

$$\text{(8)} \qquad \sum_{M_{p-1} \leqq u_{p-1} \leqq m-1} \cdots \sum_{M_j \leqq u_j \leqq u_{j+1} - 1} u_j\psi(u_j, \cdots, u_{p-1} : M_j, m_{j+1}, \cdots, m_p)$$

$$= M_j \prod_{j+1 \leqq i \leqq p} m_i/(M_{i-1} + 1).$$

Equations (7) and (8) yield:

$$\sum_{M_{p-1}\leqq u_{p-1}\leqq m-1} \cdots \sum_{M_1\leqq u_1\leqq u_2-1} (u_1+\cdots+u_{p-1})\psi(u_1,\cdots,u_{p-1}: m_1,\cdots,m_p)$$

$$\text{(9)} \qquad = \left[m \prod_{1\leqq i\leqq p} m_i/M_i\right] \sum_{1\leqq j\leqq p-1} \prod_{j\leqq i\leqq p-1} M_i/(1+M_i).$$

Setting $M_{\pi_i} = m_{\pi_1}+\cdots+m_{\pi_i}$, we obtain the following expression for the expected number of comparisons, E:

$$\text{(10)} \qquad m\left[\prod_{1\leqq i\leqq p} m_i\right] \sum_{\pi} \prod_{1\leqq i\leqq p} (M_{\pi_i})^{-1} \sum_{1\leqq j\leqq p-1} \left[\prod_{j\leqq k\leqq p-1} M_{\pi_k}/(1+M_{\pi_k})\right].$$

In the case where all $\{m_i\}$ are equal, we can evaluate E. Let $m_i = m/p$, $1\leqq i\leqq p$. Then (10) reduces to:

$$\text{(11)} \qquad E = m\cdot \sum_{j=1}^{p-1} \prod_{k=j}^{p-1} \frac{mk}{mk+p}.$$

It can be shown easily by induction on p that, for any x:

$$\text{(12)} \qquad \sum_{j=1}^{p-1} \prod_{k=j}^{p-1} \frac{mk}{mk+x} = \frac{m(p-1)}{m+x} \quad \text{for } p\geqq 1.$$

Hence we obtain:

$$\text{(13)} \qquad E = \frac{m^2(p-1)}{m+p}.$$

Comparing this with the sentinel key implementation, we see that the latter performs an expected number of $mp(p-1)/(m+p)$ "unnecessary" comparisons out of the total $m(p-1)$ comparisons, thus "wasting" $p/(m+p)$ of its total effort. This fraction is close to zero for p small. For larger values of p, the selection tree method of [2, § 5.4.1] gives better performance ($m \log p$ comparisons).

Thus it appears that in the case of equal file sizes (the case of most likely practical interest), the p-way merge without sentinel keys provides no clear-cut performance improvement over other methods.

Acknowledgment. I would like to end this note by expressing my appreciation to Professor Alan Konheim for helping me formulate this problem.

REFERENCES

[1] D. E. KNUTH, *The Art of Computer Programming, Volume* 1, *Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1968.

[2] ———, *The Art of Computer Programming, Volume* 3, *Sorting and Searching*, Addison-Wesley, Reading, MA, 1970.

SIAM J. COMPUT.
Vol. 6, No. 2, June 1977

CONSTRAINTS ON STRUCTURAL DESCRIPTIONS: LOCAL TRANSFORMATIONS*

ARAVIND K. JOSHI† AND LEON S. LEVY‡

Abstract. It is very often more convenient and more meaningful to specify a set of structural descriptions analytically rather than generatively, i.e., by specifying a set of constraints each structured description in the set has to satisfy. Peters and Ritchie [7] have shown that if context-sensitive rules are used only for "analysis" then the string language of the set of trees is still context-free. In this paper, we have generalized this result by considering context-free rules constrained by Boolean combinations of proper analysis predicates and domination predicates. These rules, called "local transformations" not only make precise an informal and briefly discussed notion of Chomsky [2], but also, generalize it in an appropriate manner. It is shown that the Peters–Ritchie result can be generalized to local transformations. Linguistic relevance of this result has been also briefly discussed. Results in this paper are relevant to the following situation: Patterns of a class, say A, may be difficult to characterize generatively; but, it may be possible to specify a suitable (nontrivial) augmentation, say B, of the class A, such that B can be easily characterized generatively, and then A is characterized by stating some restrictions on the class B. This suggests possible applications to programming languages and pattern description languages.

Key words. constraints on structural descriptions, immediate constituent analysis, local sets and recognizable sets, local transformations, proper analysis and domination predicates, rules for generation and for analysis, structural descriptions, transformational grammars, tree automata

1. Introduction. When structural descriptions are in the form of labeled trees (e.g., phrase structure trees) it is very often more convenient, and in fact, more meaningful to specify a set of structural descriptions analytically rather than generatively, i.e., by specifying a set of constraints each tree in the set must satisfy. Such descriptions are useful for specification of transformational grammars. Peters and Ritchie [7] have shown that if context-sensitive rules are used for "analysis" only i.e., for specifying constraints on a set of trees, then these rules, in effect, have no more power than context-free rules (more precisely, the string language of the set of trees is a context-free language). In this paper, we have generalized this result by considering context-free rules constrained by Boolean combinations of proper analysis predicates and domination predicates. We call these "local transformations." The main result is that local transformations when used for "analysis" do not have any more power than context-free rules (Theorem 3.1).

First, we will briefly discuss the result of Peters and Ritchie in § 2. Local transformations will be defined in § 3 and the main result concerning local transformations will be stated without proof. In § 4, we will give a proof of Theorem 3.1, our main result, which follows from Theorem 4.8. We have

* Received by the editors April 22, 1975, and in final revised form September 21, 1976.

† Department of Computer and Information Science, The Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia, Pennsylvania 19174. Partially supported by the National Science Foundation under Grants SOC 72-05465A01 and MCS 76-19466. Part of the work was carried out while the author was at The Institute for Advanced Study, Princeton, New Jersey, under a Guggenheim Fellowship.

‡ Department of Computer Science and Statistics, University of Delaware, Newark, Delaware 19711.

generalized some finite state string automata results in an obvious manner to tree automata; these results together imply some further generalizations of the Peters–Ritchie result. We will also talk about recognizable sets in terms of features of nodes. We have deliberately taken this leisurely approach because we believe that the discussion in § 4 gives some additional insights into the structure of recognizable sets.

Before proceeding with our formal results, we will briefly state their linguistic relevance.

1. Context-sensitive rules are used by linguists in many ways, particularly in strict subcategorization and in lexical insertion (e.g., $V \rightarrow V_{\text{singular}}/NP_{\text{singular}}$— i.e., V (verb) is rewritten as V_{singular} in the left-context of a singular noun-phrase (NP)). These rules are indeed used "analytically" and hence, the relevance of the Peters–Ritchie result.

2. Chomsky [2, p. 215] has discussed, very briefly and informally, a notion of a local transformation in the context of rules which affect only a substring dominated by a single category. For example, the restrictions on the choice of elements in an *Adverbial Phrase* consisting of *Preposition* $^\frown$ *Determiner* $^\frown$ *Noun**, and certain nominalizing transformations could be stated as local transformations. In fact, Chomsky [2, p. 215] speculates that an extension of the theory of context free grammars may be possible by restricting rewriting by local transformations.

3. Our notion of "local transformations" not only captures and makes precise Chomsky's notion mentioned above, but also generalizes it in an appropriate manner. Our main result concerning local transformations is that if used only for "analysis" then these rules also have no more power than context-free rules. Thus, by incorporating some local transformations in the base component of a transformational generative grammar, we do not increase the weak generative power of the base component.

Adopting the linguistic term "immediate constituent analysis" for parsing, we might call the analysis obtained by using local transformations, a "local transformational immediate constituent analysis".

The applications of such a specification of constraints in areas other than linguistics are worth exploring. Results in this paper are relevant to the following situation: Patterns of a class, say A, may be difficult to characterize generatively; but, it may be possible to specify a suitable (nontrivial) augmentation, say B, of the class A, such that B can be easily characterized generatively, and then A is characterized by stating some restrictions on the class B. This suggests applications to programming languages and pattern description languages.

Extension of our results to specification of constraints on structural descriptions which are not trees (e.g., graphs) is also a possibility.

2. Context-sensitive rules for analysis. Context-sensitive grammars, in general, are more powerful (with respect to weak generative capacity) than context-free grammars. A fascinating result of Peters and Ritchie [7] is that if a context-sensitive grammar G is used for "analysis" then the language "analyzed" by G is context-free. This result has significance in transformational linguistics because

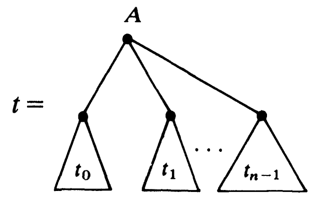
* $\frown$ denotes concatenation.

the context-sensitive rules in the base component of a phrase structure transformational grammar are used essentially in this manner.

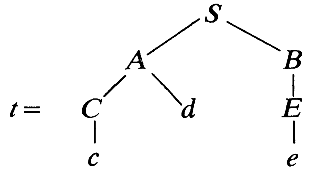
First, we will describe what we mean by a context-sensitive grammar, G, used for "analysis". Given a tree t, we define a set of "proper analyses" of t. Roughly speaking, a proper analysis of a tree is a "slice" across the tree with the condition that if a node is included in the slice then all its siblings are included; if a sibling is not included then all its siblings are included and so on. More precisely, the set of proper analyses of t, $P(t)$ is defined as follows:

(i) If $t = \cdot a$ (i.e., a single node labeled a) then $P(t) = \{a\}$

(ii) If



then $P(t) = \{A\} \cup P(t_0) \cdot P(t_1) \cdot \cdots \cdot P(t_{n-1})$ where $t_0, t_1, \cdots, t_{n-1}$ are trees and $\cdot$ denotes set product. Let t be a tree as shown below:



The set of proper analyses of t, $P(t)$, is

$$P(t) = \{S, AB, AE, Ae, CdB, CdE, Cde, cdB, cdE, cde\}.$$

Let G be a context sensitive grammar, i.e., its rules are of the form

$$A \to \omega | \phi \_\_ \psi$$

where $A \in V - \Sigma$ (V is the alphabet and Σ is the set of terminal symbols), $\omega \in V^{\dagger}$ (set of nonnull strings on V), and $\phi, \Psi \in V^*$ (set of all strings on V). If ϕ and Ψ are both null then the rule is a context-free rule. A tree t said to be "analyzable" with respect to G is for each node of t, some rule of G "holds". It is obvious how to check whether a context-free rule holds of a node or not. A context sensitive rule $A \to \omega | \phi \_\_ \Psi$ holds of a node labeled A, if the string corresponding to the immediate descendants of that node is ω, and there is a proper analysis of t of the form $\rho_1 \phi A \Psi \rho_2$ which "passes through" the node ($\rho_1, \rho_2 \in V^*$). Let $\tau(G)$ be the set of all trees analyzable by G, and let $L(\tau(G))$ be the string language corresponding to the terminal strings of trees in $\tau(G)$. Then the result of Peters and Ritchie mentioned above can be stated as follows.

THEOREM 2.1. *Let G be a context-sensitive grammar. The language analyzed by G, i.e., $L(\tau(G))$, is context-free.*

3. Local transformations. Since we will now be interested in using context-sensitive rule for analysis only, let us call the contextual condition $\phi \_\_ \Psi$ in a rule

$A \rightarrow \omega | \phi \_\_ \Psi$, a "proper analysis predicate." Let G be a finite set of rules of the form

$$A \rightarrow \omega | P_A,$$

where P_A is a Boolean combination of proper analysis predicates. The subscript A indicates that the proper analyses have to pass through a node labeled A. The following is an example of such a rule:

$$A \rightarrow \omega | (\phi_1 \_\_ \Psi_1) \wedge ((\phi_2 \_\_ \Psi_2) \vee (\neg (\phi_3 \_\_ \Psi_3))).$$

Thus, the above rule holds of a node labeled A in a tree t if the string corresponding to the immediate descendants of A is ω, and a proper analysis of the form $\rho_1 \phi_1 A \Psi_1 \rho_2$ passes through A, and either a proper analysis of the form $\rho_3 \phi_2 A \Psi_2 \rho_4$ passes through A or no proper analysis of the form $\rho_5 \phi_3 A \Psi_3 \rho_6$ passes through A. Clearly, such rules allow us to state quite complex contextual conditions. Let $\tau(G)$ be the set of trees "analyzable" by G, where G is a finite set of rules of the type mentioned above. Let $L(\tau(G))$ be the string language of G. Then we show that $L(\tau(G))$ is still context-free. This result holds even when the right and left contexts are not necessarily immediately to the right or left respectively. The right (or left) context may even be "scattered" on the right (or left).

A context sensitive rule allows us to specify context on the "right" and "left". Often, we need to specify context on the "top" (or perhaps even at the "bottom"). Given a node labeled A in a tree t, we will say that $\delta(A, B)$ holds of the node labeled A if a node labeled B immediately dominates the node labeled A, i.e., B is the immediate ancestor of A. We will call δ a "domination predicate". The most general form of a domination predicate we will allow is

$$\delta(A, \phi \_\_ \Psi), \qquad \phi, \Psi \in V^*.$$

δ holds of a node labeled A if there is a path from the root of the tree to some terminal node, which passes through this node (labeled A), and is of the form

$$\rho_1 \phi A \Psi \rho_2, \qquad \rho_1, \rho_2 \in V^*.$$

We might call such a path a "vertical proper analysis" passing through A. The top and bottom contexts (i.e., ϕ and Ψ) may even be "scattered". It is clear that this generalization allows us to state conditions involving immediate or nonimmediate domination as well as a sequence of dominations. Now let G be a finite set of rules of the form

$$A \rightarrow \omega | D_A,$$

where D_A is a Boolean combination of domination predicates (generalized) mentioned above. Let $\tau(G)$ be the set of trees analyzable by G. We also show that $L(\tau(G))$, the language corresponding to the terminal strings of trees in t, is context-free.

We are now ready to define a "local transformation". A local transformation is a rule of the form

$$A \rightarrow \omega | C_A,$$

where C_A is a Boolean combination of proper analysis predicates and domination predicates. Conditions on transformations are usually stated in terms of such predicates and their Boolean combinations. Hence, we call these rules "transformations". Since the string ω is dominated by a single category, we use the modifier "local". Our main result can now be stated as follows.

THEOREM 3.1. *Let G be a finite set of local transformations and $\tau(G)$, the set of trees analyzable by G. Then the string language $L(\tau(G))$ is context-free.*

Example 3.1. Let $V=\{S, T, a, b, c, e\}$ and $\Sigma=\{a, b, c, e\}$. Let G be a finite set of local transformations:

1. $S\rightarrow e$,
2. $S\rightarrow aT$,
3. $T\rightarrow aS$,
4. $S\rightarrow bTc/(a\_\_)\wedge\delta(S, T)$,
5. $T\rightarrow bSc/(a\_\_)\wedge\delta(T, S)$.

In rules 1, 2, and 3 the context is null, i.e., these are context-free rules. In rule 4, S in $\delta(S, T)$ refers to the label of the node at which the condition is to be checked and the condition is that this occurrence of S is dominated (immediately) by a node labeled T. In rule 5, $\delta(T, S)$ refers to a node labeled T and this node is to be dominated (immediately) by a node labeled S.

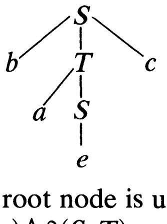
The language generated by this example can be derived by the context-free grammar:

$$\begin{array}{llll} G'\colon & S\rightarrow e, & S\rightarrow aT_1, & S_1\rightarrow aT_1, \\ & S\rightarrow aT, & T\rightarrow aS_1, & T_1\rightarrow aS_1. \\ & T\rightarrow aS, & T_1\rightarrow bSc, & \\ & S_1\rightarrow bTc, & & \end{array}$$

In G' there are additional variables S_1 and T_1 which enable the context checking of the local transformations, G, in the generation process.

It is easy to see that under the homomorphism which removes the subscripts on the variable T and S, each tree generable in G' is analyzable in G. Also each tree analyzable in G has a homomorphic pre-image in G'.

However, the tree



is not analyzable in G, since the root node is used in rule $S\rightarrow bTc$, but does not satisfy the analysis condition $(a\_\_)\wedge\delta(S, T)$.

Example 3.2 (Due to Kang Yueh). Let $V=\{E, +, *, [\,,], a\}$ and $\Sigma=\{+, *, [,], a\}$. Let G be a finite set of local transformations:

1. $E\rightarrow E+E/\neg(+\text{——})\wedge\neg(*\text{——})\wedge\neg(\text{——}*)$
2. $E\rightarrow E*E/\neg(*\text{——})$
3. $E\rightarrow[E]$
4. $E\rightarrow a$

G gives unambiguous structural descriptions, in accordance with the precedence relationships, for the arithmetic expressions.

4. Proof of Theorem 3.1. We will take a somewhat leisurely approach for the reasons indicated in § 1. Let us start with a finite state string automaton. Let Σ be the input alphabet, Q the finite set of states, and M the transition function, $M:\Sigma \times Q \to Q$.

There are some properties of strings which a finite automaton can check, and we shall call these *regular features.* A feature is checked by a finite automaton if there is a two block partition of the state set of the automaton, $\pi = \{\pi_f, \pi_{\bar{f}}\}$, such that for all strings possessing f, the automaton after processing, is in a state in π_f, and for all strings not possessing f the state of the automaton after processing, is in $\pi_{\bar{f}}$.[1]

We write $f(x)$ to denote the fact that string x has feature f, and $F = \{x \mid f(x)\}$. F is the (regular) set of strings possessing feature f. If f_1 and f_2 are regular features then $F_1 \cap F_2$ is the set of strings possessing both f_1 and f_2, and $F_1 \cup F_2$ is the set of strings having f_1 or f_2.

THEOREM 4.1 (Kleene). *Let f_1 and f_2 be any regular features; then $F_1, \bar{F}_1, F_1 \cap F_2$, and $F_1 \cup F_2$ are regular.*

COROLLARY 4.1. *Let $g_1 = B(\{f_1\})$ be any Boolean combination of a finite set of regular features; then G_1 is a regular set, and g_1 is a regular feature.*

By a Boolean combination of sets we mean the set of sets formed from the finite set by complementation, union, and intersection. By a Boolean combination of features, we mean the features $e_1(x) \wedge e_2(x)$, $e_1(x) \vee e_2(x)$, and $\neg e(x)$ where e, e_1, e_2 are in the given finite set of features or are Boolean combinations of features.

COROLLARY 4.2. *Let $g_2 = (f_1 \supset f_2)$. Then if f_1 and f_2 are regular, G_2 is a regular set, and g_2 is a regular feature.*

We may describe G_2 as the set of strings which if they possess the feature f_1 must also possess f_2.

We note that while the features are defined for strings, an automaton doing the computation of features of a particular string processes the string from left to right and is actually computing the features at each step of the processing. This motivates the following definition:

DEFINITION 4.1. If $x \in F$, we write $f(x)$, making explicit the fact that features are predicates. Let Prefix $(x) = \{y \mid (\exists z)\ (yz = x)\}$; Prefix (x) is the set of prefixes of the string x. If f is a regular feature, the following are defined for strings:

(i) $g(x) \Leftrightarrow ((\forall y \in \text{Prefix }(x)) f(y))$, i.e. every initial substring possesses f.

(ii) $h(x) \Leftrightarrow ((\exists\ y \in \text{Prefix }(x)) f(y))$, i.e. some initial substring possesses f.

(iii) $j(x) \Leftrightarrow ((\exists ! y \in \text{Prefix }(x)) f(y))$, i.e. exactly one initial substring possesses f.

[1] The reader may find it helpful to keep the following feature in mind while reading the discussion on strings:

$$f(x) = x \text{ has a substring } 01011$$

We then have the following:

THEOREM 4.2. *Let f be a regular feature; then g, h, j as defined above, are regular features.*

COROLLARY 4.3. *Any first order predicate expressed in terms of regular features of strings or their prefixes is a regular feature.*

A formulation of these results which is more directly applicable to trees is as follows:

DEFINITION 4.2. *A string domain* $S \subset 1^*$ is a finite set of elements such that if $x1$ is in S, then x is in S. Further, we map S onto $\{0, 1, \cdots, N-1\}$ by the homomorphism $h(\lambda) = 0$, $h(x, y) = h(x) + h(y)$. A *string* is a mapping from S to Σ.

We imagine the symbols of the string being numbered from $N-1$ to 0 going from left to right.[2] Hence, we can talk about the automaton being in position n of string x, and x has feature f at n, i.e., $f(x, n) \Leftrightarrow$ the state of the automaton is in π_f, when the automaton is in position n of string x. (0 denotes the position of the head when processing is complete, so that $f(x, 0) \Leftrightarrow f(x)$; and $N = \text{length}(x)$ denotes the initial position of the head.)

Accordingly, we rewrite the features g, h, and j, as follows:

$$\begin{aligned} &g(x) \Leftrightarrow ((\forall n \leqq \text{length } (x)) f(x, n)),\\ &h(x) \Leftrightarrow ((\exists n \leqq \text{length } (x)) f(x, n)),\\ &j(x) \Leftrightarrow ((\exists ! n \leqq \text{length } (x)) f(x, n)). \end{aligned}$$

We consider next the case of an automaton checking multiple features on a string, in particular those features which once identified in a prefix are features of the entire string. We call these *persistent features.* (An example of a persistent regular feature is the occurrence of some fixed substring. An example of a nonpersistent regular feature is the evenness of the number of occurrences of some fixed letter.) We want to determine the relative positions at which two persistent features, f_1 and f_2, occur. (In the tree case this will correspond to computing domination predicates.)

Informally, we can visualize two automata A_1 and A_2, A_1 checking f_1, and A_2 checking f_2. We first let A_1 process the input string going from left to right and record on each square adjacent to the original symbol in the square the state A_1 was in on that square. This sequence of states of A_1 is called a *state string.* A_2 can then be readily modified to process the combined original data and state string data to determine whether f_1 occurs earlier than f_2.

DEFINITION 4.3. A *finite relabeling transformation,* ρ, induced by an automaton A is a mapping from strings to strings defined by

$$\begin{aligned} &\rho(\lambda) = \lambda,\\ &\rho(\sigma) = M(\sigma, q_0) \quad \text{for } \sigma \in \Sigma,\\ &\rho(x\sigma) = \rho(x) M(x\sigma, q_0) \quad \text{for } x \in \Sigma^*, \quad \sigma \in \Sigma. \end{aligned}$$

[2] This numbering is chosen because of our string automata processes from left-to-right; it will later be generalized to tree automata going from the bottom of a tree, depth N, to the root, depth 0.

In a finite relabeling transformation, each string is mapped into the sequence of states which occurs in its processing. If F is the set of string having feature f, the

$$\rho(F)=\{y(\exists x \in F)(y=\rho(x))\}.$$

$\rho(F)$ is thus the image of F under the finite relabeling transformation, ρ, induced by A. $\rho(F)$ is a set of state strings.

Again, anticipating the tree automata case and considering strings as mappings from string domains to finite alphabets, let $q(x, n)$ denote the state of the automaton at position n of string x. Then $q(x, N)=q_0$ and $M(q_0, x)=q(x, 0)$; and the finite relabeling transformation generates the string $\rho(x)$.

Consider next the case of two automata A_1 and A_2 scanning the string x from left to right, where the input sequence to A_2 is the state sequence of A_1, as in the usual cascade construction. Then the output sequence from A_2 is just the composition of finite relabeling transformations. Since the cascade construction of A_1 and A_2 is just another finite state machine, this shows that finite relabeling transformations are closed under composition. Finite relabeling transformations are just a convenient way of sequentially decomposing the cascade connection of finite state machines.

We now apply the finite relabeling transformations to get an alternative development of Corollary 4.3.

DEFINITION 4.4. Let $\rho_1(x)$ and $\rho_2(x)$ be state strings induced by A_1 and A_2, respectively, and let $\rho_i(x, n)$ be the nth element of $\rho_i(x)$. Then $(\rho_1(x), \rho_2(x))$ denotes the string whose nth element is $(\rho_1(x, n), \rho_2(x, n))$.

THEOREM 4.3. *Let f_1 and f_2 be regular features, and g_1 and g_2 defined as follows*:

$$g_1(x, n) \Leftrightarrow (f_1(x, n) \supset (\exists m<n) f_2(x, n-m),$$

$$g_2(x, n) \Leftrightarrow (f_1(x, n) \supset (\exists m<n) f_2(x, n-m)).$$

Then g_1 and g_2 are regular features and $g_1(x, n)=\rho_1(x, n)$ and $g_2(x, n)=\rho_2(x, n)$, where ρ_1 and ρ_2 are two finite relabeling transformations.

Proof. The finite relabeling transformations ρ_1 and ρ_2 record the (truth) values of the features g_1 and g_2 respectively. Each g_i is a regular feature since it is computed by a finite relabeling transformation ρ_i. The computation of ρ_1 may be decomposed as follows. First, compute f_1 and f_2 by finite relabeling transformations taking x to $(f_1(x), f_2(x))$. The computation checks that $(\forall n) f_1(x, n) \supset (\exists(m>n) f_2(x, n-m))$ which is a check that the ordered pair (T, F) never occurs in the intermediate result.

We now give tree automata versions of these results and then establish an extension of the Peters–Ritchie result.

A (bottom-up) tree automaton is an automaton operating on (rooted, labeled, directed and left-to-right ordered) trees. Assuming that the states of the automaton at all descendants of a node are known, the next state at the node is determined by the transition function. At the beginning, initial states are assumed as descendants of the frontier nodes of the tree. Such a tree automata are referred to as bottom-up, since the states may be visualized as moving up the tree from the frontier to the root. Thus we have

DEFINITION 4.5. *A (bottom-up) tree automaton, A,* is a system $A = (T_\Sigma, Q, M, Q_0, F)$. Σ is a finite alphabet, T_Σ is the set of finite rooted, directed, left-to-right ordered trees labeled with elements of Σ, Q is a finite state set, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and $M: \Sigma \times Q^n \rightarrow Q$ determines the state of a node, v, in terms of the states of its direct descendants. $M: T_\Sigma \rightarrow Q$ is then defined as follows: If v is a leaf, labeled σ, then $M(v) = M(\sigma, q_0)$; otherwise, if v has direct descendants $x_1 \cdots x_n$ then $M(v) = M(\sigma, M(x_1), \cdots, M(x_n))$.

A recognizable feature of a set of nodes is a condition which can be verified at those nodes by a finite state (bottom-up) tree automaton. A recognizable feature of a set of trees is a feature recognized at the root nodes of the trees in the set.

Thus there is a two block partition of the state set of the automaton, $\pi = \{\pi_f, \pi_{\bar{f}}\}$, such that for all trees possessing f, the state of the automaton after processing is a state in π_f, and for all trees not possessing f, the state of the automaton after processing is in $\pi_{\bar{f}}$.

If f is a recognizable feature, then F is the set of trees possessing feature f, and $\bar{F}$ is the set of trees not possessing feature f. A set of trees possessing a recognizable feature is said to be a recognizable set.

Remark. The word "feature" could have been replaced in our definition by the word "predicate". However, feature is preferred because of its descriptive connotation. Both of these are closely related to the word "state".

The following result is well-known [3], [6], [11], [12].

THEOREM 4.4. *Let f_1 and f_2 be any recognizable features; then F_1, $F_1 \cap F_2$, $F_1 \cup F_2$, and $\bar{F}_1$ are recognizable.*

COROLLARY 4.4. *Let $g_1 = B(\{f_i\})$ be any Boolean combination of a finite set of recognizable features, then G_1 is a recognizable set and g_1 is a recognizable feature.*

COROLLARY 4.5. *If f_1 and f_2 are recognizable features and $g_2 = (f_1 \supset f_2)$, then G_2 is a recognizable set and g_2 is a recognizable feature.*

It is also well-known that [3], [6], [11], [12]

THEOREM 4.5. *If F is a recognizable set of trees then* yield (F) *is a context-free language.*

DEFINITION 4.6. Let Subtree $(x) = \{y \mid y$ is a subtree of $x\}$. (If

$$x = \begin{array}{c}\sigma\\ \diagup\ \diagdown\\ x_1 \cdots x_n\end{array},$$

then x is a subtree of x, each x_i is a subtree of x, and each subtree of each x_i is a subtree of x.) A more formal definition follows the definition of tree domain below (Definition 4.8).

If f is a recognizable feature, the following are defined for trees:

(i) $g(x) \Leftrightarrow ((\forall y \in \text{Subtree } (x)) f(y))$, i.e. every subtree possesses f.

(ii) $h(x) \Leftrightarrow ((\exists y \in \text{Subtree } (x)) f(y))$, i.e. some subtree possesses f.

(iii) $j(x) \Leftrightarrow ((\exists! y \in \text{Subtree } (x)) f(y))$, i.e. exactly one subtree possesses f.

As a generalization of Theorem 4.2 we now have

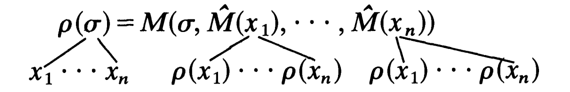
THEOREM 4.6. *Let f be a recognizable feature, then g, h, j as defined above, are recognizable features.*

We will now consider, in somewhat more detail, the case of an automaton which is checking multiple (persistent) features on trees. This will be the case, for example, when verifying that a tree is in $F_1 \cap F_2$. Again, the development parallels the string case.

DEFINITION 4.7. A *finite relabeling transformation* is a mapping which assigns to each node of the tree the state of the automaton, A, processing it [3] as follows:

$$\rho(\lambda)=\lambda,$$

$$\rho(\sigma)=M(\sigma, q_0) \quad \text{for } \sigma\in\Sigma,$$



Example:

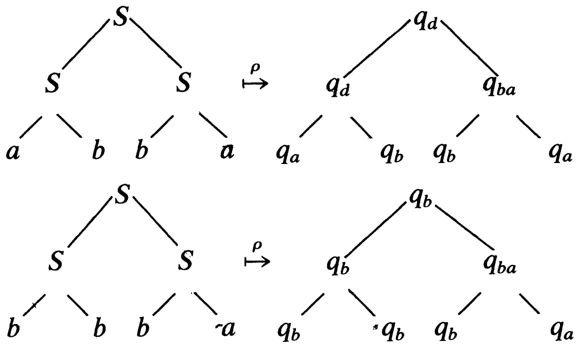
$M(a, q_0)=q_a$, $M(b, q_0)=q_b$, $M(S, q_a, q_a)=q_a$,

$M(S, q_b, q_a)=q_b$, $M(S, q_b, q_a)=q_{ba}$, $M(S, q_{ba}, q_a)=q_{ba}$,

$M(S, q_b, q_{ba})=q_b$; otherwise $M(\cdot)=q_d$ and

$F=\{q_a, q_b, q_{ba}\}$.

Then



It is known that [3]

THEOREM 4.7. *If F is recognizable then $\rho(F)$ is recognizable.*

Let J^* be the free monoid generated by J, the set of all natural numbers. Let the binary operation be denoted by $\cdot$ and the identity by 0. For p, q in J^*, $p\leqq q$ if and only if there is an r in J^* such that $q=p\cdot r$ and $p<q$ if and only if $p\leqq q$ and $p\neq q$.

DEFINITION 4.8 (see [5]). A *tree domain D* is a finite subset of J^* such that

1. if q D and p q, then p D; and
2. if $p\cdot j\in D, j\in J$, then $p\cdot r\in D$ for $r\in J$, r is equal to or less than j.

Elements of D are called *tree addresses*. A labeled tree is a function from a tree domain to a finite set of labels. We can now define a subtree formally as follows. If the tree x is a function $x\colon D\to\Sigma$, then the subtree at p, denoted x_p, has tree domain $D_p=\{q|p\cdot q\in D\}$ and $x_p\colon D_p\to\Sigma$, with $x_p(q)=x(p\cdot q)$.

DEFINITION 4.9 (*Left-to-right order*). $p\lessdot q$, p is to the left of q, for $p, q\in D$, if $p=r\cdot j_1\cdot s_1$, $q=r\cdot j_2\cdot s_2$, $j_i\in J$, s_i, $r\in J^*$, $i=1, 2$, and j_1 is less than j_2.

We now extend the tree notation of features, as we did in the string case, so that $f(x, v)$ is the feature computed in the tree x at tree address v.

LEMMA 4.1. If f_1, f_2, and f_3 are recognizable features, and

$$g_1(x, p) \Leftrightarrow (f_1(x, p) \supset (\exists q) f_2(x, p \cdot q)),$$

$$g_2(x, p) \Leftrightarrow (f_1(x, p) \supset \neg(\exists q) f_2(x, p \cdot q),$$

$$g_3(x, p) \Leftrightarrow (f_1(x, p) \supset (\exists q)(\exists r) f_2(p \cdot q) \wedge f_3(p \cdot r) q \dot{<} r))),$$

then g_1, g_2, and g_3 are recognizable features.

Proof. The argument is essentially that given in the proof of Theorem 4.3, using finite relabeling to record intermediate steps of the computation. Thus $(f_1(x, p), f_2(x, p))$ can be computed, and existential quantification applied as in Theorem 4.6 noting that $(\exists q) f_2(x, q)$ is an alternative notation for $(y \in \text{Subtree }(x)) f(y)$. The feature g_3 is obtained similarly noting that an automaton can easily check the left-to-right order at a given node.

THEOREM 4.8. *Any first order predicate, with bounded quantification as in Definition* 4.1 *and Lemma* 4.1, *expressed in terms of recognizable features on trees, their subtrees, or left-to-right ordering is a recognizable feature.*

Remarks. 1. Any condition on the proper analyses of a set of trees which can be verified by a tree automaton yields a recognizable feature. Any combination of such features as given by Theorem 4.8 will still be recognizable. From this and the next remark, Theorem 3.1 follows.

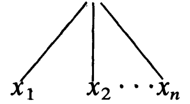
2. Recognizable sets are related to local sets [9], [10], [11]. Given a recognizable set one can develop a context-free grammar so that the recognizable set is (the homomorphic image of) the set of derivation trees of the context-free grammar.

In the following we consider some other generalizations of the Peters–Ritchie result.

LEMMA 4.2. *Let R be a regular set; then the feature $f(t) = (P(t) \subseteq R)$ is recognizable, where $P(t)$ is the set of proper analyses of t.*

Proof. (This method of proof is due to Rounds [9]). Each regular set, R, is the union of complete congruences of a finite congruence relation. We evaluate the tree (bottom-up) keeping track of the congruence classes of proper analyses. Since there are only finitely many congruence classes, the set of congruence classes having nonnull intersection with the proper analyses of a subtree can be represented in the state of the tree automaton when it reaches the root of the subtree.

If



is a node of the tree and its direct descendants, and if we know the set of congruence classes with which each x_i has nonnull intersection, then we can determine the set of congruence classes with which the proper analysis at x has nonnull intersection. Now the proper analysis of t, $P(t)$, is a subset of R, if and only if the set of congruence classes with which the root has nonnull intersection is a subset of the congruence classes defining R.

Remark. Lemma 4.2 does not imply that we can check if $P(t)=R$ because the above proof also shows that we can tell if $L \subseteq R$ where L is a context-free language and R is an arbitrary regular set; however, whether or not $L=R$ is known to be undecideable.

From Theorem 4.8, Lemma 4.2 and Theorem 4.5, we have

THEOREM 4.9. *The yield of the set of trees analyzed by productions of the form* $A \rightarrow B/\alpha \_\_ \lambda$ *(read as "A becomes* β *in the context* $\alpha \_\_ \lambda$*") is context-free, where* A *is a nonterminal,* β *is a string of terminals and nonterminals, and* α, λ *are regular expressions over terminals and nonterminals.*

Remarks. 1. Theorem 4.9 generalizes the Peters–Ritchie result which is just Theorem 2.7 specialized to the case where α, λ are strings of terminals and nonterminals.

2. By Theorem 4.8, a finite or regular set of proper analyses may be excluded. This is so because of the closure of recognizable features under complementation and intersection.

Another type of feature on trees which we often like to check is the relative positions of occurrences of given recognizable subtrees. Theorem 4.9 can be generalized to this case and the proof methods carry over directly. The following proposition is an example of such a generalization. First, we need a new definition.

DEFINITION 4.10. Let β be a set of recognizable trees. By a rule $A \rightarrow B$ we understand the (possibly infinite) set of rules $\{A \rightarrow x | x \in \beta\}$. Similarly, a rule $A \rightarrow \beta/\alpha \_\_ \lambda$ is understood to represent the set of rules $\{A \rightarrow x/\alpha \_\_ \lambda | x \in \beta\}$, where α and λ are expressions denoting recognizable sets.

THEOREM 4.10. *The yield of trees analyzed by productions of the form* $A \rightarrow \beta/\alpha \_\_ \lambda$ *is context-free, where* A, α, β, *and* λ *are as in Definition* 5.1. *The set of trees analyzed by these productions is recognizable.*

Remark. An immediate interpretation of Theorem 4.10 is that ALGOL-like languages requiring declarations are still context-free if a bound is placed on the size of the names, since the automaton can check for the presence of the declaration.

5. Conclusions. We can summarize the main argument as follows: Finite string (tree) automata compute features of strings (trees). Using standard techniques of automata theory, computations of several finite string (tree) automata can be combined so that any features described in a first order language are themselves computable by finite string (tree) automata. In particular, the proper analysis predicates of the Peters–Ritchie result are computable by finite tree automata, and Boolean combinations of these proper analysis predicates also are finite-tree-automata computable, including the subtree computation (Theorem 4.6) required for domination predicates. Some further generalizations are suggested by Theorems 4.9 and 4.10.

Acknowledgment. We are deeply indebted to the two referees of this paper whose very detailed comments have helped us to improve greatly both the accuracy and the style of presentation. We also want to thank Barry Soroka and Kang Yueh for their valuable comments. Barry Soroka has also implemented, in MTS-LISP, a system for local transformational analysis (M.S. Thesis, University of Pennsylvania, 1976).

REFERENCES

[1] B. Baker, *Tree transductions and families of tree languages*, Proc. Fifth ACM Symp. on Theory of Computing, Association for Computing Machinery, New York, 1973, pp. 200–206.

[2] N. Chomsky, *Aspects of the Theory of Syntax*, The M.I.T. Press, Cambridge, MA, 1965, p. 215.

[3] J. Donner, *Tree acceptors and some of their applications*, J. Comput. System Sci., 4 (1970), pp. 406–451.

[4] J. Engelfriet, *Bottom up and topdown tree transformations: a comparison*, Tech. Rep., Technische Hogeschool Twente, Holland, 1971.

[5] S. Gorn, *Explicit definitions and linguistic dominoes*, Proc. Systems and Computer Science Conference, Univ. of Western Ontario, London, Ont., 1965, pp. 77–115.

[6] L. S. Levy, *Automata on trees: A tutorial survey*, Egyptian Computer J., 1976, to appear.

[7] P. S. Peters and R. W. Ritchie, *Context sensitive immediate constituent analysis—context free languages revisited*, Proc. ACM Symp. Theory of Computing, Association for Computing Machinery, New York, 1969, pp. 1–10.

[8] M. O. Rabin and D. Scott, *Finite automata and their decision problems*, reprinted in Sequential Machines: Selected Papers, E. F. Moore, ed., Addison-Wesley, Reading, MA, pp. 63–91.

[9] W. C. Rounds, *Tree oriented proofs of some theorems on context-free and indexed languages*, Proc. ACM Symp. on Theory of Computing, Association for Computing Machinery, New York, 1970, pp. 210–216.

[10] J. W. Thatcher, *Characterizing derivation trees of context-free grammars through a generalization of finite automata theory*, J. Comput. System Sci., 1 (1967), pp. 317–322.

[11] J. W. Thatcher and J. B. Wright, *Generalized finite automata theory with applications to a problem of second order logic*, Math. Systems Theory, 2 (1968), pp. 57–81.

[12] J. W. Thatcher, *Tree automata: an informal survey*, Currents in the Theory of Computing, A. A. Aho, ed., Prentice-Hall, Englewood Cliffs, NJ, 1973, pp. 143–172.

SIAM J. COMPUT.
Vol. 6, No. 2, June 1977

A COMPLETENESS CRITERION FOR SPECTRA*

T. HIKITA† AND A. NOZAKI‡

Abstract. A *spectrum* is an algebraic representation of a set of "switching elements" each of which carries out an operation in a definite time lag. The notion of functional completeness ($\sim$-*completeness*) in the family of spectra was introduced by V. B. Kudryavcev and A. Nozaki. In this paper we investigate the maximal incomplete spectra and thus give a new effective criterion for a spectrum to be functionally complete.

Key words. switching element, spectrum, functional completeness, maximal incomplete set

1. Introduction. In this paper we are concerned with "completeness problem" on sets of *indexed operators.*

The classical completeness problem in logic and switching theory deals with sets of (many-valued) *logical functions* defined on the set $\{0, 1, \cdots, k-1\}$. A set of logical functions is said to be *functionally complete* (or simply, *complete*) if it can generate by the operation of composition the set of all logical functions. Several criteria have been given for the completeness of sets of logical functions, among which are those given by E. L. Post [5] for the binary case, and J. Slupecki [8] and J. W. Butler [1] for general case. S. V. Yablonskii [10] and I. Rosenberg [6], [7] have refined these results by determining maximal incomplete sets for ternary and general case respectively.

V. B. Kudryavcev [2], [3] generalized the above completeness problem by considering a "function with time delay," that is, a pair (f, d) of a logical function f and a nonnegative integer d (which we call *indexed operator*). An indexed operator may be considered to be a model of a switching element which carries out an operation in a definite time delay. A set of these pairs is said to be functionally complete ($\sim$-*complete* in Nozaki's terminology) if for any logical function f there exists a nonnegative integer d such that a pair (f, d) can be obtained by composing the elements of the set "combinationally" (for the precise definition, see § 2.2). V. B. Kudryavcev gave a solution for the binary case by giving all the maximal incomplete sets of these pairs explicitly. A. Nozaki [4] redefined this problem in mathematically clear form and has given a criterion for general values of k.

The purpose of this paper is to give a new effective criterion for $\sim$-completeness by combining the result of A. V. Kuznecov (see [11]) and the characterization of the maximal incomplete spectra. Our result corresponds to the one of J. W. Butler for the classical case of logical functions, and it deepens the results of both V. B. Kudryavcev and A. Nozaki.

2. Preliminaries. In this section we prepare several notations, definitions and basic results on functional completeness of logical functions and indexed operators.

* Received by the editors May 30, 1975, and in revised form May 10, 1976.

† Department of Information Science, Faculty of Science, University of Tokyo, Tokyo 113 Japan.

‡ Department of Computer Science, Faculty of Engineering, Yamanashi University, Kofu, Yamanashi 400 Japan.

2.1. Logical functions and completeness. Let $M(X, Y)$ denote the set of all mappings from a set X to a set Y.

DEFINITION. For the set $X=\{0, 1, \cdots, k-1\}$, where k is greater than one, let

(i) $\Omega_n(k)=M(X^n, X)$, where n is a positive integer,

(ii) $\Omega(k)=\bigcup_{n=1}^{\infty}\Omega_n(k)$.

We call an element of $\Omega(k)$ a *(k-valued) logical function.*

The set $\Omega_1(k)$ forms a semigroup by the operation of composition of functions.

DEFINITION. Let f be in $\Omega_p(k)$, and $G=\{g_1, \cdots, g_p\}$ be a set of logical functions where $g_i\in\Omega_{q_i}(k)$. A logical function h in $\Omega_N(k)$ is said to be *constructable from f and G* if and only if h can be written as follows:

$$h(x_1, \cdots, x_N)=f(g_{i_1}(x_{t(1,1)}, \cdots, x_{t(1,q_{i_1})}), \cdots,$$
$$g_{i_p}(x_{t(p,1)}, \cdots, x_{t(p,q_{i_p})})),$$

where the $x_{t(i,j)}$'s are variables chosen from $x_1, \cdots, x_N$, and each $g_{i_1}, \cdots, g_{i_p}\in G$.

DEFINITION. Let F and G be subsets of $\Omega(k)$. We denote by $F\otimes G$ the set of all the logical functions constructable from elements in F and subsets of G.

DEFINITION. For any subset F of $\Omega(k)$, let

(i) $F^{(1)}=F\otimes\{I\}$, where I denotes the identity function in $\Omega_1(k)$,

(ii) $F^{(d)}=F^{(d-1)}\otimes F^{(1)}$ for $d\geqq 2$,

(iii) $\bar{F}=\bigcup_{d=1}^{\infty}(F\cup\{I\})^{(d)}$.

$\bar{F}$ is said to be the *closure* of F. F is said to be *closed* if and only if $\bar{F}=F$. F is said to be *complete (in $\Omega(k)$)* if and only if $\bar{F}=\Omega(k)$. F is said to be *maximal incomplete* (or simply *maximal*) if and only if F is not complete and $\bar{G}=\Omega(k)$ for any G properly containing F.

We remark that if a set F is maximal then it is closed. The set $\{\text{AND}\in\Omega_2(2), \text{INVERT}\in\Omega_1(2)\}$ is a classical example (among others) of complete sets in the binary case. For the general values of k, D. L. Webb [9] showed that the set $\{\text{webb}\in\Omega_2(k)\}$ is complete, where

$$\text{webb}(x, y)=\min\{x, y\}\oplus 1,$$

where $\oplus$ denotes the addition modulo k. This function will be used later.

Now we introduce some important notation and results useful to completeness problem.

DEFINITION. For any subsets S_1 and S_2 of $\Omega_1(k)$, define a set of logical functions

$$F(S_1, S_2)=\{f\in\Omega(k);(\{f\}\otimes S_1)\cap\Omega_1(k)\subseteq S_2\}.$$

This set for the special case $S_1=S_2$ was first defined by J. W. Butler [1]. We state its easy properties without proof.

LEMMA 2.1. (i) *If $S_1\subseteq S_2$ then $F(S_1, S)\supseteq F(S_2, S)$;*

(ii) *If $S_1\subseteq S_2$ then $F(S, S_1)\subseteq F(S, S_2)$;*

(iii) $F(S_1, S_2)\cap F(S_3, S_4)\subseteq F(S_1\cap S_3, S_2\cap S_4)$.

We also give a condition for a set $F(S, S)$ to be equal to the set $\Omega(k)$:

LEMMA 2.2. *Let S be a nonempty subset of $\Omega_1(k)$. Then $F(S, S)$ is closed. Moreover, $F(S, S)$ is complete if and only if there is a direct sum decomposition $X = \sum_{m=1}^{s} X_m$ of the base set $X = \{0, 1, \cdots, k-1\}$ such that the following equality holds*:

$$S = \{h \in \Omega_1(k); |h(X_m)| = 1 \textit{ for all } m = 1, \cdots, s\},$$

where $|Y|$ denotes the cardinality of a set Y.

Proof. Closedness of the set $F(S, S)$ can be shown immediately, so that we have $\overline{F(S, S)} = F(S, S)$. The sufficiency part of the second assertion is easy, so we will show the necessity part. Assume that $\overline{F(S, S)} = \Omega(k)$ and let $S = \{h_1, \cdots, h_t\}$. For any two elements x, y of the base set X define a binary relation R as follows: $x\mathrm{R}y$ if and only if $h_i(x) = h_i(y)$ for all $i = 1, \cdots, t$. It is easily seen that this is an equivalence relation. Let $X = \sum_{m=1}^{s} X_m$ be a direct sum decomposition of the base set X associated with this equivalence relation. Let

$$S' = \{h \in \Omega_1(k); |h(X_m)| = 1 \text{ for all } m = 1, \cdots, s\}.$$

It is obvious that $S \subseteq S'$, so we show the converse. For any function h in S' define a logical function f in $\Omega_t(k)$ by

$$f(x_1, \cdots, x_t) = \begin{cases} h(x) & \text{if } x_i = h_i(x) \text{ for some } x \\ & \text{and any } i = 1, \cdots, t; \\ 0 & \text{otherwise.} \end{cases}$$

We can check the well-definedness of f. Since $F(S, S) = \Omega(k)$ by assumption, $h(x) = f(h_1(x), \cdots, h_t(x))$ is in S. Therefore $S' \subseteq S$. Thus $S = S'$ and the proof is completed.

Now we introduce some special classes of logical functions.

DEFINITION. A logical function f in $\Omega_n(k)$ is said to be *linear* if and only if f can be written in the following form:

$$f(x_1, \cdots, x_n) = a_0 \oplus a_1 x_1 \oplus \cdots \oplus a_n x_n,$$

where multiplications and additions are taken modulo k, and $a_0, a_1, \cdots, a_n$ are constants in $\{0, 1, \cdots, k-1\}$. We denote by $L(k)$ the set of all the linear logical functions.

Obviously the set $L(k)$ is closed.

DEFINITION. Let f be a logical function in $\Omega_n(k)$.

(i) f is said to be *simple* if and only if

$$f(x_1, \cdots, x_n) = f'(x_i)$$

for some $f' \in \Omega_1(k)$ and some i.

(ii) f is said to be *semi-simple* if and only if f is simple or nonsurjective.

We denote by $S(k)$ the set of all the semi-simple functions. The set $S(k)$ is also closed.

DEFINITION. We denote by $N(k)$ a set of logical functions defined as follows:

$$N(k) = \begin{cases} L(2) & \text{if } k = 2; \\ S(k) & \text{if } k \geqq 3. \end{cases}$$

Note that $N(k) \cap \Omega_1(k) = \Omega_1(k)$. We state two easy lemmas without proof:

LEMMA 2.3. *Let $f \in \Omega_n(k)$ and $g \in \Omega_m(k)$, and define h in $\Omega_{mn}(k)$ as follows:*

$$h(x_1, \cdots, x_{mn}) = f(g(x_1, \cdots, x_m), \cdots, g(x_{mn-m+1}, \cdots, x_{mn})).$$

If $f, g \notin N(k)$ then $h \notin N(k)$.

LEMMA 2.4. *If a logical function f is surjective then we can find $h_1, \cdots, h_n \in \Omega_1(k)$ such that*

$$f(h_1(x), \cdots, h_n(x)) = I(x).$$

We note that if $f \notin N(k)$ then f is surjective; hence Lemma 2.4 is applicable.

Now we cite the main results due to J. Slupecki [8] and J. W. Butler [1] on completeness for sets of logical functions.

THEOREM 2.5. *A set of logical functions is complete if and only if it is not contained in any maximal set.*

THEOREM 2.6. *Let M be a maximal set of logical functions in $\Omega(k)$. Then one of the following holds:*

(i) *There is a proper subsemigroup S of $\Omega_1(k)$ including the identity function I such that*

$$M = F(S, S)$$

and

$$S = M \cap \Omega_1(k).$$

(ii) $M = N(k)$.

Consequently, there are only a finite number of maximal sets of logical functions for each value of k.

Remark. One of the main goals in the completeness problem has been to determine all the maximal sets explicitly. In the binary case E. L. Post [5] determined five maximal sets in all, and for the ternary case S. V. Yablonskii [10] determined eighteen maximal sets in all. Finally, I. Rosenberg [6], [7] determined all the types of maximal sets for general values of k.

2.2. Indexed operators, spectra and ~-completeness.

DEFINITION. A pair (f, d) of a logical function f in $\Omega(k)$ and a nonnegative integer d is said to be an *indexed operator* over $\Omega(k)$.

DEFINITION. Let Φ be a set of indexed operators. We denote by $\tilde{\Phi}$ the minimum set of indexed operators defined recursively as follows:

(i) $(I, 0) \in \tilde{\Phi}$;

(ii) If $(f, d) \in \Phi$ and $(g_1, d'), \cdots, (g_n, d') \in \tilde{\Phi}$ then $(h, d+d') \in \tilde{\Phi}$ for any h in $\{f\} \otimes \{g_1, \cdots, g_n\}$.

$\tilde{\Phi}$ is called the ~-*closure* of Φ.

DEFINITION. A set Φ of indexed operators over $\Omega(k)$ is said to be ~-*closed* if and only if $\tilde{\Phi} = \Phi$. Φ is said to be ~-*complete* if and only if for any f in $\Omega(k)$ there exists a nonnegative integer d such that $(f, d) \in \tilde{\Phi}$. Φ is said to be ~-*maximal* if and only of Φ is not ~-complete and Ψ is ~-complete for any Ψ properly including Φ.

We note as an example that, although the set $\{\mathrm{NAND} \in \Omega_2(2)\}$ is complete, the set $\{(\mathrm{NAND}, 1)\}$ is *not* $\sim$-complete. We remark that if Φ is $\sim$-maximal then it is $\sim$-closed.

Now we introduce the notion of spectra, which is conceptually equivalent to sets of indexed operators.

DEFINITION. A *spectrum* $\mathscr{F}$ is a semi-infinite sequence

$$\mathscr{F} = (F_0, F_1, \cdots, F_d, \cdots)$$

of subsets of $\Omega(k)$. For two spectra $\mathscr{F} = (F_d)_{d \geqq 0}$ and $\mathscr{G} = (G_d)_{d \geqq 0}$ we say that $\mathscr{F}$ *includes* $\mathscr{G}$ (and denote $\mathscr{F} \supseteq \mathscr{G}$) if $F_d \supseteq G_d$ for all $d \geqq 0$.

Let Φ be a set of indexed operators, and define subsets of $\Omega(k)$ by

$$F_d = \{f \in \Omega(k); (f, d) \in \Phi\}$$

for each $d \geqq 0$. Then $\mathscr{F} = (F_0, F_1, \cdots)$ is a spectrum. Conversely, from any spectrum $\mathscr{F}$ we can define a set of indexed operators Φ. That is, mathematically these two notions are equivalent. Since the notion of spectra is easier to treat, we will mainly use it in the rest of the paper.

DEFINITION. Let $\mathscr{F} = (F_d)_{d \geqq 0}$ be a spectrum. Then the *$\sim$-closure* $\tilde{\mathscr{F}} = (\tilde{F}_d)_{d \geqq 0}$ of $\mathscr{F}$ is defined by

(i) $\tilde{F}_0 = \bar{F}_0$,

(ii) $\tilde{F}_d = (\tilde{F}_0 \otimes (F_d \otimes \tilde{F}_0)) \cup_{i=1}^{d-1} (\tilde{F}_i \otimes \tilde{F}_{d-i})$ for all $d \geqq 1$.

DEFINITION. Let $\mathscr{F} = (F_d)_{d \geqq 0}$ be a spectrum. $\mathscr{F}$ is said to be *$\sim$-closed* if and only if $\tilde{\mathscr{F}} = \mathscr{F}$. $\mathscr{F}$ is said to be *$\sim$-complete* if and only if $\bigcup_{d=0}^{\infty} \tilde{F}_d = \Omega(k)$. $\mathscr{F}$ is said to be *$\sim$-maximal* if and only if $\mathscr{F}$ is not $\sim$-complete and $\mathscr{G}$ is $\sim$-complete for any $\mathscr{G}$ properly including $\mathscr{F}$.

These definitions for spectra can be checked to be equivalent to those for sets of indexed operators.

Now we state an easy proposition on $\sim$-completeness for spectra without proof.

PROPOSITION 2.7. *Let $\mathscr{F} = (F_d)_{d \geqq 0}$ be a spectrum. If there is some $d \geqq 0$ such that F_d is complete and contains the identity function I, then $\mathscr{F}$ is $\sim$-complete.*

DEFINITION. For any distinct elements $a_1, \cdots, a_i$ $(i \geqq 2)$ of the base set $X = \{0, 1, \cdots, k-1\}$, define a set of logical functions by

$$K(a_1, \cdots, a_i) = \{f \in \Omega(k); f(a_1, \cdots, a_1) = \cdots = f(a_i, \cdots, a_i)\}.$$

The following theorem is due to A. Nozaki [4]. We prove it here, since the proof is not published yet. Actually the conditions in the theorem are sufficient for $\sim$-completeness of a spectra as announced in [4], but we will need only the necessity part.

THEOREM 2.8. *Let $\mathscr{F} = (F_d)_{d \geqq 0}$ be a spectrum, and assume that F_0 is not complete. If $\mathscr{F}$ is $\sim$-complete, then $\mathscr{F}$ satisfies the following two conditions*:

(i) *For any maximal set M of logical functions there exists a nonnegative integer q such that, for any positive integer p,*

$$\tilde{F}_{pq} \not\subseteq M;$$

(ii) *There exists a positive integer d such that for any two distinct elements a, b of the base set* $X=\{0, 1, \cdots, k-1\}$,

$$\tilde{F}_d \not\subseteq K(a, b).$$

Proof. (i) We consider the following two possible cases.

Case 1. There exists a constant-valued function c in $\Omega_1(k)$ such that $c \notin M$. Since $\mathscr{F}$ is $\sim$-complete, there is a nonnegative integer q such that $c \in \tilde{F}_q$. Then obviously $c = c^p \in \tilde{F}_{pq}$ for any $p > 0$. Therefore we have $\tilde{F}_{pq} \not\subseteq M$.

Case 2. All the constant-valued functions in $\Omega_1(k)$ belong to M. We define a function g in $\Omega_4(k)$ as follows:

$$g(x, y, u, v) = \begin{cases} \mathrm{webb}(x, y) & \text{if } u \neq v, \\ x & \text{if } u = v. \end{cases}$$

Now, since $\mathscr{F}$ is $\sim$-complete, we have $g \in \tilde{F}_q$ for some $q \geqq 0$. And we have

$$I(x) = g(x, x, x, x) \in \tilde{F}_q.$$

Therefore, for any $p > 0$, we have $g \in \tilde{F}_{pq}$. By the way, we can show that $g \notin M$ for any M. For, if $g \in M$, since the constant-valued functions 0 and 1 are in M and M is closed, we have

$$\mathrm{webb}(x, y) = g(x, y, 0, 1) \in M.$$

But this is a contradiction since the set $\{\mathrm{webb}\}$ is complete. Thus we have $\tilde{F}_{pq} \not\subseteq M$.

(ii) Since $\mathscr{F}$ is $\sim$-complete, we have $\mathrm{webb} \in \tilde{F}_d$ for some d. If $d = 0$ then F_0 is necessarily complete, so that d must be positive. Now, we have $\mathrm{webb}' \in \tilde{F}_d$ where

$$\mathrm{webb}'(x) = \mathrm{webb}(x, x) = x \oplus 1.$$

Therefore $\tilde{F}_d \not\subseteq K(a, b)$ for any two distinct elements a and b. Thus the proof is completed.

3. Completeness criterion for spectra. An analogy of Theorem 2.5 holds for the case of $\sim$-completeness.

THEOREM 3.1. *A spectrum $\mathscr{F}$ is $\sim$-complete if and only if it is not included in any $\sim$-maximal spectrum.*

A more generalized form of this theorem is due to A. V. Kuznecov (see [11]), and is presented in V. B. Kudryavcev [2], [3].

Now we define special types of spectra.

DEFINITION. Let $\mathscr{T} = (T_d)_{d \geqq 0}$ be a spectrum. We call $\mathscr{T}$ of

(i) *first type* if there exist $p(\geqq 1)$ subsets of $\Omega_1(k)$ denoted by $S_0, S_1, \cdots, S_{p-1}$ which satisfy the conditions 1–3 listed below, such that

$$T_d = \bigcap_{j=0}^{p-1} F(S_j, S_{\overline{j+d}})$$

for all $d \geqq 0$, where $\overline{j+d}$ denotes the residue of $j+d$ modulo p;

1. S_0 is a proper subsemigroup of $\Omega_1(k)$ containing the identity function I;
2. $I \notin S_j$ for $j = 1, \cdots, p-1$;
3. $S_j \otimes S_m \subseteq S_{\overline{j+m}}$ for any j, m;

(ii) *second type* if

$$T_d = N(k)$$

for all $d \geqq 0$;

(iii) *third type* if there exist two subsets of $\Omega_1(k)$ denoted by S_0 and S, which satisfy the conditions 1–3 listed below, such that

$$T_0 = F(S_0, S_0)$$

and

$$T_d = F(S_0, S)$$

for all $d \geqq 1$;

1. S_0 is a proper subsemigroup of $\Omega_1(k)$ containing I;

2. $S = \Omega_1(k) \cap K(a_{11}, a_{12}, \cdots) \cap K(a_{21}, a_{22}, \cdots) \cap \cdots$ for more than one distinct element $a_{11}, a_{12}, \cdots$ of the base set $X = \{0, 1, \cdots, k-1\}$;

3. $S \otimes S_0 \subseteq S$.

These three types are mutually disjoint. Now we have the following lemma:

LEMMA 3.2. *A spectrum which is of one of these three types is* $\sim$*-closed and* $\sim$*-incomplete.*

Proof. We use the notations in the definition above. The $\sim$-closedness is easy to show, so we only prove the $\sim$-incompleteness. Assume that $\mathcal{T}$ is a spectrum of first type. Since S_0 is a proper subsemigroup of $\Omega_1(k)$ containing I, $F(S_0, S_0)$ is incomplete by Lemma 2.2. Hence there is some maximal set M such that $F(S_0, S_0) \subseteq M$. Therefore

$$\begin{aligned} T_{pq} &= \bigcap_{j=0}^{p-1} F(S_j, S_j) \\ &\subseteq M \end{aligned}$$

for all $q \geqq 0$. Hence $\mathcal{T}$ is $\sim$-incomplete by Theorem 2.8.

If $\mathcal{T}$ is of second type then $\mathcal{T}$ is obviously $\sim$-incomplete.

If $\mathcal{T}$ is of third type, then for all $d \geqq 1$

$$\begin{aligned} T_d &= F(S_0, S) \\ &\subseteq K(a_{11}, a_{12}, \cdots) \cap K(a_{21}, a_{22}, \cdots) \cap \cdots \\ &\subseteq K(a_{11}, a_{12}); \end{aligned}$$

therefore $\mathcal{T}$ is $\sim$-incomplete by Theorem 2.8. Thus the proof is completed.

With these preparations we can now state our key proposition.

PROPOSITION 3.3. *Let* $\mathcal{F}$ *be a spectrum. Assume that* $\mathcal{F}$ *is* $\sim$*-closed and* $\sim$*-incomplete. Then there exists some spectrum* $\mathcal{T}$ *which is of one of the three types, such that* $\mathcal{T} \supseteq \mathcal{F}$.

Since the proof is lengthy and complicated, it is deferred to the next section. By Proposition 3.3 we can prove our main result:

THEOREM 3.4. *Any* $\sim$*-maximal spectrum* $\mathcal{M} = (M_d)_{d \geqq 0}$ *is of one of the three types. Moreover,*

(i) *if $\mathscr{M}$ is of first type, then for $j = 0, 1, \cdots, p-1$,*

$$S_j = M_j \cap \Omega_1(k),$$

where S_j and p are as in the definition of first type;

(ii) *if $\mathscr{M}$ is of third type, then*

$$S_0 = M_0 \cap \Omega_1(k)$$

and

$$S = M_1 \cap \Omega_1(k)$$

where S_0 and S are as in the definition of the third type.

Proof. By assumption $\mathscr{M}$ is $\sim$-maximal; hence $\mathscr{M}$ is $\sim$-closed and $\sim$-incomplete, so that Proposition 3.3 can be applied and there exists a spectrum $\mathscr{T}$ of one of the three types such that $\mathscr{T} \supseteq \mathscr{M}$. $\mathscr{M}$ is $\sim$-maximal and $\mathscr{T}$ is $\sim$-incomplete so it necessarily follows that $\mathscr{M} = \mathscr{T}$; therefore $\mathscr{M}$ is of one of the three types.

Now we show (i). Assume that $\mathscr{M}$ is of first type. Put $S'_j = M_j \cap \Omega_1(k)$ for $j = 0, 1, \cdots, p-1$. We can check that S'_j's also satisfy the properties 1–3 required for S_j's in the definition of first type. Let $\mathscr{M}' = (M'_d)_{d \geqq 0}$ be a spectrum where

$$M'_d = \bigcap_{j=0}^{p-1} F(S'_j, S'_{\overline{j+d}})$$

for all $d \geqq 0$. We can easily show that $\mathscr{M}' \supseteq \mathscr{M}$. $\mathscr{M}'$ is $\sim$-incomplete since $\mathscr{M}'$ is of first type. Since $\mathscr{M}$ is $\sim$-maximal it necessarily follows that $\mathscr{M} = \mathscr{M}'$. Therefore we can take S'_j instead of S_j and we have shown the assertion. Assertion (ii) can be shown in the same way. Thus the proof is completed.

4. Proof of Proposition 3.3.

In this section we will settle Proposition 3.3. Let $\mathscr{F} = (F_d)_{d \geqq 0}$ be a spectrum and assume that $\mathscr{F}$ is $\sim$-closed and $\sim$-incomplete. Put $U_d = F_d \cap \Omega_1(k)$ for all $d \geqq 0$. For any positive integer p let $\Sigma(p)$ denote a set of subsets of $\Omega_1(k)$ defined by

$$\Sigma(p) = \{U_{pq}; q = 1, 2, \cdots\}.$$

Then we have the following

LEMMA 4.1. *There exists some positive integer p' such that*

$$\Sigma(p') = \Sigma(p'r)$$

holds for any $r \geqq 1$.

Proof. Suppose that the assertion does not hold. Take any $p'_0 \geqq 1$; then we have $\Sigma(p'_0) \supseteq \Sigma(p'_0 r)$ for any $r \geqq 1$. If $\Sigma(p'_0) = \Sigma(p'_0 r)$ for any $r \geqq 1$ then the assertion holds; hence there exists some $r_0 \geqq 1$ such that $\Sigma(p'_0) \supsetneqq \Sigma(p'_0 r_0)$. Put $p'_1 = p'_0 r_0$ and repeat the same argument for p'_1 instead of p'_0; then we have $\Sigma(p'_0) \supsetneqq \Sigma(p'_1) \supsetneqq \Sigma(p'_1 r_1)$. Therefore, continuing the above arguments, we have the following sequence of inclusions of infinite length:

$$\Sigma(p'_0) \supsetneqq \Sigma(p'_1) \supsetneqq \Sigma(p'_2) \supsetneqq \cdots.$$

But it is a contradiction since every $\Sigma(p'_i)$ is a finite set. Thus the proof of Lemma 4.1 is completed.

Hereafter this p' of Lemma 4.1 will be considered to be fixed.

Next, for each $j = 0, 1, \cdots, p'-1$, consider the following sequence of subsets of $\Omega_1(k)$:

$$(U_j, U_{j+p'}, U_{j+2p'}, \cdots),$$

and let $V_j^1, \cdots, V_j^{c_j}$ be all the distinct subsets of $\Omega_1(k)$ that appear an *infinite* number of times as a component in this sequence. Put $V_j = \bigcap_{i=1}^{c_j} V_j^i$ for each j. Then we have the following

LEMMA 4.2. *If $h \in U_d$ for some $d \geqq 1$, then we can find some $u \geqq 1$ such that $h^u \in V_0$, where h^u denotes the composite $h \circ \cdots \circ h$ (u times) of the function h.*

Proof. Since the set $\Omega_1(k)$ is a finite semigroup under the operation of composition, for $h \in \Omega_1(k)$ we can find some v, $w \geqq 1$ such that $h^{v+w} = h^v$. Put $u = p'vw$. Then we can check that $(h^u)^2 = h^u$. Also we have $h^u \in U_{p'(dvw)}$ by $\sim$-closedness of $\mathscr{F}$. Therefore we see that $h^u \in U_{p'(dvw)q}$ for any $q \geqq 1$, namely we have $h^u \in V$ for any $V \in \Sigma(p'(dvw))$. Since $\Sigma(p'(dvw)) = \Sigma(p')$ by Lemma 4.1, we have $h^u \in V$ for any $V \in \Sigma(p')$. Therefore by definition of V_0 we can conclude that $h^u \in V_0$. Thus the proof of Lemma 4.2 is completed.

Now put

$$G_d = \bigcap_{j=0}^{p'-1} F(V_j, V_{\overline{j+d}})$$

for all $d \geqq 0$, where $\overline{j+d}$ denotes the residue of $j+d$ modulo p', and let $\mathscr{G} = (G_d)_{d \geqq 0}$ be a spectrum. Then we have the following

LEMMA 4.3. *$\mathscr{G}$ is $\sim$-closed and $\mathscr{G}$ includes $\mathscr{F}$.*

Proof. It is easily verified that $\mathscr{G}$ is $\sim$-closed, so we show the second assertion. Take any $d \geqq 0$ and any $j = 0, 1, \cdots, p'-1$. Since $\mathscr{F}$ is $\sim$-closed, for each $i = 1, \cdots, c_j$ there is some e_i such that

$$F_d \otimes V^{e_i}_{\overline{j-d}} \subseteq V^i_j,$$

namely

$$F_d \subseteq F(V^{e_i}_{\overline{j-d}}, V^i_j).$$

By Lemma 2.1 (i) we have

$$F(V^{e_i}_{\overline{j-d}}, V^i_j) \subseteq F(V_{\overline{j-d}}, V^i_j);$$

therefore

$$F_d \subseteq F(V_{\overline{j-d}}, V^i_j).$$

This inclusion holds for each $i = 1, \cdots, c_j$, so that

$$F_d \subseteq \bigcap_{i=1}^{c_j} F(V_{\overline{j-d}}, V^i_j).$$

Then by Lemma 2.1 (iii) we have

$$\bigcap_{i=1}^{c_j} F(V_{\overline{j-d}}, V_j^i) \subseteq F(V_{\overline{j-d}}, V_j);$$

therefore

$$F_d \subseteq F(V_{\overline{j-d}}, V_j).$$

This inclusion holds for each $j=0, 1, \cdots, p'-1$, so that

$$\begin{aligned} F_d &\subseteq \bigcap_{j=0}^{p'-1} F(V_{\overline{j-d}}, V_j) \\ &= \bigcap_{j=0}^{p'-1} F(V_j, V_{\overline{j+d}}) \\ &= G_d. \end{aligned}$$

Thus $F_d \subseteq G_d$ holds for all $d \geqq 0$; hence the proof of Lemma 4.3 is completed.

Now we will show that there exists a spectrum of one of three types which contains $\mathscr{F}$. There are five possible cases to consider according to the property of the set V_0.

Case 1. $F(V_0, V_0) \subseteq M$ for some maximal set M other than $N(k)$.

Case 2. $F(V_0, V_0) \subseteq N(k)$.

Case 3. $F(V_0, V_0) = \Omega(k)$ and $V_0 \neq \Omega_1(k), \phi$.

Case 4. $V_0 = \Omega_1(k)$.

Case 5. $V_0 = \phi$.

Let us begin with

Case 1. $F(V_0, V_0) \subseteq M$ for some maximal set M other than $N(k)$. Put $S_j = G_j \cap \Omega_1(k)$ for $j = 0, 1, \cdots, p'-1$. Then we have

$$\begin{aligned} S_0 &= \bigcap_{j=0}^{p'-1} F(V_j, V_j) \cap \Omega_1(k) \\ &\subseteq M \cap \Omega_1(k); \end{aligned}$$

hence S_0 is a proper subset of $\Omega_1(k)$ by Theorem 2.6. Moreover, since $\mathscr{G}$ is $\sim$-closed, S_0 is a subsemigroup of $\Omega_1(k)$ including the identity function I and $S_j \otimes S_m \subseteq S_{\overline{j+m}}$ for any j, m.

Now, put

$$p = \begin{cases} \min\{j;\ 1 \leqq j \leqq p'-1,\ I \in S_j\} & \text{if this set is not empty;} \\ p' & \text{if this set is empty.} \end{cases}$$

Then $p|p'$ holds. For, if $I \notin S_j$ for any j then $p = p'$. If $I \in S_{j_0}$ for some $j_0 \geqq 1$, then we can find some $q \geqq 1$ such that both $\overline{j_0 q}|p'$ and $\overline{j_0 q} \leqq j_0$ hold. Then we have $I = I^q \in S_{\overline{j_0 q}}$. By this argument it necessarily follows that $p|p'$. Moreover we can see that if $j \equiv m (\mathrm{mod}\, p)$ then $S_j = S_m$, since $\mathscr{G}$ is $\sim$-closed and $I \in S_p$.

We put

$$T_d = \bigcap_{j=0}^{p-1} F(S_j, S_{\overline{j+d}})$$

for all $d \geqq 0$, where $\overline{j+d}$ denotes the residue of $j+d$ modulo p, and let $\mathscr{T} = (T_d)_{d \geqq 0}$ be a spectrum. We already have that $\mathscr{T}$ is of first type. Also, we can directly show that $\mathscr{T} \supseteq \mathscr{G}$. Therefore $\mathscr{T} \supseteq \mathscr{F}$ holds by Lemma 4.3.

Case 2. $F(V_0, V_0) \subseteq N(k)$. By Lemma 4.3, for all $q \geqq 0$ we have

$$\begin{aligned} F_{p'q} &\subseteq G_{p'q} \\ &= \bigcap_{j=0}^{p'-1} F(V_j, V_j) \\ &\subseteq N(k). \end{aligned}$$

Then by Lemma 2.3 and $\sim$-closedness of $\mathscr{F}$ it necessarily follows that $F_d \subseteq N(k)$ for all $d \geqq 0$. Therefore obviously $\mathscr{F}$ is contained in a spectrum of the second type.

Case 3. $F(V_0, V_0) = \Omega(k)$ and $V_0 \neq \Omega_1(k)$, ϕ. Since $F(V_0, V_0) = \Omega(k)$ and $V_0 \neq \phi$, by Lemma 2.2 there is a direct sum decomposition $X = \sum_{m=1}^{s} X_m$ of the base set $X = \{0, 1, \cdots, k-1\}$ such that the following holds:

$$(*) \qquad V_0 = \{h \in \Omega_1(k);\ |h(X_m)| = 1 \text{ for all } m = 1, \cdots, s\}.$$

Put

$$\Pi = \{(a, b) \in X^2;\ a \neq b \text{ and both } a \text{ and } b \text{ are contained in the same subset } X_m\}.$$

Since $V_0 \neq \Omega_1(k)$ there is some X_m such that $|X_m| \geqq 2$; hence $\Pi \neq \phi$. With these preparations we can show the following basic lemma for this case.

LEMMA 4.4. *There exist two distinct elements a, b of the base set $X = \{0, 1, \cdots, k-1\}$ such that*

$$F_d \subseteq K(a, b)$$

for all $d \geqq 1$.

Proof. Suppose that the assertion does not hold. Hence for any $(a_i, b_i) \in \Pi$ there is some $h_i \in U_{d_i}$ where $d_i \geqq 1$ such that $h_i(a_i) \neq h_i(b_i)$. Under these situations we can show the following:

$(**)$ There is some $h' \in U_{d'}$ where $d' \geqq 1$ such that both $h'(a') = a'$ and $h'(b') = b'$ hold for some $(a', b') \in \Pi$.

We show $(**)$ for the following two possible cases.

Case A. For any $(a_i, b_i) \in \Pi$, $(h_i(a_i), h_i(b_i)) \in \Pi$. In this case, since Π is a finite set, we can find some $(a', b') \in \Pi$ and some h' which is a composite of several h_i's such that $h'(a') = a'$ and $h'(b') = b'$.

Case B. There is some $(a_i, b_i) \in \Pi$ such that $(h_i(a_i), h_i(b_i)) \notin \Pi$. In this case, by $(*)$ there must be some $h'' \in V_0$ such that $h''(h_i(a_i)) = a_i$ and $h''(h_i(b_i)) = b_i$. Put $h' = h'' \circ h_i$, $a' = a_i$ and $b' = b_i$.

Now, by Lemma 4.2 we have $h'^u \in V_0$ for some $u \geqq 1$. But, by $(**)$ we have $h'^u(a') = a'$, $h'^u(b') = b'$ and $a' \neq b'$; hence $h'^u(a') \neq h'^u(b')$, which contradicts to $(*)$. Thus the proof of Lemma 4.4 is completed.

Next, let

$$K = K(a_{11}, a_{12}, \cdots) \cap K(a_{21}, a_{22}, \cdots) \cap \cdots$$

be the minimal set that contains $\bigcup_{d=1}^{\infty} F_d$ and that can be expressed as the form above, where $a_{11}, a_{12}, \cdots$ are more than one distinct element of the base set $X=\{0, 1, \cdots, k-1\}$. By Lemma 4.4 such K exists, and clearly such K is unique.

First we consider the case that the set U_0 is a proper subset of $\Omega_1(k)$. Put $S_0=U_0$ and $S=K\cap\Omega_1(k)$. Since $\mathscr{F}$ is $\sim$-closed, S_0 is a subsemigroup of $\Omega_1(k)$ including I. Also we have $S\otimes S_0\subseteq S$ by $\sim$-closedness of $\mathscr{F}$ and definition of K.

Put

$$T_0=F(S_0, S_0)$$

and

$$T_d=F(S_0, S)$$

for all $d\geqq 1$. Then $\mathscr{T}=(T_d)_{d\geqq 0}$ is a spectrum and it is already obvious that $\mathscr{T}$ is of the third type. It is easily verified that $\mathscr{T}$ contains $\mathscr{F}$.

Now we consider the case that $U_0=\Omega_1(k)$. In this case we can show that $F_d\subseteq N(k)$ for all $d\geqq 0$. For, if there is some d such that $f\in F_d$ where $f\notin N(k)$, then by Lemma 2.4 and $\sim$-closedness of $\mathscr{F}$ we have $I\in F_d$; hence $F_d\supseteq\Omega_1(k)\cup\{f\}$. Then F_d is complete by Theorem 2.6; therefore $\mathscr{F}$ is $\sim$-complete by Proposition 2.7, which contradicts our first assumption. So it is clear that $\mathscr{F}$ is contained in a spectrum of the second type.

Case 4. $V_0=\Omega_1(k)$. By definition of V_0 there exists some q^* such that $F_{p'q}\supseteq\Omega_1(k)$ for all $q\geqq q^*$. Then we can show that $F_d\subseteq N(k)$ for all $d\geqq 0$. For, if not, by Lemma 2.4 there is some large d such that $f_d\supseteq\Omega_1(k)\cup\{f\}$ where $f\notin N(k)$. Then $\mathscr{F}$ is $\sim$-complete by the same argument as before, which contradicts our first assumption. So it is clear that there is a spectrum of second type which contains $\mathscr{F}$.

Case 5. $V_0=\phi$. By Lemma 4.2 and $\sim$-closedness of $\mathscr{F}$, $F_d=\phi$ for all $d\geqq 1$. Since $\mathscr{F}$ is not $\sim$-complete, there is a maximal set M which includes F_0. Then there is a spectrum of the first or second type which includes $\mathscr{F}$.

Thus the whole proof of Proposition 3.3 is completed.

REFERENCES

[1] J. W. BUTLER, *On complete and independent sets of operations in finite algebras*, Pacific J. Math., 10 (1960), pp. 1169–1179.

[2] V. B. KUDRYAVCEV, *Completeness theorem for a class of automata without feedback couplings*, Problemy Kibernet., 8 (1959), pp. 91–115. (In Russian.)

[3] ———, *Completeness theorem for a class of automata without feedback couplings*, Dokl. Akad. Nauk SSSR, 132 (1960), pp. 272–274 = Soviet Math. Dokl., 1 (1960), pp. 537–539.

[4] A. NOZAKI, *Réalisation des fonctions definies dans un ensemble fini à l'aide des organes élémentaires d'entrée-sortie*, Proc. Japan Acad., 46 (1970), pp. 478–482.

[5] E. L. POST, *The Two-valued Iterative Systems of Mathematical Logic*, Princeton Annals of Math. Studies, 5, Princeton University Press, Princeton, N.J., 1941.

[6] I. ROSENBERG, *La structure des fonctions de plusieurs variables sur un ensemble fini*, C. R. Acad, Sci. Paris Sér. A-B, 260 (1965), pp. 3817–3819.

[7] ———, *Über die funktionale Vollständigkeit in den mehrwertigen Logiken*, Czechoslovakia Academy, Prague, 1970.

[8] J. SLUPECKI, *Completeness criterion in many-valued logical system*, Comptes Rendus des séances de la Société des Sciences et des Lettres de Varsovie, Cl. III, 32 (1939), pp. 102–109. (In Polish.)

[9] D. L. WEBB, *Definition of Post's generalized negative and maximum in terms of one binary operation*, Amer. J. Math., 58 (1936), pp. 193–194.

[10] S. V. YABLONSKII, *Functional structures in k-valued logic*, Trudy Mat. Inst. Steklov., 51 (1958), pp. 5–142. (In Russian.)

[11] S. A. YANOVSKAYA, *Mathematical logic and foundations of mathematics*, Mathematics in USSR for Forty Years 1917–1957, vol. 1, M., Fizmatgiz, Moscow, 1959, pp. 13–120. (In Russian.)

SIAM J. COMPUT.
Vol. 6, No. 2, June 1977

SUPERCONCENTRATORS*

NICHOLAS PIPPENGER†

Abstract. An n-superconcentrator is an acyclic directed graph with n inputs and n outputs for which, for every $r \leqq n$, every set of r inputs, and every set of r outputs, there exists an r-flow (a set of r vertex-disjoint directed paths) from the given inputs to the given outputs. We show that there exist n-superconcentrators with $39n + O(\log n)$ (in fact, at most $40n$) edges, depth $O(\log n)$, and maximum degree (in-degree plus out-degree) 16.

Key words. superconcentrator, concentrator, directed graph

Superconcentrators were defined by Valiant [1] who showed that there exist n-superconcentrators with at most $238n$ edges. Superconcentrators have proved useful in counterexemplifying conjectures [1] and in demonstrating the optimality of algorithms [2].

Valiant's proof was based on a complicated recursive construction which used a related type of graph, called a "concentrator," as a basic element. Concentrators were defined by Pinsker [3], who showed that there exist (n, m)-concentrators (which we shall not define here), with at most $29n$ edges. Pinsker's proof was based on another rather complicated recursive construction which used a nonconstructive existence theorem concerning bipartite graphs as a basic element. This theorem, though not the recursive construction for concentrators, was also obtained independently by the author [4].

The purpose of this note is to give a sharpened version of the nonconstructive existence theorem and a simple recursive construction, using this theorem as a basic element, for superconcentrators. This yields four benefits. First, the proof that n-superconcentrators with $O(n)$ edges exist is greatly simplified; our construction is simpler than Pinsker's, let alone its composition with Valiant's. Second, our n-superconcentrators have depth $O(\log n)$; Valiant's have depth $O((\log n)^2)$. Third, our superconcentrators have maximum degree (in-degree plus out-degree) 16; Pinsker's concentrators (and thus Valiant's superconcentrators) do not have maximum degree $O(1)$. Finally, our n-superconcentrators have $39n + O(\log n)$ (in fact, at most $40n$) edges.

LEMMA. *For every m, there exists a bipartite graph with $6m$ inputs and $4m$ outputs in which every input has out-degree at most 6, every output has in-degree at most 9, and, for every $k \leqq 3m$ and every set of k inputs, there exists a k-flow (a set of r vertex-disjoint directed paths) from the given inputs to some set of k outputs.*

Proof. Let π be a permutation on $\mathcal{M} = \{0, 1, \cdots, 36m-1\}$. From π we obtain a bipartite graph $G(\pi)$ by taking $\{0, 1, \cdots, 6m-1\}$ as inputs, $\{0, 1, \cdots, 4m-1\}$ as outputs, and, for every x in $\mathcal{M}$, an edge from $(x \bmod 6m)$ to $(\pi(x) \bmod 4m)$. In $G(\pi)$, every input has out-degree at most 6 (since there are only 6 elements of $\mathcal{M}$ in each residue class mod $6m$) and each output has in-degree at most 9 (since there are only 9 elements of $\mathcal{M}$ in each residue class mod $4m$).

* Received by the editors April 14, 1976.

† Mathematical Sciences Department, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.

We shall say that a graph $G(\pi)$ is "good" if there do not exist a $k \leqq 3m$, a set A of k inputs, and a set B of k outputs such that every edge directed out of A is directed into B; we shall say that it is "bad" otherwise. If $G(\pi)$ is good, the marriage theorem (see Hall [5]) ensures that for every $k \leqq 3m$ and every set of k inputs, there exists a k-flow from the given inputs to some set of k outputs, so that $G(\pi)$ satisfies the requirements of the lemma. We shall show that there exists such a graph by obtaining an upper bound, less than unity for all m, on the fraction of all permutations π for which $G(\pi)$ is bad.

Any set A of k inputs corresponds to a set $\mathcal{A}$ of $6k$ elements of $\mathcal{M}$, and any set of B of k outputs corresponds to a set $\mathcal{B}$ of $9k$ elements of $\mathcal{M}$. Every edge of $G(\pi)$ directed out of A will be directed into B only if π sends every element of $\mathcal{A}$ into $\mathcal{B}$. Of the $(36m)!$ permutations of $\mathcal{M}$, there are $[9k]_{6k}\,(36m-6k)!$ that satisfy this condition, where $[n]_r = n(n-1)\cdots(n-r+1)$. For a given value of k, there are $\binom{6m}{k}$ possible choices for A and $\binom{4m}{k}$ possible choices for B.

Thus an upper bound on the fraction of all permutations π for which $G(\pi)$ is bad is

$$I_m = \sum_{1 \leqq k \leqq 3m} \binom{6m}{k}\binom{4m}{k}\frac{[9k]_{6k}(36m-6k)!}{(36m)!}$$

$$= \sum_{1 \leqq k \leqq 3m} \frac{\binom{6m}{k}\binom{4m}{k}\binom{9k}{6k}}{\binom{36m}{6k}}$$

We shall show that I_m is less than unity.

1. We first observe that

$$\binom{36m}{6k} \geqq \binom{6m}{k}\binom{4m}{k}\binom{26m}{4k},$$

for the number of ways of choosing $6k$ out of $36m$ objects is not less than the number of ways of choosing k out of the first $6m$, k out of the next $4m$, and $4k$ out of the last $26m$. Thus I_m is at most

$$J_m = \sum_{1 \leqq k \leqq 3m} \frac{\binom{9k}{6k}}{\binom{26m}{4k}}.$$

2. To find the largest term in J_m, we set

$$L_k = \frac{\binom{9k}{6k}}{\binom{26m}{4k}}$$

and observe that the ratio of successive terms can be written as

$$\frac{L_{k+1}}{L_k}=\frac{(9k+9)\cdots(9k+7)(9k+6)\cdots(9k+1)(4k+4)(4k+3)\cdots(4k+1)}{(6k+6)\cdots(6k+1)\qquad(3k+3)\cdots(3k+1)(26m-4k)\cdots(26m-4k-3)}.$$

Each vertically aligned factor or pair of factors is an increasing function of k. Thus L_{k+1}/L_k is increasing, $L_{k-1}L_{k+1}/L_k^2$ is greater than unity, and the largest term of J_m must be either the first (L_1) or the last (L_{3m}).

3. If the largest term is the first, then J_m is at most

$$3mL_1=3m\frac{\binom{9}{6}}{\binom{26m}{4}}=\frac{3024}{13(26m-1)(26m-2)(26m-3)},$$

which is less than unity for all $m\geqq 1$.

4. If the largest term is the last, then J_m is at most

$$3mL_{3m}=3m\frac{\binom{27m}{18m}}{\binom{26m}{12m}}=3m\frac{(27m)!\,(12m)!\,(14m)!}{(18m)!\;(9m)!\;(26m)!}.$$

We shall use Stirling's formula in the form

$$(2\pi n)^{1/2}e^{-n}n^n\leqq n!\leqq e^{1/12n}(2\pi n)^{1/2}e^{-n}n^n$$

(see Robbins [6]), together with

$$e^x\leqq\frac{1}{1-x}\qquad\qquad(x\leqq 1)$$

which implies

$$n!\leqq\left(\frac{12n}{12n-1}\right)(2\pi n)^{1/2}e^{-n}n^n.$$

These inequalities give

$$3mL_{3m}\leqq 3m\left(\frac{324m}{324m-1}\right)\left(\frac{144m}{144m-1}\right)\left(\frac{168m}{168m-1}\right)\cdot\left(\frac{27\;12\;14}{18\;\;9\;26}\right)^{1/2}\left(\frac{27^{27}\;12^{12}\;14^{14}}{18^{18}\;\;9^{9}\;26^{26}}\right)^{m},$$

which is less than unity for all $m\geqq 3$. (The bound for $m=3$ is easily evaluated with a table of logarithms and a calculator. Futhermore, the bound is a decreasing function of m, since if m is increased by 1, the first factor increases by a factor of at most 4/3, the next three factors decrease, and the last factor decreases by a factor exceeding 2.)

5. In the remaining cases, $m=1$ and $m=2$, I_m can be evaluated with a table of binomial coefficients (for example, Miller [7]), and is less than unity. □

COROLLARY. *For every m, there exists a bipartite graph with $4m$ inputs and $6m$ outputs in which every input has out-degree at most 9, every output has in-degree at most 6, and, for every $k \leqq 3m$ and every set of k outputs, there exists a k-flow to the given outputs from some set of k inputs.*

Proof. Exchange the roles of inputs and outputs and reverse the directions of edges and flows in the lemma. □

Let $s(n)$ denote the minimum possible number of edges in an n-superconcentrator. Let

$$\theta(n) = 4\left\lceil \frac{n}{6} \right\rceil,$$

where $\lceil \cdot \rceil$ denotes "the smallest integer not less than".

THEOREM. *For any n, $s(n) \leqq 13n + s(\theta(n))$.*

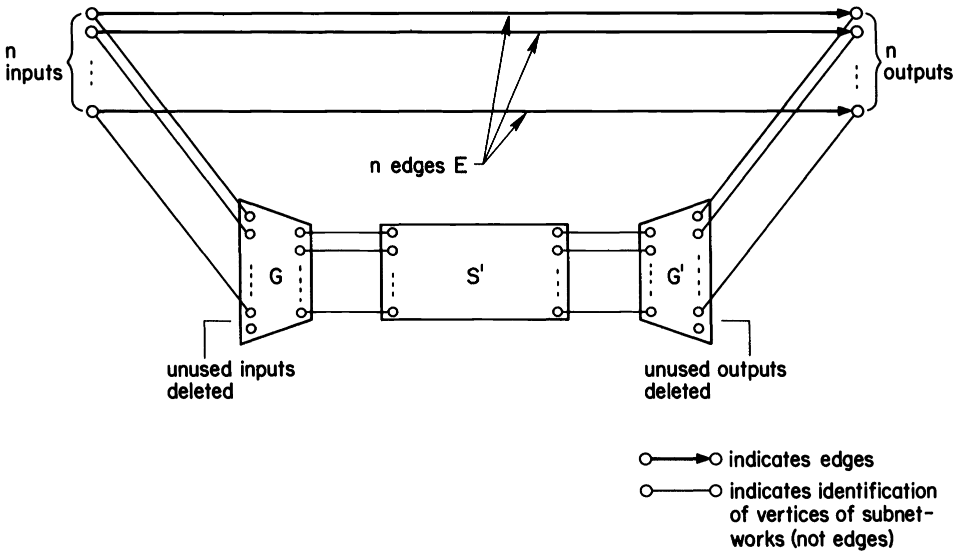
Proof. Let

$$m = \left\lceil \frac{n}{6} \right\rceil.$$

Let G and G' be bipartite graphs satisfying the lemma and corollary, respectively, and let S' be a $4m$-superconcentrator with $s(4m)$ edges. The graph S is obtained by deleting $6m - n$ inputs (and the edges directed out of them) from G, identifying the outputs of G with the inputs of S', identifying the outputs of S' with the inputs of G', deleting $6m - n$ outputs (and the edges directed into them) from G', and adding a set E of n edges from the surviving inputs of G to the surviving outputs of G'. This is illustrated in the figure below.

The graph S clearly has $13n + s(\theta(n))$ edges. All that remains is to verify that S is an n-superconcentrator.

For some $r \leqq n$, let X be a set of r inputs and let Y be a set of r outputs. Let X be partitioned into two parts: X_0, the vertices of X that correspond through E to



vertices in Y, and X_1, the vertices of X that correspond through E to vertices not in Y. Similarly, let Y be partitioned into Y_0 (corresponding to vertices in X) and Y_1 (corresponding to vertices not in X). There is an l-flow from X_0 through E to Y_0, where l is the common cardinality of X_0 and Y_0. The set X_1 corresponds through E to a set of vertices disjoint from and equinumerous with Y_1. Thus X_1 and Y_1 have a common cardinality $k \leqq n/2 \leqq 3m$. By the lemma, there is a k-flow from X_1 to some set X' of k outputs of G, and by the corollary, there is a k-flow from some set Y' of k inputs of G' to Y_1. Finally, by inductive hypothesis, there is a k-flow from X' through S' to Y'. These four flows together constitute an r-flow from X to Y. □

From this theorem it is clear that $s(n) \leqq 39n + O(\log n)$, and that this can be accomplished by graphs with depth $O(\log n)$ and maximum degree 16. Since it is often helpful to have an explicit bound, we shall show that $s(n) \leqq 40n$.

For small values of n we shall use a "rearrangeable connection network" or "permutation network." Such a network contains an n-flow following any prescribed mapping from its inputs to its outputs, and is, *a fortiori*, an n-superconcentrator. A well-known recursive construction for these networks gives

$$s(n) \leqq 3n(2\lceil \log_3 n \rceil - 1)$$

(see Beneš [8, Thm. 3.1]; in the outer stages use 3-by-3 switches, with at most one smaller switch when n is not a multiple of 3; in the inner stages use this construction recursively). This gives $s(n) \leqq 39n$ for $n \leqq N = 3^7 = 2187$.

For large values of n we shall apply the theorem recursively. Define

$$\theta^0(n) = n,$$
$$\theta^{t+1}(n) = \theta(\theta^t(n)).$$

Then applying the theorem $t+1$ times gives

$$s(n) \leqq 13(\theta^0(n) + \theta^1(n) + \cdots + \theta^t(n)) + s(\theta^{t+1}(n)).$$

Let us choose t such that

$$\theta^t(n) > N \geqq \theta^{t+1}(n).$$

Then by the result of the preceding paragraph

$$s(n) \leqq 13(\theta^0(n) + \theta^1(n) + \cdots + \theta^t(n)) + s(\theta^{t+1}(n)).$$

We note that

$$\begin{aligned} \theta(n) &= 4\left\lceil \frac{n}{6} \right\rceil \\ &\leqq 4\left(\frac{n}{6} + \frac{5}{6}\right) \\ &= \frac{2}{3}n + \frac{10}{3}, \end{aligned}$$

and $\theta(n)$ is even. Furthermore, if n is even,

$$\begin{aligned}\theta(n)&=4\left\lceil\frac{n}{6}\right\rceil\\ &\leqq 4\left(\frac{n}{6}+\frac{2}{3}\right)\\ &=\frac{2}{3}n+\frac{8}{3},\end{aligned}$$

and again $\theta(n)$ is even. Thus, by induction on t,

$$\theta^t(n)\leqq\left(\frac{2}{3}\right)^t n+8.$$

Applying this to the result of the preceding paragraph gives

$$s(n)\leqq 39n+104(t+3).$$

Next we note that if $n\geqq N=3^7=2187$,

$$\begin{aligned}\theta(n)&=4\left\lceil\frac{n}{6}\right\rceil\\ &\leqq 4\left(\frac{n}{6}+\frac{5}{6}\right)\\ &=\left(\frac{2}{3}+\frac{10}{3n}\right)n\\ &\leqq\frac{4384}{6561}n.\end{aligned}$$

Thus, by induction on t, if $\theta^0(n), \theta^1(n), \cdots, \theta^{t-1}(n)\geqq N$,

$$\theta^t(n)\leqq\left(\frac{4384}{6561}\right)^t n.$$

From the condition defining t it follows that

$$t\leqq\frac{\log\frac{n}{2187}}{\log\frac{6561}{4384}}.$$

Now

$$\log\tfrac{6561}{4384}\geqq\tfrac{1}{3}\log 3,$$

and therefore

$$t\leqq 3\log_3 n-21.$$

Furthermore, if $n\geqq N$,

$$\frac{3\log_3 n}{n}\leqq\frac{3\log_3 N}{N}=\frac{7}{729},$$

and therefore

$$t \leqq \frac{7}{729}n - 21,$$

or

$$104(t+3) \leqq \frac{728}{729}n - 1872.$$

Combining this with the result of the preceding paragraph gives

$$s(n) \leqq 40n.$$

REFERENCES

[1] L. G. VALIANT, *On nonlinear lower bounds in computational complexity*, Proc. 7th Annual ACM Symposium on Theory of Computing, Albuquerque, NM, May 1975, pp. 45–53.

[2] W. J. PAUL, R. E. TARJAN AND J. R. CELONI, *Space bounds for a game on graphs*, Proc. 8th Annual ACM Symposium on Theory of Computing, Hershey, PA, May 1976, pp. 149–160.

[3] M. S. PINSKER, *On the complexity of a concentrator*, Proc. 7th International Teletraffic Conference, Stockholm, June 1973, pp. 318/1–318/4.

[4] N. J. PIPPENGER, *The complexity theory of switching networks*, Ph.D. Thesis, Dept. of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, MA, August 1973.

[5] P. HALL, *On representatives of subsets*, J. London Math. Soc., 10 (1935), pp. 26–30.

[6] H. ROBBINS, *A remark on Stirling's formula*, Amer. Math. Monthly, 62 (1955), pp. 26–29.

[7] J. C. P. MILLER, ed., *Royal Society Mathematical Tables, Volume 3: Table of Binomial Coefficients*, University Press, Cambridge, England, 1954.

[8] V. E. BENEŠ, *Mathematical Theory of Connecting Networks and Telephone Traffic*, Academic Press, New York, 1965.

SIAM J. COMPUT.
Vol. 6, No. 2, June 1977

ON ISOMORPHISMS AND DENSITY OF *NP* AND OTHER COMPLETE SETS*

L. BERMAN AND J. HARTMANIS†

Abstract. If all *NP* complete sets are isomorphic under deterministic polynomial time mappings (p-isomorphic) then $P \neq NP$ and if all PTAPE complete sets are p-isomorphic then $P \neq$ PTAPE. We show that all *NP* complete sets known (in the literature) are indeed p-isomorphic and so are the known PTAPE complete sets. This shows that, in spite of the radically different origins and attempted simplification of these sets, all the known *NP* complete sets are identical but for simple isomorphic codings computable in deterministic polynomial time.

Furthermore, if all *NP* complete sets are p-isomorphic then they all must have similar densities and, for example, no language over a single letter alphabet can be *NP* complete, nor can any sparse language over an arbitrary alphabet be *NP* complete. We show that complete sets in EXPTIME and EXPTAPE cannot be sparse and therefore they cannot be over a single letter alphabet. Similarly, we show that the hardest context-sensitive languages cannot be sparse. We also relate the existence of sparse complete sets to the existence of simple combinatorial circuits for the corresponding truncated recognition problem of these languages.

Key words. polynomial time computations, polynomial tape computations, *NP* complete problems, polynomial time isomorphisms, sparse sets

1. Introduction. During the past years the importance of the $P = NP$? problem has been fully realized and today it is one of the most important problems in theoretical computer science [1], [2], [8], [10], [13]. The importance of the $P = NP$? problem derives from the fact that *NP*, the family of languages accepted by nondeterministic Turing machines in polynomial time, contains complete problems to which all other problems in *NP* can be easily reduced and from the fact that very many problems of practical interest in computing are in *NP* and many of them are *NP* complete [1], [2], [5], [10], [13], [17]. Thus the search for fast algorithms for a bewildering variety of problems can be reduced to search for a fast algorithm for a single problem. As a matter of fact, during the last years considerable effort has been expended in discovering new *NP* complete problems, and it is quite impressive how many diverse problems from many different problem areas have turned out to be *NP* complete [1], [2], [13], [17]. Furthermore, among the known *NP* complete problems some have been simplified and found still to be *NP* complete [5].

In this paper we show that regardless of their origins and attempted simplifications, all the "known" *NP* complete sets are essentially the same set. More specifically, we prove that all the known *NP* complete sets are isomorphic under deterministic polynomial time mappings. Thus these *NP* complete sets, except for a deterministic polynomial time recoding, are identical. The proof of this result follows from two technical lemmas which give necessary and sufficient conditions that a set is isomorphic under polynomial time mappings to a given *NP* complete set, say the set of all satisfiable Boolean functions in conjunctive normal form. To

* Received by the editors September 9, 1975, and in revised form July 11, 1976.

† Department of Computer Science, Cornwell University, Ithaca, New York, 14853. This research was supported in part by National Science Foundation under Grants GJ-33171X and DCR 75-09433.

establish polynomial time isomorphisms (p-isomorphism) between NP complete sets we just have to check that these sets satisfy the sufficient conditions of our lemmas, which turn out to be easy to verify for all the NP complete sets found in the literature. We exhibit the proof of the existence of p-isomorphism for the best known NP complete problems and they can be easily supplied for the other NP complete problems which have been described up to date in the literature. Since so far no NP complete problems have been found which are not p-isomorphic and since all attempts to construct such sets have failed, we are forced to conjecture that all NP complete sets are isomorphic under deterministic polynomial time mappings.

It should be observed that a proof of this conjecture implies that $P \neq NP$. To see this, we just have to note that $P = NP$ if and only if every nonempty finite set is NP complete. Since finite sets cannot be isomorphic to infinite sets, the isomorphism of all NP complete sets implies that $P \neq NP$. As a matter of fact, $P \neq NP$ if and only if all NP complete sets are isomorphic under recursive mappings.

It still could happen that $P \neq NP$ but that there exist NP complete sets which are not p-isomorphic. We conjecture that this is not the case.

By the same methods we also show that all the known PTAPE complete sets are isomorphic under deterministic polynomial time mappings. Furthermore, if all PTAPE complete sets are p-isomorphic then $P \neq$ PTAPE, since $P =$ PTAPE if and only if every nonempty finite set is PTAPE complete.

Next we look at the density of NP and PTAPE complete sets. We say that a set A, $A \subseteq \Sigma^*$, is *p-sparse* iff the number of elements in A up to length n is bounded by a polynomial in n. It is easily seen that the known NP and PTAPE complete sets are not p-sparse and that they cannot be p-isomorphic to p-sparse sets. We suspect that neither NP nor PTAPE complete sets can be p-sparse. Note that a proof that p-sparse sets cannot be NP nor PTAPE complete would prove that

$$P \neq NP \quad \text{and} \quad P \neq \text{PTAPE}.$$

On the other hand, we show that p-sparse sets cannot be complete in EXPTIME and EXPTAPE (as first observed by A. R. Meyer). Our proof actually shows that in EXPTIME there exist sets which are not p-sparse and whose reduction to another set must be one-one almost everywhere; thus, no p-sparse set can be EXPTIME complete. The corresponding result also holds for EXPTAPE and more complex time and tape bounded families of languages. It is still an open problem whether the EXPTIME and EXPTAPE complete sets are all p-isomorphic, respectively.

It should be observed that the existence of p-sparse complete sets for NP (or PTAPE) would imply that we could prepare a tape growing only polynomially in the length of the input such that all NP (or PTAPE) problems could be solved in deterministic polynomial time using this fixed tape for table-look up. Thus the existence of sparse NP complete sets would permit, for all practical purposes, the recognition of NP sets in deterministic polynomial time (using a precomputed, polynomially long tape segment). This seems to be quite unlikely, and the above mentioned results shows that this is not the case for EXPTIME and EXPTAPE:

there does not exist any sparse set to which complete problems in EXPTIME and EXPTAPE can be reduced in polynomial time.

Furthermore, we show that there is a very close relation between the existence of sparse oracle sets (not necessarily sets in *NP*) with which *NP* problems can be recognized in deterministic polynomial time and the existence of polynomially complex switching circuits which recognize the members up to length n, $n = 1, 2, \cdots$, of an *NP* complete set. As a matter of fact, as originally shown by A. R. Meyer, there exist sparse oracle sets for *NP* complete problems if and only if there exist polynomially complex circuits for the recognition of the corresponding truncated *NP* complete problem.

Finally, we turn to context-sensitive languages. We say (following R. Book) that a context-sensitive language L is *hardest* if every other context-sensitive language can be reduced to L by a linear-time mapping. It is well known that hardest context-sensitive languages (csl) exist [7] and that hardest context-free languages also exist [6]. Clearly, the context-sensitive languages are contained in *P* or *NP* iff a hardest csl is in *P* or *NP*, respectively. Similarly, the deterministic context-sensitive languages are equal to the nondeterministic context-sensitive languages iff a hardest csl is a deterministic csl. We prove that no p-sparse language can be a hardest csl and show that all known hardest csl's are p-isomorphic. These results easily generalize to hardest languages of other families of tape bounded languages.

2. Preliminaries. In this section, we make precise some of the objects which we will treat. Our terminology is reasonably standard and so this section may be skipped by those familiar with the terminology of complexity theory.

A *transducer* is a deterministic $k+2$ tape Turing machine with one two-way read-only input tape, k two-way read-write work tapes, and one one-way write only output tape.

Our *acceptor* will be a k-tape Turing machine. The input will be written on one of the tapes and all tapes are two-way read-write. Acceptance will be indicated by entering a final state and halting. If the machine has just one tape we call it a *single tape Turing machine*, otherwise, it is called a *multi-tape Turing machine*. If the next move function associated with the Turing machine is single-valued, we call it *deterministic*, otherwise, it is called *nondeterministic*. We note that a deterministic TM may be considered to be nondeterministic in a trivial fashion.

The *amount of time used by a* TM on input x is the number of steps in the shortest accepting computation if x is accepted; the number of steps in the longest computation if x is not accepted (if some computation does not halt it is undefined).

The *amount of tape used by a* TM is the smallest amount of tape used by an accepting computation if x is accepted, or the largest amount used by any computation if x is not accepted (again, if some computation uses unbounded tape, it is undefined).

A TM, M, *runs in time* (*tape*) $t(n)$ for some function $t(n)$ if for all $n \geqq 0$ for every x of length n, M uses no more than $t(n)$ time (tape) on input x.

$(N)\mathrm{DTIME}[t(n)]=\{A \mid A$ is accepted by a (non)deterministic TM which runs in time $t(n)\}$.

$(N)\mathrm{DTAPE}[t(n)]=\{A \mid$ is accepted by a (non)deterministic TM which runs on tape $t(n)\}$.

$$P=\bigcup_{i\geqq 0} \mathrm{DTIME}(n^i),$$

$$NP=\bigcup_{i\geqq 0} \mathrm{NDTIME}(n^i),$$

$$\mathrm{PTAPE}=\bigcup_{I\geqq 0} \mathrm{DTAPE}(n^i)=\bigcup_{i\geqq 0} \mathrm{NDTAPE}(n^i)=\mathrm{NPTAPE},$$

$$(N)\mathrm{DEXPTIME}=\bigcup_{i\geqq 0} (N)\mathrm{DTIME}(2^{in}),$$

$$\mathrm{DEXPTAPE}=\bigcup_{i\geqq 0} \mathrm{DTAPE}(2^{in}).$$

A transducer, T, is said to be *polynomial time bounded* if there is some polynomial $p(n)$ so that T runs in time $p(n)$.

A transducer, T, is said to be a *linear time transducer* if there is some constant, $c>0$, so that T runs in time cn.

A set $A\subseteq\Sigma^*$ is said to be *reducible* to a set $B\subseteq\Gamma^*$ if there is some transducer T such that T: $\Sigma^*\rightarrow\Gamma^*$ and $T(x)\in B$ iff $x\in A$. A is said to be reducible to B in *polynomial time* (p-reducible) if the transducer T runs in polynomial time. Similarly if T runs in linear time A is said to be *linearly reducible* to B.

A set, B, is *C-hard* for some class of sets C (e.g. NP or NTAPE(n)) if for every A in C, A is p-reducible to B.

A set, B, is *complete* for C if it is C-hard and B is in C.

A set, B, is *C-hardest* if B is in C and every A in C is linearly reducible to B. For example, hardest languages exist for the families of context-free languages, context-sensitive languages, deterministic context-sensitive languages, etc. but not for NP or PTAPE.

We say that a set A, $A\subseteq\Sigma^*$, is *p-sparse* iff there exists a polynomial $p(n)$ such that

$$|\{w \mid w\in A, |w|\leqq n\}|\leqq p(n),$$

where $|w|$ denotes the length of the sequence w and $|S|$ denotes the cardinality of the set S.

3. Polynomial time isomorphisms. One of the major concepts used in the classification of recursive sets is that of polynomial time reducibility. From the polynomial reducibilities of Cook [2] and Karp [10], it is straightforward to define two corresponding equivalence relations on the recursively enumerable sets. There is, however, a major limitation to the usefulness of these relations in the study of complete sets: two sets A and B which are complete for a class C, with respect to one of the reducibilities, are automatically equivalent, with respect to the same reducibility.

In this section, we define a new equivalence relation, polynomial time isomorphism, which is a proper refinement of the above two equivalences. We consider a subgroup of the group of recursive permutations:

$G_p = \{f | f$ is a one-to-one onto map and f is computable in p-time and f^{-1} is also computable in p-time$\}$.

We say that A and B are polynomial time isomorphic if they are related by an element of G_p. We formalize this in the following definition.

DEFINITION. A and B, $A \subseteq \Sigma^*$ and $B \subseteq \Gamma^*$, are *p-isomorphic* iff there exists a bijection $f: \Sigma^* \to \Gamma^*$ such that f is a p-reduction of A to B and f^{-1} is a p-reduction of B to A.

We say that a p-reduction f is *invertible* iff f is a one-one mapping (not necessarily onto) and f's left inverse is also computable in polynomial time. Finally, f is said to be *length increasing* if for all w we have $|f(w)| > |w|$.

We now prove a polynomial time bounded equivalent of the Cantor–Bernstein–Myhill theorem.

THEOREM 1. *Let p and q be length increasing invertible p-reductions of A to B and B to A, respectively. Then A and B are p-isomorphic.*

Proof. From p and q we will construct a bijection ϕ such that ϕ and ϕ^{-1} are p-time computable and

$$w \in A \quad \text{iff} \quad \phi(w) \in B.$$

We note that

$$\Sigma^* = R_1 \cup R_2 \quad \text{and} \quad \Gamma^* = S_1 \cup S_2$$

with

$$R_1 = \{(q \circ p)^k x | k \geqq 0 \text{ and } x \notin q(\Gamma^*)\},$$
$$R_2 = \{q \circ (p \circ q)^k x | k \geqq 0 \text{ and } x \notin p(\Sigma^*)\},$$
$$S_1 = \{(p \circ q)^k x | k \geqq 0 \text{ and } x \notin p(\Sigma^*)\},$$
$$S_2 = \{p \circ (q \circ p)^k x | k \geqq 0 \text{ and } x \notin q(\Gamma^*)\}.$$

Let $s(n)$ be a polynomial such that p, q, p^{-1} and q^{-1} can all be computed by deterministic TM's within $s(n)$ steps for inputs of length n. We assume that p^{-1} and q^{-1} both output a special symbol, $*$, if they are undefined. This is permissible since they are polynomial time bounded. Let ϕ and ϕ^{-1} be computed by the following:

$$\phi(z) = \begin{cases} p(z) & \text{if } z \in R_1, \\ q^{-1}(z) & \text{if } z \in R_2, \end{cases}$$

$$\phi^{-1}(z) = \begin{cases} p^{-1}(z) & \text{if } z \in S_2, \\ q(z) & \text{if } z \in S_1. \end{cases}$$

First notice that ϕ maps R_1 onto S_2 and R_2 onto S_1 and that these mappings are one-one. Furthermore, ϕ and ϕ^{-1} are inverses.

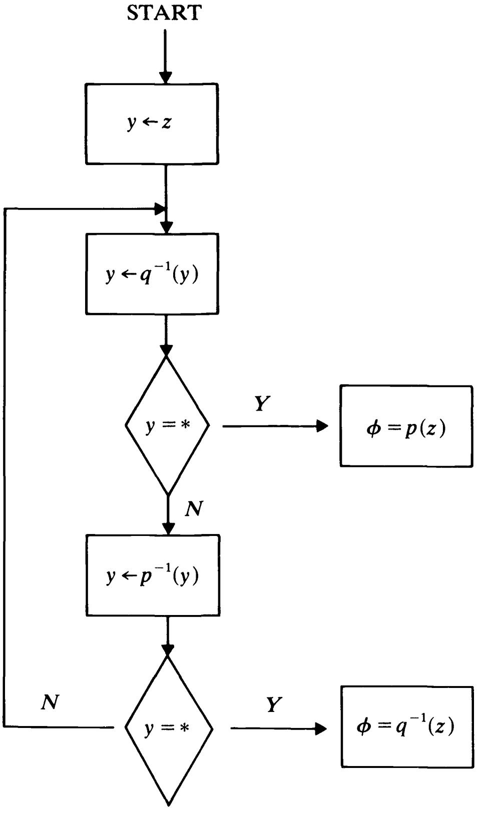


FIG. 1. *Flowchart computing* $T(z)=\phi(z)$

We will now describe a transducer, T, which computes ϕ and is polynomial time bounded. We describe T by means of the following flowchart (see Fig. 1). As p and q are both strictly length increasing, p^{-1} and q^{-1} are length decreasing and so T need cycle through the loop at most $|z|/2$ times. At most $(|z|+1)$ evaluations of p^{-1}, q^{-1}, or p are therefore required and so T runs in time at most $(n+2)\ s(n)$ which is a polynomial.

We note that identical considerations show that ϕ^{-1} is also p-time bounded.

COROLLARY 2. *If p, q, p^{-1}, q^{-1} of Theorem 1 are computable in linear time then ϕ and ϕ^{-1} are computable in n^2-time.*

Proof. The proof is similar to the previous proof.

In order to simplify the application of Theorem 1 we now establish two technical results which can easily be applied to show that many complete sets are p-isomorphic. We first define padding functions and show that if either set A or B of Theorem 1 (or Corollary 2) have padding functions satisfying some simple hypotheses, then we can remove the length increasing restrictions from the hypothesis of these results.

DEFINITION. Let $A \subseteq \Sigma^*$. Then $Z_A : \Sigma^* \to \Sigma^*$ is a *padding function* for a set A if it satisfies the following two properties:

1. $Z_A(x)$ in A iff x in A.
2. Z_A is invertible (i.e. one-one).

We say that a padding function, Z_A, has *time complexity* $t(n)$ if both Z_A and Z_A^{-1} may be computed by deterministic TM's in time $t(n)$.

LEMMA 3. *Let f be a one-one, p-time reduction of A to B and let f^{-1} also be computable in p-time. Assume also that either A or B has a padding function Z_X $(X = A,B)$ which satisfies conditions*:

(i) *Z_X has polynomial time complexity $s(n)$.*

(ii) $(\forall y)[|Z_X(y)|>|y|^2+1]$.

Then there exists a reduction f' of A to B which is one-one, p-time, length increasing and has $(f')^{-1}$ computable in p-time.

LEMMA 4. *If f, f^{-1} have linear time complexity, Z_X has linear time complexity and condition* (ii) *of Lemma* 3 *is replaced by*

(ii') $(\forall y)[|Z_X(y)|>2|y|+1]$,

then f' of Lemma 3 *exists and has linear time complexity.*

Proof of Lemma 3. Let $X=A$ and let f, f^{-1} be computable in polynomial time. Let $q(n)$ be a polynomial time bound in which f and f^{-1} can be computed. Then, by condition (ii) on the padding function we know there exists an integer r such that for all x, $|Z_A^r(x)|>q(|x|)$; therefore it follows that $|f\circ Z_A^r(x)|>|x|$, since if $|f\circ Z_A^r(x)|\leqq|x$ then (as f^{-1} can output at most one digit per move) $|f^{-1}\circ f\circ Z_A^r(x)|\leqq q(|x|)$, which is a contradiction. So define $f'=f\circ Z_A^r$, by the reasoning given above, f' is length increasing and clearly satisfies the other requirements of Lemma 3.

For $X=B$ and f, f^{-1} polynomial computable, we again know that there exists an integer r such that for all x, $|Z_B^r(x)|>q(|x|)$ and also as above $q(|f(x)|)\geqq|x|$. Let $f'=Z_B^r\circ f$; then $|f'(x)|=|Z_B^r(f(x))|>q(|f(x)|)\geqq|x|$ as needed.

For the linear time bounds a more careful time analysis yields the desired proof.

The primary difficulty in applying Theorem 1 is now seen to be verifying that a given reduction can be inverted in polynomial time. The next lemma states two technical, but easily verified, conditions which guarantee the existence of polynomial time invertible reductions. The next result shows that the existence of an invertible reduction depends solely on the richness of the structure of the target set.

LEMMA 5. *Let A be a set for which two p-time computable functions $S_A(-,-)$ and $D_A(-)$ exist with the following properties*:

(i) $(\forall x, y)[S_A(x,y)\in A$ *iff* $x\in A]$,

(ii) $(\forall x, y)[D_A(S_A(x,y))=y]$.

Then if f is any p-time reduction of some set C to A, the map $f'(x)=S_A(f(x),x)$ is one-one and invertible in p-time and reduces C to A.

Proof. Assume $f'(x)=f'(y)$. Then

$$x=D_A(f'(x))=D_A(S_A(f(x),x))=D_A(f'(y))=D_A(S_A(f(y),y))=y,$$

so f' is one-one. If we define

$$q(x)=\textbf{if } x=S_A(f(D_A(x)),D_A(x)) \textbf{ then } D_A(x) \textbf{ else } \quad *$$

we see that $q(f'(x))=x$ so $q=(f')^{-1}$ and a straightforward time analysis shows f' and q are both p-time computable.

To see that f' reduces C to A note that (i) above implies $f(x) \in A$ if and only if $f'(x) \in A$.

LEMMA 6. *If S_A, D_A, and f of Lemma 5 are all linear time computable then so is f'.*

Proof. The proof is straightforward.

We now combine these results in our next theorem which is easy to apply to show that many complete sets are p-isomorphic.

THEOREM 7. *Let the set A be p-reducible to B and B p-reducible to A; furthermore let the set A have a padding function Z_A satisfying Lemma 3 and functions S_A and D_A satisfying Lemma 5. Then B is p-isomorphic to A iff B has functions S_B and D_B satisfying Lemma 5.*

Proof. If A and B are p-isomorphic under the bijection ϕ then we can define

$$S_B[x, y] = \phi^{-1} \circ S_A[\phi(x), y]$$

and

$$D_B(x) = D_A[\phi(x)].$$

Clearly, S_B and D_B are p-time computable since S_A, D_A, ϕ and ϕ^{-1} are; furthermore for all y

$$\begin{aligned} x \in B \quad \text{iff} \quad \phi(x) \in A \quad &\text{iff} \quad S_A[\phi(x), y] \in A \\ &\text{iff} \quad \phi^{-1} \circ S_A[\phi(x), y] \in B. \end{aligned}$$

and

$$D_B[S_B(x, y)] = y.$$

Conversely, if the functions D_B and S_B exist then by combining Lemmas 3 and 5 we get that A and B are p-isomorphic. This completes the proof.

In the next section we apply these results to prove that large numbers of *NP* complete and PTAPE complete problems are p-isomorphic. The proof strategy is very simple; for example, we will show that the classic *NP* complete set of satisfiable Boolean functions in conjunctive normal form A (=CNF-SAT) has the three required functions Z_A, S_A and D_A and therefore any other *NP* complete set B which has two functions S_B and D_B, satisfying Lemma 5, is immediately seen to be p-isomorphic to A. Thus we will have proven our main result about *NP* complete sets; equivalent results hold for PTAPE and other complete sets.

MAIN THEOREM. *An NP complete set B is p-isomorphic to* CNF-SAT *if and only if there exist two p-time computable functions S_B and D_B such that*

(i) $(\forall x, y)[S_B(x, y) \in B \textit{ iff } x \in B]$,

(ii) $(\forall x, y)[D_B(S_B(x, y)) = y]$.

We stress that we know no *NP* or PTAPE complete problems which are not, respectively, isomorphic to the classic complete problems in the corresponding classes. Furthermore, all our attempts to construct such problems have failed.

For a related study of invertible reducibilities and padding see [16], and also the Appendix to this paper.

4. Applications. We first define a number of known *NP* complete problems:

1. UNIV—$\{\text{CODE}(x_1x_2 \cdots x_n) \# M_i \# ^{3|M_i|t} | M_i \text{ accepts } x_1 \cdots x_n \text{ in } t \text{ steps}\}$,

where $\mathrm{CODE}(x_1x_2\cdots x_n)$ is a simple digit by digit encoding of $x_1, x_2, \cdots, x_n$ so that $|\mathrm{CODE}(x)|=|M_i|$ [7].

2. CNF-SAT—given an encoding of a Boolean expression in conjunctive normal form, is there some assignment of truth values to the variables which gives the expression the value **true** [2]?
3. INEQ[0, 1), (,+, ·]—given an encoding of two regular expressions over 0, 1,), (, + , · do they represent different sets [11]?
4. CLIQUE—given an encoding of an undirected graph and an integer k, is there a subset of k-mutually adjacent nodes [10]?
5. HAMILTON CIRCUIT—given an encoding of a directed graph is there a cycle including all nodes which does not intersect itself [10]?

THEOREM 8. *The following NP complete problems are p-time isomorphic.*

1. UNIV,
2. CNF-SAT,
3. INEQ[0, 1,), (,+, ·],
4. CLIQUE,
5. HAMILTON CIRCUIT.

Proof. We first show that CNF-SAT has a padding function satisfying Lemma 3 and functions $S_A(-,-)$ and $D_A(-)$ satisfying Lemma 5. Then Theorem 7 will show that any set having functions satisfying Lemma 5 is p-isomorphic to CNF-SAT.

Consider the function $S_A(w, y)$, which is computed as follows: it examines w to determine if w is a Boolean formula, B, in CNF. If not $r=0$. If yes, let $x_1, \cdots, x_r$ be variables appearing in B (or at least including every variable in B). The value of r can be determined in p-time. Let y denote a binary string and let $y(j)$ be the jth digit of this string. Let

$$S_A(w, y)=w\wedge(x_{r+1}\vee\neg x_{r+1})\wedge z_1\wedge z_2\wedge\cdots\wedge z_n$$

where z_j is the literal

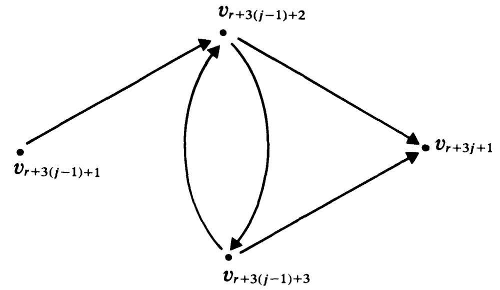
$$z_j=\begin{cases} x_{r+1+j} & \text{if } y(j)=1,\\ \neg x_{r+1+j} & \text{if } y(j)=0.\end{cases}$$

Thus, w is satisfiable iff $S_A(w, y)$ is satisfiable. $S_A(-,-)$ is clearly p-time computable and a function D_A which examines a string to determine if it has a suffix of the proper form and if so translates it appropriately will also be in p-time. $S_A(-,-)$ and $D_A(-)$ together satisfy Lemma 5. The padding function needed for Lemma 3 is defined by $Z_A(w)=S_A(w, 0^{|w|^2+1})$, which clearly satisfies Lemma 3.

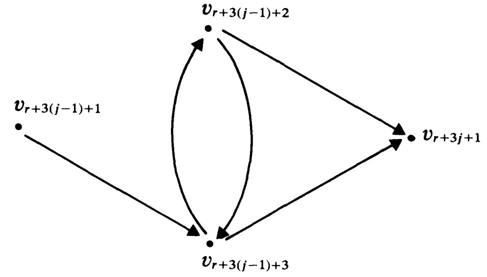
We now show INEQ satisfies Lemma 5 and so is p-time isomorphic to CNF-SAT: $S_A(w, y)$ first checks that w has the right format, $w=R_1\neq R_2$ when R_1 and R_2 are encodings of regular expressions. If so it outputs $(R_1+0^n1y)\neq(R_2+0^n1y)$ [where $n=|R_1\neq R_2|$] if not it outputs $w\#\#y$. The obvious D_A works. So INEQ is p-isomorphic to CNF-SAT. For the other sets we proceed as follows.

UNIV. $S_A(w, y)$ encodes y in inaccessible new states of M_i and adjusts #'s at end to accommodate new states.

CLIQUE. $S_A(w, y)$ checks that w has format $k\# G$, where G is the encoding of some graph, determines highest labeled vertex, r, used in G. G has vertices v_1,



Section of graph indicating $y(j)=1$



Section of graph indicating $y(j)=0$

FIG. 2. *Encodings in Hamilton circuit*

$v_2, \cdots, v_r$. S_A outputs $(k+1)\#G'$ where G' has vertices $v_1, v_2, \cdots, v_r, v_{r+1}, v_{r+2}, \cdots, v_{r+2|y|+1}$ where G' has the edges of G plus $\forall j \leqq r$, $\forall i \leqq |y|$ G' contains edges (v_j, v_{r+2i}) if $y(i)=1$ and (v_j, v_{r+2i-1}) if $y(i)=0$, and $(v_j, v_{r+2|y|+1})$. This S_A and the obvious D_A work.

HAMILTON CIRCUIT. $S_A(w, y)$ checks that format is correct and inserts vertices, where r is the highest numbered vertex in G, v_{r+1}, v_{r+2}, $v_{r+3}, \cdots, v_{r+3|y|+1}$, an edge from $v_r \to v_{r+1}$ and $\forall j \leqq |y|$ inserts edges

$$(v_{r+3(j-1)+k}, v_{r+3j+1}) \quad \text{for } k=2,3,$$
$$(v_{r+3(j-1)+2}, v_{r+3(j-1)+3}),$$
$$(v_{r+3(j-1)+3}, v_{r+3(j-1)+2}),$$

and

$$\text{if } y(j)=1 \qquad (v_{r+3(j-1)+1}, v_{r+3(j-1)+2}),$$
$$\text{if } y(j)=0 \qquad (v_{r+3(j-1)+1}, v_{r+3(j-1)+3})$$

and if $(v_r, v_i) \in G$ put $(v_{r+3|y|+1}, v_i) \in G'$. (See Fig. 2.)

Again the obvious D_A map also works. This completes the proof.

We note that in other known *NP* problems it is possible to encode the necessary information in a manner not affecting whether a given string is in the language. We note specifically that this technique shows that the "simplified" *NP* complete problems of Garey, Johnson, and Stockmeyer [5] are all p-time isomorphic.

One may argue that our isomorphisms are unnatural, that they were constructed through recursion theoretic techniques which are out of place in discussions of combinatorial problems. We will show, however, that with a little care our results yield not only isomorphisms between the various problems, but in fact, isomorphisms that preserve the underlying combinatorics.

Given a Boolean formula in CNF, we ask how many distinct variable assignments there are which produce a true value for the formula. Similarly, if we were given the encodings of a pair of regular expressions, $R_1 \# R_2$, we might ask how many strings are accepted by one and not the other. For a language L and a fixed $w \in L$, we will call each "piece of information" which evidences $w \in L$ a solution to the (w, L) problem. We use the following notation:

$$\text{Sol}\,(w, L) = \begin{cases} \{x \mid x \text{ encodes a solution to the } (w, L) \text{ problem}\}, \\ \phi \quad \text{if } w \notin L. \end{cases}$$

It is, in fact, solutions of the various problems which are of practical importance in computing. We are interested in the elements of Sol (w, L) and not merely whether $|\text{Sol}\,(w, L)| > 0$. It is of little use to a multi-process scheduling algorithm to know that there is a schedule of a given cost; the schedulor must determine the optimal schedule.

DEFINITION. If A and B are *NP* complete problems and $f: A \to B$ is a polynomial time reduction, we say that f is *parsimonious* if for all w, $|\text{Sol}\,(w, A)| = |\text{Sol}\,(f(w), B)|$.

Parsimonious reductions have been studied before [14] and it turns out that many of the well known *NP* complete problems are related by parsimonious reductions. We feel that parsimonious reductions should be considered natural since they do not introduce "new" solutions but yield translated problems whose solutions are in one-one correspondence with the solutions of the original problem.

We now state and prove our main result concerning parsimonious reductions:

THEOREM 9. *Let A be an NP complete set for which there exist parsimonious p-time reductions $f: A \to$ CNF-SAT and $g:$ CNF-SAT $\to A$. Let A have functions $S_A(-, -)$ and D_A as in Lemma* 5, *and furthermore assume that $\forall x \in \{0, 1\}^*$ $S_A(-, x): A \to A$ is parsimonious; then the isomorphism $\phi: A \to$ CNF-SAT guaranteed by Theorem* 1 *is parsimonious.*

Proof. We first note that the composition of parsimonious reductions is parsimonious. Unfortunately, the $S_{\text{CNF}}(-, -)$ function defined earlier is not parsimonious; however the function $S_{\text{CNF}}(w, y) = w \wedge (x_{r+1} \vee x_{r+1}) \wedge z_1 \wedge \cdots \wedge z_n$ with z_j as before is parsimonious.

Since $S_{\text{CNF}}(-, -)$ and $S_A\,(-, -)$ are both parsimonious we know, via Lemma 5 and the observation above that the f' and g' of Lemma 5 will in fact be parsimonious.

Again S_{CNF} gives us a padding function for CNF-SAT (the function is now parsimonious) which by Lemma 3 tells us the conditions of Theorems 1 are now satisfied. This time, however, all constituents of our isomorphism are parsimonious and since the isomorphism is constructed by application of these reductions, we have that the isomorphism is parsimonious.

We note that the encoding functions of the *NP* complete problems INEQ, UNIV, and CLIQUE are all parsimonious, and also that they are each related to CNF-SAT via parsimonious reductions [14]; therefore we have

THEOREM 10. *The following NP complete problems are p-time isomorphic via parsimonious mappings*:

1. CNF-SAT,
2. INEQ,
3. UNIV,
4. CLIQUE.

Proof. The proof is by Theorem 9.

We now turn our attention to languages complete for PSPACE. We again first define a number of PSPACE complete problems:

1. UNIV—$\{M_i \# \mathrm{CODE}(x_1 \cdots x_n) \#^{t|M_i|} | M_i$ accepts $x_1 \cdots x_n$ on $t+n$ tape squares$\}$ [7].

2. QBF—Given a quantified Boolean formula, e.g., $\forall x_1 \exists x_2 \forall x_3 (x_1 \vee \neg x_2) \wedge (x_1 \vee x_2 \vee x_3)$, is it true [11].

3. HEX—Given a graph and two distinguished vertices a game is defined in which the two players alternatively choose vertices. Player 1 wins if he is able to choose vertices which define a path in the graph between the two distinguished vertices. Player 2 wins otherwise [3]. The set HEX consists of all those graphs for which player 1 has a winning strategy.

4. $L_{\Sigma*}$—$\{R | R$ is a regular expression and $L(R) \neq \Sigma^*\}$ [11].

THEOREM 11. *The following* PSPACE *complete problems are p-time isomorphic*:

1. UNIV,
2. QBF,
3. HEX,
4. $L_{\Sigma*}$.

Proof. By the same method used to show CNF could be padded in Theorem 8 we see QBF has padding and $S_A(-, -)$, $D_A(-)$ functions.

S_{UNIV} encodes the second argument in inaccessible states as before. S_{HEX} encodes the second argument in dead end paths. $S_{\Sigma*}(x, y) = [y + (\Lambda + 0 + 1)^n + (0+1)^{n+1} x]$, $n = |y|$. In all cases the obvious $D(-)$ function works.

We now prove a metatheorem which extends the previous result to tape complete decision problems concerning regular expressions. Define

$$x \backslash L = \{w | xw \in L\} \quad \text{and} \quad L/x = \{w | wx \in L\}.$$

THEOREM 12. *Let P be any predicate in the regular sets over* $\{0, 1\}$ *such that*

1. $P(\{0, 1\}^*) = \mathrm{TRUE}$,

2. $P_L = \bigcup_{x \in \{0,1\}^*} \{x \backslash L | P(L) = \mathrm{TRUE}\}$[*or* $P_R = \bigcup_{x \in \{0,1\}^*} \{L/x | P(L) = T\}$] *is not the set of all regular sets over* $\{0, 1\}$,

3. *$L_P = \{R \mid R$ is a regular expression over $\{0, 1\}$ and $P(L(R)) =$* FALSE*$\}$ is in* PTAPE.

Then L_P is p-time isomorphic to L_{Σ^*}.

Proof. Any L_P where P satisfies conditions 1 and 2 above is PTAPE-hard [7] and 3 above then guarantees L_P is PTAPE complete.

In order to show the isomorphism we must find S and D. Let ϕ be the p-time map such that $R \in L_P$ iff $\phi(R) \in L_{\Sigma^*}$ and let L_0 be a regular set over $\{0, 1\}$ not in P_L as in [7]; define $h_0(0) = 00$ and $h_0(1) = 01$. We note that the map

$$R_i \overset{\psi}{\rightarrow} h_0(R_i) \cdot 10 \cdot (0+1)^* + (00+01)^* \cdot 10 \cdot R_{L_0}$$
$$+ (00+01)^* \cdot [\Lambda + 0 + 1 + 11(0+1)^*]$$

has the properties:

(i) $R_i \in L_{\Sigma^*}$ iff $\psi(R_i) \in L_P$,

(ii) ψ is p-time invertible.

We now define $S_P(x, y) = \psi(S_{\Sigma^*}(\phi(x), y))$; we note $D_P(x) = D_{\Sigma^*}(\psi^{-1}(z))$ is the required D function. Since L_P is PSPACE complete and satisfies Lemma 5 the sets L_P and L_{Σ^*} are p-time isomorphic, as was to be shown.

5. Density considerations. We recall that a proof that no p-sparse set can be *NP* complete would imply that $P \neq NP$. We cannot solve this problem but we can show that some other complete and hardest sets cannot be p-sparse.

We prove next that complete sets for EXPTIME and EXPSPACE cannot be p-sparse.[1] Furthermore, using a recent result from [9] we show that the hardest context-sensitive languages are not p-sparse. Thus showing that a single letter alphabet language cannot be a hardest csl. We also conjecture that the hardest context-free languages cannot be p-sparse. The following result was first obtained by A. R. Meyer.

THEOREM 13. *No p-sparse language A can be complete in* EXPTIME *or* EXPTAPE. *Thus $A \subseteq a^*$ cannot be complete in* EXPTIME *or* EXPTAPE.

Proof. We will prove this result by constructing a set A_0 with the following properties:

(a) A_0 is in EXPTIME,

(b) A_0 is not p-sparse,

(c) If A_0 is p-reduced to a set B by the mapping ρ then ρ is a one-one mapping almost everywhere.

We now describe a TM, M, which accepts A_0. M will be a multitape TM which on input w computes as follows:

1. on one of its tapes M writes down $1 \# 10 \# 11 \# \cdots \# |w| \#$. Each integer will be treated as the encoding of a transducer, T_i and $T_i(x)$ will be limited to $|x|^i$ steps so this list will eventually cover all polynomial time bounded transducers. The list can be written down in time $O(|w|^3)$ so there is some c such that for all $n \geqq 1$ the time required to carry out step 1 on input of length n is less than 2^{cn}.

[1] The second author has recently shown that sets complete for EXPTIME and EXPSPACE cannot be a.e. complex.

```
2. for $i = 1$ to $|w|$
       for each $x$ such that $2^i \leqq x < w$ do
       (a) compute $|x|^i$ for $2^{|x|}$ steps.
       (b) if computation (a) completed then:
           (I) compute $T_i(x)$ for $|x|^i$ simulated steps or
           (II) compute $T_i(x)$ for $2^{|x|}$ actual steps whichever comes first
       (c) if case I above occurred store $(x, i, T_i(x))$ on a storage tape
```

(since (a), (b), and (c) are each limited to $2^{|w|}$ steps and they must be carried out at most $w2^{|w|}$ times there is some c' such that step 2 can be completed within $2^{c'|w|}$. Also for large $|x|$ case II never occurs).

```
3. Construct two lists $L_1$ and $L_2$ as follows:
   for $i = 1$ to $|w|$
     find (if it exists) the smallest $x < w$ which satisfies the following two
     conditions:
     (a) there is some $y < x$ for which $T_i(x) = T_i(y)$ ($T_i(x)$ and $T_i(y)$ must be
     listed in step 2)
     (b) for all $z$, $(x, z)$ is not on $L_1$,
     if such an $x$ is found, take smallest $y$ for which $T_i(x) = T_i(y)$ and put $(x, y)$
     on $L_1$; if no such $x$ is found put $i$ on $L_2$.
```

(The length of $T_i(x)$ is less than $2^{|x|}$ and comparisons on a multi-tape machine can be done in linear time so step 3 can be carried out in time $2^{c''|w|}$ for some c'').

```
4. In ascending order, for each $i \in L_2$:
   for $x = w$ perform steps 2(a) and 2(b). If computations complete find
   smallest $y < w$ for which $T_i(y) = T_i(w)$, determines length of longest chain
   $(y, x_1)(x_1, x_2) \cdots (x_{l-1}, x_l)$ entirely on $L_1$, $M$ accepts $w$ iff for the first $i$ for
   which a $y$ is found $l = 2m + 1$ for some $m$ (if the chain is empty, do not
   accept) if no such $y$ has been found for any $i$ on list $L_2$ then $M$ accepts $w$.
```

This last step can also be carried out in exponential time and so the entire machine has $T(M)$ in DEXPTIME.

It should be clear that for every polynomial time bounded machine T_i there is some integer n_i so that for all x, $|x| > n_i$, the simulation of T_i on x will be completed within $2^{|x|}$ steps and that therefore no p-time reduction, f, for which there are infinitely many pairs (x_i, y_i) with $f(x_i) = f(y_i)$ can reduce A_0.

Note that step 3 in the process constructs the past history of M relevant to M's action on w.

Since for every EXPTIME (and EXPTAPE) complete set, C, we know there must be some p-time reduction $f: A_0 \rightarrow C$ and since A_0 contains about $2^n - n$ elements of length less than or equal to n, we have that there is some n_c for which $|\{x | x \in C \text{ and } |x| < r^{n_c}\}| > 2^r$. This immediately implies that no p-sparse set can be EXPTIME or EXPTAPE complete. This completes the proof.

We now return our attention to hardest context-sensitive languages.

LEMMA 14. *There exists a recursive function σ such that for all linear time transducers M_i, the machine $M_{\sigma(i)}$ is a* 3 *tape transducer satisfying*:

1. $\forall x M_i(x) = M_{\sigma(i)}(x)$,

2. $\exists c$ *depending only on σ such that $M_{\sigma(i)}$ never scans more than $c|M_i|(|x|/\log|x|)$ squares on its work tape when processing* x.

Proof. The proof follows from efficient simulation techniques in [9].

We can now make use of the above transducer to enable us to diagonalize over linear time transductions on linear tape and get the following result.

THEOREM 15. *The hardest context-sensitive languages cannot be p-sparse. Thus no language over a single letter alphabet can be a hardest* csl.

Proof. We describe a TM M which accepts a csl which is not p-sparse and such that any linear-time reduction of this language to another language must be one-one almost everywhere. M behaves as follows on input w, $w \in \Sigma^*$, $|\Sigma| > 1$:

1. writes down as many of the $n/\log n$ space bounded transducers which simulate the linear-time transducers as possible on linear tape,

2. determines which of the transducers M_i have been eliminated while processing $x < w$, (i.e. M recomputes what it did for all previous inputs and keeps a list of the transducers which were eliminated),

3. for each M_i listed but not eliminated in increasing order M find smallest $x < w$ such that

(i) $M_i(x) = M_i(w)$,

(ii) $M_i(x) = M_i(w)$ can each be computed (although not necessarily written down) in $|w|$ tape.

If such an x is found, eliminate M_i and w is accepted iff x was rejected. If no such x is found for any i accept w.

The set accepted by this TM clearly has the property that if $f: T(M) \rightarrow A$ is a linear time reduction of $T(M)$ to A then f is one-one a.e. Since for every n there are at least 2^n inputs of length n and at most n transducers have been checked, we see that $T(M)$ is not a p-sparse set.

This shows that no hardest csl's can be sla languages nor can they be p-sparse, as was to be shown.

These results can easily be extended to the following:

COROLLARY 16. *Let $L(n) \geqq n$ be tape constructable. Then the hardest language for* TAPE$[L(n)]$ *cannot be p-sparse. Let $L(n)$ be tape constructable and such that for every k, $k \geqq 0$,*

$$\lim_{n \to \infty} \frac{n^k}{L(n)} = 0,$$

then the complete languages of $L(n)$ cannot be p-sparse.

Note that the question of sparseness for hardest cfl's has implications for the question of linear time recognition of cfl's.

We conclude this section by deriving a result due to A. R. Meyer which relates the complexity of switching circuits for the recognition of truncated *NP* complete problems to the sparseness of oracle sets for the recognition of these problems. For the set A, $A \subseteq \{0, 1\}^*$, let $C_A(n)$ be the size (number of gates) of the smallest switching circuit which realizes the function

$$f_A^{(n)}: \{0, 1\}^n \rightarrow \{0, 1\}$$

such that $f_A^{(n)}(w) = 1$ iff $w \in A \cap \Sigma^n$.

It is easily seen that if for some *NP* complete language A the function $C_A(n)$ cannot be bounded by a polynomial in n, then $P \neq NP$. As a matter of fact, considerable effort has been expended in trying to prove that $C_A(n)$ cannot be bounded by a polynomial when A is the clique problem.

THEOREM 17. *For a set A the function C_A is bounded by a polynomial iff there exists a sparse set B such that a Turing machine with oracle set B can recognize A in deterministic polynomial time.*

Proof. Assume that B is a sparse oracle set with which the TM M_A can recognize A in deterministic polynomial time. Then, since B is sparse, during the processing of inputs of length n, $n = 1, 2, \cdots, M_A$ can consult only polynomially many different members of B and therefore we can construct for each n a polynomially complex circuit which simulates the computation of M_A on w, $|w| = n$. Thus C_A is polynomially bounded.

Conversely, if there exist polynomially complex circuits which for each n recognize the set $A \cap \Sigma^n$, then we can use these circuits to get a sparse oracle set B as follows. Encode for each n all (polynomially many) "prefixes" of the description of nth circuit so that with polynomially many oracle questions the TM can reconstruct the nth circuit description. Thus for input w the TM constructs in polynomial time the nth circuit, $n = |w|$, and then in polynomial time checks whether the input w yields "one" as output. Thus A can be recognized in p-time by means of a sparse oracle B, as was to be shown.

From the above theorem, we see immediately that no set complete for EXPTAPE can be recognized in p-time by a TM with a sparse oracle. At this time it is still open whether sets from NP or EXPTIME can be recognized in deterministic polynomial time by sparse oracle sets [15].

6. Conclusion. A number of interesting and apparently difficult problems suggest themselves immediately from this work. As we have noted, if all NP complete problems are p-isomorphic then $P \neq NP$ and if all PTAPE complete problems are p-isomorphic then $P \neq$ PTAPE. Thus, the question whether all NP and PTAPE complete sets, respectively, are p-isomorphic could be a very important and hard question. Similarly, the problem about the existence of sparse complete sets for NP and PTAPE seems very difficult and could help solve the $P = NP =$ PTAPE? problem.

What about EXPTIME and EXPTAPE complete problems? We known that they cannot be sparse; are they all p-isomorphic? Similarly, are hardest context-sensitive languages all p-isomorphic?

Appendix. Our work shows that the invertibility of p-time reductions is an important question. As we noted earlier, if all NP complete sets are related by invertible maps then $P \neq NP$. In this section, we mention some questions related to the invertibility of reductions of various types.

The first question we ask is easily answered:

I. Do all one to one polynomial time computable maps have inverses which are computable in polynomial time?

The answer to this is no; however, all known counterexamples to this statement have the property that on some sequence of inputs, the outputs are more than a polynomial amount shorter than the corresponding inputs. We are primarily interested in mappings which do not alter lengths by more than a polynomial amount and our next two questions are designed to limit changes in length.

II. If f is a polynomial time length nondecreasing bijection, is f^{-1} computable in polynomial time?

III. If f is a polynomial time length increasing injection is f^{-1} (extended by * to be total) computable in polynomial time?

We have partial answers to these questions. (These results follow immediately from results of Gary Miller [12].)

$$\text{If } P = NP \cap \text{co} - Np \text{ then II.}$$

$$\text{If } P = NP \text{ then III.}$$

Whether or not the converses of the above statements hold is an interesting open question.

As observed above polynomial time maps need not have polynomial time inverses, one might, however, hope for the following:

IV. If f is a bijection which p-reduces A to B, is B p-reducible to A?

However, the answer to this question is no also. Let $A \subseteq (0+1)^*$ be a set in DSPACE(2^n) which requires 2^n space almost everywhere. Let $B = \{0^j|\log(\log(\log j))$ in $A\}$. Let $X = \{1w|w$ in $A\}$. Then there is a bijection $f:(0+1)^* \to (0+1)^*$ which reduces B to X; however, there can be no polynomial time reduction of X to B.

The final question we raise here concerns the comparison of one-one and isomorphic equivalences.

V. If A and B are one to one polynomial time equivalent, are they p-isomorphic?

A solution to this question would tell us how much our Theorem 1 can be strengthened.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] S. A. Cook, *The complexity of theorems proving procedures*, Proc. 3rd Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1971, pp. 151–158.

[3] S. Even and R. E. Tarjan, *A combinatorial problem which is complete in polynomial space*, Proc. 7th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1975, pp. 66–71.

[4] Z. Galil, *The complexity of resolution procedures for theorem proving in the propositional calculus*, Rep. TR 75-239, Cornell University, Ithaca, NY, 1975.

[5] M. R. Garey, D. S. Johnson and L. Stockmeyer, *Some simplified polynomial complete problems*, Proc. 6th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1974, pp. 47–63.

[6] S. A. Greibach, *The hardest context-free language*, this Journal, 1973, pp. 304–310.

[7] J. Hartmanis and H. B. Hunt, III, *The LBA problem and its importance in the theory of computing*, SIAM-AMS Proceedings, vol. 7, American Mathematical Society, Providence, RI, 1974, pp. 1–26.

[8] J. Hartmanis and J. Simon, *On the structure of feasible computations*, Advances in Computers, vol. 14, M. Rubinoff and M. C. Yovits, eds., Academic Press, New York, 1976.

[9] J. E. Hopcroft, W. Paul and L. Valiant, *On time versus space and related problems*, IEEE 16th Annual Symp. on Foundations of Computer Science, IEEE, New York, 1975, pp. 57–64.

[10] R. M. KARP, *Reducibilities among combinatorial problems*, Complexity of Computer Computations, R. Miller and J. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.

[11] A. R. MEYER AND L. STOCKMEYER, *The equivalence problem for regular expressions with squaring requires exponential space*, IEEE 13th Annual Symp. on Switching and Automata Theory, IEEE, New York, 1972, pp. 125–129.

[12] G. L. MILLER, *Riemann's Hypothesis and tests for primality*, Rep. CS 75–27, University of Waterloo, Waterloo, Ontario, 1975.

[13] S. SAHNI, *Some related problems from network flows, game theory, and integer programming*, IEEE 13th Annual Symp. on Switching and Automata Theory, IEEE, New York, 1972, pp. 130–138.

[14] J. SIMON, *On some central problems in computational complexity*, Rep. TR 75–224, Cornell University, Ithaca, NY, 1975.

[15] R. SOLOVAY, *On sets Cook reducible to sparse sets*, IBM Research Rep. RC5215, Yorktown Heights, NY, January 1975.

[16] L. J. STOCKMEYER, *The complexity of decision problems in automata theory and logic*, Ph.D. dissertation, Mass. Inst. of Tech., Cambridge, MA, July 1974.

[17] J. D. ULLMAN, *Polynomial complete scheduling problems*, Proc. 4th Annual ACM Symp. on Operating Systems Principles, Association for Computing Machinery, New York, 1973, pp. 96–101.

SIAM J. COMPUT.
Vol. 6, No. 2, June 1977

FAST PATTERN MATCHING IN STRINGS*

DONALD E. KNUTH†, JAMES H. MORRIS, JR.‡ AND VAUGHAN R. PRATT¶

Abstract. An algorithm is presented which finds all occurrences of one given string within another, in running time proportional to the sum of the lengths of the strings. The constant of proportionality is low enough to make this algorithm of practical use, and the procedure can also be extended to deal with some more general pattern-matching problems. A theoretical application of the algorithm shows that the set of concatenations of even palindromes, i.e., the language $\{\alpha\alpha^R\}^*$, can be recognized in linear time. Other algorithms which run even faster on the average are also considered.

Key words. pattern, string, text-editing, pattern-matching, trie memory, searching, period of a string, palindrome, optimum algorithm, Fibonacci string, regular expression

Text-editing programs are often required to search through a string of characters looking for instances of a given "pattern" string; we wish to find all positions, or perhaps only the leftmost position, in which the pattern occurs as a contiguous substring of the text. For example, *c a t e n a r y* contains the pattern *t e n*, but we do not regard *c a n a r y* as a substring.

The obvious way to search for a matching pattern is to try searching at every starting position of the text, abandoning the search as soon as an incorrect character is found. But this approach can be very inefficient, for example when we are looking for an occurrence of *a a a a a a a b* in *a a a a a a a a a a a a a a b*. When the pattern is a^nb and the text is $a^{2n}b$, we will find ourselves making $(n+1)^2$ comparisons of characters. Furthermore, the traditional approach involves "backing up" the input text as we go through it, and this can add annoying complications when we consider the buffering operations that are frequently involved.

In this paper we describe a pattern-matching algorithm which finds all occurrences of a pattern of length m within a text of length n in $O(m+n)$ units of time, without "backing up" the input text. The algorithm needs only $O(m)$ locations of internal memory if the text is read from an external file, and only $O(\log m)$ units of time elapse between consecutive single-character inputs. All of the constants of proportionality implied by these "O" formulas are independent of the alphabet size.

* Received by the editors August 29, 1974, and in revised form April 7, 1976.

† Computer Science Department, Stanford University, Stanford, California 94305. The work of this author was supported in part by the National Science Foundation under Grant GJ 36473X and by the Office of Naval Research under Contract NR 044-402.

‡ Xerox Palo Alto Research Center, Palo Alto, California 94304. The work of this author was supported in part by the National Science Foundation under Grant GP 7635 at the University of California, Berkeley.

¶ Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. The work of this author was supported in part by the National Science Foundation under Grant GP-6945 at University of California, Berkeley, and under Grant GJ-992 at Stanford University.

We shall first consider the algorithm in a conceptually simple but somewhat inefficient form. Sections 3 and 4 of this paper discuss some ways to improve the efficiency and to adapt the algorithm to other problems. Section 5 develops the underlying theory, and § 6 uses the algorithm to disprove the conjecture that a certain context-free language cannot be recognized in linear time. Section 7 discusses the origin of the algorithm and its relation to other recent work. Finally, § 8 discusses still more recent work on pattern matching.

1. Informal development. The idea behind this approach to pattern matching is perhaps easiest to grasp if we imagine placing the pattern over the text and sliding it to the right in a certain way. Consider for example a search for the pattern *a b c a b c a c a b* in the text *b a b c b a b c a b c a a b c a b c a b c a c a b c*; initially we place the pattern at the extreme left and prepare to scan the leftmost character of the input text:

```
a b c a b c a c a b
b a b c b a b c a b c a a b c a b c a b c a c a b c
↑
```

The arrow here indicates the current text character; since it points to *b*, which doesn't match that *a*, we shift the pattern one space right and move to the next input character:

```
  a b c a b c a c a b
b a b c b a b c a b c a a b c a b c a b c a c a b c
  ↑
```

Now we have a match, so the pattern stays put while the next several characters are scanned. Soon we come to another mismatch:

```
  a b c a b c a c a b
b a b c b a b c a b c a a b c a b c a b c a c a b c
        ↑
```

At this point we have matched the first three pattern characters but not the fourth, so we know that the last four characters of the input have been *a b c x* where $x \neq a$; we don't have to remember the previously scanned characters, since *our position in the pattern yields enough information to recreate them.* In this case, no matter what x is (as long as it's not a), we deduce that the pattern can immediately be shifted four more places to the right; one, two, or three shifts couldn't possibly lead to a match.

Soon we get to another partial match, this time with a failure on the eighth pattern character:

```
          a b c a b c a c a b
b a b c b a b c a b c a a b c a b c a b c a c a b c
                        ↑
```

Now we know that the last eight characters were *a b c a b c a x*, where $x \neq c$. The pattern should therefore be shifted three places to the right:

```
                a b c a b c a c a b
b a b c b a b c a b c a a b c a b c a b c a c a b c
                        ↑
```

We try to match the new pattern character, but this fails too, so we shift the pattern four (not three or five) more places. That produces a match, and we continue scanning until reaching *another* mismatch on the eighth pattern character:

```
                        a b c a b c a c a b
b a b c b a b c a b c a a b c a b c a b c a c a b c
                                      ↑
```

Again we shift the pattern three places to the right; this time a match is produced, and we eventually discover the full pattern:

```
                              a b c a b c a c a b
b a b c b a b c a b c a a b c a b c a b c a c a b c
                                                ↑
```

The play-by-play description for this example indicates that the pattern-matching process will run efficiently if we have an auxiliary table that tells us exactly how far to slide the pattern, when we detect a mismatch at its *j*th character *pattern*[*j*]. Let *next*[*j*] be the character position in the pattern which should be checked next after such a mismatch, so that we are sliding the pattern $j - next[j]$ places relative to the text. The following table lists the appropriate values:

$j=$	1	2	3	4	5	6	7	8	9	10
pattern[*j*] =	*a*	*b*	*c*	*a*	*b*	*c*	*a*	*c*	*a*	*b*
next[*j*] =	0	1	1	0	1	1	0	5	0	1

(Note that *next*[*j*] = 0 means that we are to slide the pattern all the way *past* the current text character.) We shall discuss how to precompute this table later; fortunately, the calculations are quite simple, and we will see that they require only $O(m)$ steps.

At each step of the scanning process, we move either the text pointer or the pattern, and each of these can move at most *n* times; so at most $2n$ steps need to be performed, after the *next* table has been set up. Of course the pattern itself doesn't really move; we can do the necessary operations simply by maintaining the pointer variable *j*.

2. Programming the algorithm. The pattern-match process has the general form

```
place pattern at left;
while pattern not fully matched
    and text not exhausted do
    begin
        while pattern character differs from
                current text character
                do shift pattern appropriately;
        advance to next character of text;
    end;
```

For convenience, let us assume that the input text is present in an array $text[1:n]$, and that the pattern appears in $pattern[1:m]$. We shall also assume that $m>0$, i.e., that the pattern is nonempty. Let k and j be integer variables such that $text[k]$ denotes the current text character and $pattern[j]$ denotes the corresponding pattern character; thus, the pattern is essentially aligned with positions $p+1$ through $p+m$ of the text, where $k=p+j$. Then the above program takes the following simple form:

```
$j := k := 1$;
while $j \leqq m$ and $k \leqq n$ do
    begin
        while $j > 0$ and $text[k] \neq pattern[j]$
            do $j := next[j]$;
        $k := k+1; j := j+1$;
    end;
```

If $j>m$ at the conclusion of the program, the leftmost match has been found in positions $k-m$ through $k-1$; but if $j \leqq m$, the text has been exhausted. (The **and** operation here is the "conditional and" which does not evaluate the relation $text[k] \neq pattern[j]$ unless $j>0$.) The program has a curious feature, namely that the inner loop operation "$j := next[j]$" is performed no more often than the outer loop operation "$k := k+1$"; in fact, the inner loop is usually performed somewhat *less* often, since the pattern generally moves right less frequently than the text pointer does.

To prove rigorously that the above program is correct, we may use the following invariant relation: "Let $p=k-j$ (i.e., the position in the text just preceding the first character of the pattern, in our assumed alignment). Then we have $text[p+i]=pattern[i]$ for $1 \leqq i<j$ (i.e., we have matched the first $j-1$ characters of the pattern, if $j>0$); but for $0 \leqq t<p$ we have $text[t+i] \neq pattern[i]$ for some i, where $1 \leqq i \leqq m$ (i.e., there is no possible match of the entire pattern to the left of p)."

The program will of course be correct only if we can compute the *next* table so that the above relation remains invariant when we perform the operation $j := next[j]$. Let us look at that computation now. When the program sets

$j := \mathit{next}[j]$, we know that $j > 0$, and that the last j characters of the input up to and including $\mathit{text}[k]$ were

$$\mathit{pattern}[1] \dots \mathit{pattern}[j-1]\, x$$

where $x \neq \mathit{pattern}[j]$. What we want is to find the least amount of shift for which these characters can possibly match the shifted pattern; in other words, we want $\mathit{next}[j]$ to be the largest i less than j such that the last i characters of the input were

$$\mathit{pattern}[1] \dots \mathit{pattern}[i-1]\, x$$

and $\mathit{pattern}[i] \neq \mathit{pattern}[j]$. (If no such i exists, we let $\mathit{next}[j] = 0$.) With this definition of $\mathit{next}[j]$ it is easy to verify that $\mathit{text}[t+1] \dots \mathit{text}[k] \neq \mathit{pattern}[1] \dots \mathit{pattern}[k-1]$ for $k - j \leqq t < k - \mathit{next}[j]$; hence the stated relation is indeed invariant, and our program is correct.

Now we must face up to the problem we have been postponing, the task of calculating $\mathit{next}[j]$ in the first place. This problem would be easier if we didn't require $\mathit{pattern}[i] \neq \mathit{pattern}[j]$ in the definition of $\mathit{next}[j]$, so we shall consider the easier problem first. Let $f[j]$ be the largest i less than j such that $\mathit{pattern}[1] \dots \mathit{pattern}[i-1] = \mathit{pattern}[j-i+1] \dots \mathit{pattern}[j-1]$; since this condition holds vacuously for $i = 1$, we always have $f[j] \geqq 1$ when $j > 1$. By convention we let $f[1] = 0$. The pattern used in the example of § 1 has the following f table:

$j=1$	2	3	4	5	6	7	8	9	10
$\mathit{pattern}[j]=a$	b	c	a	b	c	a	c	a	b
$f[j]=0$	1	1	1	2	3	4	5	1	2.

If $\mathit{pattern}[j] = \mathit{pattern}[f[j]]$ then $f[j+1] = f[j]+1$; but if not, we can use essentially the same pattern-matching algorithm as above to compute $f[j+1]$, with $\mathit{text} = \mathit{pattern}$! (Note the similarity of the $f[j]$ problem to the invariant condition of the matching algorithm. Our program calculates the largest j less than or equal to k such that $\mathit{pattern}[1] \dots \mathit{pattern}[j-1] = \mathit{text}[k-j+1] \dots \mathit{text}[k-1]$, so we can transfer the previous technology to the present problem.) The following program will compute $f[j+1]$, assuming that $f[j]$ and $\mathit{next}[1], \dots, \mathit{next}[j-1]$ have already been calculated:

```
$t := f[j]$;
while $t > 0$ and $pattern[j] \neq pattern[t]$
    do $t := next[t]$;
$f[j+1] := t+1$;
```

The correctness of this program is demonstrated as before; we can imagine two copies of the pattern, one sliding to the right with respect to the other. For example, suppose we have established that $f[8] = 5$ in the above case; let us consider the computation of $f[9]$. The appropriate picture is

```
      a b c a b c a c a b
a b c a b c a c a b
              ↑
```

Since $pattern[8] \neq b$, we shift the upper copy right, knowing that the most recently scanned characters of the lower copy were $a\,b\,c\,a\,x$ for $x \neq b$. The *next* table tells us to shift right four places, obtaining

```
              a b c a b c a c a b
a b c a b c a c a b
              ↑
```

and again there is no match. The next shift makes $t = 0$, so $f[9] = 1$.

Once we understand how to compute f, it is only a short step to the computation of $next[j]$. A comparison of the definitions shows that, for $j > 1$,

$$next[j] = \begin{cases} f[j], & \text{if } pattern[j] \neq pattern[f[j]]; \\ next[f[j]], & \text{if } pattern\ [j] = pattern[f[j]]. \end{cases}$$

Therefore we can compute the *next* table as follows, without ever storing the values of $f[j]$ in memory.

```
$j := 1; t := 0; next[1] := 0;$
while $j < m$ do
    begin comment $t = f[j]$;
        while $t > 0$ and $pattern[j] \neq pattern[t]$
            do $t := next[t]$;
        $t := t + 1; j := j + 1$;
        if $pattern[j] = pattern[t]$
        then $next[j] := next[t]$
        else $next[j] := t$;
    end.
```

This program takes $O(m)$ units of time, for the same reason as the matching program takes $O(n)$: the operation $t := next[t]$ in the innermost loop always shifts the upper copy of the pattern to the right, so it is performed a total of m times at most. (A slightly different way to prove that the running time is bounded by a constant times m is to observe that the variable t starts at 0 and it is increased, $m-1$ times, by 1; furthermore its value remains nonnegative. Therefore the operation $t := next[t]$, which always decreases t, can be performed at most $m-1$ times.)

To summarize what we have said so far: Strings of text can be scanned efficiently by making use of two ideas. We can precompute "shifts", specifying how to move the given pattern when a mismatch occurs at its jth character; and this precomputation of "shifts" can be performed efficiently by using the same principle, shifting the pattern against itself.

3. Gaining efficiency. We have presented the pattern-matching algorithm in a form that is rather easily proved correct; but as so often happens, this form is not very efficient. In fact, the algorithm as presented above would probably not be competitive with the naive algorithm on realistic data, even though the naive algorithm has a worst-case time of order m times n instead of m plus n, because the chance of this worst case is rather slim. On the other hand, a well-implemented form of the new algorithm should go noticeably faster because there is no backing up after a partial match.

It is not difficult to see the source of inefficiency in the new algorithm as presented above: When the alphabet of characters is large, we will rarely have a partial match, and the program will waste a lot of time discovering rather awkwardly that $text[k] \neq pattern[1]$ for $k = 1, 2, 3, \ldots$. When $j = 1$ and $text[k] \neq pattern[1]$, the algorithm sets $j := next[1] = 0$, then discovers that $j = 0$, then increases k by 1, then sets j to 1 again, then tests whether or not 1 is $\leqq m$, and later it tests whether or not 1 is greater than 0. Clearly we would be much better off making $j = 1$ into a special case.

The algorithm also spends unnecessary time testing whether $j > m$ or $k > n$. A fully-matched pattern can be accounted for by setting $pattern[m+1] =$ "@" for some impossible character @ that will never be matched, and by letting $next[m+1] = -1$; then a test for $j < 0$ can be inserted into a less-frequently executed part of the code. Similarly we can for example set $text[n+1] =$ "$\perp$" (another impossible character) and $text[n+2] = pattern[1]$, so that the test for $k > n$ needn't be made very often. (See [17] for a discussion of such more or less mechanical transformations on programs.)

The following form of the algorithm incorporates these refinements.

```
     a := pattern[1];
     pattern[m+1] := '@' ; next[m+1] := −1;
     text[n+1] := '⊥' ; text[n+2] := a;
     j := k := 1;
get started: comment j = 1;
     while text[k] ≠ a do k := k+1;
     if k > n then go to input exhausted;
char matched: j := j+1; k := k+1;
loop: comment j > 0;
     if text[k] = pattern[j] then go to char matched;
     j := next[j];
     if j = 1 then go to get started;
     if j = 0 then
          begin
               j := 1; k := k+1;
               go to get started;
          end;
     if j > 0 then go to loop;
comment text[k − m] through text[k − 1] matched;
```

This program will usually run faster than the naive algorithm; the worst case occurs when trying to find the pattern $a\,b$ in a long string of a's. Similar ideas can be used to speed up the program which prepares the *next* table.

In a text-editor the patterns are usually short, so that it is most efficient to translate the pattern directly into machine-language code which implicitly contains the *next* table (cf. [3, Hack 179] and [24]). For example, the pattern in § 1 could be compiled into the machine-language equivalent of

```
L0:  $k := k+1$;
L1:  if $text[k] \neq a$ then go to L0;
     $k := k+1$;
     if $k > n$ then go to input exhausted;
L2:  if $text[k] \neq b$ then go to L1;
     $k := k+1$;
L3:  if $text[k] \neq c$ then go to L1;
     $k := k+1$;
L4:  if $text[k] \neq a$ then go to L0;
     $k := k+1$;
L5:  if $text[k] \neq b$ then go to L1;
     $k := k+1$;
L6:  if $text[k] \neq c$ then go to L1;
     $k := k+1$;
L7:  if $text[k] \neq a$ then go to L0;
     $k := k+1$;
L8:  if $text[k] \neq c$ then go to L5;
     $k := k+1$;
L9:  if $text[k] \neq a$ then go to L0;
     $k := k+1$;
L10: if $text[k] \neq b$ then go to L1;
     $k := k+1$;
```

This will be slightly faster, since it essentially makes a special case for *all* values of j.

It is a curious fact that people often think the new algorithm will be slower than the naive one, even though it does less work. Since the new algorithm is conceptually hard to understand at first, by comparison with other algorithms of the same length, we feel somehow that a computer will have conceptual difficulties too—we expect the machine to run more slowly when it gets to such subtle instructions!

4. Extensions. So far our programs have only been concerned with finding the leftmost match. However, it is easy to see how to modify the routine so that all matches are found in turn: We can calculate the *next* table for the extended pattern of length $m+1$ using $pattern[m+1] =$ "@", and then we set $resume := next[m+1]$ before setting $next[m+1]$ to -1. After finding a match and doing whatever action is desired to process that match, the sequence

$$j := resume;\ \textbf{go to}\ \text{loop};$$

will restart things properly. (We assume that *text* has not changed in the meantime. Note that *resume* cannot be zero.)

Another approach would be to leave $next[m+1]$ untouched, never changing it to -1, and to define integer arrays $head[1:m]$ and $link[1:n]$ initially zero, and to insert the code

$$link[k] := head[j];\ head[j] := k;$$

at label “char matched”. The test “**if** $j>0$ **then**” is also removed from the program. This forms linked lists for $1\leq j\leqq m$ of all places where the first j characters of the pattern (but no more than j) are matched in the input.

Still another straightforward modification will find the longest initial match of the pattern, i.e., the maximum j such that *pattern*[1] . . . *pattern*[j] occurs in *text*.

In practice, the text characters are often packed into words, with say b characters per word, and the machine architecture often makes it inconvenient to access individual characters. When efficiency for large n is important on such machines, one alternative is to carry out b independent searches, one for each possible alignment of the pattern's first character in the word. These searches can treat *entire words* as “supercharacters”, with appropriate masking, instead of working with individual characters and unpacking them. Since the algorithm we have described does not depend on the size of the alphabet, it is well suited to this and similar alternatives.

Sometimes we want to match two or more patterns in sequence, finding an occurrence of the first followed by the second, etc.; this is easily handled by consecutive searches, and the total running time will be of order n plus the sum of the individual pattern lengths.

We might also want to match two or more patterns in parallel, stopping as soon as any one of them is fully matched. A search of this kind could be done with multiple *next* and *pattern* tables, with one j pointer for each; but this would make the running time kn plus the sum of the pattern lengths, when there are k patterns. Hopcroft and Karp have observed (unpublished) that our pattern-matching algorithm can be extended so that the running time for simultaneous searches is proportional simply to n, plus the alphabet size times the sum of the pattern lengths. The patterns are combined into a “trie” whose nodes represent all of the initial substrings of one or more patterns, and whose branches specify the appropriate successor node as a function of the next character in the input text. For example, if there are four patterns $\{a\,b\,c\,a\,b,\ a\,b\,a\,b\,c,\ b\,c\,a\,c,\ b\,b\,c\}$, the trie is shown in Fig. 1.

node	substring	**if** *a*	**if** *b*	**if** *c*
0		1	7	0
1	*a*	1	2	0
2	*a b*	5	10	3
3	*a b c*	4	7	0
4	*a b c a*	1	*a b c a b*	*b c a c*
5	*a b a*	1	6	0
6	*a b a b*	5	10	*a b a b c*
7	*b*	1	10	8
8	*b c*	9	7	0
9	*b c a*	1	2	*b c a c*
10	*b b*	1	10	*b b c*

FIG. 1

Such a trie can be constructed efficiently by generalizing the idea we used to calculate *next*[j]; details and further refinements have been discussed by Aho and Corasick [2], who discovered the algorithm independently. (Note that this algorithm depends on the alphabet size; such dependence is inherent, if we wish to keep the coefficient of n independent of k, since for example the k patterns might each consist of a single unique character.) It is interesting to compare this approach to what happens when the LR(0) parsing algorithm is applied to the regular grammar $S \to a\,S \mid b\,S \mid c\,S \mid a\,b\,c\,a\,b \mid a\,b\,a\,b\,c \mid b\,c\,a\,c \mid b\,b\,c$.

5. Theoretical considerations. If the input file is being read in "real time", we might object to long delays between consecutive inputs. In this section we shall prove that the number of times $j := next[j]$ is performed, before k is advanced, is bounded by a function of the approximate form $\log_\phi m$, where $\phi = (1+\sqrt{5})/2 \approx 1.618\ldots$ is the golden ratio, and that this bound is best possible. We shall use lower case Latin letters to represent characters, and lower case Greek letters $\alpha, \beta, \ldots$ to represent strings, with ε the empty string and $|\alpha|$ the length of α. Thus $|a| = 1$ for all characters a; $|\alpha\beta| = |\alpha| + |\beta|$; and $|\varepsilon| = 0$. We also write $\alpha[k]$ for the kth character of α, when $1 \leqq k \leqq |\alpha|$.

As a warmup for our theoretical discussion, let us consider the *Fibonacci strings* [14, exercise 1.2.8–36], which turn out to be especially pathological patterns for the above algorithm. The definition of Fibonacci strings is

$$(1) \qquad \phi_1 = b, \quad \phi_2 = a; \quad \phi_n = \phi_{n-1}\phi_{n-2} \quad \text{for } n \geqq 3.$$

For example, $\phi_3 = a\,b$, $\phi_4 = a\,b\,a$, $\phi_5 = a\,b\,a\,a\,b$. It follows that the length $|\phi_n|$ is the nth Fibonacci number F_n, and that ϕ_n consists of the first F_n characters of an infinite string ϕ_∞ when $n \geqq 2$.

Consider the pattern ϕ_8, which has the functions $f[j]$ and $next[j]$ shown in Table 1.

TABLE 1

$j=$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
$pattern[j]=$	a	b	a	a	b	a	b	a	a	b	a	a	b	a	b	a	a	b	a	b	a
$f[j]=$	0	1	1	2	2	3	4	3	4	5	6	7	5	6	7	8	9	10	11	12	8
$next[j]=$	0	1	0	2	1	0	4	0	2	1	0	7	1	0	4	0	2	1	0	12	0

If we extend this pattern to ϕ_∞, we obtain infinite sequences $f[j]$ and $next[j]$ having the same general character. It is possible to prove by induction that

$$(2) \qquad f[j] = j - F_{k-1} \quad \text{for } F_k \leqq j < F_{k+1},$$

because of the following remarkable near-commutative property of Fibonacci strings:

$$(3) \qquad \phi_{n-2}\phi_{n-1} = c(\phi_{n-1}\phi_{n-2}), \quad \text{for } n \geqq 3,$$

where $c(\alpha)$ denotes changing the two rightmost characters of α. For example, $\phi_6 = a\,b\,a\,a\,b \cdot a\,b\,a$ and $c(\phi_6) = a\,b\,a \cdot a\,b\,a\,a\,b$. Equation (3) is obvious when $n = 3$; and for $n > 3$ we have $c(\phi_{n-2}\phi_{n-1}) = \phi_{n-2}c(\phi_{n-1}) = \phi_{n-2}\phi_{n-3}\phi_{n-2} = \phi_{n-1}\phi_{n-2}$ by induction; hence $c(\phi_{n-2}\phi_{n-1}) = c(c(\phi_{n-1}\phi_{n-2})) = \phi_{n-1}\phi_{n-2}$.

Equation (3) implies that

$$(4) \qquad next[F_k - 1] = F_{k-1} - 1 \quad \text{for } k \geqq 3.$$

Therefore if we have a mismatch when $j = F_8 - 1 = 20$, our algorithm might set $j := next[j]$ for the successive values 20, 12, 7, 4, 2, 1, 0 of j. Since F_k is $(\phi^k/\sqrt{5})$ rounded to the nearest integer, it is possible to have up to $\sim\log_\phi m$ consecutive iterations of the $j := next[j]$ loop.

We shall now show that Fibonacci strings actually are the worst case, i.e., that $\log_\phi m$ is also an upper bound. First let us consider the concept of *periodicity* in strings. We say that p is a *period* of α if

$$(5) \qquad \alpha[i] = \alpha[i+p] \quad \text{for } 1 \leqq i \leqq |\alpha| - p.$$

It is easy to see that p is a period of α if and only if

$$(6) \qquad \alpha = (\alpha_1\alpha_2)^k\alpha_1$$

for some $k \geqq 0$, where $|\alpha_1\alpha_2| = p$ and $\alpha_2 \neq \varepsilon$. Equivalently, p is a period of α if and only if

$$(7) \qquad \alpha\theta_1 = \theta_2\alpha$$

for some θ_1 and θ_2 with $|\theta_1| = |\theta_2| = p$. Condition (6) implies (7) with $\theta_1 = \alpha_2\alpha_1$ and $\theta_2 = \alpha_1\alpha_2$. Condition (7) implies (6), for we define $k = \lfloor |\alpha|/p \rfloor$ and observe that if $k > 0$, then $\alpha = \theta_2\beta$ implies $\beta\theta_1 = \theta_2\beta$ and $\lfloor |\beta|/p \rfloor = k - 1$; hence, reasoning inductively, $\alpha = \theta_2^k\alpha_1$ for some α_1 with $|\alpha_1| < p$, and $\alpha_1\theta_1 = \theta_2\alpha_1$. Writing $\theta_2 = \alpha_1\alpha_2$ yields (6).

The relevance of periodicity to our algorithm is clear once we consider what it means to shift a pattern. If $pattern[1] \dots pattern[j-1] = \alpha$ ends with $pattern[1] \dots pattern[i-1] = \beta$, we have

$$(8) \qquad \alpha = \beta\theta_1 = \theta_2\beta$$

where $|\theta_1| = |\theta_2| = j - i$, so the amount of shift $j - i$ is a period of α.

The construction of $i = next[j]$ in our algorithm implies further that $pattern[i]$, which is the first character of θ_1, is unequal to $pattern[j]$. Let us assume that β itself is subsequently shifted leaving a residue γ, so that

$$(9) \qquad \beta = \gamma\psi_1 = \psi_2\gamma$$

where the first character of ψ_1 differs from that of θ_1. We shall now prove that

$$(10) \qquad |\alpha| > |\beta| + |\gamma|.$$

If $|\beta| + |\gamma| \geqq |\alpha|$, there is an overlap of $d = |\beta| + |\gamma| - |\alpha|$ characters between the occurrences of β and γ in $\beta\theta_1 = \alpha = \theta_2\psi_2\gamma$; hence the first character of θ_1 is $\gamma[d+1]$. Similarly there is an overlap of d characters between the occurrences of β and γ in $\theta_2\beta = \alpha = \gamma\psi_1\theta_1$; hence the first character of ψ_1 is $\beta[d+1]$. Since these characters are distinct, we obtain $\gamma[d+1] \neq \beta[d+1]$, contradicting (9). This establishes (10), and leads directly to the announced result:

THEOREM. *The number of consecutive times that $j := next[j]$ is performed, while one text character is being scanned, is at most* $1 + \log_\phi m$.

Proof. Let L_r be the length of the shortest string α as in the above discussion such that a sequence of r consecutive shifts is possible. Then $L_1 = 0$, $L_2 = 1$, and we have $|\beta| \geqq L_{r-1}$, $|\gamma| \geqq L_{r-2}$ in (10); hence $L_2 \geqq F_{r+1} - 1$ by induction on r. Now if r shifts occur we have $m \geqq F_{r+1} \geqq \phi^{r-1}$. □

The algorithm of § 2 would run correctly in linear time even if $f[j]$ were used instead of *next*$[j]$, but the analogue of the above theorem would then be false. For example, the pattern a^n leads to $f[j]=j-1$ for $1 \leqq j \leqq m$. Therefore if we matched a^m to the text $a^{m-1}b\alpha$, using $f[j]$ instead of *next*$[j]$, the mismatch *text*$[m] \neq$ *pattern*$[m]$ would be followed by m occurrences of $j:=f[j]$ and $m-1$ redundant comparisons of *text*$[m]$ with *pattern*$[j]$, before k is advanced to $m+1$.

The subject of periods in strings has several interesting algebraic properties, but a reader who is not mathematically inclined may skip to § 6 since the following material is primarily an elaboration of some additional structure related to the above theorem.

LEMMA 1. *If p and q are periods of α, and $p+q \leqq |\alpha|+\gcd(p, q)$, then $\gcd(p, q)$ is a period of α.*

Proof. Let $d=\gcd(p, q)$, and assume without loss of generality that $d<p<q=p+r$. We have $\alpha[i]=\alpha[i+p]$ for $1 \leqq i \leqq |\alpha|-p$ and $\alpha[i]=\alpha[i+q]$ for $1 \leqq i \leqq |\alpha|-q$; hence $\alpha[i+r]=\alpha[i+q]=\alpha[i]$ for $1+r \leqq i+r \leqq |\alpha|-p$, i.e.,

$$\alpha[i]=\alpha[i+r] \quad \text{for } 1 \leqq i \leqq |\alpha|-q.$$

Furthermore $\alpha=\beta\theta_1=\theta_2\beta$ where $|\theta_1|=p$, and it follows that p and r are periods of β, where $p+r \leqq |\beta|+d=|\beta|+\gcd(p, r)$. By induction, d is a period of β. Since $|\beta|=|\alpha|-p \geqq q-d \geqq q-r=p=|\theta_1|$, the strings θ_1 and θ_2 (which have the respective forms $\beta_2\beta_1$ and $\beta_1\beta_2$ by (6) and (7)) are substrings of β; so they also have d as a period. The string $\alpha=(\beta_1\beta_2)^{k+1}\beta_1$ must now have d as a period, since any characters d positions apart are contained within $\beta_1\beta_2$ or $\beta_1\beta_1$. □

The result of Lemma 1 but with the stronger hypothesis $p+q \leqq |\alpha|$ was proved by Lyndon and Schützenberger in connection with a problem about free groups [19, Lem. 4]. The weaker hypothesis in Lemma 1 turns out to give the best possible bound: If $\gcd(p, q)<p<q$ we can find a string of length $p+q-\gcd(p, q)-1$ for which $\gcd(p, q)$ is *not* a period. In order to see why this is so, consider first the example in Fig. 2 showing the most general strings of lengths 15 through 25 having both 11 and 15 as periods. (The strings are "most general" in the sense that any two character positions that can be different *are* different.)

```
a b c d e f g h i j k a b c d
a b c d a f g h i j k a b c.d a
a b c d a b g h i j k a b c d a b
a b c d a b c h i j k a b c d a b c
a b c d a b c d i j k a b c d a b c d
a b c d a b c d a j k a b c d a b c d a
a b c d a b c d a b k a b c d a b c d a b
a b c d a b c d a b c a b c d a b c d a b c
a b c a a b c a a b c a b c a a b c a a b c a
a a c a a a c a a a c a a c a a a c a a a c a a
a a a a a a a a a a a a a a a a a a a a a a a a a
```

FIG. 2

Note that the number of degrees of freedom, i.e., the number of distinct symbols, decreases by 1 at each step. It is not difficult to prove that the number cannot decrease by *more* than 1 as we go from $|\alpha|=n-1$ to $|\alpha|=n$, since the only new relations are $\alpha[n]=\alpha[n-q]=\alpha[n-p]$; we decrease the number of distinct symbols by one if and only if positions $n-q$ and $n-p$ contain distinct symbols in the most general string of length $n-1$. The lemma tells us that we are left with at most $\gcd(p, q)$ symbols when the length reaches $p+q-\gcd(p, q)$; on the other hand we always have exactly p symbols when the length is q. Therefore each of the $p-\gcd(p, q)$ steps *must* decrease the number of symbols by 1, and the most general string of length $p+q-\gcd(p, q)-1$ must have exactly $\gcd(p, q)+1$ distinct symbols. In other words, the lemma gives the best possible bound.

When p and q are relatively prime, the strings of length $p+q-2$ on two symbols, having both p and q as periods, satisfy a number of remarkable properties, generalizing what we have observed earlier about Fibonacci strings. Since the properties of these pathological patterns may prove useful in other investigations, we shall summarize them in the following lemma.

LEMMA 2. *Let the strings $\sigma(m, n)$ of length n be defined for all relatively prime pairs of integers $n \geqq m \geqq 0$ as follows:*

$$\sigma(0, 1)=a, \quad \sigma(1, 1)=b, \quad \sigma(1, 2)=ab;$$

$$(11) \qquad \left.\begin{aligned} \sigma(m, m+n)&=\sigma(n \bmod m, m)\sigma(m, n)\\ \sigma(n, m+n)&=\sigma(m, n)\sigma(n \bmod m, m)\end{aligned}\right\} \quad \textit{if } 0<m<n.$$

These strings satisfy the following properties:

(i) $\sigma(m, qm+r)\sigma(m-r, m)=\sigma(r, m)\sigma(m, qm+r)$, *for* $m>2$;

(ii) $\sigma(m, n)$ *has period* m, *for* $m>1$;

(iii) $c(\sigma(m, n))=\sigma(n-m, n)$, *for* $n>2$.

(The function $c(\alpha)$ was defined in connection with (3) above.)

Proof. We have, for $0<m<n$ and $q \geqq 2$,

$$\sigma(m+n, q(m+n)+m)=\sigma(m, m+n)\,\sigma(m+n, (q-1)(m+n)+m),$$

$$\sigma(m+n, q(m+n)+n)=\sigma(n,m+n)\,\sigma(m+n, (q-1)(m+n)+n),$$

$$\sigma(m+n, 2m+n)=\sigma(m, m+n)\,\sigma(n \bmod m, m),$$

$$\sigma(m+n, m+2n)=\sigma(n, m+n)\,\sigma(m, n);$$

hence, if $\theta_1=\sigma(n \bmod m, m)$ and $\theta_2=\sigma(m, n)$ and $q \geqq 1$,

$$(12) \quad \sigma(m+n, q(m+n)+m)=(\theta_1\theta_2)^q\theta_1, \qquad \sigma(m+n, q(m+n)+n)=(\theta_2\theta_1)^q\theta_2.$$

It follows that

$$\sigma(m+n, q(m+n)+m)\sigma(n, m+n)=\sigma(m, m+n)\,\sigma(m+n, q(m+n)+m),$$

$$\sigma(m+n, q(m+n)+n)\sigma(m, m+n)=\sigma(n, m+n)\,\sigma(m+n, q(m+n)+n),$$

which combine to prove (i). Property (ii) also follows immediately from (12), except for the case $m=2$, $n=2q+1$, $\sigma(2, 2q+1)=(ab)^q a$, which may be verified directly. Finally, it suffices to verify property (iii) for $0<m<\frac{1}{2}n$, since $c(c(\alpha))=\alpha$; we must show that

$$c(\sigma(m, m+n))=\sigma(m, n)\,\sigma(n \bmod m, m) \quad \text{for } 0<m<n.$$

When $m \leqq 2$ this property is easily checked, and when $m > 2$ it is equivalent by induction to

$$\sigma(m, m+n) = \sigma(m, n)\,\sigma(m-(n \bmod m), m) \quad \text{for } 0 < m < n, \quad m > 2.$$

Set $n \bmod m = r$, $\lfloor n/m \rfloor = q$, and apply property (i). □

By properties (ii) and (iii) of this lemma, $\sigma(p, p+q)$ minus its last two characters is the string of length $p+q-2$ having periods p and q. Note that Fibonacci strings are just a very special case, since $\phi_n = \sigma(F_{n-1}, F_n)$. Another property of the σ strings appears in [15]. A completely different proof of Lemma 1 and its optimality, and a completely different definition of $\sigma(m, n)$, were given by Fine and Wilf in 1965 [7]. These strings have a long history going back at least to the astronomer Johann Bernoulli in 1772; see [25, § 2.13] and [21].

If α is any string, let $P(\alpha)$ be its shortest period. Lemma 1 implies that all periods q which are not multiples of $P(\alpha)$ must be greater than $|\alpha| - P(\alpha) + \gcd(q, P(\alpha))$. This is a rather strong condition in terms of the pattern matching algorithm, because of the following result.

LEMMA 3. *Let* $\alpha = pattern[1] \ldots pattern[j-1]$ *and let* $a = pattern[j]$. *In the pattern matching algorithm,* $f[j] = j - P(\alpha)$, *and* $next[j] = j - q$, *where* q *is the smallest period of* α *which is not a period of* αa. (*If no such period exists,* $next[j] = 0$.) *If* $P(\alpha)$ *divides* $P(\alpha a)$ *and* $P(\alpha a) < j$, *then* $P(\alpha) = P(\alpha a)$. *If* $P(\alpha)$ *does not divide* $P(\alpha a)$ *or if* $P(\alpha a) = j$, *then* $q = P(\alpha)$.

Proof. The characterizations of $f[j]$ and $next[j]$ follow immediately from the definitions. Since every period of αa is a period of α, the only nonobvious statement is that $P(\alpha) = P(\alpha a)$ whenever $P(\alpha)$ divides $P(\alpha a)$ and $P(\alpha a) \neq j$. Let $P(\alpha) = p$ and $P(\alpha a) = mp$; then the (mp)th character from the right of α is a, as is the $(m-1)p$th, . . . , as is the pth; hence p is a period of αa. □

Lemma 3 shows that the $j := next[j]$ loop will almost always terminate quickly. If $P(\alpha) = P(\alpha a)$, then q must not be a multiple of $P(\alpha)$; hence by Lemma 1, $P(\alpha) + q \geqq j + 1$. On the other hand $q > P(\alpha)$; hence $q > \frac{1}{2}j$ and $next[j] < \frac{1}{2}j$. In the other case $q = P(\alpha)$, we had better not have q too small, since q will be a period in the residual pattern after shifting, and $next[next[j]]$ will be $< q$. To keep the loop running it is necessary for new small periods to keep popping up, relatively prime to the previous periods.

6. Palindromes.

One of the most outstanding unsolved questions in the theory of computational complexity is the problem of how long it takes to determine whether or not a given string of length n belongs to a given context-free language. For many years the best upper bound for this problem was $O(n^3)$ in a general context-free language as $n \to \infty$; L. G. Valiant has recently lowered this to $O(n^{\log_2 7})$. On the other hand, the problem isn't known to require more than order n units of time for any particular language. This big gap between $O(n)$ and $O(n^{2.81})$ deserves to be closed, and hardly anyone believes that the final answer will be $O(n)$.

Let Σ be a finite alphabet, let Σ^* denote the strings over Σ, and let

$$P = \{\alpha\alpha^R \mid \alpha \in \Sigma^*\}.$$

Here α^R denotes the reversal of α, i.e., $(a_1 a_2 \ldots a_n)^R = a_n \ldots a_2 a_1$. Each string π in P is a *palindrome* of even length, and conversely every even palindrome over

Σ is in P. At one time it was popularly believed that the language P^* of "even palindromes starred", namely the set of *palstars* $\pi_1 \ldots \pi_n$ where each π_i is in P, would be impossible to recognize in $O(n)$ steps on a random-access computer.

It isn't especially easy to spot members of this language. For example, $a\ a\ b\ b\ a\ b\ b\ a$ is a palstar, but its decomposition into even palindromes might not be immediately apparent; and the reader might need several minutes to decide whether or not

$$b\ a\ a\ b\ b\ a\ b\ b\ a\ a\ b\ a\ b\ b\ a\ a\ b\ b\ a\ b\ b\ a\ b\ a\ a$$
$$b\ b\ a\ b\ b\ a\ b\ b\ a\ b\ b\ a\ a\ b\ a\ b\ a\ b\ b\ a\ b\ b\ a\ a\ b$$

is in P^*. We shall prove, however, that palstars can be recognized in $O(n)$ units of time, by using their algebraic properties.

Let us say that a nonempty palstar is *prime* if it cannot be written as the product of two nonempty palstars. A prime palstar must be an even palindrome $\alpha\alpha^R$ but the converse does not hold. By repeated decomposition, it is easy to see that every palstar β is expressible as a product $\beta_1 \ldots \beta_t$ of prime palstars, for some $t \geqq 0$; what is less obvious is that such a decomposition into prime factors is unique. This "fundamental theorem of palstars" is an immediate consequence of the following basic property.

LEMMA 1. *A prime palstar cannot begin with another prime palstar.*

Proof. Let $\alpha\alpha^R$ be a prime palstar such that $\alpha\alpha^R = \beta\beta^R\gamma$ for some nonempty even palindrome $\beta\beta^R$ and some $\gamma \neq \varepsilon$; furthermore, let $\beta\beta^R$ have minimum length among all such counterexamples. If $|\beta\beta^R| > |\alpha|$ then $\alpha\alpha^R = \beta\beta^R\gamma = \alpha\delta\gamma$ for some $\delta \neq \varepsilon$; hence $\alpha^R = \delta\gamma$, and $\beta\beta^R = (\beta\beta^R)^R = (\alpha\delta)^R = \delta^R\alpha^R = \delta^R\delta\gamma$, contradicting the minimality of $|\beta\beta^R|$. Therefore $|\beta\beta^R| \leqq |\alpha|$; hence $\alpha = \beta\beta^R\delta$ for some δ, and $\beta\beta^R\gamma = \alpha\alpha^R = \beta\beta^R\delta\delta^R\beta\beta^R$. But this implies that γ is the palstar $\delta\delta^R\beta\beta^R$, contradicting the primality of $\alpha\alpha^R$. □

COROLLARY (Left cancellation property.) *If $\alpha\beta$ and α'are palstars, so is β.*

Proof. Let $\alpha = \alpha_1 \ldots \alpha_r$ and $\alpha\beta = \beta_1 \ldots \beta_s$ be prime factorizations of α and $\alpha\beta$. If $\alpha_1 \ldots \alpha_r = \beta_1 \ldots \beta_r$, then $\beta = \beta_{r+1} \ldots \beta_s$ is a palstar. Otherwise let j be minimal with $\alpha_j \neq \beta_j$; then α_j begins with β_j or vice versa, contradicting Lemma 1. □

LEMMA 2. *If α is a string of length n, we can determine the length of the longest even palindrome $\beta \in P$ such that $\alpha = \beta\gamma$, in $O(n)$ steps.*

Proof. Apply the pattern-matching algorithm with *pattern* $= \alpha$ and *text* $= \alpha^R$. When $k = n + 1$ the algorithm will stop with j maximal such that *pattern*$[1] \ldots$ *pattern* $[j-1] =$ *text*$[n+2-j] \ldots$ *text*$[n]$. Now perform the following iteration:

while $j \geqq 3$ **and** j even **do** $j := f(j)$.

By the theory developed in § 3, this iteration terminates with $j \geqq 3$ if and only if α begins with a nonempty even palindrome, and $j-1$ will be the length of the largest such palindrome. (Note that $f[j]$ must be used here instead of *next*$[j]$; e.g. consider the case $\alpha = a\ a\ b\ a\ a\ b$. But the pattern matching process takes $O(n)$ time even when $f[j]$ is used.) □

THEOREM. *Let L be any language such that L^* has the left cancellation property and such that, given any string α of length n, we can find a nonempty $\beta \in L$*

such that α begins with β or we can prove that no such β exists, in $O(n)$ steps. Then we can determine in $O(n)$ time whether or not a given string is in L^*.

Proof. Let α be any string, and suppose that the time required to test for nonempty prefixes in L is $\leqq Kn$ for all large n. We begin by testing α's initial subsequences of lengths $1, 2, 4, \dots, 2^k, \dots$, and finally α itself, until finding a prefix in L or until establishing that α has no such prefix. In the latter case, α is not in L^*, and we have consumed at most $(K+K_1)+(2K+K_1)+(4K+K_1)+\cdots+(|\alpha|K+K_1)<2Kn+K_1 \log_2 n$ units of time for some constant K_1. But if we find a nonempty prefix $\beta \in L$ where $\alpha=\beta\gamma$, we have used at most $4|\beta|K+K(\log_2|\beta|)$ units of time so far. By the left cancellation property, $\alpha \in L^*$ if and only if $\gamma \in L^*$, and since $|\gamma| = n-|\beta|$ we can prove by induction that at most $(4K+K_1)n$ units of time are needed to decide membership in L^*, when $n>0$. □

COROLLARY. *P^* can be recognized in $O(n)$ time.*

Note that the related language

$$P_1^* = \{\pi \in \Sigma^* \mid \pi = \pi^R \text{ and } |\pi| \geqq 2\}^*$$

cannot be handled by the above techniques, since it contains both $a\,a\,a\,b\,b\,b$ and $a\,a\,a\,b\,b\,b\,b\,a$; the fundamental theorem of palstars fails with a vengeance. It is an open problem whether or not P_1^* can be recognized in $O(n)$ time, although we suspect that it can be done.[1] Once the reader has disposed of this problem, he or she is urged to tackle another language which has recently been introduced by S. A. Greibach [11], since the latter language is known to be as hard as possible; no context-free language can be harder to recognize except by a constant factor.

7. Historical remarks. The pattern-matching algorithm of this paper was discovered in a rather interesting way. One of the authors (J. H. Morris) was implementing a text-editor for the CDC 6400 computer during the summer of 1969, and since the necessary buffering was rather complicated he sought a method that would avoid backing up the text file. Using concepts of finite automata theory as a model, he devised an algorithm equivalent to the method presented above, although his original form of presentation made it unclear that the running time was $O(m+n)$. Indeed, it turned out that Morris's routine was too complicated for other implementors of the system to understand, and he discovered several months later that gratuitous "fixes" had turned his routine into a shambles.

In a totally independent development, another author (D. E. Knuth) learned early in 1970 of S. A. Cook's surprising theorem about two-way deterministic pushdown automata [5]. According to Cook's theorem, any language recognizable by a two-way deterministic pushdown automaton, in *any* amount of time, can be recognized on a random access machine in $O(n)$ units of time. Since D. Chester had recently shown that the set of strings beginning with an even palindrome could be recognized by such an automaton, and since Knuth couldn't imagine how to recognize such a language in less than about n^2 steps on a conventional computer, Knuth laboriously went through all the steps of Cook's construction as applied to Chester's automaton. His plan was to "distill off" what

[1] (Note added April, 1976.) Zvi Galil and Joel Seiferas have recently resolved this conjecture affirmatively.

was happening, in order to discover why the algorithm worked so efficiently. After pondering the mass of details for several hours, he finally succeeded in abstracting the mechanism which seemed to be underlying the construction, in the special case of palindromes, and he generalized it slightly to a program capable of finding the longest prefix of one given string that occurs in another.

This was the first time in Knuth's experience that automata theory had taught him how to solve a real programming problem better than he could solve it before. He showed his results to the third author (V. R. Pratt), and Pratt modified Knuth's data structure so that the running time was independent of the alphabet size. When Pratt described the resulting algorithm to Morris, the latter recognized it as his own, and was pleasantly surprised to learn of the $O(m+n)$ time bound, which he and Pratt described in a memorandum [22]. Knuth was chagrined to learn that Morris had already discovered the algorithm, *without* knowing Cook's theorem; but the theory of finite-state machines had been of use to Morris too, in his initial conceptualization of the algorithm, so it was still legitimate to conclude that automata theory had actually been helpful in this practical problem.

The idea of scanning a string without backing up while looking for a pattern, in the case of a two-letter alphabet, is implicit in the early work of Gilbert [10] dealing with comma-free codes. It also is essentially a special case of Knuth's LR(0) parsing algorithm [16] when applied to the grammar

$$\begin{aligned} &S \rightarrow aS, \qquad \text{for each } a \text{ in the alphabet,}\\ &S \rightarrow \alpha, \end{aligned}$$

where α is the pattern. Diethelm and Roizen [6] independently discovered the idea in 1971. Gilbert and Knuth did not discuss the preprocessing to build the *next* table, since they were mainly concerned with other problems, and the preprocessing algorithm given by Diethelm and Roizen was of order m^2. In the case of a binary (two-letter) alphabet, Diethelm and Roizen observed that the algorithm of § 3 can be improved further: we can go immediately to "char matched" after $j := next[j]$ in this case if $next[j] > 0$.

A conjecture by R. L. Rivest led Pratt to discover the $\log_\phi m$ upper bound on pattern movements between successive input characters, and Knuth showed that this was best possible by observing that Fibonacci strings have the curious properties proved in § 5. Zvi Galil has observed that a real-time algorithm can be obtained by letting the text pointer move ahead in an appropriate manner while the j pointer is moving down [9].

In his lectures at Berkeley, S. A. Cook had proved that P^* was recognizable in $O(n \log n)$ steps on a random-access machine, and Pratt improved this to $O(n)$ using a preliminary form of the ideas in § 6. The slightly more refined theory in the present version of § 6 is joint work of Knuth and Pratt. Manacher [20] found another way to recognize palindromes in linear time, and Galil [9] showed how to improve this to real time. See also Slisenko [23].

It seemed at first that there might be a way to find the *longest common substring* of two given strings, in time $O(m+n)$; but the algorithm of this paper does not readily support any such extension, and Knuth conjectured in 1970 that such efficiency would be impossible to achieve. An algorithm due to Karp, Miller, and Rosenberg [13] solved the problem in $O((m+n)\log(m+n))$ steps, and this tended to support the conjecture (at least in the mind of its originator). However, Peter Weiner has recently developed a technique for solving the longest common substring problem in $O(m+n)$ units of time with a fixed alphabet, using tree structures in a remarkable new way [26]. Furthermore, Weiner's algorithm has the following interesting consequence, pointed out by E. McCreight: a text file can be processed (in linear time) so that it is possible to determine exactly how much of a pattern is necessary to identify a position in the text uniquely; as the pattern is being typed in, the system can be interrupt as soon as it "knows" what the rest of the pattern must be! Unfortunately the time and space requirements for Weiner's algorithm grow with increasing alphabet size.

If we consider the problem of scanning finite-state languages in general, it is known [1 § 9.2] that the language defined by any regular expression of length m is recognizable in $O(mn)$ units of time. When the regular expression has the form

$$\Sigma^*(\alpha_{1,1}+\cdots+\alpha_{1,s(1)})\Sigma^*(\alpha_{2,1}+\cdots+\alpha_{2,s(2)})\Sigma^* \ldots \Sigma^*(\alpha_{r,1}+\cdots+\alpha_{r,s(r)})\Sigma^*$$

the algorithm we have discussed shows that only $O(m+n)$ units of time are needed (considering Σ^* as a character of length 1 in the expression). Recent work by M. J. Fischer and M. S. Paterson [8] shows that regular expressions of the form

$$\Sigma^*\alpha_1\Sigma\alpha_2\Sigma \ldots \Sigma\alpha_r\Sigma^*,$$

i.e., patterns with "don't care" symbols, can be identified in $O(n \log m \log\log m \log q)$ units of time, where q is the alphabet size and $m = |\alpha_1\alpha_2 \ldots \alpha_r|+r$. The constant of proportionality in their algorithm is extremely large, but the existence of their construction indicates that efficient new algorithms for general pattern matching problems probably remain to be discovered.

A completely different approach to pattern matching, based on hashing, has been proposed by Malcolm C. Harrison [12]. In certain applications, especially with very large text files and short patterns, Harrison's method may be significantly faster than the character-comparing method of the present paper, on the average, although the redundancy of English makes the performance of his method unclear.

8. Postscript: Faster pattern matching in strings.[2] In the spring of 1974, Robert S. Boyer and J. Strother Moore and (independently) R. W. Gosper noticed that there is an even faster way to match pattern strings, by skipping more rapidly over portions of the text that cannot possibly lead to a match. Their idea was to look first at *text*[m], instead of *text*[1]. If we find that the character *text*[m] does not appear in the pattern at all, we can immediately shift the pattern right m places. Thus, when the alphabet size q is large, we need to inspect only about n/m characters of the text, on the average! Furthermore if *text*[m] does occur in the pattern, we can shift the pattern by the minimum amount consistent with a match.

[2] This postscript was added by D. E. Knuth in March, 1976, because of developments which occurred after preprints of this paper were distributed.

Several interesting variations on this strategy are possible. For example, if *text*$[m]$ does occur in the pattern, we might continue the search by looking at *text*$[m-1]$, *text*$[m-2]$, etc.; in a random file we will usually find a small value of r such that the substring *text*$[m-r]$ ... *text*$[m]$ does not appear in the pattern, so we can shift the pattern $m-r$ places. If $r = \lfloor 2 \log_q m \rfloor$, there are more than m^2 possible values of *text*$[m-r]$ ... *text*$[m]$, but only $m-r$ substrings of length $r+1$ in the pattern, hence the probability is $O(1/m)$ that *text*$[m-r]$ ... *text*$[m]$ occurs in the pattern; If it doesn't, we can shift the pattern right $m-r$ places; but if it does, we can determine all matches in positions $<m-r$ in $O(m)$ steps, shifting the pattern $m-r$ places by the method of this paper. Hence the expected number of characters examined among the first $m - \lfloor 2 \log_q m \rfloor$ is $O(\log_q m)$; this proves the existence of a linear worst-case algorithm which inspects $O(n(\log_q m)/m)$ characters in a random text. This upper bound on the average running time applies to all patterns, and there are some patterns (e.g., a^m or $(a\,b)^{m/2}$) for which the expected number of characters examined by the algorithm is $O(n/m)$.

Boyer and Moore have refined the skipping-by-m idea in another way. Their original algorithm may be expressed as follows using our conventions:

```
k := m;
while k ≦ n do
    begin
        j := m;
        while j > 0 and text[k] = pattern[j] do
            begin
                j := j − 1; k := k − 1;
            end;
        if j = 0 then
            begin
                match found at (k);
                k := k + m + 1
            end else
            k := k + max (d[text[k]], dd[j]);
    end;
```

This program calls *match found at* (k) for all $0 \leqq k \leqq n-m$ such that *pattern*$[1]$ ... *pattern*$[m]$ = *text*$[k+1]$ ... *text*$[k+m]$. There are two precomputed tables, namely

$$d[a] = \min\{s \mid s = m \text{ or } (0 \leqq s < m \text{ and } \textit{pattern}[m-s] = a)\}$$

for each of the q possible characters a, and

$$dd[j] = \min\{s+m-j \mid s \geqq 1 \text{ and } ((s \geqq 1 \text{ or } \textit{pattern}[i-s] = \textit{pattern}[i]) \text{ for } j < i \leqq m)\},$$

for $1 \leqq j \leqq m$.

The d table can clearly be set up in $O(q+m)$ steps, and the dd table can be precomputed in $O(m)$ steps using a technique analogous to the method in § 2 above, as we shall see. The Boyer–Moore paper [4] contains further exposition of the algorithm, including suggestions for highly efficient implementation, and gives both theoretical and empirical analyses. In the remainder of this section we shall show how the above methods can be used to resolve some of the problems left open in [4].

First let us improve the original Boyer–Moore algorithm slightly by replacing $dd[j]$ by

$$\begin{aligned} dd'[j] = \min\{s+m-j \mid s \geqq 1 \text{ and } (s \geqq j \text{ or } \mathit{pattern}[j-s] \neq \mathit{pattern}[j]) \\ \text{and } ((s \geqq i \text{ or } \mathit{pattern}[i-s] = \mathit{pattern}[i]) \text{ for } j < i \leqq m)\}. \end{aligned}$$

(This is analogous to using $next[j]$ instead of $f[j]$; Boyer and Moore [4] credit the improvement in this case to Ben Kuipers, but they do not discuss how to determine dd' efficiently.) The following program shows how the dd' table can be precomputed in $O(m)$ steps; for purposes of comparison, the program also shows how to compute dd, which actually turns out to require slightly *more* operations than dd':

```
for $k$ := 1 step 1 until $m$ do $dd[k]$ := $dd'[k]$ := $2 \times m - k$;
$j$ := $m$; $t$ := $m+1$;
while $j > 0$ do
    begin
        $f[j]$ := $t$;
        while $t \leqq m$ and $pattern[j] \neq pattern[t]$ do
            begin
                $dd'[t]$ := min $(dd'[t], m-j)$;
                $t$ := $f[t]$;
            end;
        $t$ := $t-1$; $j$ := $j-1$;
        $dd[t]$ := min $(dd[t], m-j)$;
    end;
for $k$ := 1 step 1 until $t$ do
    begin
        $dd[k]$ := min $(dd[k], m+t-k)$;
        $dd'[k]$ := min $(dd'[k], m+t-k)$;
    end;
```

In practice one would, of course, compute only dd', suppressing all references to dd. The example in Table 2 illustrates most of the subtleties of this algorithm.

TABLE 2

$j=1$	2	3	4	5	6	7	8	9	10	11
$pattern[j]=b$	a	d	b	a	c	b	a	c	b	a
$f[j]=10$	11	6	7	8	9	10	11	11	11	12
$dd[j]=19$	18	17	16	15	8	7	6	5	4	1
$dd'[j]=19$	18	17	16	15	8	13	12	8	12	1

To prove correctness, one may show first that $f[j]$ is analogous to the $f[j]$ *in* § 2, but with right and left of the pattern reversed; namely $f[m]=m+1$, and for $j<m$ we have

$$f[j]=\min\{i \mid j<i\leqq m \text{ and } \\ pattern[i+1]\ldots pattern[m]=pattern[j+1]\ldots pattern[m+j-i]\}.$$

Furthermore the final value of t corresponds to $f[0]$ in this definition; $m-t$ is the maximum overlap of the pattern on itself. The correctness of $dd[j]$ and $dd'[j]$ for all j now follows without much difficulty, by showing that the minimum value of s in the definition of $dd[j_0]$ or $dd'[j_0]$ is discovered by the algorithm when $(t,j)=(j_0,j_0-s)$.

The Boyer–Moore algorithm and its variants can have curiously anomalous behavior in unusual circumstances. For example, the method discovers more quickly that the pattern $a\,a\,a\,a\,a\,a\,a\,c\,b$ does not appear in the text $(a\,b)^n$ if it suppresses the d heuristic entirely, i.e., if $d[t]$ is set to $-\infty$ for all t. Likewise, dd actually turns out to be better than dd' when matching $a^{15}\,b\,c\,b\,a\,b\,a\,b$ in $(b\,a\,a\,b\,a\,b)^n$, for large n.

Boyer and Moore showed that their algorithm has quadratic behavior in the worst case; the running time can be essentially proportional to pattern length times text length, for example when the pattern $c\,a(b\,a)^m$ occurs together with the text $(x^{2m}\,a\,a(b\,a)^m)^n$. They observed that this particular example was handled in linear time when Kuiper's improvement (dd' for dd) was made; but they left open the question of the true worst case behavior of the improved algorithm.

There are trivial cases in which the Boyer–Moore algorithm has qųadratic behavior, when matching all occurrences of the pattern, for example when matching the pattern a^m in the text a^n. But we are probably willing to accept such behavior when there are so many matches; the crucial issue is how long the algorithm takes in the worst case to scan over a text that does *not* contain the pattern at all. By extending the techniques of § 5, it is possible to show that the modified Boyer–Moore algorithm is *linear* in such a situation:

THEOREM. *If the above algorithm is used with dd' replacing dd, and if the text does not contain any occurrences of the pattern, the total number of characters matched is at most* $6n$.

Proof. An execution of the algorithm consists of a series of *stages*, in which m_k characters are matched and then the pattern is shifted s_k places, for $k=1,2,\ldots$. We want to show that $\sum m_k\leqq 6n$; the proof is based on breaking this cost into three parts, two of which are trivially $O(n)$ and the third of which is less obviously so.

Let $m'_k=m_k-2s_k$ if $m_k>2s_k$; otherwise let $m'_k=0$. When $m'_k>0$, we will say that the leftmost m'_k text characters matched during the kth stage have been "tapped". It suffices to prove that the algorithm taps characters at most $4n$ times, since $\sum m_k\leqq\sum m'_k+2\sum s_k$ and $\sum s_k\leqq n$. Unfortunately it is possible for some characters of the text to be tapped roughly $\log m$ times, so we need to argue carefully that $\sum m'_k\leqq 4n$.

Suppose the rightmost m''_k of the m'_k text characters tapped during the kth stage are matched again during some later stage, but the leftmost $m'_k-m''_k$ are being matched for the last time. Clearly $\sum (m'_k - m''_k) \leqq n$, so it remains to show that $\sum m''_k \leqq 3n$.

Let p_k be the amount by which the pattern would shift after the kth stage if the $d[a]$ heuristic were not present ($d[a] = -\infty$); then $p_k \leqq s_k$, and p_k is a period of the string matched at stage k.

Consider a value of k such that $m''_k > 0$, and suppose that the text characters matched during the kth stage form the string $\alpha = \alpha_1\alpha_2$ where $|\alpha| = m_k$ and $|\alpha_2| = m''_k + 2s_k$; hence the text characters in α_1 are matched for the last time. Since the pattern does not occur in the text, it must end with $x\alpha$ and the text scanned so far must end with $z\alpha$, where $x \neq z$. At this point the algorithm will shift the pattern right s_k positions and will enter stage $k+1$. We distinguish two cases: (i) The pattern length m exceeds $m_k + p_k$. Then the pattern can be written $\theta\beta\alpha$, where $|\beta| = p_k$; the last character of β is x and the last character of θ is $y \neq x$, by definition of dd'. Otherwise (ii) $m \leqq m_k + p_k$; the pattern then has the form $\beta\alpha$, where $|\beta| \leqq p_k \leqq s_k$. By definition of m''_k and the assumption that the pattern does not occur in the text, we have $|\beta\alpha| > s_k + |\alpha_2|$, i.e., $|\beta| > s_k - |\alpha_1|$. In both cases (i) and (ii), p_k is a period of $\beta\alpha$.

Now consider the first subsequent stage k' during which the *leftmost* of the m''_k text characters tapped during stage k is matched again; we shall write $k \to k'$ when the stages are in this relation. Suppose the mismatch occurs this time when text character z' fails to match pattern character x'. If z' occurs in the text within α_1, regarding α as fixed in its stage k position, then x' cannot be within $\beta\alpha$ where $\beta\alpha$ now occurs in the stage k' position of the pattern, since p_k is a period of $\beta\alpha$ and the character p_k positions to the right of x' is a z' (it matches a z' in the text). Thus x' now appears within θ. On the other hand, if z' occurs to the left of α, we must have $|\alpha_1| = 0$, since the characters of α_1 are never matched again. In either event, case (ii) above proves to be impossible. Hence case (i) always occurs when $m''_k > 0$, and x' always appears within θ.

To complete the argument, we shall show that $\sum_{k \to k'} m''_k$, for all fixed k', is at most $3s_{k'}$. Let $p' = p_{k'}$ and let α' denote the pattern matched at stage k'. Let $k_1 < \cdots < k_r$ be the values of k such that $k \to k'$. If $|\alpha'| + p' \leqq m$, let $\beta'\alpha'$ be the rightmost $p' + |\alpha'|$ characters of the pattern. Otherwise let α'' be the leftmost $|\alpha'| + p' - m$ characters of α'; and let $\beta'\alpha'$ be α'' followed by the pattern. Note that in both cases α' is an initial substring of $\beta'\alpha'$ and $|\beta'| = p'$. In both cases, the actions of the algorithm during stages $k_1 + 1$ through k' are completely known if we are given the pattern and β', and if we know z' and the place within β' where stage $k_1 + 1$ starts matching. This follows from the fact that β' by itself determines the text, so that if we match the pattern against the string $z'\beta'\beta'\beta' \ldots$ (starting at the specified place for stage $k_1 + 1$) until the algorithm first tries to match z' we will know the length of α'. (If $|\alpha'| < p'$ then β' begins with α' and this statement holds trivially; otherwise, α' begins with β' and has period p'; hence $\beta'\beta'\beta' \ldots$ begins with α'.) Note that the algorithm cannot begin two different stages at exactly the same position within β', for then it would loop indefinitely, contradicting the fact that it does terminate. This property will be out key tool for proving the desired result.

Let the text strings matched during stages $k_1, \ldots, k_r$ be $\alpha_1, \ldots, \alpha_r$, and let their periods determined as in case (i) be $p_1, \ldots, p_r$ respectively; we have $p_j < \frac{1}{2}|\alpha_j|$

for $1 \leqq j \leqq r$. Suppose that during stage k_j the mismatch of $x_j \neq z_j$ implies that the pattern ends with $y_j\beta_j\alpha_j$, where $|\beta_j| = p_j$. We shall prove that $|\alpha_1| + \cdots + |\alpha_r| \leqq 3p'$. First let us prove that $|\alpha_j| < p'$ for all j: We have observed that x' always occurs within θ_j; hence $y_j\beta_j\alpha_j$ occurs as a rightmost substring of $x'\alpha'$. If $|\alpha_j| \geqq p'$ then $p_j + p' \leqq |\beta_j\alpha_j|$; hence the character p_j positions to the right of y_j in $x'\alpha'$ is x_j, as is the character $p_j + p'$ positions to the right of y_j. But the character p' positions to the right of y_j in $x'\alpha'$ is a y_j, since p' is a period of $x'\alpha'$; hence the character $p' + p_j$ positions to the right of y_j is also y_j, contradicting $x_j \neq y_j$.

Since $|\alpha_j| < p'$, each string α_j for $j \geqq 2$ appears somewhere within β', when β' is regarded as a cyclic string, joined end-for-end. (It follows from the definition of $k \rightarrow k'$ that $z_j\alpha_j$ is a substring of α' for $j \geqq 2$.) We shall prove that the rightmost halves of these strings, namely the rightmost $\lceil \frac{1}{2}|\alpha_j| \rceil$ characters as they appear in β', are disjoint. This implies that $\frac{1}{2}|\alpha_2| + \cdots + \frac{1}{2}|\alpha_r| \leqq p'$, and the proof will be complete (since $|\alpha_1| \leqq p'$).

Suppose therefore that the right half of the appearance of α_i overlaps the right half of the appearance of α_j within β', for some $i \neq j \geqq 2$, where the rightmost character of α_i is within α_j. This means that the algorithm at stage k_i begins to match characters starting within α_j at least p_j characters to the right of z_j where $z_j\alpha_j$ appears in β', when the text α' is treated modulo p'. (Recall that $p_j < \frac{1}{2}|\alpha_j|$.) The pattern ends with $x_j\alpha_j$, and p_j is a period of $x_j\alpha_j$. The algorithm must work correctly when the text equals the pattern, so there must come a stage, before shifting the pattern to the right of the appearance of α_j, where the algorithm scans left until hitting z_j. At this point, call it stage k'', there must be a mismatch of $z_j \neq x_j$, since p_j or more characters have been matched. (The character p_j positions to the right of z_j is x_j, by periodicity.) Hence $k'' < k'$; and it follows that $k'' = k_i$. (If $k'' > k_i$ we have $z_i\alpha_i$ entirely contained within α'', but then $k_i \rightarrow k'$ implies that $k'' = k'$.) Now $k'' = k_i$ implies that $z_j = z_i$ and $x_j = x_i$. We shall obtain a contradiction by showing that the algorithm "synchronizes" its stage $k_i + 1$ behavior with its stage $k_j + 1$ behavior, modulo p', causing an infinite loop as remarked above. The main point is that the dd' table will specify shifting the pattern p_j steps, so that y_j is brought into the position corresponding to z_j, in stage k_i as well as in stage k_j. (Any lesser shift brings an x_j into position p_j spaces to the right of z_j; hence it puts $y_i = x_j$ into the position corresponding to z_j, by periodicity, contradicting $x_i \neq y_i$.) The amount of shift depends on the maximum of the d and dd' entries, and the d entry will be chosen (in either k_i or k_j) if and only if z_j is not a character of β_j; but in this case, the d entry will also specify the same shift both for stage k_i and stage k_j. □

The constant 6 in the above theorem is probably much too large, and the above proof seems to be much too long; the reader is invited to improve the theorem in either or both respects. An interesting example of the rather complex behavior possible with this algorithm occurs when the pattern is $b\psi_r$ and the text is $\psi_r a \psi_r$ for large r, where

$$\psi_0 = a, \qquad \psi_{n+1} = \psi_n\psi_n b \psi_n.$$

COROLLARY. *The worst case running time of the Boyer–Moore algorithm with dd' replacing dd is $O(n + rm)$ character comparisons, if the pattern occurs r times in the text.*

Proof. Let $T(n, r)$ be the worst case running time as a function of n and r, when m is fixed. The theorem implies that $T(n, 0) \leqq 7n$, counting the mismatched characters as well as the matched ones. Furthermore, if $r > 0$ and if the first appearance of the pattern ends at position n_0 we have $T(n, r) \leqq 7(n_0 - 1) + m + T(n - n_0 + m - 1, r - 1)$. It follows that $T(n, r) \leqq 7n + 8rm - 14r$. □

When the Boyer–Moore algorithm implicitly shifts the pattern to the right, it forgets all it "knows" about characters already matched; this is why the linearity theorem is not trivial. A more complex algorithm can be envisaged, with a finite number of states corresponding to which text characters are known to match the pattern in its current position; when in state q we fetch the character $x := text[k - t[q]]$, then we set $k := k + s[q, x]$ and go to state $q'[q, x]$. For example, consider the pattern *a b a c b a b a*, and the specification of t, s, and q' in Table 3; exactly 41 distinguishable states can arise. An asterisk (*) in that table shows where the pattern has been fully matched.

The number of states in this generalization of the Boyer–Moore algorithm can be rather large, as the example shows, but the patterns which occur most often in practice probably do not imply many states. The number of states is always less than 2^m, and perhaps a much smaller upper bound is possible; it is unclear which patterns of a given length lead to the most states, and it does not seem obvious that this maximum number of states is exponential in m.

If the characters of the pattern are distinct, say $a_1 a_2 \ldots a_m$, this generalization of the Boyer–Moore algorithm leads to exactly $\frac{1}{2}(m^2 + m)$ states. (Namely, all states of the form $\bullet \ldots \bullet\, a_k \bullet \ldots \bullet\, a_{j+1} \ldots a_m$ for $0 \leqq k < j \leqq m$, with a_k suppressed if $k = 0$.) By merging several of these states we obtain the following simple algorithm, which uses a table $c[x]$ where

$$c[x] = \begin{cases} m - j, & \text{if } x = a_j; \\ -1, & \text{if } x \notin \{a_1, \ldots, a_m\}. \end{cases}$$

The algorithm works only when all pattern characters are distinct, but it improves slightly on the Boyer–Moore technique in this important special case.

```
$j := k := m$;
**while** $k \leqq n$ **do**
    **begin** $i := c[text[k]]$;
        **if** $i < 0$ **then** $j := m$
        **else if** $i = 0$ **then**
        **begin for** $i := 1$ **step** 1 **until** $m - 1$ **do**
            **if** $text[k - i] \neq pattern[m - i]$ **then go to** nomatch;
            *match found at* $(k - m)$;
        nomatch: $j := m$;
        **end else if** $i + j \geq m$ **then** $j := i$ **else** $j := m$;
        $k := k + j$;

    **end**;
```

Let us close this section by making a preliminary investigation into the question of "fastest" pattern matching in strings, i.e., *optimum* algorithms. What algorithm minimizes the number of text characters examined, over all conceivable algorithms for the problem we have been considering? In order to make this question nontrivial, we shall ask for the minimum *average* number of characters examined when finding *all* occurrences of the pattern in the text, where the average is taken uniformly with respect to strings of length n over a given alphabet. (The minimum worst case number of characters examined is of no interest, since it is between $n-m$ and n for all patterns[3]; therefore we ask for the minimum average number. It might be argued that the minimum average number, taken over random strings, is of little interest, since people rarely search in random strings; they usually search for patterns that actually appear. However, the random-string model is a reasonable approximation when we consider those stretches of text that do not contain the pattern, and the algorithm obviously must examine every character in those places where the pattern does occur.)

The case of patterns of length 2 can be solved exactly; it is somewhat surprising to find that the analysis is not completely trivial even in this case. Consider first the pattern $a\,b$ where $a \neq b$. Let q be the alphabet size, $q \geqq 2$. Let $f(n)$ denote the minimum average number of characters examined by an algorithm which finds all occurrences of the pattern in a random text of length n; and let $g(n)$ denote the minimum average number of characters examined in a random text of length $n+1$ which is known to begin with a, not counting the examination of the known first character. These functions can be computed by the following recurrence relations:

$$f(0)=f(1)=g(0)=0, \qquad g(1)=1.$$

$$f(n)=1+\min_{1\leqq k\leqq n}\left(\frac{1}{q}(f(k-1)+g(n-k))+\frac{1}{q}(g(k-1)+f(n-k)) + \left(1-\frac{2}{q}\right)(f(k-1)+f(n-k))\right),$$

$$g(n)=1+\frac{1}{q}g(n-1)+\left(1-\frac{1}{q}\right)f(n-1), \qquad n\geqq 2.$$

The recurrence for f follows by considering which character is examined first; the recurrence for g follows from the fact that the second character must be examined in any case, so it can be examined first without loss of efficiency. It can be shown that the minimum is always assumed for $k=2$; hence we obtain the closed form solution

$$f(n)=\frac{n(q^2+q-1)}{q(2q-1)}-\frac{(q-1)(q^2+2q-1)}{q(2q-1)^2}+\frac{(1-q)^n}{q^{n-3}(q-1)(2q-1)^2},$$

$$g(n)=\frac{n(q^2+q-1)}{q(2q-1)}+\frac{(q-1)(q^2-3q+1)}{q(2q-1)^2}-\frac{(1-q)^n}{q^{n-2}(2q-1)^2}, \qquad n\geqq 1.$$

(To prove that these functions satisfy the stated recurrences reduces to showing that the minimum of

$$\left(\frac{1-q}{q}\right)^{k-1}+\left(\frac{1-q}{q}\right)^{n-k}$$

for $1\leqq k\leqq n$ occurs for $k=2$, whenever $n\geqq 2$ and $q\geqq 2$.)

[3] This is clear when we must find *all* occurrences of the pattern; R. L. Rivest has recently proved it also for algorithms which stop after finding *one* occurrence. (*Information Processing Letters*, to appear.)

TABLE 3

			$s[q, x], q'[q, x]$			
state q	known characters	$t[q]$	$x=a$	$x=b$	$x=c$	other x
0	● ● ● ● ● ● ● ● ●	0	0, 1	1, 8	4, 9	8, 0
1	● ● ● ● ● ● ● ● *a*	1	7, 10	0, 2	7, 10	7, 10
2	● ● ● ● ● ● ● *b a*	2	0, 3	7, 10	2, 11	7, 10
3	● ● ● ● ● ● *a b a*	3	5, 12	0, 4	5, 12	5, 12
4	● ● ● ● ● *b a b a*	4	5, 12	5, 12	0, 5	5, 12
5	● ● ● ● *c b a b a*	5	0, 6	5, 12	5, 12	5, 12
6	● ● ● *a c b a b a*	6	5, 12	0, 7	5, 12	5, 12
7	● ● *b a c b a b a*	7	*5, 12	5, 12	5, 12	5, 12
8	● ● ● ● ● ● ● *b* ●	0	0, 2	8, 0	8, 0	8, 0
9	● ● ● ● *c* ● ● ● ●	0	0, 13	6, 14	4, 9	8, 0
10	*a* ● ● ● ● ● ● ● ●	0	0, 15	1, 8	4, 9	8, 0
11	● ● ● ● *c b a* ● ●	0	0, 16	6, 14	8, 0	8, 0
12	*a b a* ● ● ● ● ● ●	0	0, 17	3, 18	4, 9	8, 0
13	● ● ● ● *c* ● ● ● *a*	1	7, 10	0, 19	7, 10	7, 10
14	● *b* ● ● ● ● ● ● ●	0	0, 20	3, 18	4, 9	8, 0
15	*a* ● ● ● ● ● ● ● *a*	1	7, 10	0, 21	7, 10	7, 10
16	● ● ● ● *c b a* ● *a*	1	7, 10	0, 5	7, 10	7, 10
17	*a b a* ● ● ● ● ● *a*	1	7, 10	0, 22	7, 10	7, 10
18	● ● ● ● ● *b* ● ● ●	0	0, 23	3, 24	8, 0	8, 0
19	● ● ● ● *c* ● ● *b a*	2	0, 25	7, 10	7, 10	7, 10
20	● *b* ● ● ● ● ● ● *a*	1	7, 10	0, 26	7, 10	7, 10
21	*a* ● ● ● ● ● ● *b a*	2	0, 27	7, 10	2, 11	7, 10
22	*a b a* ● ● ● ● *b a*	2	0, 28	7, 10	2, 29	7, 10
23	● ● ● ● ● *b* ● ● *a*	1	7, 10	0, 30	7, 10	7, 10
24	● *b* ● ● ● *b* ● ● ●	0	0, 31	3, 24	8, 0	8, 0
25	● ● ● ● *c* ● *a b a*	3	5, 12	0, 5	5, 12	5, 12
26	● *b* ● ● ● ● ● *b a*	2	0, 32	7, 10	2, 11	7, 10
27	*a* ● ● ● ● ● *a b a*	3	5, 12	0, 33	5, 12	5, 12
28	*a b a* ● ● ● *a b a*	3	5, 12	0, 34	5, 12	5, 12
29	*a* ● ● ● *c b a* ● ●	0	0, 35	6, 14	8, 0	8, 0
30	● ● ● ● ● *b* ● *b a*	2	0, 4	7, 10	7, 10	7, 10
31	● *b* ● ● ● *b* ● ● *a*	1	7, 10	0, 36	7, 10	7, 10
32	● *b* ● ● ● ● *a b a*	3	5, 12	0, 37	5, 12	5, 12
33	*a* ● ● ● ● *b a b a*	4	5, 12	5, 12	0, 38	5, 12
34	*a b a* ● ● *b a b a*	4	5, 12	5, 12	*5, 12	5, 12
35	*a* ● ● ● *c b a* ● *a*	1	7, 10	0, 38	7, 10	7, 10
36	● *b* ● ● ● *b* ● *b a*	2	0, 37	7, 10	7, 10	7, 10
37	● *b* ● ● ● *b a b a*	4	5, 12	5, 12	0, 39	5, 12
38	*a* ● ● ● *c b a b a*	5	0, 40	5, 12	5, 12	5, 12
39	● *b* ● ● *c b a b a*	5	0, 7	5, 12	5, 12	5, 12
40	*a* ● ● *a c b a b a*	6	5, 12	*5, 12	5, 12	5, 12

If the pattern is $a\ a$, the recurrence for f changes to

$$f(n)=1+\min_{1\leqq k\leqq n}\left(\frac{1}{q}(g(k-1)+g(n-k))+\left(1-\frac{1}{q}\right)(f(k-1)+f(n-k))\right),\qquad n\geqq 2;$$

but this is actually no change!

Hence the following is an optimum algorithm for all patterns of length 2, in the sense of minimum average text characters inspected to find all matches in a random string:

```
$k := 2$;
while $k \leqq n$ do
    begin $c := text[k]$;
        if $c = pattern[2]$ and $text[k-1] = pattern[1]$
        then match found at $(k-2)$;
        while $c = pattern[1]$ do
            begin $k := k+1$; $c := text[k]$;
                if $c = pattern[2]$ then match found at $(k-2)$;
            end;
        $k := k+2$;
    end;
```

For patterns of length 3 the recurrence relations become more complex; they depend on more than simply the length of the strings and knowledge about characters at the boundaries. The determination of an optimum strategy in this case remains an open problem. The algorithm sketched at the beginning of this section shows that an average of $O(n(\log m)/m)$ bit inspections suffices over a binary alphabet. Clearly $\lfloor n/m \rfloor$ is a lower bound, since the algorithm must inspect at least one bit in any block of n consecutive bits. The pattern a^m can be handled with $O(n/m)$ bit inspections on the average; but it seems reasonable to conjecture that patterns of length m exist for arbitrarily large m, such that an average of at least $cn(\log m)/m$ bits must be inspected for all large n. Here c denotes a positive constant, independent of m and n.

Acknowledgment. Robert S. Boyer and J. Strother Moore suggested many important improvements to early drafts of this postscript, especially in connection with errors in the author's first attempts at proving the linearity theorem.

REFERENCES

[1] Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

[2] Alfred V. Aho and Margaret J. Corasick, *Efficient string matching: An aid to bibliographic search*, Comm. ACM, 18 (1975), pp. 333–340.

[3] M. Beeler, R. W. Gosper and R. Schroeppel, *HAKMEM*, Memo No. 239, M.I.T. Artificial Intelligence Laboratory, Cambridge, Mass., 1972.

[4] Robert S. Boyer and J. Strother Moore, *A fast string searching algorithm*, manuscript dated December 29, 1975; Stanford Research Institute, Menlo Park, Calif., and Xerox Palo Alto Research Center, Palo Alto, Calif.

[5] S. A. Cook, *Linear time simulation of deterministic two-way pushdown automata*, Information Processing 71, North-Holland, Amsterdam, 1972, pp. 75–80.

[6] Pascal Diethelm and Peter Roizen, *An efficient linear search for a pattern in a string*, unpublished manuscript dated April, 1972; World Health Organization, Geneva, Switzerland.

[7] N. J. Fine and H. S. Wilf, *Uniqueness theorems for periodic functions*, Proc. Amer. Math. Soc., 16 (1965), pp. 109–114.

[8] Michael J. Fischer and Michael S. Paterson, *String matching and other products*, SIAM-AMS Proc., vol. 7, American Mathematical Society, Providence, R.I., 1974, pp. 113–125.

[9] ZVI GALIL, *On converting on-line algorithms into real-time and on real-time algorithms for string-matching and palindrome recognition*, SIGACT News, 7 (1975), No. 4, pp. 26–30.

[10] E. N. GILBERT, *Synchronization of binary messages*, IRE Trans. Information Theory, IT-6 (1960), pp. 470–477.

[11] SHEILA A. GREIBACH, *The hardest context-free language*, this Journal, 2 (1973), pp. 304–310.

[12] MALCOLM C. HARRISON, *Implementation of the substring test by hashing*, Comm. ACM, 14 (1971), pp. 777–779.

[13] RICHARD M. KARP, RAYMOND E. MILLER AND ARNOLD L. ROSENBERG, *Rapid identification of repeated patterns in strings, trees, and arrays*, ACM Symposium on Theory of Computing, vol. 4, Association for Computing Machinery, New York, 1972, pp. 125–136.

[14] DONALD E. KNUTH, *Fundamental Algorithms, The Art of Computer Programming*, Vol. 1, Addison-Wesley, Reading, Mass., 1968; 2nd edition 1973.

[15] ———, *Sequences with precisely $k+1$ k-blocks*, Solution to problem E2307, Amer. Math. Monthly, 79 (1972), pp. 773–774.

[16] ———, *On the translation of languages from left to right*, Information and Control, 8 (1965), pp. 607–639.

[17] ———, *Structured programming with* **go to** *statements*, Computing Surveys, 6 (1974), pp. 261–301.

[18] DONALD E. KNUTH, JAMES H. MORRIS, JR. AND VAUGHAN R. PRATT, *Fast pattern matching in strings*, Tech. Rep. CS440, Computer Science Department, Stanford Univ., Stanford, Calif., 1974.

[19] R. C. LYNDON AND M. P. SCHÜTZENBERGER, *The equation $a^M = b^N c^P$ in a free group*, Michigan Math. J., 9 (1962), pp. 289–298.

[20] GLENN MANACHER, *A new linear-time on-line algorithm for finding the smallest initial palindrome of a string*, J. Assoc. Comput. Mach., 22 (1975), pp. 346–351.

[21] A. MARKOFF, *Sur une question de Jean Bernoulli*, Math. Ann., 19 (1882), pp. 27–36.

[22] J. H. MORRIS, JR. AND VAUGHAN R. PRATT, *A linear pattern-matching algorithm*, Tech. Rep. 40, Univ. of California, Berkeley, 1970.

[23] A. O. SLISENKO, *Recognition of palindromes by multihead Turing machines*, Dokl. Steklov Math. Inst., Akad. Nauk SSSR, 129 (1973), pp. 30–202. (In Russian.)

[24] KEN THOMPSON, *Regular expression search algorithm*, Comm. ACM, 11 (1968), pp. 419–422.

[25] B. A. VENKOV, *Elementary Number Theory*, Wolters-Noordhoff, Groningen, the Netherlands, 1970.

[26] PETER WEINER, *Linear pattern matching algorithms*, IEEE Symposium on Switching and Automata Theory, vol. 14, IEEE, New York, 1973, pp. 1–11.

SIAM J. COMPUT.
Vol. 6, No. 2, June 1977

STABLE SORTING AND MERGING WITH OPTIMAL SPACE AND TIME BOUNDS*

LUIS TRABB PARDO†

Abstract. This work introduces two algorithms for stable merging and stable sorting of files. The algorithms have optimal worst case time bounds, the merge is linear and the sort is of order $n \log n$. Extra storage requirements are also optimal, since both algorithms make use of a fixed number of pointers. Files are handled only by means of the primitives exchange and comparison of records and basic pointer transformations.

Key words. stable sorting, stable merging, minimal storage, optimal time bounds

1. Introduction. An algorithm which rearranges a file is said to be stable if it keeps records with equal keys in their initial relative order. This work presents an algorithm for merging two contiguous files in a stable manner (the PARTITION MERGE). As an immediate application of this, a stable algorithm to sort a file (the PARTITION MERGE SORT) is given.

The algorithms attain optimal worst case bounds with respect to time, the merge is of order n and the sort is of order $n \log n$. Both algorithms require only a fixed number of pointers for auxiliary storage.

While D. E. Knuth was preparing his book about sorting techniques, he noted that the known algorithms for stable sorting either were of order n^2 or they used approximately n pointers for additional memory space. Therefore he asked ([3 § 5.5, exercise 3]) whether it was possible to do stable sorting in less time than order n^2, using at most $O(\log n)$ pointers for additional storage. The first progress on this problem was made by R. B. K. Dewar [1], who developed a stable sorting algorithm or order $n^{1.5}$, using $O(\log n)$ pointers. Further improvements in the running time were made by V. Pratt [6], F. Preparata [7], R. Rivest [8], and A. Nijenhuis [5]. E. C. Horvath [2] constructed stable merging and sorting algorithms with optimal time and space bounds; however, his algorithms involve the operation of key modification; thus they apply only to files in which the key is explicitly stored within the record.

In that respect the algorithms here presented are completely general, since they treat files as sequences of unmodifiable records, with the keys evaluated from the record contents and not necessarily stored within them.

The algorithms in the present paper make use of a minimum set of primitive operations on files (exchange and comparison) and in this sense appear to offer the final solution to Knuth's problem, except of course for questions dealing with the optimum constants of proportionality in the time and space bounds.

The partition merge strategy (presented in § 3) is the key to the PARTITION MERGE algorithm and was inspired by the work of Kronrod on a nonstable

* Received by the editors December 19, 1974, and in revised form January 6, 1976.

† Computer Science Department, Stanford University, Stanford, California 94305. This research was supported by the National Science Foundation Grant GJ-36473X.

merging algorithm with optimal time and space bounds ([4]; see also [3, answer to exercise 5.2.4-18]).

2. Basic concepts. This section presents the notation used throughout the paper and describes a set of elementary operations on files that will be used for further definitions of more complex transformations.

2.1. Notation. A *record* R is a unit of information; its contents cannot be altered. The *key* k of a given record R results from the evaluation of a certain function K applied to R, $k=K(R)$. A *file* $\mathcal{F}$ is a sequence of records, $\mathcal{F}=\langle R_1, R_2, \cdots, R_i, \cdots, R_n\rangle$. Each position in a file has associated with it a *pointer value*, an integer in the range $[1, n]$.

If i and j are pointers, two primitive operations (and only these) may be used to access the file:

(a) an *exchange primitive*, denoted by exchange (i, j) that exchanges R_i with R_j;

(b) a *comparison primitive*, denoted by $F(i)\leqq F(j)$, whose value is true if and only if $K(R_i)\leqq K(R_j)$. Since the other relations $<$, $=$, $\neq$, $\geqq$, $>$ can be easily expressed in terms of one or two $\leqq$'s they will be used in the definition of algorithms, as a shorthand for the corresponding relation expressed in terms of the $\leqq$ primitive.

A *block* U (of *length* p) is a subsequence of p consecutive elements of $\mathcal{F}$, $U=\langle R_m, R_{m+1}, \cdots, R_{m+p-1}\rangle$. The length of U will be denoted by $|U|$; thus in the above case $|U|=p$. The block U will be also identified by the pointers to its first and last elements and denoted by $F[m:m+p-1]$. The first and last records of U will be first$(U)=R_m$ and last$(U)=R_{m+p-1}$. The term *prefix* (*suffix*) of U will refer to an initial (final) sequence of contiguous records of the block U.

The *number of distinct keys* in a block U will be $\lambda(U)$. Obviously $\lambda(U)\leqq|U|$, with the case $\lambda(U)=|U|$ corresponding to a block composed of records with distinct keys.

A *segment* X is a sequence of contiguous blocks

$$X=X_1X_2\cdots X_l\cdots X_q.$$

A segment will be also recorded as a block with the notations $|X|$, first(X), last(X) and $\lambda(X)$ having the previous meaning.

Normally only nondecreasing order will be considered. The predicate ordered(U) is true if and only if the block U is ordered in nondecreasing order.

A *stable transformation* is a permutation of a file, that preserves the relative order of these records with equal keys. In particular, this work is concerned with two stable transformations: the *stable merge* of two contiguous ordered blocks U and V denoted by merge(U, V), and the *stable sort* of a block U (denoted by sort(U)).

In the examples a file will be represented by the actual sequence of records, with the keys explicitly written down.

Algorithms will be presented in ALGOL-like procedures. The language used will be ALGOL W with the addition of a new type **pointer**, whose range depends only on the length of the common file.

2.2. Some basic transformations using minimal extra storage. These transformations are used throughout the rest of this work so their definition and time bounds are included here, though they have been presented previously (especially see [2]). The reader is referred to [2] or [9] for a formal description of the algorithm and derivation of time bounds for each transformation.

In the following paragraphs U and V will denote the blocks $U=F[u_1:u_2]$ and $V=F[v_1:v_2]$.

2.2.1. Permutation of two contiguous blocks: PERMUTE(u_1, u_2, v_1, v_2). The permuting process is done by application of three successive reversals, and takes

$$T_{\mathrm{PERM}}(U, V)=O(|U|+|V|). \tag{2.1}$$

2.2.2. Stable insertion of two contiguous ordered blocks: INSERT(u_1, u_2, v_1, v_2, f_1, f_2). The file $\mathcal{F}=AUVB$ is transformed into $\mathcal{F}=AV'UV''B$, where $V'V''=V$ and last $(V')<\mathrm{first}(U)\leqq\mathrm{first}(V'')$. The pointers are set in such a way that

$$U=F[u_1:u_2],\quad V'=F[v_1:v_2]\quad\text{and}\quad V''=F[f_1:f_2].$$

As a direct consequence of the above definition, we have

CLAIM 2.1. *Let U, V, V' and V'' be as above, and $U=U'U''$; then* merge$(U, V)=$ merge(U', V') merge(U'', V'').

The time bounds result:

$$T_{\mathrm{INS}}(U, V)=O(|U|+|V'|)\quad[=O(|U|+|V|)]. \tag{2.2}$$

2.2.3. Direct merge of two contiguous ordered blocks: BLOCK_MERGE_FORWARD(u_1, u_2, v_1, v_2) and BLOCK_MERGE BACKWARD(u_1, u_2, v_1, v_2). The file $\mathcal{F}=AUVB$ becomes $\mathcal{F}=A$ merge$(U, V)B$. The forward merge is accomplished by an iterative process of insertions of successively smaller suffixes of U into successively smaller suffixes of V. Thus, after a stable insertion of U into V as in § 2.2.2 yielding V_1UV_2, U is partitioned U_1U_2 (where U_2 is the largest subblock with first$(U_2)>$ first(V_2)), and the problem reduces to merge(U_2, V_2). The backward merge is similar, but the insertions are done in a backwards direction.

The time bounds for the forward and backward merges are:

$$T^{(\mathrm{forw.})}_{\mathrm{BLOCKM}}(U, V)=O(|U|\lambda(U))+O(|V'|) \tag{2.3}$$

where V' is that prefix of V $(V=V'V'')$ such that last$(V')<$ last$(U)\leqq$ first(V'');

$$T^{(\mathrm{back.})}_{\mathrm{BLOCKM}}(U, V)=O(|V|\lambda(V))+O(|U''|) \tag{2.4}$$

where U'' is that suffix of U $(U=U'U'')$ such that last$(U')\leqq$ first$(V)<$ first(U'').

Instead of introducing the definitions of V' and U'' the block merge processes could have been bounded by the overall lengths $|V|$ and $|U|$, but these bounds pretend to emphasize the fact that the running time is only a function of the elements that are actually exchanged by the process.

3. The partition merge strategy. This section outlines the basic strategy on which the partition merge algorithm is based, without considering either storage requirements or time bounds.

The first subsection introduces the segment insertion process, a stable transformation that is basic to the stable merge, while the second subsection analyzes the strategy itself.

3.1. The segment insertion process. This stable transformation deals with two contiguous ordered blocks U and V, of length equal to a multiple of a given value f. This last condition on the length allows treating U and V as segments of blocks of length f, and thus

$$
\begin{aligned}
&U = U_1 \cdots U_i \cdots U_k \quad \text{and}\\
&V = V_1 \cdots V_j \cdots V_l\\
(3.1)\qquad &\quad \text{for some } k>0 \quad \text{and} \quad l>0,\\
&\quad \text{with the block length}\\
&\quad |U_i| = |V_j| = f\\
&\qquad\qquad\qquad\qquad \text{for } 1 \leqq i \leqq k \quad \text{and} \quad 1 \leqq j \leqq l.
\end{aligned}
$$

Informally the segment insertion can be described as a permutation of the sequence of blocks $U_1 \cdots U_k V_1 \cdots V_l$ yielding the minimum number of inversions, but, of course, being stable.

In order to characterize such a permutation it can be argued that any block U_i in U cannot go after any block in V that could contain a record with key equal to any key of the records in U_i. Thus a block U_i should be positioned between the contiguous blocks V_j and V_{j+1} such that

$$(3.2) \qquad \text{last}(V_j) < \text{first}(U_i) \leqq \text{last}(V_{j+1}).$$

(In order to make the above equation hold in every case, the fictitious blocks V_0 and V_{l+1} must be assumed, with $\text{last}(V_0) = -\infty$ and $\text{last}(V_{l+1}) = +\infty$.) Since (3.2) might yield the same value of j for various consecutive blocks $U_i, U_{i+1}, \cdots, U_{i+p}$, it must also be stated that the permutation must retain the original relative ordering of blocks in U and V. So in this case the final layout will contain the segment $V_j U_i U_{i+1} \cdots U_{i+p} V_{j+1}$.

Example 3.1. As an example, let us consider U and V as below, for a block size $f = 2$:

U					V			
1 2	2 2	2 3	4 5	6 8	1 1	2 3	3 3	5 5
a b	c d	e f	g h	i j	A B	C D	E F	G H
U_1	U_2	U_3	U_4	U_5	V_1	V_2	V_3	V_4

Applying (3.2) to U_1 we see that

$$\text{last}(V_0) = -\infty < \text{first}(U_1) = 1 \leqq \text{last}(V_1) = 1.$$

Thus U_1 will go before V_1. For the blocks U_2 and U_3,

$$\text{last}(V_1) = 1 < \text{first}(U_2) = 2 \leqq \text{last}(V_2) = 3$$

and

$$\text{last}(V_1) < \text{first}(U_3) = 2 \leqq \text{last}(V_2),$$

so U_2 and U_3 will be positioned between V_1 and V_2, with U_2 preceding U_3. After considering U_4 and U_5 it can be seen that the final permutation will be

1 2	1 1	2 2	2 3	2 3	3 3	4 5	5 5	6 8
a b	A B	c d	e f	C D	E F	g h	G H	i j
U_1	V_1	U_2	U_3	V_2	V_3	U_4	V_4	U_5

The final result of the segment insertion can be characterized as the sequence of segments

$$Y_1Z_1Y_2Z_2 \cdots Y_dZ_d \cdots Y_tZ_t \tag{3.3}$$

where $Y_1Y_2 \cdots Y_d \cdots Y_t = U$ and $Z_1Z_2 \cdots Z_d \cdots Z_t = V$ and all the segments Y_d and Z_d containing at least one block, with the possible exception of Y_1 and Z_t.

Renaming Y_d and Z_d as

$$Y_d = Y_d'L_d \quad \text{with } |L_d| = f \quad \text{and} \quad Z_d = F_dZ_d' \quad \text{with } |F_d| = f \tag{3.4}$$

(that is, L_d is the last block in Y_d and F_d is the first one in Z_d), the following restrictions apply to the layout in (3.3):

$$\text{(i)} \quad \text{last}(Z_{d-1}) < \text{first}(Y_d), \qquad 1 < d \leqq t, \tag{3.5}$$

$$\text{(ii)} \quad \text{first}(L_d) \leqq \text{last}(F_d), \qquad 1 \leqq d \leqq t. \tag{3.6}$$

The characterization given by (3.3) to (3.6) is no more than a formal statement of the initial considerations. Thus in Example 3.1,

$$\begin{aligned} &Y_1 = U_1, && Z_1 = V_1, \\ &Y_2 = U_2U_3, && Z_2 = V_2V_3, \\ &Y_3 = U_4, && Z_3 = V_4, \\ &Y_4 = U_5, && Z_4\text{: empty.} \end{aligned}$$

Equations (3.5) and (3.6) state boundary relations between contiguous segments. Somehow they give us the hint that a merge of U and V could be reduced after segment inserting U and V, to a sequence of "local" merges of the pairs of segments Y_d and Z_d. That is the idea underneath the partition merge strategy and so it is the topic of the next subsection.

3.2. Description of the partition merge strategy. Let U and V be two contiguous ordered blocks of length greater than a given value f

$$|U| > f \quad \text{and} \quad |V| > f. \tag{3.7}$$

For the sake of simplicity, and only for the time being, it will be assumed that U is of length equal to a multiple of f

$$|U| = k \cdot f \quad \text{for } k \geqq 1. \tag{3.8}$$

The partition merge will proceed in the following way:

(a) Segment insert U and the longest prefix of V of length equal to a multiple of f.

(b) "Finish up" the merge, by means of local merges.

So let

$$\begin{aligned} U &= U_1 \cdots U_i \cdots U_k, \\ V &= V_1 \cdots V_j \cdots V_l T_v \end{aligned} \tag{3.9}$$

$$\text{with } |V_i| = |V_j| = f \quad \text{and} \quad |T_v| < f.$$

The *segment insertion* of U and $V_1 \cdots V_l$ yields

$$Y_1 Z_1 \cdots Y_d Z_d \cdots Y_{t-1} Z_{t-1} Y_t Z_t T_v$$

with the segments Y_d and Z_d as described in (3.3) to (3.6) of the previous subsection.

In order to analyze the *finish up* process we shall first consider the rightmost portion of the file, in particular the situation at the boundary of Y_t and Z_t. It is assumed that Z_t is not empty. The case Z_t empty will be quite similar.

By comparing last(Y_t) with first(Z_t) two cases may arise:

(i) If last$(Y_t) \leqq$ first(Z_t) then the segment $Y_t Z_t T_v$ is already in order and, what is more important, in its *final position within the merged file.* This last statement is a direct consequence of the segment insertion definition, since by (3.5)

$$\text{last}(Z_{t-1}) < \text{first}(Y_t) \tag{3.10}$$

and so all records of $Z_1 \cdots Z_{t-1}$ must precede first(Y_t). But also last $(Y_{t-1}) \leqq$ first(Y_t) because U was originally in order. Thus, all the elements to the left of first(Y_t) must precede it, so the above statement is true. Then nothing needs to be done about this segment, and the finish up proceeds by replacing t by $t-1$.

(ii) If last$(Y_t) >$ first(Z_t) it is going to be necessary to proceed with the finish up of the segment $Y_t Z_t T_v$, as described below.

The finish up of $Y_t Z_t T_v$ will consist of three steps. In order to describe them, let us adopt the notation of the previous subsection, and for reasons that will be immediately clear, let us rename T_v as C_{t+1}. By doing so, the rightmost portion of the file can be written as

$$Z_{t-1} Y'_t L_t F_t Z'_t C_{t+1} \tag{3.11}$$

where $Y'_t L_t = Y_t$ and $F_t Z'_t = Z_t$ with $|L_t| = |F_t| = f$.

This initial disposition is depicted in Fig. 3.1(a). Notice that Fig. 3.1 shows the values of the keys along the vertical axis, thus displaying the relative ordering of records.

The *first step* in the finish up process is to stable insert L_t into F_t, thus transforming $L_t F_t$ into $F'_t L_t F''_t$, such that

$$\text{last}(F'_t) < \text{first}(L_t) \leqq \text{first}(F''_t). \tag{3.12}$$

Figure 3.1(b) shows the situation after the first step. It can be seen that all the elements in L_t and $F''_t Z'_t C_{t-1}$ are greater or equal to those towards the left of L_t. This last assertion can be formally stated as the following claim.

CLAIM 3.1. *After step* 1, first(L_t) *is already in its final position within the merged file, and the overall merge has been reduced to the respective merge of the records to the left and to the right of* first(L_t).

Proof. All the elements to the right of first(L_t) are greater than or equal to it since

(a) those originally in U are greater than or equal to first(L_t), by the initial order of U;

(b) those originally in V are greater than or equal to first(F''_t), and, by (3.12), it is first(L_t) $\leqq$ first(F''_t). (The block F''_t is never empty, since first(L_t) $\leqq$ last(F_t), by (3.6), and then by (3.12) at least last(F_t) must belong in F''_t.)

Similarly the elements to the left of first(L_t) are less than or equal to it:

(a) those originally in U by the initial ordering;

(b) those originally in V are less than or equal to last(F'_t) and by (3.12)

$$\text{last}(F'_t) < \text{first}(L_t).$$

(In the case that F'_t resulted empty, the first element originally in V to the left of L_t is last(Z_{t-1}), and by (3.5) and the initial order of U last(Z_{t-1}) < first(Y_t) $\leqq$ first(L_t).)

Hence, the stability of the merge imposes that first(L_t) remain in its current place, since it was originally in U. And clearly the overall merge is reduced as stated in our claim. □

So, the *second step* in the finish up is the merge of L_t with $F''_tZ'_tC_{t+1}$.

Now let us consider Y'_t and F'_t, if Y'_t is nonempty. Assume that F'_t is of the form

$$(3.13) \qquad F'_t = C_tC'_t \quad \text{where last}(C_t) < \text{first}(Y'_t) \leqq \text{first}(C'_t).$$

(This partition of F'_t is identical to the one that would have been obtained by stable inserting Y'_t into F'_t.)

The third and last step in the finish up process of $Y_tZ_tC_{t+1}$ is the merge of Y'_t and F'_t. But by Claim 2.1 the merge of Y'_t and F'_t yields

$$(3.14) \qquad \text{merge}(Y'_t, F'_t) = C_t\ \text{merge}(Y'_t, C'_t).$$

If Y'_t is empty, the third step does not take place, and C_t is simply taken to be F'_t.

It is possible now to issue the following claim.

CLAIM 3.2. *After step* 3 *all the elements to the right of* C_t *are already in their final position.*

Proof. Only the case Y'_t nonempty needs to be considered. When Y'_t is empty the claim follows trivially from Claim 3.1.

Consider first(Y'_t). By (3.13) and the stability of the merge it must occupy the first position in merge(Y'_t, C'_t). Also by a similar reasoning as in Claim 3.1 (but applying (3.13) instead of (3.12)) it can be seen that it is in its final position within the merge. Clearly the rest of the elements in Y'_t and those in C'_t must be placed to the right of first(Y'_t), and by Claim 3.1 to the left of first(L_t). Then, all the elements in

$$\text{merge}(Y'_t, C'_t)\ \text{merge}(L_t, F''_tZ'_tC_{t+1})$$

must be in their final positions. □

The final result of the finish up of $Y_tZ_tC_{t+1}$ is shown in Fig. 3.1(c).

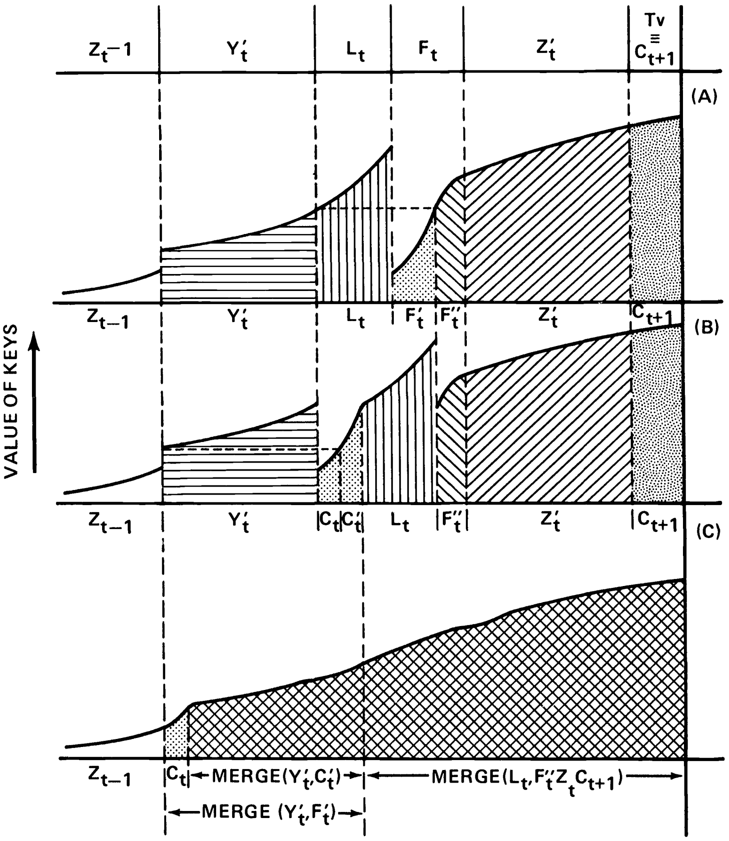


FIG. 3.1. *"Finish up" merges for the rightmost section of the file*
(a) *Initial layout*
(b) *After inserting L_t into F_t*
(c) *After merging forwards L_t into $F''_tZ_tC_{t+1}$ and merging backwards F'_t into Y'_t*

It is left to the reader to verify that the above process is valid also in the case of empty Z_t. The only difference is that C_{t+1} plays the role of F_t, and F''_t can therefore be empty.

The *overall finish up* will consist of the application of the above process successively to $Y_tZ_tC_{t+1}$, $Y_{t-1}Z_{t-1}C_t, \cdots, Y_1Z_1C_2$. The proof that this process yields the merge of U and V is a straightforward induction on t, using Claim 3.2.

A remark must be made about the initial restriction on the length of U, given by (3.8),

$$|U| = k \cdot f.$$

The general case

$$|U| \bmod f \neq 0$$

can be reduced to the one considered here by partitioning

$$U = U'U'' \tag{3.15}$$

(with $|U'| \bmod f = 0$ and $|U''| < f$) and stable inserting U'' into V, thus yielding $U'\ V'\ U''\ V''$. By Claim 2.1 the overall merge is reduced to

$$\text{merge}(U, V) = \text{merge}(U', V')\,\text{merge}(U'', V'')$$

and now the partition merge strategy can be applied to merge U' and V'.

So, in the general case the partition merge strategy will be:

(a) Insert the suffix U'' into V yielding $U'V'U''V''$.

(b) Segment insert U' into V'.

(c) Finish up the merge of U' and V': for $d = t, t-1, \cdots, 1$;

(c-1) Stable insert L_d into F_d.

(c-2) Merge L_d and $F''_d Z'_d C_{d+1}$.

(c-3) Merge Y'_d and F'_d

(d) Merge U'' and V''.

To conclude it must be noticed that in all the merge processes, at least one of the blocks to be merged is of length f or less. As it will be seen later this is a key fact in order to achieve linear time bounds.

4. Keeping storage requirements minimal. So far, no analysis has been made about extra storage needs for the actual implementation of the partition merge, and it is not obvious how to implement it using only absolute minimum ($O(\log n)$ bits) extra storage.

This section presents the concept of internal buffer, the implementation of another merging technique (the BUFFER MERGE), later used as a local merge for the finish up phase, and an implementation of the segment insertion process.

The usage of an internal buffer for a nonstable form of the buffer merge was first introduced by Kronrod [4]. Horvath further developed these ideas for the stable case (see [2]).

The segment insertion was also implemented by Horvath but this implementation (called the "stable Russian merge") requires the modification of keys. This last fact, explicitly avoided in this paper (records are considered nonmodifiable) also establishes constraints on the length of the files to be merged. The implementation here presented avoids those problems.

Timing analysis for the algorithms presented in subsections 4.1 and 4.2 can be found in [2] or [9]. Section 4.3 is treated in more detail in [9].

4.1. The concept of internal buffer. Let B be an ordered block consisting of records with distinct keys, that is,

$$\text{ordered}(B) \quad \text{and} \quad \lambda(B) = |B|. \tag{4.1}$$

Then B will be called an *internal buffer.*

Two useful characteristics of internal buffers may be singled out in advance:

(i) Permutations of an internal buffer do not affect the stability of a sorting or merging process (since the internal buffer might always be sorted back in a stable manner). This property is the basis of the buffer merge technique presented in the next subsection.

(ii) A given permutation of $|B|$ or less elements can be "stored" in a buffer B by simply permuting its elements correspondingly. This will be the key to the implementation of the segment insertion process, appearing in § 4.3.

Both properties could be used provided an internal buffer is present in the file being processed. Nevertheless, whenever a buffer is needed to process a block U it is possible to rearrange U in order to produce the desired buffer. Such a process will be called *buffer extraction.*

DEFINITION 4.1. Given an ordered block U, the extraction of a buffer B of at most l records transforms U into $U'B$, with U' and B also ordered blocks, such that

$$\text{merge}(U', B) = U \tag{4.2}$$

and B is an internal buffer

$$\text{(4.3) and} \quad \begin{gathered} |B| = \lambda(B) \text{ and ordered}(B) \\ |B| = \min(l, \lambda(U)). \end{gathered}$$

That is, the buffer extraction collects at most l distinct keyed records (or if the block U has only $\lambda(U) < l$ records with distinct keys, only $\lambda(U)$ are collected) placing them at the end of the original block; the rest of the records are compressed in U'.

In order to satisfy condition (4.2), for any sequence of records with equal keys in U, the last one is picked, so when merging U' and B, the original block U is obtained.

The following facts (analyzed in detail in [9]) must be pointed out:

(i) The extraction process needs only a fixed amount of pointers as extra storage.

(ii) The buffer extraction technique can be applied to a fixed number of contiguous ordered blocks; we will be interested in the extraction of a buffer out of the two blocks to be merged, and in this case the time bounds are

$$T_{\mathrm{BE}}(U, V, U', V', B, l) = O(|U| + |V|) + O(|B|^2). \tag{4.4}$$

4.2. Merging using an internal buffer: The BUFFER_MERGE. The BUFFER_MERGE of two contiguous ordered blocks U and V requires an internal buffer B of length $|B| \geqq \min(|U|, |V|)$.

In order to describe the buffer merge, let us assume first that $|V| \leqq |B|$, and let $B = B'B''$ with $|B'| = |V|$. Then the file $\mathscr{F}$ looks like

$$\mathscr{F} = \cdots UV \cdots B'B'' \cdots .$$

We first exchange V and B', obtaining

$$\mathscr{F} = \cdots UB' \cdots VB'' \cdots .$$

Now the actual merge takes place by repeatedly exchanging last(U) or last(V) (according to the relative values of their keys) with last(B') and redefining U, V and B' correspondingly. Once this process is exhausted we get

$$\mathscr{F} = \cdots \text{merge}(U, V) \cdots B'''B'' \cdots$$

(where B''' is a permutation of B').

The process briefly described above will be called BUFFER_MERGE_BACKWARD. A completely similar one applies when $|U| \leqq |B|$ (BUFFER_MERGE_BACKWARD).

The time bounds are

$$T^{(\text{back})}_{\text{BUFM}}(U, V, B) = O(|U''|) + O(|V|) \tag{4.5}$$

where $U = U'U''$ and $\text{last}(U') \leqq \text{first}(V) < \text{first}(U'')$.

(Equation (4.5) reiterates a point already considered when discussing the block merge (§ 2.2.3): The running time is bounded by the number of elements that are actually exchanged.)

Similarly, for the forward merge

$$T^{(\text{forw.})}_{\text{BUFM}}(U, V, B) = O(|U|) + O(|V'|) \tag{4.6}$$

where $V = V'V''$ and $\text{last}(V') < \text{last}(U) \leqq \text{first}(V'')$.

4.3. Implementation of the segment insertion process. This subsection describes how the segment insertion can be implemented with the aid of an internal buffer, using as extra storage only a fixed number of pointers.

Recalling the definition stated in § 3.1, the two contiguous ordered segments U and V

$$\begin{aligned} U &= U_1 \cdots U_i \cdots U_k, \\ V &= V_1 \cdots V_j \cdots V_l \end{aligned} \tag{4.7}$$

where $|U_i| = |V_j| = f$

are transformed into

$$Y_1 Z_1 \cdots Y_d Z_d \cdots Y_t Z_t$$

where the segments Y_d and Z_d are defined by (3.3)–(3.6).

With the segments $Z_{d-1} Y_d Z_d Y_d$ considered as

$$\begin{aligned} Z_{d-1} &= V_{j-r} \cdots V_{j-1}, \\ Y_d &= U_i \cdots U_{i+p}, \\ Z_d &= V_j \cdots V_{j+q}, \\ Y_{d+1} &= U_{i+p+1} \cdots U_{i+p+s}, \end{aligned} \tag{4.8}$$

equations (3.5) and (3.6) yield

$$\begin{aligned} &\text{last}(V_{j-1}) \\ &\quad < \text{first}(U_i) \leqq \cdots \leqq \text{first}(U_{i+p}) \\ &\quad \leqq \text{last}(V_j) \leqq \cdots \leqq \text{last}(V_{j+q}) \\ &\qquad < \text{first}(U_{i+p+1}). \end{aligned} \tag{4.9}$$

Equation (4.9) indicates an easy method to determine the final order of the blocks. Consider sequentially U_1, U_2, etc. until reaching the smallest p with

$$\text{last}(V_1) < \text{first}(U_p).$$

Then $U_1 \cdots U_{p-1}$ are the first blocks in the final permutation. Now consider V_1, V_2, etc. until reaching the smallest q with

$$\text{first}(U_p) \leqq \text{last}(V_q),$$

thus establishing that the sequence $V_1 \cdots V_{q-1}$ will come after U_{p-1}. The process is now repeated until U and V are exhausted.

The above process gives us a method to compute the permutation that must be applied to the blocks in UV. But somehow that permutation must be stored before permuting the blocks, since its definition is based on the original ordering of the blocks. Thus the algorithm will have two phases:

(a) Compute and "store" the permutation.

(b) Permute the blocks.

In order to "store" the permutation, an internal buffer will be used. The key point is that the permutation as defined in (4.9) can be computed by inspecting the blocks in the exact order in which they are going to be permuted. Then it is possible to "remember" the final position of each block by exchanging one of its elements (say the first one) with the element in the buffer that corresponds to its final position (recall that a buffer is an ordered block). After that, the permuting phase becomes simply a sorting process in which each block has as its key the key of its first element.

In order to permute the blocks, a sorting method that minimizes the number of exchanges, since they are block exchanges, must be chosen. The "straight selection sort" [3, § 5.2.3] is well suited for our purposes.

After this the records that were exchanged with those in the buffer are restored to their final position.

The dominant factor in the running time is the time needed to sort the blocks (marking the blocks and restoring exchanged records can be done in time linear on the number of blocks). With N blocks, each one of f records, the straight selection sort runs in $O(N^2) + O(N \cdot f)$ time. So since $|U| + |V| = Nf$ the time bounds for the segment insertion result,

$$T_{\text{SEGIN}}(U, V, f) = O((|U| + |V|)^2/f^2) + O(|U| + |V|). \tag{4.10}$$

5. The partition merge algorithm. Section 3 presented the partition merge strategy. In § 4 the necessary tools to keep storage requirements minimal were considered. With that background it is now possible to introduce the partition merge algorithm and bound its running time.

5.1. Description. The algorithm here presented closely follows the process introduced in § 3, except for the addition of an initial buffer extraction step and, of course, a final merging step to merge back the internal buffer previously obtained. Figures 5.1–5.7 illustrate the process on a particular example.

Let U and V be two contiguous ordered blocks to be merged. The following procedure defines the partition merge algorithm:

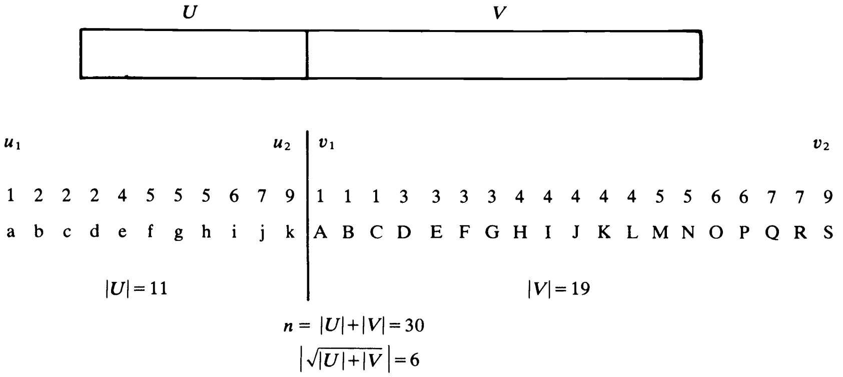
```
procedure partition merge (pointer value $u_1$, $u_2$, $v_1$, $v_2$);
begin comment: $U$ is $F[u_1 : u_2]$ and $V$ is $F[v_1 : v_2]$;
    pointer $n$, $f$, $b$, $t_1$, $t_2$, $v_3$, $v_4$, $l_1$, $l_2$, $w_1$, $w_2$, $w_3$, $w_4$, $p$;
    $n := v_2 - u_1 + 1$;
```

Step 1. Extract an internal buffer of length at most

$$\lceil\sqrt{|U|+|V|}\rceil.$$

```
buffer_extraction:
    BUFFER_EXTRACT2($u_1$, $u_2$, $v_1$, $v_2$, ceiling(sqrt($n$)), $b_1$, $b_2$);
    $b := b_2 - b_1 + 1$; $f :=$ floor($n/b$);
```

This step transforms UV into $U'V'B$, where B is an internal buffer of length $b = b_2 - b_1 + 1$. (See Figs. 5.1 and 5.2.)



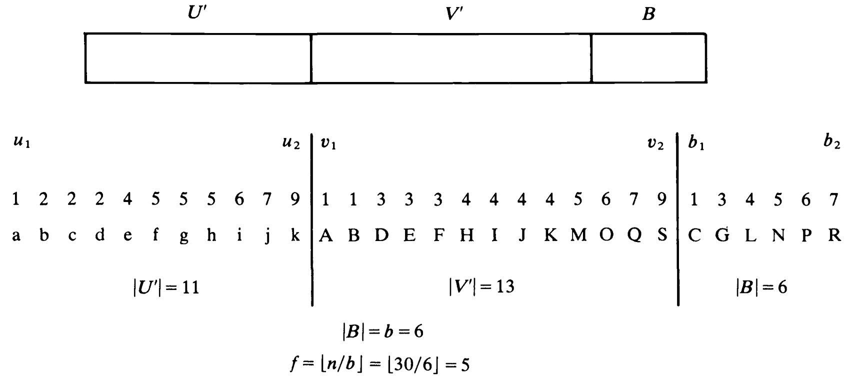
u_1										u_2	v_1																		v_2
1	2	2	2	4	5	5	5	6	7	9	1	1	1	3	3	3	3	4	4	4	4	4	5	5	6	6	7	7	9
a	b	c	d	e	f	g	h	i	j	k	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S

$|U| = 11$ $|V| = 19$

$n = |U| + |V| = 30$

$\lfloor\sqrt{|U|+|V|}\rfloor = 6$

FIG. 5.1. *Initial layout*



u_1										u_2	v_1												v_2	b_1					b_2
1	2	2	2	4	5	5	5	6	7	9	1	1	3	3	3	4	4	4	4	5	6	7	9	1	3	4	5	6	7
a	b	c	d	e	f	g	h	i	j	k	A	B	D	E	F	H	I	J	K	M	O	Q	S	C	G	L	N	P	R

$|U'| = 11$ $|V'| = 13$ $|B| = 6$

$|B| = b = 6$

$f = \lfloor n/b \rfloor = \lfloor 30/6 \rfloor = 5$

FIG. 5.2. *After Step* 1

Let $f = \lfloor n/b \rfloor$.

Step 2. If either $|U'|$ or $|V'|$ has length less than or equal to f, then merge them directly and proceed with the final step (merging back B).

```
check_lengths:
  **if** $(u_2 - u_1 + 1) \leqq f$
  **then begin**
      **if** $(u_2 - u_1 + 1) > b$
```

```
            then BLOCK_MERGE_FORWARD ($u_1, u_2, v_1, v_2$)
            else BUFFER_MERGE_FORWARD ($u_1, u_2, v_1, v_2, b_1, b_2$);
         go to merge_back_B;
      end
   else if $(v_2 - v_1 + 1) \leqq f$ then
      begin
         if $(v_2 - v_1 + 1) > b$
            then BLOCK_MERGE_BACKWARD ($u_1, u_2, v_1, p_2$)
            else BUFFER_MERGE_BACKWARD ($u_1, u_2, v_1, v_2, b_1, b_2$);
         go to merge_back_B;
      end;
```

Notice that depending on the length of the buffer, the algorithm chooses either block_merge or buffer_merge. This choice allows linear running time as will be analyzed below.

Step 3. Prepare things for segment insertion by getting rid of that suffix T_u of U' of length

$$|T_u| = |U'| \bmod f.$$

```
   insert_suffix:
      $t_2 := u_2$; $u_2 := u_2 - (u_2 - u_1 + 1)$ mod $f$; $t_1 := u_2 + 1$;
      comment: $U''$ is $F[u_1 : u_2]$ and $T_u$ is $F[t_1 : t_2]$;
      INSERT ($t_1, t_2, v_1, v_2, v_3, v_4$);
      comment: $V''$ is now $F[v_1 : v_2]$ and $V'''$ is $F[v_3 : v_4]$;
```

After the insertion $U'V'$ becomes $U''V''T_uV'''$, where $U''T_u = U'$ and $V''V''' = V'$. By the characteristics of stable insertion the merge of U' and V' is now reduced to the merge of U'' and V'' and that of T_u and V'''.

Now $|U''| \bmod f = 0$, by the choice of T_u, so U'' and V'' can be viewed as segments such that:

$$U'' = U_1 \cdots U_i \cdots U_k$$

and

$$V'' = V_1 \cdots V_j \cdots V_l T_v$$

where $|U_i| = |V_j| = f$ for $1 \leqq i \leqq k$ and $1 \leqq j \leqq l$ and $|T_v| < f$. (See Fig. 5.3.)

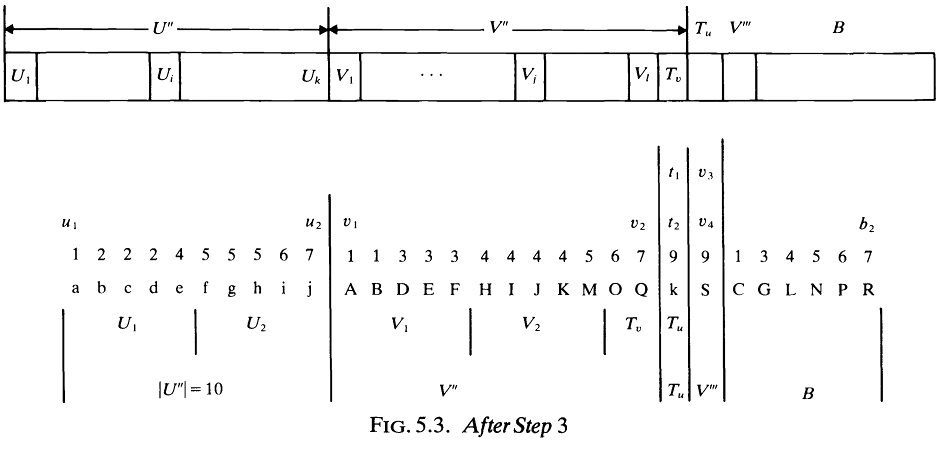


FIG. 5.3. *After Step* 3

Step 4. Segment insert $U_1 \cdots U_i \cdots U_k$ into $V_1 \cdots V_j \cdots V_l$.

segment_insertion:

SEGMENT_INSERT $(u_1, u_2, v_1, v_2-(v_2-v_1+1) \textbf{ mod } f, f, b_1, b_2)$;

The next step will be the finish up process (see § 3), but some discussion is needed first.

Assume that the layout after the segment insertion is $W_1 \cdots W_m \cdots W_{k+l}T_vT_uV'''B$, where $W_1 \cdots W_m \cdots W_{k+l}$ corresponds to $Y_1Z_1 \cdots Y_dZ_d \cdots Y_tZ_t$ as presented in § 3. Unfortunately there is no explicit information about the way the blocks W_m are grouped to form the segments Y_dZ_d. But fortunately the local merges must be performed only on those pairs Y_dZ_d such that last$(Y_d)>$first(Z_d); hence the finish up can be done by repeating the following sequence until the whole segment $W_1 \cdots W_m \cdots W_{k+l}T_v$ has been processed:

(a) In order to locate the next pair Y_dZ_d to be merged, scan to the left until a block W_m, such that last $(W_m)>$first(W_{m+1}) is found.

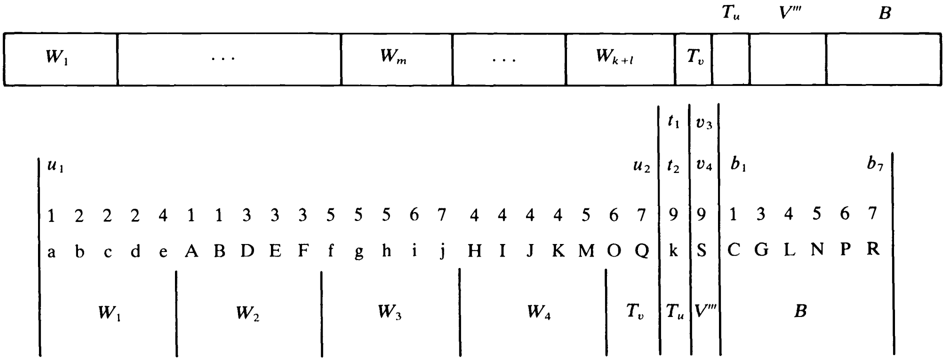
(b) Perform the local merge:

1st *step.* Insert W_m in W_{m+1}, thus transforming W_mW_{m+1} into $W'W_mW''$.

2nd *step.* Merge W_m forward.

3rd *step.* Merge W' backward.

Both definitions result in equivalent operation, if the merging method stops once the merge is complete. In this case the bounds are preserved simply by the existing order in the file, thus making it unnecessary to keep track of them. In other words, the grouping of $W_1 \cdots W_m \cdots W_{k+l}$ into $Y_1Z_1 \cdots Y_dZ_d \cdots Y_tZ_t$ is useful to prove that the algorithm works (and, as will be seen later, to compute its time bounds) but it is not needed to take it into account for implementation purposes. (See Fig. 5.4.)


T_u V''' B
W_1 ... W_m ... W_{k+l} T_v
t_1 v_3
u_1 u_2 t_2 v_4 b_1 b_7
1 2 2 2 4 1 1 3 3 3 5 5 5 6 7 4 4 4 4 5 6 7 9 9 1 3 4 5 6 7
a b c d e A B D E F f g h i j H I J K M O Q k S C G L N P R
W_1 W_2 W_3 W_4 T_v T_u V''' B

In the notation of § 4,

$$W_1W_2W_3W_4 \equiv Y_1Z_1Y_2Z_2 \qquad (t=2)$$

with the following grouping:

$$Y_1 = W_1 = U_1 \qquad Z_1 = W_2 = V_1$$

$$Y_2 = W_3 = U_2 \qquad Z_2 = W_4 = V_2.$$

FIG. 5.4. *After Step* 4

Step 5. Finish up the merge of $W_1 \cdots W_m \cdots W_{l+k}T_v$.

```
finish_up:
p := $v_2-(v_2-v_1+1) \bmod f$; if $p=v_2$ then $p := p-f$;
while $p>u_1$ do
  begin
    comment: find next pair $Y_dZ_d$ to be merged;
      while $(p>u_1) \wedge (F(p) \leqq F(p+1))$ do $p := p-f$;
    if $p>u_1$ then
      begin comment: local merge;
        comment: $W_m$ is $F[l_1:l_2]$, $W_{m+1}$ is $F[w_1:w_2]$;
        $l_1 := p-f+1$; $l_2 := p$;
        $w_1 := p+1$; $w_2 := \min(p+f, v_2)$;
        INSERT $(l_1, l_2, w_1, w_2, w_3, w_4)$;
        comment: now $W'$ is $F[w_1:w_2]$ and $W''$ is $F[w_3:w_4]$;
        comment: in order to do the merges
                 "forward (of $W_m$)" means $F[w_3:v_2]$,
                 "backward (of $W'$)" means $F[u_1:w_1-1]$;
        comment: depending on the size $b$ of the buffer the
                 algorithm chooses:
                     BUFFER_MERGE if $b \geqq f$
                     BLOCK_MERGE if $b<f$;
        if $b \geqq f$
          then begin
                 BUFFER_MERGE_FORWARD $(l_1, l_2, w_3, v_2, b_1, b_2)$;
                 BUFFER_MERGE_BACKWARD $(u_1, w_1-1, v_1, w_2,$
                   $b_1, b_2)$
               end
          else begin
                 BLOCK_MERGE_FORWARD $(l_1, l_2, w_3, v_2)$;
                 BLOCK_MERGE_BACKWARD $(u_1, w_1-1, w_1, w_2)$
               end;
        $p := p-f$
      end if_p
  end while_p;
```

It must be noticed that the algorithm chooses either BLOCK_MERGE or BUFFER_MERGE depending on the relative sizes of the blocks and the buffer.

Step 5 transforms the layout into

$$\text{merge}\,(U'', V'')T_uV'''B'$$

where B' is a permutation of B (and $B'=B$ if block_merge was used in Step 5). (See Fig. 5.5.)

Layout after segment insertion:

					t_1	v_3	
u_1 p_4	p_3	p_2	p_1	v_2	t_2	v_4	b_1 b_2
1 2 2 2 4	1 1 3 3 3	5 5 5 6 7	4 4 4 4 5	6 7	9	9	1 3 4 5 6 7
a b c d e	ABDEF	f g h i j	HI J KM	OQ	k	S	CGLNPR

When $p = p_2$, $F(p) > F(p+1)$, and the first local merge is done:

	p			
u_1	l_1 l_2	w_1 w_2	v_2	
1 2 2 2 4 1 1 3 3 3	5 5 5 6 7	4 4 4 4 5	6 7	9 . . .
a b c d e ABDEF	f g h i j	HI J KM	OQ	k

			w_3			
	w_1 w_2	l_1 l_2	w_4	v_2		
. . .	4 4 4 4	5 5 5 6 7	5	6 7 9	. . .	After inserting W_m into W_{m+1}
	HI J K	f g h i j	M	OQk		
	W'	W_m	W''			

1 2 2 2 4 1 1 3 3 3 4 4 4 4	5 5 5 5 6 6 7 7 9 . . .	After merging (BUFFER_MERGE is used)
a b c d e ABDEF HIJK	f g h Mi Oj Qk	
merge backward of W' stops here ↑	merge forward of W_m stops here ↑	

When $p = p_4$, again $F(p) > F(p+1)$, so

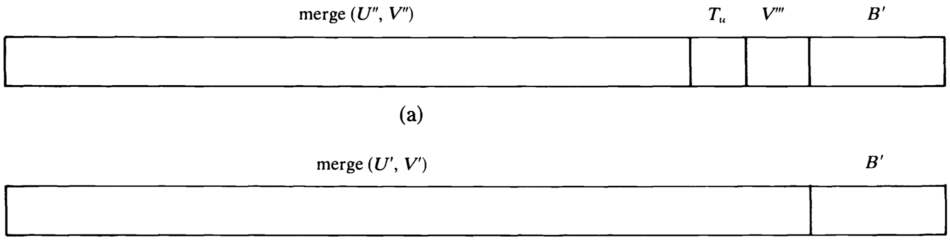
1 2 2 2 4	1 1 3 3 3	. . .	After inserting W_m (in this case yielding W' empty and $W'' = W_{m+1}$
a b c d e	ABDEF		
W_m	W''		

After merging:

	t_1	v_3	
u_1 v_2	t_2	v_4	b_1 b_2
1 1 1 2 2 2 3 3 3 4 4 4 4 4 5 5 5 5 6 6 7 7	9	9	1 6 5 3 4 7
a ABbcd DEFe HI J Kf g h Mi Oj Q	k	S	CPNGLR
merge (U'', V'')	T_u	V''	B'

FIG. 5.5. *The finish up process applied to the example*

This step completes the merge of U' and V', thus yielding merge(U', V')B'. (See Fig. 5.6.)



u_1											merge (U', V')												v_4			B'			
1	1	1	2	2	2	3	3	3	4	4	4	4	4	5	5	5	5	6	6	7	7	9	9	1	6	5	3	4	7
a	A	B	b	c	d	D	E	F	e	H	I	J	K	f	g	h	M	i	O	j	Q	k	S	C	P	N	G	L	R

(b)

FIG. 5.6. (a) *The situation after Step* 5 (b) *The result after Step* 6

Step 6. Merge T_u and V'''.

```
merge_T_u_V''':
  if $b \geqq t_2 - t_1 + 1$
    then BUFFER_MERGE_FORWARD ($t_1$, $t_2$, $v_3$, $v_4$, $b_1$, $b_2$)
    else BLOCK_MERGE_FORWARD ($t_1$, $t_2$, $v_3$, $v_4$);
```

Step 7. Sort B' and merge it backward.

```
 merge_back_B:
   STRAIGHT_INSERTION_SORT ($b_1$, $b_2$);
   BLOCK_MERGE_BACKWARD ($u_1$, $v_4$, $b_1$, $b_2$)
end partition_merge;
```

Here STRAIGHT_INSERTION_SORT (b_1, b_2) sorts B' into the original buffer B, according to the techniques described in [3, § 5.2.1]. And Step 7 finally yields the desired merge of U and V. (See Fig. 5.7.)

merge (U, V)

1	1	1	1	2	2	2	3	3	3	3	4	4	4	4	4	4	5	5	5	5	5	6	6	6	7	7	7	9	9
a	A	B	C	b	c	d	D	E	F	G	e	H	I	J	K	L	f	g	h	M	N	i	O	P	j	Q	R	k	S

FIG. 5.7. *Final result*

The storage requirements for the partition merge are the fixed number of pointers declared at the beginning (14 in total, though a more careful usage could have saved some) plus those needed by the different procedures called. Since those procedures (BUFFER_EXTRACT2, BLOCK_MERGE's, BUFFER_MERGE's, INSERT, SEGMENT_INSERT, and STRAIGHT_

INSERTION_SORT) require also a fixed amount of pointers (and clearly there is no recursive call involved) the overall storage requirements are absolute minimum, that is $O(\log n)$.

5.2. Time bounds for the partition merge algorithm. Each of the steps of the partition merge runs in time linear on the length of the merged file. Hence the overall time bounds result:

$$T_{\mathrm{PARTM}}(U, V) = O(|U| + |V|). \tag{5.1}$$

This subsection will very briefly present the derivation of bounds for the different steps (with Step 5 treated in a slightly more detailed way). The reader is referred to [9] for a complete analysis. Throughout this subsection, n will stand for $n = |U| + |V|$.

5.2.1. Time bounds for Steps 1, 2, 3, 4, 6 and 7. Since $|B| \leqq \lceil\sqrt{n}\rceil$, bounds for Step 1 are

$$T_1 = O(n) + O(\lceil\sqrt{n}\rceil^2) = O(n). \tag{5.2}$$

Step 2 yields the merge of U' and V' when $|U'| \leqq f$ or $|V'| \leqq f$. Assume $|U'| \leqq f$. When buffer merge is used, the time is trivially $O(n)$. Block merge is used when $|U'| > b$, and in this case it can be shown that $\lambda(U') \leqq b$. Hence

$$T_2 = 0(|U'|\lambda(U)) + O(|V'|) = O(f \cdot b) + O(|V|) = O(n).$$

The case $|V'| \leqq f$ is similar, and thus

$$T_2 = O(n). \tag{5.3}$$

In Step 3, the insertion takes

$$T_3 = O(|T_u| + |V|) = O(|U| + |V|) = O(n). \tag{5.4}$$

The fact that $f > (n-b)/b$, and $|U''| + |V_1 \cdots V_l| \leqq n - b$ bound the segment insertion in Step 4, by

$$T_4 = O((n-b)^2/f^2) + O(n-b) = O(n). \tag{5.5}$$

By a completely similar analysis to the one for Step 2

$$T_6 = O(n). \tag{5.6}$$

Finally, in Step 7, the sort takes $O(b^2)$ and the block merge $O(n-b) + O(b\lambda(B))$, but $\lambda(B) = b$

$$T_7 = O(b^2) + O(n-b) + O(b^2) = O(n). \tag{5.7}$$

5.2.2. Time bounds for Step 5. In order to compute these bounds, it is convenient to resort to the notation in § 3. The segment insertion in Step 4 transforms $U''V''$ into $Y_1Z_1 \cdots Y_dZ_d \cdots Y_tZ_tC_{t+1}$, where

$$U'' = Y_1 \cdots Y_d \cdots Y_t \quad \text{and} \quad V'' = Z_1 \cdots Z_d \cdots Z_tC_{t+1}.$$

Also, let L_d be the last block of Y_d, and F_d the first one of Z_d, thus renaming

$$Y_d = Y'_dL_d \quad \text{and} \quad Z_d = F_dZ'_d \quad \text{with } |L_d| = |F_d| = f. \tag{5.8}$$

The finish up process of Step 5 can be viewed as:

```
for $d := t$ step $-1$ until 1 do
  begin
    insert $L_d$ into $F_d$, transforming $L_dF_d$ into $F'_dL_dF''_d$;
    merge $L_d$ forward;
    merge $F'_d$ backward;
  end
```

Time bounds for insertion (T_I). Clearly

$$T_I=\sum_d T_{\mathrm{INS}}(L_d, F_d)=O\Big(\sum_d (|L_d|+|F_d|)\Big)=O(|U''|+|V''|). \tag{5.9}$$

Time bounds for merges $(T_F$ and $T_B)$. The time bounds for block and buffer merge are functions of those records that are actually exchanged (see remarks at the end of §§ 2.2.3 and 4.2).

Claim 3.2 shows that during the finish up of Y_dZ_d, all the elements to the right of C_{d+1} are already in their final position. Hence when merging L_d forward, it merges into $F''_dZ'_dC_{d+1}$, regardless of how far to the right of C_{d+1} the merge limits point. So in order to bound the running time the process "merge L_d forward" will be regarded as "merge L_d forward into $F''_dZ'_dC_{d+1}$".

A quite similar reasoning shows that "merge F'_d backward" is equivalent to "merge F'_d backward into Y'_d".

There are two cases depending on whether block or buffer merge is used, and they will be analyzed separately:

(a) Case $b \geqq f$, BUFFER_MERGE: By (4.5) and (4.6)

$$\begin{aligned} T_F+T_B&=\sum_d O(|L_d|)+\sum_d O(|F''_dZ'_dC_{d+1}|)+\sum_d O(|F'_d|)+\sum_d O(|Y'_d|)\\ &=\sum_d O(|Y'_dL_d|)+\sum_d O(|F'_dF''_dZ'_d|)+\sum_d O(|C_{d+1}|) \qquad (5.10)\\ &=O(|U''|+|V''|). \end{aligned}$$

(b) Case $b<f$, BLOCK_MERGE: Here it is

$$\begin{aligned} T_F+T_B&=\sum_d O(|L_d|\lambda(L_d))+\sum_d O(|F''_dZ'_dC_{d+1}|)\\ &\quad+\sum_d O(|Y_d|)+\sum_d (|F'_d|\lambda(F'_d))\\ &=f\cdot O\Big(\sum_d \lambda(L_d)\Big)+O(V'')+O(U'')+fO\Big(\sum_d \lambda(F'_d)\Big). \end{aligned}$$

Now, it can be shown that if U is a segment $U=U_1U_2\cdots U_k$, then $\lambda(U)+k\geqq\sum_{1\leqq i\leqq k}\lambda(U_i)$, and then

$$T_F+T_B=O(|U''|+|V''|)+fO(\lambda(U'')+t)+fO(\lambda(V'')+t).$$

And since $f > b$ implies $\lambda(UV) = b$ and $t \leqq b$

$$T_F + T_B = O(n) + 2fO(b+b) = O(n). \tag{5.11}$$

Overall bounds for Step 5. Clearly by (5.10) and (5.11)

$$T_5 = O(n). \tag{5.12}$$

6. The partition merge sort. The availability of a linear time merge algorithm gives rise to the possibility of an $(n \cdot \log n)$ time bounded sort. A few slightly different variations of the same basic strategy are possible, and [9] presents one of them in detail.

The sorting strategy consists of successive merging passes over the entire block to be sorted, each pass merging pairs of blocks of length 1, 2, $4, \cdots, 2^i, \cdots, 2^k$ until the entire file is sorted.

The time bounds for this sorting method are very easily computed since when sorting a block U, the merging passes are repeated up to

$$k = \lceil \log |U| \rceil - 1,$$

and each pass is linear on the length of U. Then

$$T_{\text{P\_M\_SORT}}(U) = O(|U| \log |U|). \tag{6.1}$$

Clearly the partition merge sort can be implemented using a fixed amount of pointers over those used by the partition merge.

7. Conclusions. The most interesting of the results presented here is the partition merge algorithm, since as the reader was able to see, the partition merge sort resulted as a straightforward consequence of it.

By analyzing the previously published results, especially the work by Horvath [2], it can be concluded that there were two considerations that led to the general result presented here.

First, the utilization of an internal buffer, without any modification of the keys, to "mark" a permutation of a segment, allowed the segment insertion process to be implemented within extra storage bounds of $O(\log n)$ bits.

Secondly, the adaptivity of the algorithm to the characteristics of the file being processes (by proper choice of either BUFFER or BLOCK_MERGE) resulted in a linear time "finish up".

It is interesting to note that the operation "$p+q$" on pointers is strictly needed only for the permutation of blocks in the SEGMENT_INSERT process (§ 4.3). All the other sums of pointer values could have been realized by successive "$p+1$" operations within the same time and space bounds. It remains an open question whether these minimum time and space bounds are obtainable only with the primitives "exchange(p, q)", "$F(p) \leqq F(q)$", "$p \pm 1$", "$p = q$", and "$p := q$".

Acknowledgments. The author is greatly indebted to Professor Donald E. Knuth, not only for his help and encouragement in the present work, but for the many things the author learned from him.

REFERENCES

[1] Robert B. K. Dewar, *A stable minimum storage algorithm*, Information Processing Letts., 2 (April, 1974), pp. 162–164.

[2] EDWARD C. HORVATH, *Efficient minimum extra space stable sorting*, Ph.D. thesis, Dept. of Electrical Engrg., Princeton Univ., Princeton, N.J.; also in Proc. 6th Annual Symp. on Theory of Computing (SIGACT 6), Association for Computing Machinery, New York, 1974, pp. 194–215.

[3] DONALD E. KNUTH, *The Art of Computer Programming, Vol.* 3: *Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[4] M. A. KRONROD, *An optimal ordering algorithm without a field of operation*, Dok. Akad. Nauk SSSR, 186 (1969), pp. 1256–1258.

[5] ALBERT NIJENHUIS, private communication, 1974.

[6] VAUGHAN PRATT, private communication, 1974.

[7] F. P. PREPARATA, *A fast stable sorting algorithm with absolutely minimum storage*, Theoretical Computer Science, 1 (1975), pp. 185–190.

[8] RONALD RIVEST, *A fast stable minimum storage sorting algorithm*, Rep. 43, Institute de Recherche d'Informatique et d'Automatique, Rocquencourt, France, Dec. 1973.

[9] LUIS TRABB PARDO, *Stable sorting and merging with optimal space and time bounds*, Comput. Sci. Rep. STAN-CS-74-470, Stanford Univ., Stanford, CA, Dec. 1974.

SIAM J. COMPUT.
Vol. 6, No. 2, June 1977

THE DEPTH OF ALL BOOLEAN FUNCTIONS*

W. F. McCOLL† AND M. S. PATTERSON‡

Abstract. Every Boolean function of n arguments has a circuit of depth $n+1$ over the basis $\{f|f:\{0,1\}^2\to\{0,1\}\}$.

Key words. depth, Boolean functions, complexity, circuit, formula

1. Introduction. Spira showed in [3] that any n argument Boolean function has a circuit of depth $n+\log^* n$ where

$$\log^* (n) = (\text{if } n \leqq 1 \text{ then } 0 \text{ else } \log^* (\log_2 n) + 1).$$

Upper bounds on depth for specific values of n, given by Preparata and Muller [2], are

$$\begin{array}{ll} n & \text{for } n \leqq 8, \\ n+1 & \text{for } n \leqq 2^8+8=264, \\ n+2 & \text{for } n \leqq 2^{264}+264, \\ \text{etc.;} & \end{array}$$

whereas Knuth has shown, by computer analysis, that there are four argument Boolean functions requiring depth 4.

In this paper, we describe constructions which yield an upper bound of $n+1$ for all values of n, and bounds of the form $n+O(1)$ for circuits over various restricted bases.

2. Schemes. Our present constructions, and all previous ones for minimizing depth that we know of, have the property of being "uniform" for all functions of n arguments. The same directed graph with the same assignment of arguments to inputs is used for all the functions, the necessary variation being only in the assignment of base functions to the nodes. Lupanov's construction for minimizing formula size [1] is notable for escaping this form. We formalize this restriction in our definition of "circuit scheme" and show that for schemes our construction achieves the optimal depth to within an additive constant.

Let $B=\{0,1\}$ and $B_n=\{f:B^n\to B\}$. $X_n=\langle x_0, x_1, \cdots, x_{n-1}\rangle$ is the set of formal arguments we shall use in formulae and circuits.

DEFINITION. A *circuit scheme* is a connected acyclic directed graph in which nodes have either in-degree 2 (*gates*) in which case the pair of incoming arcs are ordered, or else in-degree 0 (*input nodes*) in which case an argument x_i is assigned to the node. A *formula scheme* is a circuit scheme in which all gates have out-degree at most one.

Let $C_n \subseteq B_n$ and $b \subseteq B_2$. A circuit scheme S *covers* C_n *over basis* b if for each $f \in C_n$ there is an assignment of functions from b to the gates of S such that the

* Received by the editors October 9, 1975, and in revised form July 7, 1976.

† Computer Science Department, Strathclyde University, Glasgow G1 1XH, Scotland. This author was supported by a Science Research Council Studentship during the course of this research.

‡ Department of Computer Science, University of Warwick, Coventry, Warwickshire CV4 7AL, England.

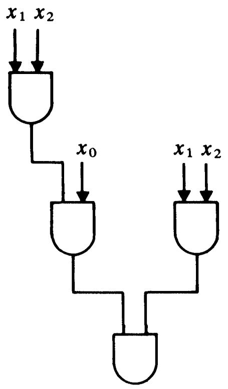


FIG. 1

resulting circuit computes f. Figure 1 shows a formula scheme which covers B_3 over basis B_2. This follows from the expansion

$$f(x_0, x_1, x_2) = (x_0 \wedge f_1(x_1, x_2)) \oplus f_0(x_1, x_2)$$

where $\oplus$ denotes sum modulo 2, $f_0(x_1, x_2) = f(0, x_1, x_2)$ and $f_1(x_1, x_2) = f(1, x_1, x_2) \oplus f(0, x_1, x_2)$. We have verified that this is the unique formula scheme (to within obvious symmetries) with fewer than five gates that covers B_3. Its depth of 3 is therefore optimal. A general lower bound on the depth of schemes can be proved by a simple counting argument.

THEOREM 1. *Any circuit scheme which covers B_n over any basis $b \subseteq B_2$ has depth at least $n-1$. Furthermore if $|b| \leqq 4$ or $|b| = 2$ the depth is at least n or $n+1$ respectively.*

Proof. A scheme of depth D has at most $2^D - 1$ gates, and so by varying the assignment to gates from b it can cover at set of at most $|b|^{2^D-1}$ different functions. Since $|B_n| = 2^{2^n}$ we have

$$|b|^{2^D-1} \geqq 2^{2^n}$$

which yields the stated bounds. □

In the next three sections we describe the main result of the paper, a scheme of depth $n+1$ to cover B_n over the basis B_2.

3. Outline of construction. Our starting point is a pair of familiar dual expansions for Boolean functions. Let $Y = \langle y_1, \cdots, y_k \rangle$ and $Z = \langle z_1, \cdots, z_m \rangle$ be sets of binary variables. Any function $f(Y, Z)$ in B_{k+m} may be expressed as a *disjunctive expansion about Z* by

$$f(Y, Z) = \bigvee_{C \in B^m} \delta_C(Z) \wedge f(Y, C)$$

where

$$\delta_C(Z) = \begin{cases} 1 & \text{if } Z = C, \\ 0 & \text{otherwise.} \end{cases}$$

The dual *conjunctive expansion about Z* is

$$f(Y, Z) = \bigwedge_{C} \bar{\delta}_C(Z) \vee f(Y, C)$$

where $\bar{\delta}_C$ is the complement (negation) of δ_C.

Each δ or $\bar{\delta}$ term requires a formula of depth only $\lceil \log_2 m \rceil$ and in each case the total depth used exceeds the maximum for δ_C, $\bar{\delta}_C$ and $f(Y, C)$ by $m+1$. The outer disjunctions or conjunctions over 2^m subformulae need depth m and one extra level is used for the single conjunction or disjunction used to attach the δ's or $\bar{\delta}$'s. It is the accumulation of these extra single levels in a recursive expansion about successive subsets of arguments which accounts for the $\log^* n$ term in Spira's bound. We plan to avoid these increments.

Consider one term $\delta_C(Z) \wedge f(Y, C)$ of the disjunctive expansion. We may ensure that $f(Y, C)$ is expressed as a conjunction of many small terms by using the conjunctive expansion for the next subset of arguments. Using the associativity of conjunction we might attempt to reassociate δ_C into $f(Y, C)$ but unfortunately the number of subterms of $f(Y, C)$ will be exactly a power of two. Our seemingly reckless solution is to discard one of these terms to make room for δ_C, and to be content with an "approximation" to the original function. To accomplish this ruse for each expansion we alternate disjunctive and conjunctive expansions about successive subsets of variables. The result of this first construction will be a formula of depth only n, but it will represent merely an approximation to the required function.

In the surprising finale we are able to show that the true function may be achieved by not-equivalencing ($\oplus$) this preliminary result with a second function which we can generate using the whole construction recursively in depth n also. The result is therefore of depth $n+1$.

In the details of the first construction, some attention must be paid to the sequence of cardinalities of the expansion subsets, and the way in which a term is omitted from the expansions is not quite straightforward. We shall describe our construction in terms of formulae rather than more abstractly as schemes. It will be clear throughout however that the formulae are uniform.

4. Initial construction. To define the subsets of arguments for the expansion, let $R_0, R_1, \cdots, R_p$ be a partition of X_n with $|R_i| = r_i$ for all i. We shall use the simple sequence $\langle r_0, \cdots, r_p \rangle$ defined by

$$r_0 = 2,$$

$$r_i = i+1 \qquad \text{for } 0 < i < p,$$

$$r_p = n - S_{p-1} \quad \text{where } S_m = \sum_{i=0}^{m} r_i$$

and where p is maximal such that

$$\frac{p(p+1)}{2} + 1 < n.$$

For example, if $n = 17$, we have $\langle 2, 2, 3, 4, 5, 1\rangle$. We can equally well use for our present purposes any sequence of positive integers satisfying

(i) $r_0 = r_1 = 2$,
(ii) $S_p = n$,
(iii) $r_m \leqq 2^{S_{m-2}}$ for $m \geqq 2$ and m even,
(iv) $r_m \leqq 2^{S_{m-2}} - 2^{S_{m-3}}$ for $m \geqq 3$ and m odd.

The fastest-growing sequence satisfying (i)–(iv) begins for large n with $\langle 2, 2, 4, 12, 256, 2^{20} - 256, 2^{276}, \cdots\rangle$.

The following definition allows us to describe the kind of function we can produce with the initial construction.

DEFINITION. Given $S = \{R_1, \cdots, R_k\}$ where $R_j \subseteq X_n$ for all $1 \leqq j \leqq k$, we define $g(X_n)$ to be *S-simple* if

$$g(X_n) = 0 \quad \text{whenever } R_j = \mathbf{0} \text{ for some } R_j \in S,$$
$$\text{where } \mathbf{0} = \langle 0, 0, \cdots, 0\rangle.$$

After the outline of § 3 we are prepared for the initial construction.

THEOREM 2. *For $n > 4$, every $\{R_1, \cdots, R_{p-1}\}$-simple function $g(X_n)$ has a formula of depth n.*

Proof. Since $n > 4$, then $p > 1$. We express g as an expansion about R_p which is disjunctive if p is odd and conjunctive if p is even. Each of the 2^{r_p} terms in this expansion is expressed in depth S_{p-1} by using the results and constructions of the following lemma. □

For an inductive proof we must incorporate a more detailed specification of the formulae at each stage.

LEMMA. *Let $R_0, R_1, \cdots, R_m$ $(m > 0)$ be disjoint sets of arguments with cardinalities $r_0, \cdots, r_m$ satisfying conditions* (i), (iii) *and* (iv) *above. Then for any function $g(R_0, \cdots, R_m)$, which is $\{R_1, \cdots, R_m\}$-simple, there is a formula for g consisting of*:

Case (a) *if m is odd, a disjunction of $2^{r_m} - 1$ subformulae each of depth S_{m-1}*;

Case (b) *if m is even, a conjunction of $2^{r_m} - 1$ subformulae each of depth S_{m-1} and another subformula of depth S_{m-2}.*

Proof. We proceed by induction on m using two alternative expansions about R_m. In Case (a),

$$g(R_0, \cdots, R_m) = \bigvee_{C \neq \mathbf{0}} (\delta_C(R_m) \wedge g_C(R_0, \cdots, R_{m-1}))$$

and in Case (b),

$$g(R_0, \cdots, R_m) = \bar{\delta}_{\mathbf{0}}(R_m) \wedge \bigwedge_{C \neq \mathbf{0}} (\bar{\delta}_C(R_m) \vee g_C(R_0, \cdots, R_{m-1}))$$

where in each case $g_C = g(R_0, \cdots, R_{m-1}, C)$.

The validity of these expansions is easily verified. If $m = 1$, then the first expansion is of the required form since both δ_C and g_C have depth 1 and so we have a disjunction of 3 formulae of depth 2. If $m > 1$ and m is odd then in the same expansion we may, by the inductive hypothesis, take g_C to be a conjunction of $2^{r_{m-1}} - 1$ subformulae of depth S_{m-2} and a smaller subformula of depth S_{m-3}.

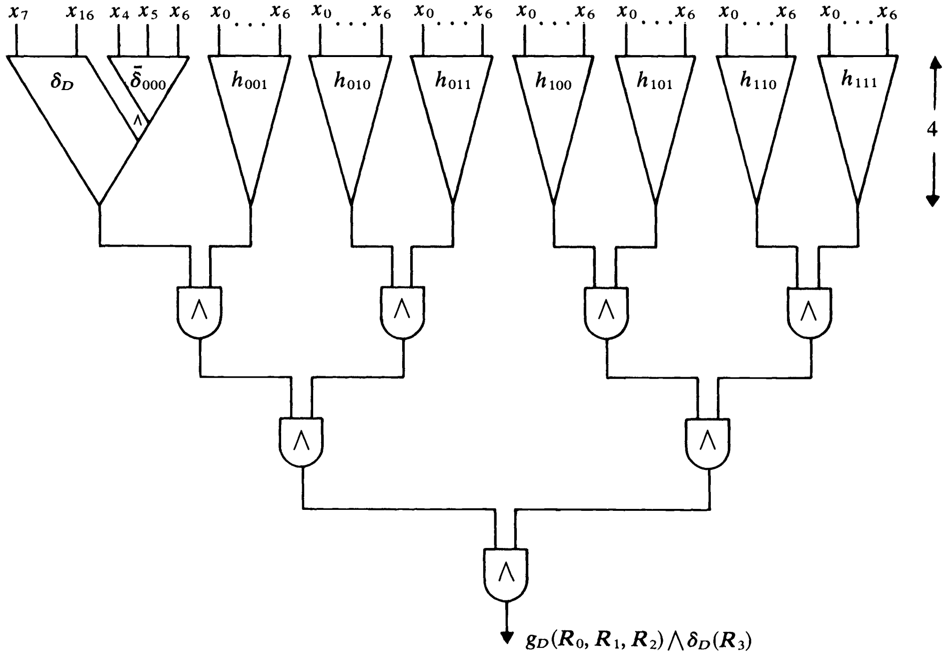


FIG. 2

Since δ_C is essentially a conjunction of r_m arguments and $r_m \leqq 2^{S_{m-2}} - 2^{S_{m-3}}$, it may be conjoined with the smaller subformula to produce a formula of depth S_{m-2}. The resulting conjunction of $2^{r_{m-1}}$ formulae of depth S_{m-2} can be written in depth $r_{m-1} + S_{m-2} = S_{m-1}$. The requirements of Case (a) are thereby met. If m is even then the second expansion is used, the $\bar{\delta}_C$ are themselves of depth S_{m-2} and Case (b) is easily satisfied. □

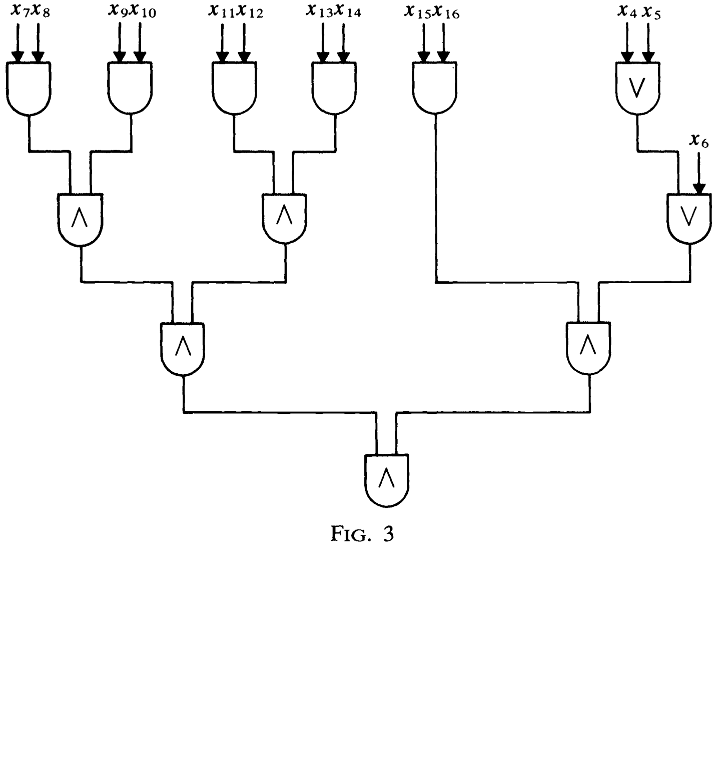


FIG. 3

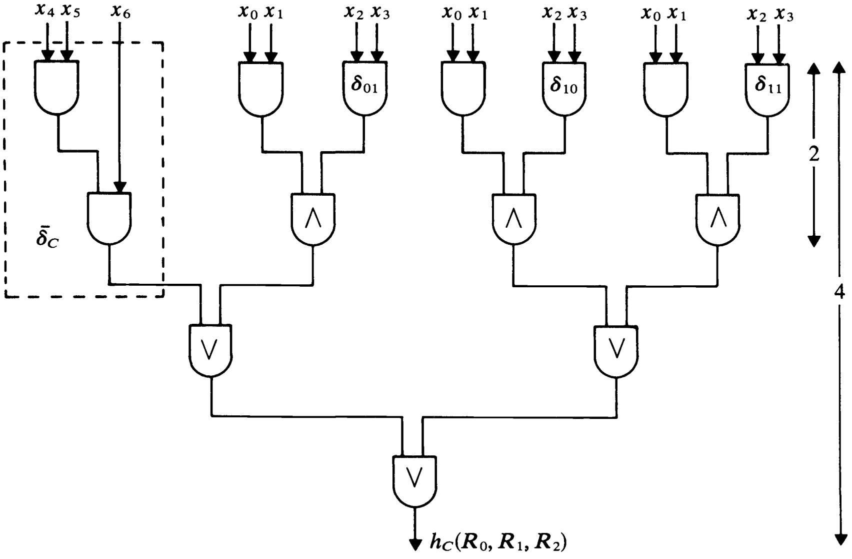


FIG. 4

The lemma may be illustrated with $n = 17$, $m = 3$ and the sequence $\langle 2, 2, 3, 10\rangle$. The resulting formula for $g(R_0, R_1, R_2, R_3)$ is a disjunction of 1023 formulae each of the form shown in Fig. 2. The leftmost subformula of Fig. 2 is given in more detail in Fig. 3, where the base functions associated with certain gates are not defined if they depend on D.

Each of the $h_C(R_0, R_1, R_2)$ subformulae from Fig. 2 are of the form illustrated in Fig. 4, where the base functions associated with unmarked gates depend on C.

5. Main result. It remains to be shown how formulae for arbitrary functions are to be derived from the construction just described for simple functions.

LEMMA. *Suppose $R_1, \cdots, R_k$ are disjoint subsets of X_n. For all $f(X_n)$, there exist $f_1(X_n - R_1), \cdots, f_k(X_n - R_k)$ such that $g_k(X_n) = f \oplus \bigoplus_{i=1}^k f_i$ is $\{R_1, \cdots, R_k\}$-simple.*

Proof. This is by induction on k. The lemma holds trivially for $k = 0$. Let $k > 0$, and suppose the result is true for $k - 1$. Then, there exists $f_1(X_n - R_1), \cdots, f_{k-1}(X_n - R_{k-1})$ such that for all j, $1 \leqq j < k$,

$$R_j = \mathbf{0} \Rightarrow g_{k-1}(X_n) = f \oplus \bigoplus_{i=1}^{k-1} f_i = 0.$$

We define

$$f_k(X_n - R_k) = \begin{cases} 0 & \text{if } \exists i,\ 1 \leqq i < k,\ R_i = \mathbf{0}, \\ g_{k-1} & \text{with arguments } R_k \text{ set to } \mathbf{0}, \text{ otherwise.} \end{cases}$$

It is evident that g_k has the required property. □

The principal result of the paper is now readily proved.

THEOREM 3. *For all n, $n \geqq 1$, there is a formula scheme with depth $n+1$ which covers B_n over B_2.*

Proof. Schemes for B_1, B_2 are obvious, while for B_3, B_4 expansions can be made about 1 and 2 arguments respectively to yield schemes of depth 3 and 4.

By the previous lemma and properties of $\oplus$, any function $f(X_n)$ may be expressed as $g(X_n) \oplus \bigoplus_{i=1}^{p-1} f_i(X_n - R_i)$ where $g(X_n)$ is $\{R_1, \cdots, R_{p-1}\}$-simple, and the R_i are defined at the beginning of § 4. For $n > 4$, Theorem 2 yields a formula of depth n for $g(X_n)$, to which we must "add" appropriate functions $f_1, \cdots, f_{p-1}$ where f_i has $n_i = n - i - 1$ arguments. Whenever $n_i \leqq 4$, a formula for f_i is constructed directly, otherwise the present construction is used recursively to yield a formula of depth $n_i + 1 = n - i$.

Thus f is expressible as

$$g(X_n) \oplus \bigoplus_{i=1}^{p-1} f_i(X_n - R_i).$$

or, after reassociation, as

$$g \oplus (f_1 \oplus (f_2 \oplus (\cdots \oplus f_{p-1})) \cdots).$$

Since f_i has depth $n - i$ for $i = 1, \cdots, p-1$, the latter represents a formula of depth $n+1$.

Again it is clear that the construction is uniform and thus yields a scheme. □

6. Restricted bases. The formulae considered so far have used all of B_2 as the basis. Provided that the basis b permits a scheme to cover B_2 and contains at least one function from each of the following three types:

$$\begin{array}{ll} \wedge\text{-type} & p^* \wedge q^*, \\ \vee\text{-type} & p^* \vee q^*, \\ \oplus\text{-type} & p^* \oplus q, \end{array}$$

where a starred variable represents either the variable or its complement, the construction can be followed more or less as before, complementing subformulae as necessary to achieve an upper bound of $n +$ (depth of a scheme to cover B_2).

An interesting basis is the set which excludes the two $\oplus$-type functions. In using this *unate* basis we may replace $\oplus$ by

$$p \oplus q = (p \wedge \bar{q}) \vee (\bar{p} \wedge q).$$

In order to fit in the correcting functions efficiently we choose a new sequence

$$\langle r_0, r_1, r_2, \cdots \rangle = \langle 2, 2, 4, 6, 8, 10, \cdots \rangle$$

so that each f_i contains $\underline{2}$ fewer arguments than the previous one. The result is a scheme of depth $n+3$.

CONJECTURE. *For any $b \subseteq B_2$, if there is a scheme over b which covers B_2 then there is a constant c such that for all n there is a scheme over b of depth $n+c$ which covers B_n.*

For $b = \{\text{NAND}, \pi_1\}$ (where $\text{NAND}(p, q) = \bar{p} \vee \bar{q}$ and $\pi_1(p, q) = p$) we have, at present, achieved no better than $n + O(\log^* n)$.

We must distinguish the notions of complete bases for formulae and for schemes. For example, $b = \{\text{NAND}\}$ is complete for formulae but obviously no singleton basis can be complete for schemes; hence the condition on b is necessary in the conjecture.

7. Conclusion. We have described a uniform scheme for expressing all n argument Boolean functions in depth $n+1$, and have matched this upper bound with a lower bound of $n-1$ under the restriction of uniformity. For a basis of unate functions only, our upper bound is $n+3$.

In our construction we used a sequence $\langle 2, 2, 3, 4, 5, \cdots \rangle$, but a sequence which grows much faster could be used instead. The effect of the choice of sequence on formula size has not been considered but easy counting arguments limit the possible size to within 2^{n-1} and 2^{n+1} for our method. Lupanov's construction [1] yields formulas of sizes about $2^n/\log_2 n$, though not of course using schemes. This raises the following question.

Open problem. Does a lower bound of $n - O(1)$ on depth hold for formulae as well as schemes?

REFERENCES

[1] O. B. Lupanov, *Complexity of formula realisation of functions of logical algebra*, Problemy Kibernet., 3 (1960), pp. 61–80; Problems of Cybernetics, 3 (1962), pp. 782–811.

[2] F. P. Preparata and D. E. Muller, *On the delay required to realise Boolean functions*, IEEE Trans. Computers, C-20 (1971), pp. 459–461.

[3] P. M. Spira, *On the time necessary to compute switching functions*, Ibid., C-20 (1971), pp. 104–105.

SIAM J. COMPUT.
Vol. 6, No. 2, June 1977

SOME PRESERVATION PROPERTIES OF NORMAL FORM GRAMMARS*

DAVID B. BENSON†

Abstract. The normal form grammars, such as those developed by Chomsky and Greibach, preserve certain properties of the original grammar. Ordinarily attention is only directed to weak equivalence, that is, that the original grammar and its normal form version both generate the same language. By paying greater attention to the functions carrying a grammar to its normal form, considerably stronger preservation properties can be proved. We demonstrate that several normal forms preserve ambiguities. More surprisingly, a variant of Chomsky's normal form, called canonical two form, forms an adjunction in connection with the original grammar. This fact shows that canonical two form preserves a large number of the structural properties of the original grammar. In particular, we show that the canonical two form is $LR(k)$ iff the original grammar is $LR(k)$, strengthening a result of Gray and Harrison. Preservation of structural properties such as ambiguity is important in semantic considerations, and the methods given for the determination of property preservation seem to be of general applicability.

Key words. normal form grammars, syntactic structures, ambiguity, syntax category, grammar functors, adjunction

1. Introduction.

The literature contains numerous instances of normal form grammars. In every case, a grammar G and its normal form G' are shown to be weakly equivalent, that is $L(G)=L(G')$. In almost every case, no more is shown. In fact, the normal form grammars preserve a large amount of the structure present in the original grammar. How much structure is preserved depends on the normal form. Intuitively, Chomsky's normal form preserves almost everything while Greibach's normal form, due to left recursion removal, preserves very little. This intuition suggests that there is a classification of normal forms by how much structure is preserved. Our intent in this paper is to give some precision to these preservation ideas.

The structures of primary interest are the syntactic structures assigned to strings by the grammar. If the grammar is context-free the syntactic structures are p-markers or derivation trees. For general phrase-structure grammars, the syntactic structures are similarity classes of inessentially distinct derivations (Hotz [18], Griffiths [16], Benson [6], [8]). Both context-free and type 0 grammars are considered in developing preservation properties of normal forms. For context-sensitive grammars there are at least three distinct notions of syntactic structure (Benson [6], Woods [31], Peters and Ritchie [24]). For this reason context-sensitive grammars are not considered in this note, although the preservation theory developed here should apply to context-sensitive grammars as well.

The interest in syntactic structures arises from syntax-directed translations and various notions of semantics for formal languages. In both cases the syntax, not the strings, are the entities of primary interest. In this situation one wishes to discover the extent to which normal form transformations preserve the syntax. The preservation theory given is basic in that we restrict attention to the

* Received by the editors January 3, 1975, and in revised form August 3, 1976.

† Computer Science Department, Washington State University, Pullman, Washington. Now at Department of Computer Science, University of Colorado, Boulder, Colorado 80302. This work was supported in part by the National Science Foundation under Grant GJ-43495.

"homomorphic" transformations. In the categorical framework used, these are the functors from one syntax category to another. There is ample evidence in the literature to show that functors are the basic homomorphisms of syntactic structures (Benson [8], Walter [30], Bertsch [9] and [10], Schnorr [26], Schnorr and Walter [27], Walter [29], Hotz and Claus [20]). Concepts such as coverings (Gray and Harrison [14], Aho and Ullman [2]) do not preserve all the structure available and are rather more difficult to work with.

These homomorphisms are exactly what are needed to transfer the semantics between grammars without going to the complexities involved in syntax-directed translations (Alagić [3], Benson [7]).

2. Categories and grammars. The basic notions of categories and functors are assumed known to the reader (MacLane [22], Herrlich and Strecker [17], Arbib and Manes [4]). However, we compose arrows (morphisms) in arrow-order. That means: if $f: \alpha \to \beta$, $g: \beta \to \gamma$ are arrows in a category then the composite arrow, "first f, then g", is denoted $f \circ g: \alpha \to \gamma$. The sign for composition, $\circ$, is frequently elided. In category $\mathbf{C}$ the set of all arrows from a to b is denoted $\mathbf{C}(a, b)$. A *strict monoidal category* (MacLane [22, p. 171]) is a triple $(\mathbf{C}, +, \lambda)$ where $\mathbf{C}$ is a category, $+: \mathbf{C} \times \mathbf{C} \to \mathbf{C}$ is a bifunctor and λ is a designated object of $\mathbf{C}$, all subject to the axioms:

(i) For all x, y, z, arrows of $\mathbf{C}$, $(x+y)+z = x+(y+z)$.

(ii) For all α, β, γ, objects of $\mathbf{C}$, $(\alpha+\beta)+\gamma = \alpha+(\beta+\gamma)$.

(iii) For all x, an arrow of $\mathbf{C}$, $x + 1_\lambda = x = 1_\lambda + x$, where 1_λ is the categorical identity at λ.

(iv) For all α, an object of $\mathbf{C}$, $\alpha + \lambda = \alpha = \lambda + \alpha$.

These axioms cause the objects of $\mathbf{C}$ to be a monoid under $+$ with unit element λ. Our interest lies in the case that this object-monoid is the free monoid over some alphabet V.

Let $(V^*, +, \lambda)$ be the free monoid over V where $+$ denotes string concatenation and λ the null string. The general notion of rewriting system used in this note is defined after we develop the notion of directed graph (diagram scheme, precategory, MacLane [22, pp. 48–49]) needed here. A *directed graph* (O, R, d, c) consists of a set of nodes O, a set of arcs R and two functions, $d, c: R \to O$. $d: R \to O$ assigns the domain (source) node to each arc, $c: R \to O$ assigns the codomain (target) node to each arc. Thus $r \in R$ is an arc from $d(r)$ to $c(r)$, $r: d(r) \to c(r)$. Note that several arcs may share the same source and target. An *indexed general rewriting system* $(V^*, R, d, c, +, \lambda)$ consists of a free monoid $(V^*, +, \lambda)$ and a directed graph (V^*, R, d, c). The entire note is stated in terms of indexed rewriting systems and grammars with indexed rules since the index proves to be technically useful. We write (V^*, R) for $(V^*, R, d, c, +, \lambda)$.

Consider the rewriting system $P = (V^*, R)$ underlying a grammar G. P may also be considered as a function, $P: R \to V^* \times V^*: r \mapsto (d(r), c(r))$. Rather than $P(r) = (\alpha, \beta)$ we follow the usual convention in grammar theory and write $P(r) = \alpha \to \beta$. In this view, the rewrite rules are indexed by R via the function $P: R \to V^* \times V^*$.

(V^*, R) generates a free strict monoidal category (Hotz [18]), which we call the *syntax category* for (V^*, R) and denote $\mathbf{S}(V^*, R)$ or just $\mathbf{S}$ when the generating indexed general rewriting system is clear. A full development of $\mathbf{S}(V^*, R)$ is given in Benson [8]. Here it suffices to note the following: An arrow in $\mathbf{S}$, say f: $\alpha \to \beta$, is a "derivation" from α to β. The arrows are isomorphic to derivation trees if the rewriting system is context-free. For general rewriting systems the arrows are similarity classes of inessentially distinct derivations. If f: $\alpha \to \beta$ and g: $\beta \to \gamma$ are derivations in $\mathbf{S}$, then the composite derivation fg: $\alpha \to \gamma$ is the derivation of first deriving by f and then deriving by g. For f: $\alpha \to \beta$ and g: $\gamma \to \delta$ in $\mathbf{S}$, the concatenate $f + g$: $\alpha + \gamma \to \beta + \delta$ is the derivation from the string $\alpha\gamma = \alpha + \gamma$ to the string $\beta\delta = \beta + \delta$ which consists of deriving by f on the left, and in parallel, by g on the right. If f: $\alpha \to \beta$ is a derivation the source, or domain, string is denoted by $d(f) = \alpha$. The target, or codomain, string is denoted by $c(f) = \beta$.

Using the notion of syntax category gives a convenient algebraic setting for explorations of normal form. Every derivation in $\mathbf{S}(V^*, R)$ has a length, the number of rewrite rule instances used in it, so every nonidentity derivation f may be written as the composite of *terms*, each term being of length one. That is,

$$f = \bigcirc_{i=1}^{n} f_i = f_1 \circ f_2 \circ \cdots \circ f_n,$$

where $f_i = 1_{\mu_i} + r_i + 1_{\nu_i}$ with $\mu_i, \nu_i \in V^*, r_i \in R, 1 \leqq i \leqq n$. If r_i: $\alpha \to \beta$ then the domain of f_i is $d(f_i) = \mu_i + d(r_i) + \nu_i = \mu_i \alpha \nu_i$ and the codomain of f_i is $c(f_i) = \mu_i + c(r_i) + \nu_i = \mu_i \beta \nu_i$. For this reason terms are denoted by $f_i = \mu_i + r_i + \nu_i$, letting the strings μ_i and ν_i denote the identities, derivations of length zero, 1_{μ_i} and 1_{ν_i}. For $f = \bigcirc_{i=1}^{n} f_i$, each f_i a term, the length of f is $l(f) = n$. The composition of terms $\bigcirc_{i=1}^{n} (\mu_i + r + \nu_i)$ is *canonical* if for all i, $1 \leqq i \leqq n$, $l(\mu_{i+1}) < l(\mu_i + c(r_i))$. The composition is *cocanonical* if for all i, $1 \leqq i < n$, $l(\nu_{i+1}) < l(c(r_i) + \nu_i)$. As Hotz [18] and others have shown, every derivation has a unique canonical representation and a unique cocanonical representation if the grammar is type 0.

The concatenate of derivations g_i may be written as

$$g = \sum_{i=1}^{n} g_i = g_1 + g_2 + \cdots + g_n.$$

With this orientation, our grammars are defined slightly differently than usual. Let $P = (V^*, R)$ be an indexed general rewriting system. Let Σ and N be sets such that $\Sigma \cup N = V, \Sigma \cap N = \varnothing$. Σ is the set of *external* or *terminal* symbols; N is the set of *internal* or *nonterminal* symbols. Call $\zeta \in N^*$ the *point* or *start string*. With this, $G = (N, \Sigma, P, \zeta)$ is a *grammar*. The syntax category of the underlying rewriting system P is denoted by $\mathbf{S}(G)$ or by $\mathbf{S}$ when the intended grammar is clear from context. The language generated by grammar $G = (N, \Sigma, P, \zeta)$ is $L(G) = \{\omega \in \Sigma^* | \mathbf{S}(\zeta, \omega) \neq \varnothing\}$. Otherwise stated, $\omega \in L(G)$ just in case there is a derivation from ζ to ω, f: $\zeta \to \omega$, and $\omega \in \Sigma^*$.

However, the primary interest in derivations is for semantic reasons. Several notions of semantics for formal languages and programming languages depend on the derivations to give semantic content to the sentences of the language (Benson [6], Goguen et al. [13]). Indeed the same dependence exists for the various versions of syntax-directed translations (Alagić [3]) and the relationships between syntax-directed translations and semantics (Benson [7], Buttelmann [11], Knuth [21]). An interpretation is a functor from the syntax category to some semantic category—the category of sets and functions in Benson [6]. Goguen et al. [13] generalize to other notions of the syntactic-semantic distinction, but always functorially.

On semantic grounds then, it is important to find the exact method of mapping between grammars to discover how the semantics or interpretation is to be preserved.

Let $\mathbf{S}$ and $\mathbf{S}'$ be syntax categories. A *syntax functor*, $F\colon \mathbf{S} \to \mathbf{S}'$ is a functor which preserves concatenation and the null string; $F(\alpha+\beta)=F(\alpha)+F(\beta)$, $F(f+g)=F(f)+F(g)$, and $F(\lambda)=\lambda$. In other words, a syntax functor is a strict monoidal functor between free strict monoidal categories. A *grammar functor*, $F\colon \mathbf{S}(G) \to \mathbf{S}(G')$, is a syntax functor carrying internal symbols to internal symbol strings, external symbols to external symbol strings, and the point of G to the point of G'. Symbolically, let $G=(N, \Sigma, P, \zeta)$ and $G'=(N', \Sigma', P', \zeta')$ be grammars. $F\colon \mathbf{S}(G) \to \mathbf{S}(G')$ is a grammar functor if F is a syntax functor and

(i) $F(A) \in (N')^*$ for all $A \in N$,

(ii) $F(a) \in (\Sigma')^*$ for all $a \in \Sigma$, and

(iii) $F(\zeta)=\zeta'$.

A grammar functor is *externally fixed* if $\Sigma=\Sigma'$ and $F(a)=a$ for all $a \in \Sigma$. This much apparatus gives the first useful result.

LEMMA 1. *Let* $G=(N, \Sigma, P, \zeta)$ *and* $G'=(N', \Sigma, P', \zeta')$ *be grammars with common external alphabet* Σ. *Let* $F\colon \mathbf{S}(G) \to \mathbf{S}(G')$ *be an externally fixed grammar functor. Then* $L(G) \subseteq L(G')$.

Proof. Let $\mathbf{S}=\mathbf{S}(G)$, $\mathbf{S}'=\mathbf{S}(G')$. Since F is a functor, if $\mathbf{S}(\alpha, \beta) \neq \varnothing$ then $\mathbf{S}'(F(\alpha), F(\beta)) \neq \varnothing$. Thus for all $\omega \in \Sigma^*$, $\mathbf{S}(\zeta, \omega) \neq \varnothing$ implies $\mathbf{S}'(\zeta, \omega) \neq \varnothing$. □

Many proofs of language containment in the literature are an application of this simple lemma, which is a variant of Hilfsatz 7 of Hotz and Claus [20, p. 73].

Appealing to the lemma is considerably more satisfactory than the usual trivial proof by induction given, or appealed to, in each separate case. As the proof of Lemma 1 shows, no induction argument is actually required.

Since $\mathbf{S}(G)$ is a freely generated strict monoidal category, to define a grammar functor F it suffices to specify F on the vocabulary $V=N \cup \Sigma$ and to specify F on the rewrite rules R. Free generation takes care of the rest of $\mathbf{S}(G)$. The only portion of this which requires care is to ensure that F is indeed a functor. That is, if $R\colon \alpha \to \beta$ is a rewrite rule then $F(r)$ must rewrite $F(\alpha)$ into $F(\beta)$. This principle may be stated precisely as follows.

Standard result. Let **RWS** be the category of all indexed general rewriting systems and homomorphisms between them. These homomorphisms preserve both the monoid structure of the nodes and the source, target structure of the arcs. Let **FSM-Cat** be the category of all strict monoidal categories whose object set is a free monoid under the bifunctor together with the strict monoidal functors between them. Let $\mathscr{F}\colon \textbf{FSM-Cat} \to \textbf{RWS}$ be the "forgetful" functor which forgets the composition and concatenation of arrows. Then there is a unique functor $\mathbf{S}\colon \textbf{RWS} \to \textbf{FSM-Cat}$ which assigns to each indexed general rewriting system (V^*, R) its syntax category $\mathbf{S}(V^*, R)$. There is an **RWS**-homomorphism $I\colon (V^*, R) \to \mathscr{F}(\mathbf{S}(V^*, R))$ which is the identity on each string and each rewrite rule. Now, let $\mathbf{C}$ be any syntax category and $f\colon (V^*, R) \to \mathscr{F}(\mathbf{C})$ a homomorphism. Then there is a *unique* syntax functor $F\colon \mathbf{S}(V^*, R) \to \mathbf{C}$ such that $f=I \circ \mathscr{F}(F)$ as in

$$\begin{tikzcd}
\mathbf{S}(V^*, R) \arrow[dd, dashed, "F"'] & (V^*, R) \arrow[rr, "I"] \arrow[ddrr, "f"'] & & \mathscr{F}(\mathbf{S}(V^*, R)) \arrow[dd, dashed, "\mathscr{F}(F)"] \\
& & & \\
\mathbf{C} & & & \mathscr{F}(\mathbf{C})
\end{tikzcd}$$

FIG. 1

the commutative diagram on the right of Fig. 1. Hence giving the functor data, f, on the rewriting system suffices to uniquely define the entire syntax functor, F. We have left out the details which may be completed in analogy to II.7 in MacLane [22]. □

The other portion of showing language equality via grammar functors requires a preliminary notion. If F: $\mathbf{C} \to \mathbf{D}$ is an arbitrary functor, F may be restricted to each of the sets $\mathbf{C}(a, b)$ where a and b are objects of $\mathbf{C}$. The definition is

$$F(a, b)\colon \mathbf{C}(a, b) \to \mathbf{D}(F(a), F(b))\colon x \mapsto F(x).$$

With this control over each hom-set, $\mathbf{C}(a, b)$, we have

LEMMA 2. *With the hypotheses of Lemma* 1, *if* $F(\zeta, \omega)$ *is a surjection for each* $\omega \in \Sigma^*$, *then* $L(G) = L(G')$.

Proof. If $F(\zeta, \omega)$ is a surjection, then $\mathbf{S}(\zeta, \omega) = \varnothing$ implies $\mathbf{S}'(\zeta', \omega) = \varnothing$. □

Lemmas 1 and 2 provide a fundamental method of demonstrating the weak equivalence of two grammars: set up a functor and show the surjective property. Surjectivity is a remarkably elusive property to characterize. Hotz and others have worked on this for several years. Hotz [19] provides a review of all but the latest work on grammar functor surjectivity. See Bertsch [9], [10] for the latest work.

Because of this difficulty it is usually easier, although not necessarily trivial, to prove the stronger property that the functor is full. A functor F: $\mathbf{C} \to \mathbf{D}$ is said to be *full* if for every pair of objects, (a, b), of $\mathbf{C}$, $F(a, b)$ is surjective (MacLane [22, p. 14]).

COROLLARY 3. *With the hypotheses of Lemma* 1, *if* F: $\mathbf{S}(G) \to \mathbf{S}(G')$ *is full then* $L(G) = L(G')$.

The existence of a full externally fixed grammar functor is considerably stronger than weak equivalence since it specifies that the derivations in $\mathbf{S}(G')$ are "homomorphic" images of those in $\mathbf{S}(G)$ and are no more numerous than those in $\mathbf{S}(G)$. As we shall illustrate in the examples, many standard normal form proofs of weak equivalence are in fact proofs of the existence of full externally fixed grammar functors.

3. Preserving ambiguities. In comparing a grammar G with its normal form version G', it is often of interest to know that G and G' have the same degree of ambiguity, that is, that for each $\omega \in \Sigma^*$ there are as many derivations from ζ to ω in $\mathbf{S}(G)$ as there are from ζ to ω in $\mathbf{S}(G')$. Again it is easiest to generalize: A functor F: $\mathbf{C} \to \mathbf{D}$ is *faithful* if for every pair of objects, (a, b), of $\mathbf{C}$, $F(a, b)$ is injective

(MacLane [22, p. 15]). If $F\colon \mathbf{C} \to \mathbf{D}$ is both full and faithful then $F(a, b)$ is a bijection from $\mathbf{C}(a, b)$ to $\mathbf{D}(F(a), F(b))$.

PROPOSITION 4. *Let $G = (N, \Sigma, P, \zeta)$ and $G' = (N', \Sigma, P', \zeta')$ be grammars with a common external alphabet Σ. Let $F\colon \mathbf{S}(G) \to \mathbf{S}(G')$ be an externally fixed grammar functor. If F is full and faithful then F preserves ambiguities. That is, for all $\omega \in \Sigma^*$, the cardinality of $\mathbf{S}(\zeta, \omega)$ is equal to the cardinality of $\mathbf{S}(\zeta', \omega)$.*

The existence of a full and faithful functor implies more than equal cardinality. The functor shows how derivations of the codomain syntax category are unique homomorphic images under the functor. The functorial homomorphism has implications for the study of semantics preservation.

Here is a useful sufficient condition for faithfulness in the current cases. First some notation: A grammar functor $F\colon \mathbf{S}_1 \to \mathbf{S}_2$ is *expansive* if $l(F(f)) \geqq l(f)$ for all derivations f. For each r in the set of rule indices R_1 generating $\mathbf{S}_1$ let

$$E(r) = \left\{ r'_i \in R_2 \middle| F(r) = \bigcirc_{i=1}^{n} (\mu_i + r'_i + \nu_i) \right\}.$$

That is, $E(r)$ is the set of all rule indices occurring in the image of r under $F\colon \mathbf{S}_1 \to \mathbf{S}_2$. For all $r \in R_1$, let

$$D(r) = E(r) - \bigcup_{\substack{s \in R_1 \\ s \neq r}} E(s).$$

The elements of $D(r)$ are said to be the *distinguishers* of r. If $\mu + r + \nu$ is a term, let $D(\mu + r + \nu) = D(r)$. x is *satiated* if for all $\rho, \sigma \in V^*$ and derivations y, $x = \rho + y + \sigma$ implies $\rho = \sigma = \lambda$. That is, x is satiated if every letter in its domain is rewritten by some rewrite rule in x.

For Proposition 5 and Lemma 6, let $F\colon \mathbf{S}_1 \to \mathbf{S}_2$ be a grammar functor between syntax categories generated by type 0 grammars $G_i = (N_i, \Sigma_i, P_i, \zeta_i)$, $i = 1, 2$.

PROPOSITION 5. *If F restricted to Σ_1^* is injective to Σ_2^*, F restricted to N_1 is injective to N_2 and for all $r \in P_1$, $d(r) \neq c(r)$, then F is expansive.*

LEMMA 6. *If*

(i) *$F\colon \Sigma_1 \to \Sigma_2$ is injective,*

(ii) *$F\colon N_1 \to N_2$ is injective,*

(iii) *F is expansive,*

(iv) *for all $r \in R_1$, $D(r) \neq \varnothing$, and*

(v) *for all $r \in R_1$, $F(r)$ is satiated,*

then F is faithful.

Proof. Each $F(\alpha, \beta)\colon \mathbf{S}_1(\alpha, \beta) \to \mathbf{S}_2(F(\alpha), F(\beta))$ is injective by induction on the length of derivations. If $l(f) = 0$ and $F(f) = F(f')$, then f is an identity and by (iii), f' is an identity. By (i) and (ii), $f = f'$. Assume for every pair of derivations f, f' in $\mathbf{S}_1$ where $l(f) = n$ that: $F(f) = F(f')$ implies $f = f'$. Consider g with $l(g) = n + 1$ and g' such that $F(g) = F(g')$. g has canonical representation $g = \bigcirc_{i=0}^{n} g_i$. Due to (iii) each $F(g_i)$ has length $\geqq 1$ so g' has canonical representation $g' = \bigcirc_{i=0}^{m} g'_i$. If $g_0 = g'_0$ then as left cancellation holds in syntax categories for type 0 grammars, (Hotz [18], Hotz and Claus [20]),

$$F\left(\bigcirc_{i=1}^{n} g_i\right) = F\left(\bigcirc_{i=1}^{m} g'_i\right)$$

and the result follows from the induction hypothesis and (iv). To this end, consider $D(g_0)\cap D(g'_0)$. If the intersection is not empty, $D(g_0)=D(g'_0)$. Therefore $g_0=\mu+r+\nu$ and $g'_0=\mu'+r+\nu'$ for some $r\in R_1$, $\mu, \nu, \mu', \nu'\in V^*$. If $\mu=\mu'$ then since $\mu+d(r)+\nu=\alpha=\mu'+d(r)+\nu'$, $\nu=\nu'$ and $g_0=g'_0$. To show that $\mu=\mu'$, consider satiated $\hat{g}, \hat{g}'$ such that $g=\psi+\hat{g}+\omega$ and $g'=\psi'+\hat{g}'+\omega'$ for $\psi, \psi', \omega, \omega'\in V^*$. $F(g)=F(\psi)+F(\hat{g})+F(\omega)=F(\psi')+F(\hat{g}')+F(\omega')=F(g')$ implies that $\psi=\psi'$ and $\omega=\omega'$. Write $\hat{g}$ and $\hat{g}'$ canonically as

$$\hat{g}=\bigcirc_{i=0}^{n}\hat{g}_i,$$

$$\hat{g}'=\bigcirc_{i=0}^{m}\hat{g}'_i$$

from which one immediately has $g_i=\psi+\hat{g}_i+\omega$, $g'_i=\psi+\hat{g}'_i+\omega$ for each i. Now $\hat{g}_0=\rho+r+\sigma$, $\hat{g}'_0=\rho'+r+\sigma'$ for $\rho, \rho', \sigma, \sigma'\in V^*$ such that $\mu=\psi+\rho$, $\mu'=\psi+\rho'$, $\nu=\sigma+\omega$, $\nu'=\sigma'+\omega$. If $\rho=\rho'$ we're done, so by symmetry assume $l(\rho)<l(\rho')$ and $\rho'=\rho+\tau$ for $\tau\in V^+$. Then

$$F(\hat{g}_0)=F(\rho)+h_1+F(\sigma),$$

$$F(\hat{g}'_0)=F(\rho)+F(\tau)+h_2+F(\sigma')$$

where the two instances of $F(r)$ have been relabeled h_1 and h_2 for clarity. As $F(\hat{g}_0)$ appears first in the composition

$$\bigcirc_{i=0}^{n}F(\hat{g}_i),$$

the rule applications in h_1 depend on the prior application of no other rules in $F(\hat{g})$. (See Benson [8] for definitions of rule application and dependency. The concepts are the obvious ones.) Similarly, h_2 depends on the prior application of no other rules in $F(g')$. Since $F(r)$ is satiated, h_1 and h_2 can be done in parallel in $F(\hat{g})=F(\hat{g}')$ and $l(F(\tau))\geqq l(d(F(r)))$. By (i) and (ii), $l(\tau)\geqq l(d(r))$. Since $F(r)$ occurs twice in $F(\hat{g})=F(\hat{g}')$, r must occur twice in both $\hat{g}$ and $\hat{g}'$, and both occurrences depend on the prior application of no other rules. Therefore, $\hat{g}'_0$ is not the canonically first term of $\hat{g}'$. This contradiction shows that $\tau=\lambda$ so that $g_0=g'_0$.

Now consider the case that $D(g_0)\cap D(g'_0)=\varnothing$. Since $F(g)=F(g')$ the rules done in the $F(g_0)$ subderivation are accomplished within $F(g'_i)$ subderivations for various $i>0$. Consider any distinguisher $s\in D(g_0)$ and the subderivation $F(g'_i)$ in which it occurs. Since s appears in $F(g_0)$, the application of s does not depend on the prior application of any rules in $F(g'_0)$. Therefore s is accomplished entirely to the right of $F(g'_0)$ and since s distinguishes g_0, all the rules in $F(g_0)$ are accomplished to the right of any rule in $F(g'_0)$. By symmetry, the same argument applies to distinguishers of $D(g'_0)$ to conclude that all the rules in $F(g'_0)$ are accomplished to the right of any rule in $F(g_0)$. This contradiction shows it is not possible to have $D(g_0)\cap D(g'_0)=\varnothing$. □

If one views "deriving" as the relation $\overset{*}{\Rightarrow}$ between strings, faithfulness reduces to a triviality and the correspondent to the above lemma is uninteresting as it would say nothing about ambiguity or semantics preservation.

Our first example is from Savitch [25]. He gives a conversion from type 0 grammars in "a standard alternative" form to type 0 grammars in *strong normal form*.

DEFINITION 1. A grammar $G=(N, \Sigma, P, S)$, $S \in N$ is in *standard alternative form* if every rewrite rule is in one of the following forms, where $A, B \in N$ and $\alpha \in V^* = (N \cup \Sigma)^*$:

(i) $BA \rightarrow \alpha A$,
(ii) $AB \rightarrow A\alpha$,
(iii) $B \rightarrow \alpha$.

DEFINITION 2. A grammar $G=(N, \Sigma, P, S)$, $S \in N$, is in *strong normal form* if there is a partition of N into N_{cf}, $\vec{N}$ and $\overleftarrow{N}$ such that every rewrite rule is in one of the forms,

(i) $A \rightarrow \alpha$, with $A \in N_{cf}$ and $\alpha \in V^*$,
(ii) $AB \rightarrow \lambda$, with $A \in \vec{N}$ and $B \in \overleftarrow{N}$, where λ is the null string.

Let $\mathscr{G}_{\text{sa}}$ be the class of grammars in "standard alternative" form and $\mathscr{G}_{\text{snf}}$ be the class of grammars in strong normal form. We give, by construction, a function SNF: $\mathscr{G}_{\text{sa}} \rightarrow \mathscr{G}_{\text{snf}}$.

Let $G \in \mathscr{G}_{\text{sa}}$, $G=(N, \Sigma, P, S)$. Define $G'=(N', \Sigma, P', S)$ as follows: $N' = N \cup \vec{N}_1 \cup \overleftarrow{N}_1 \cup \vec{N}_2 \cup \overleftarrow{N}_2$ where $\vec{N}_1$, $\overleftarrow{N}_1$, $\vec{N}_2$ and $\overleftarrow{N}_2$ are mutually disjoint sets of symbols not in N such that N' consists of five distinct copies of N. The subscript and over-arrow notation will indicate to which copy of N an internal symbol belongs, e.g., $\vec{A}_1 \in \vec{N}_1$. Let $R' = \vec{N}_1 \cup \overleftarrow{N}_1 \cup \vec{N}_2 \cup \overleftarrow{N}_2 \cup R$ where $P: R \rightarrow V^* \times V^*$. R' is the set of indices for the rewrite rules of G' and the new internal symbols are pressed into service as rule indices in this case. The indexed set of rewrite rules, $P': R' \rightarrow (V')^* \times (V')^*$, $V' = N' \cup \Sigma$, is given by:

(1) For $\vec{A}_1 \in \vec{N}_1$, $P'(\vec{A}_1) = A \rightarrow A\vec{A}_1$.
(2) For $\overleftarrow{A}_1 \in \overleftarrow{N}_1$, $P'(\overleftarrow{A}_1) = A \rightarrow \overleftarrow{A}_1 A$.
(3) For $\vec{A}_2 \in \vec{N}_2$, $P'(\vec{A}_2) = \vec{A}_2 \overleftarrow{A}_1 \rightarrow \lambda$.
(4) For $\overleftarrow{A}_2 \in \overleftarrow{N}_2$, $P'(\overleftarrow{A}_2) = \vec{A}_1 \overleftarrow{A}_2 \rightarrow \lambda$.
(5) For $r \in R$ with $P(r) = AB \rightarrow A\alpha$, $P'(r) = B \rightarrow \overleftarrow{A}_2 \alpha$.
(6) For $r \in R$ with $P(r) = BA \rightarrow \alpha A$, $P'(r) = B \rightarrow \alpha \vec{A}_2$.
(7) For $r \in R$ with $P(r) = B \rightarrow \alpha$, $P'(r) = B \rightarrow \alpha$.

As G' is in $\mathscr{G}_{\text{snf}}$, define SNF$(G) = G'$. Compare with Savitch [25].

For each $G \in \mathscr{G}_{\text{sa}}$ and $G' = \text{SNF}(G)$ there is an externally fixed grammar functor $F: \mathbf{S}(G) \rightarrow \mathbf{S}(G')$, completely determined by the following data.

(F1) For $A \in V$, $F(A) = A$.

(F2) For $r \in R$ with $P(r) = AB \rightarrow A\alpha$,

$$F(r) = ((A \xrightarrow{\vec{A}_1} A\vec{A}_1) + B) \circ (A\vec{A}_1 + (B \xrightarrow{r} \overleftarrow{A}_2 \alpha)) \circ (A + (\vec{A}_1 \overleftarrow{A}_2 \xrightarrow{\overleftarrow{A}_2} \lambda) + \alpha).$$

(F3) For $r \in R$ with $P(r) = BA \rightarrow \alpha A$,

$$F(r) = (B + (A \xrightarrow{\overleftarrow{A}_1} \overleftarrow{A}_1 A)) \circ ((B \xrightarrow{r} \alpha \vec{A}_2) + \overleftarrow{A}_1 A) \circ (\alpha + (\vec{A}_2 \overleftarrow{A}_1 \xrightarrow{\vec{A}_2} \lambda) + A).$$

(F4) For $r \in R$ with $P(r) = B \rightarrow \alpha$, $F(r) = B \xrightarrow{r} \alpha$.

The only complexity is introduced by (F2) and (F3). Consider (F2). The derivation of length one, $r\colon AB \to A\alpha$ is carried by F into a derivation of length three such that $F(r)$ rewrites AB to $A\alpha$ by first applying $\vec{A}_1\colon A \to A\vec{A}_1$ leaving B fixed, then applying $r\colon B \to \overleftarrow{A}_2\alpha$ leaving $A\vec{A}_1$ fixed, and finally applying $\overleftarrow{A}_2\colon \vec{A}_1\overleftarrow{A}_2 \to \lambda$ leaving A and α fixed.

By Lemma 1, $L(G) \subseteq L(\mathrm{SNF}(G))$. That $F\colon \mathbf{S}(G) \to \mathbf{S}(G')$ is full follows by Savitch's Lemma 2, requiring only the addition of indexing which does not change the proof in any material way. Hence the languages are equal. F is faithful by the immediate application of Lemma 6.

Not only does this result show that Savitch's strong normal form construction is ambiguity preserving, it gives an embedding of $\mathbf{S}(G)$ in $\mathbf{S}(G')$. Stated differently, a full and faithful functor means that the original $\mathbf{S}(G)$ derivations are recoverable from their images in $\mathbf{S}(G')$.

Stanat [28] gives a different normal form for type 0 grammars, called *standard form.* His Theorem 6.1, if translated into the categorical terminology, shows that there exists a full and faithful externally fixed grammar functor into standard form grammars.

4. Functors from the normal form. Functors must map every derivation of every length without a tree automaton's ability to scan other portions of a derivation before deciding how to map. Therefore some normal forms do not admit a functor between an original grammar and the normal form grammar. Greibach's prefix normal form [15] is the best example. The primary difficulty is the removal of left recursion. This does sufficient violence to the syntactic structure that there is no hope of a functor in either direction. However, the remainder of the Greibach transformation, lifting external symbols to the prefix position, does have functorial properties. Specifically, if G is a context free grammar (cfg) free of left recursion and G' is the prefix normal form of it, there is a faithful externally fixed grammar functor from $\mathbf{S}(G')$ to $\mathbf{S}(G)$. This functor fails to be full, although it possesses a weaker property (externally full) which suffices to guarantee equality of languages.

We follow Aho and Ullman's Algorithm 2.14 [1, p. 158] in producing the details and follow their notation to the extent possible. Specifically, since cfg $G = (N, \Sigma, P, S)$ is free of left recursion there exists a total ordering $A_1, A_2, \cdots, A_n$ of the internal symbols N such that for all $r \in R$, $P(r) = A_i \to A_j\alpha$ implies $i < j$. The algorithm then iterates from n down to 1 in the outer loop to produce the prefix form grammar G'. At each step i of the algorithm there is an inner loop from n to $i+1$. After each step j of the inner loop there is implicitly a grammar G_{ij}. At the end of the inner loop at step $(i, i+1)$, one uses $G_{i,i+1}$ as input to the step $(i-1, n)$. The algorithm begins with $G = G_{nn}$ and terminates with $G' = G_{12}$. We give the details for the functors $\mathbf{S}(G_{ij}) \to \mathbf{S}(G_{i,j+1})$. Since faithfulness and external fullness are preserved under functor composition, to show that each functor $\mathbf{S}(G_{ij}) \to \mathbf{S}(G_{i,j+1})$ is faithful and externally full suffices to obtain the result that the composite functor $\mathbf{S}(G') \to \mathbf{S}(G)$ is faithful and externally full.

Specifically, at step (i, j) of the algorithm all the rules of the form $A_i \to A_j\alpha$ in $G_{i,j+1}$ are eliminated and replaced by the derived rules $A_i \to \beta_k\alpha$ where $A_j \to \beta_k$ is the kth rule rewriting A_j. Let $G_{i,j+1} = (N, \Sigma, P_{i,j+1}, S)$ where $P_{i,j+1}\colon R_{i,j+1} \to$

$N\times V^*$. Let

$$R'_{i,j+1}=\{r|P_{i,j+1}(r)=A_i\to A_j\alpha,\ \alpha\in V^*\},$$

and

$$R^{(j)}_{i,j+1}=\{r|P_{i,j+1}(r)=A_j\to\beta,\ \beta\in V^*\},$$

$$R''_{i,j+1}=\{(r,r')|r\in R'_{i,j+1}\text{ and }r'\in R^{(j)}_{i,j+1}\}.$$

The rule indices of $G_{i,j}$ are $R_{i,j}=(R_{i,j+1}-R'_{i,j+1})\cup R''_{i,j+1}$. The rule function for $G_{i,j}$ is $P_{i,j}\colon R_{i,j}\to N\times V^*$ such that $P_{i,j}$ restricted to the indices common to both grammars is equal to $P_{i,j+1}$, i.e., $P_{i,j}|R_{i,j+1}\cap R_{i,j}=P_{i,j+1}|R_{i,j+1}\cap R_{i,j}$. On the new indices in $R''_{i,j+1}$, $P_{i,j}(r,r')=A_1\to\beta\alpha$ iff $P_{i,j+1}(r)=A_i\to A_j\alpha$ and $P_{i,j+1}(r')=A_j\to\beta$. This completely determines $G_{i,j}=(N,\Sigma,P_{i,j},S)$.

The functor $H\colon \mathbf{S}(G_{i,j})\to\mathbf{S}(G_{i,j+1})$ is given by

(H1) for all $X\in V$, $H(X)=X$,

(H2) for all $r\in R_{i,j}\cap R_{i,j+1}$, $H(r)=r$,

(H3) for all $(r,r')\in R''_{i,j+1}$,

$$H(r,r')=(A_i\xrightarrow{r}A_j\alpha)\circ((A_j\xrightarrow{r'}\beta)+\alpha).$$

$H\colon \mathbf{S}(G_{i,j})\to\mathbf{S}(G_{i,j+1})$ is faithful by an application of Lemma 6.

Let $G=(N,\Sigma,P,\zeta)$, $G'=(N',\Sigma,P,\zeta')$ be grammars, $\mathbf{S}'=\mathbf{S}(G')$, $\mathbf{S}=\mathbf{S}(G)$. An externally fixed grammar functor, $H\colon\mathbf{S}'\to\mathbf{S}$ is *externally full* if for all $\alpha\in V^*$ and all $\omega\in\Sigma^*$, $H(\alpha,\omega)\colon\mathbf{S}(\alpha,\omega)\to\mathbf{S}(H(\alpha),\omega)$ is surjective.

PROPOSITION 7. *$H\colon\mathbf{S}(G_{i,j})\to\mathbf{S}(G_{i,j+1})$ is externally full.*

Proof. Let $\mathbf{S}'=\mathbf{S}(G_{i,j})$, $\mathbf{S}=\mathbf{S}(G_{i,j+1})$. The proof is by strong induction on the length of derivations in $\mathbf{S}$. The base is trivial since for $\omega\in\Sigma^*$ the only derivation is the identity. Let $g\colon\alpha\to\omega$ be a derivation of length $n+1$ in $\mathbf{S}$ with $\omega\in\Sigma^*$, and let $g=\bigcirc_{i=1}^{n+1}g_i$ be the canonical representation of g. Consider $g_1=\mu+r+\nu$, where μ, $\nu\in V^*$ and $r\in R_{i,j+1}$. If $r\notin R'_{i,j+1}$, then g_1 is a derivation in $\mathbf{S}'$ such that $H(g_1)=g_1$. By induction then, there is a derivation $f\colon\alpha\to\omega$ in $\mathbf{S}'$ such that $H(f)=g$. If $r\in R'_{i,j+1}$ then $P_{i,j+1}(r)=A_i\to A_j\gamma$ for some $\gamma\in V^*$. By the fact that $c(g)=\omega\in\Sigma^*$ and the canonicity of $\bigcirc_{i=1}^{n+1}g_i$, $g_2=\mu+r'+\gamma\nu$ for some $r'\in R^{(j)}_{i,j+1}$. The derivation $f_1=\mu+(r,r')+\nu$ in $\mathbf{S}'$ maps into $g_1\circ g_2$ as $H(f_1)=H(\mu)+H(r,r')+H(\nu)=\mu+H(r,r')+\nu=g_1\circ g_2$ by (H3). Induction again establishes the result. □

Composing the functors $H\colon\mathbf{S}(G_{i,j})\to\mathbf{S}(G_{i,j+1})$ gives the preservation properties of the prefix normal form.

COROLLARY 8. *Let G be a* cfg *free of left recursion and let G' be the corresponding prefix normal form grammar. There is a faithful and externally full, externally fixed grammar functor $H\colon\mathbf{S}(G')\to\mathbf{S}(G)$. In particular $L(G')=L(G)$ and prefix normal form preserves ambiguities.*

The existence of a functor from the normal form to the original grammar makes it particularly easy to carry the semantics of the original language to the normal form. Suppose $T\colon\mathbf{S}(G)\to\mathbf{A}$ is an interpretation of G. The corresponding interpretation of G' is given by composition with H:

$$HT\colon\mathbf{S}(G')\to\mathbf{A}=\mathbf{S}(G')\xrightarrow{H}\mathbf{S}(G)\xrightarrow{T}\mathbf{A}.$$

Let $G^*_{i,j}=(N,\Sigma,P^*,S)$ be a cfg with nonindexed productions $P^*=\{A\to\beta|\exists r\in R_{i,j}$ with $d(r)=A$, $c(r)=\beta\}$. There is always a functor $S\colon\mathbf{S}(G_{i,j})\to$

$\mathbf{S}(G^*_{i,j})$ which is the identity on objects and which maps each rule r into the unique corresponding rule in P^*. While $S: R_{i,j} \to P^*$ is surjective, it is not in general injective, since there may be distinct (r, r') and (s, s') in $R''_{i,j+1} \subseteq R_{i,j}$ for which

$$P_{i,j}(r, r') = A_i \to \beta\alpha \quad \text{and} \quad P_{i,j}(s, s') = A_i \to \beta'\alpha'$$

while $\beta\alpha = \beta'\alpha'$. Hence in the diagram

$$\begin{tikzcd}
& \mathbf{S}(G_{i,j}) \arrow[dl, "S"'] \arrow[dr, "H"] & \\
\mathbf{S}(G^*_{i,j}) \arrow[rr, dashed] & & \mathbf{S}(G_{i,j+1})
\end{tikzcd}$$

there is no canonical choice of functor $\dashrightarrow$ since S is not faithful. Without indexing then, lifting external symbols to the prefix position does not in general preserve ambiguities. The above analysis, mentioned by a referee, shows that indexing is important in semantically motivated studies of syntactic relationships. There may very well be two rules $P(r) = A \to \alpha$, $P(s) = A \to \alpha$ with different interpretations $T(r) \neq T(s)$. Ambiguity preservation guarantees that the syntaxes remain free to accept arbitrary interpretations mediated by the connecting functor $H: \mathbf{S}(G') \to \mathbf{S}(G)$.

5. Adjoint situations. Let $\mathbf{C}$, $\mathbf{D}$ be categories and $F: \mathbf{C} \to \mathbf{D}$, $G: \mathbf{C} \to \mathbf{D}$ be functors. A *natural transformation* from F to G, $\eta: F \dot{\to} G$, is a family of arrows in $\mathbf{D}$, indexed by the objects of $\mathbf{C}$, such that for each object a of $\mathbf{C}$, $\eta(a)$ is an arrow from $F(a)$ to $G(a)$, $\eta(a): F(a) \to G(a)$. These arrows are subject to the axiom: for all arrows $f: a \to b$ in $\mathbf{C}$,

$$\begin{tikzcd}
F(a) \arrow[r, "\eta(a)"] \arrow[d, "F(f)"'] & G(a) \arrow[d, "G(f)"] \\
F(b) \arrow[r, "\eta(b)"'] & G(b)
\end{tikzcd}$$

commutes, that is $\eta(a) \circ G(f) = F(f) \circ \eta(b)$.

For each category $\mathbf{C}$, the identity functor on $\mathbf{C}$ is denoted $I_{\mathbf{C}}$.

An *adjunction* from $\mathbf{C}$ to $\mathbf{D}$ consists of two functors and two natural transformations, $(H, F; \eta, \varepsilon): \mathbf{C} \rightharpoonup \mathbf{D}$, where $H: \mathbf{C} \to \mathbf{D}$, $F: \mathbf{D} \to \mathbf{C}$ are functors and $\eta: I_{\mathbf{C}} \dot{\to} HF$ and $\varepsilon: FH \dot{\to} I_{\mathbf{D}}$ are natural transformations. The data renders commutative the *triangular identity* diagrams in Fig. 2.

In this, $HF: \mathbf{C} \to \mathbf{C}$ and $FH: \mathbf{D} \to \mathbf{D}$ are the composite functors, where HF means "first H, then F" and FH means "first F, then H.". ηH, $H\varepsilon$, $F\eta$ and εF are all natural transformations. ηH means use η to find an arrow and then apply H to that arrow. Similar meanings are attached to the other three natural transformations. In $(H, F; \eta, \varepsilon)$ H is called the left adjoint and F is called the right adjoint. MacLane [22] contains a detailed exposition of adjoints.

$$\begin{tikzcd}
H \arrow[r, "\eta H"] \arrow[dr, equal] & HFH \arrow[d, "H\varepsilon"] & FHF \arrow[d, "\varepsilon F"'] & F \arrow[l, "F\eta"'] \arrow[dl, equal] \\
& H & F &
\end{tikzcd}$$

FIG. 2

If $FH = I_{\mathbf{D}}$ then H is said to be right-inverse to F. If $(H, F; \eta, \varepsilon)$ is an adjunction and H is right-inverse to F, then H is called the left-adjoint-right-inverse to F. In this case ε is the identity natural transformation and the adjunction is written $(H, F; \eta, =)$. The existence of an adjunction $(H, F; \eta, =)$: $\mathbf{C} \rightharpoonup \mathbf{D}$ implies a great deal about the structural similarity of $\mathbf{C}$ and $\mathbf{D}$.

A cfg $G = (N, \Sigma, P, S)$ is in *canonical two form* if for $P\colon R \to N \times V^*$, $P(R) \subseteq N \times (\{\lambda\} \cup V \cup N^2)$. Canonical two form is a slight extension of Chomsky's normal form. If G is an arbitrary cfg and G' is its canonical two form equivalent, there is an adjunction $(H, F; \eta, =)$: $\mathbf{S}(G') \rightharpoonup \mathbf{S}(G)$. While intuition says that canonical two form preserves the structure of G, the existence of an adjunction is surprising. We proceed to the details, with notation following that of Gray and Harrison [14].

Let $\mathcal{G}_2$ be the class of cfg and let $\mathcal{C}_2 \subset \mathcal{G}_2$ be the subclass of grammars in canonical two form. Let $C_2\colon \mathcal{G}_2 \to \mathcal{C}_2$ be the function such that for each $G = (N, \Sigma, P, S) \in \mathcal{G}_2$ results in $C_2(G) = (N', \Sigma, P', [S])$ where

$$N' = N_1 \cup N_2,$$

$$N_1 = \{[A] \mid A \in V\}, \text{ a new copy of } V,$$

$$N_2 = \{[\beta] \mid \exists r \ni P(r) = A \to \alpha\beta,\ \alpha \in V^*,\ \beta \in V^2V^*\},$$

$$P'\colon R' \to N' \times (V')^*,$$

in which $R' = R \cup R_1 \cup R_2$, R is the set of indices of rules in G, $R_1 = \{\overline{[\beta]} \mid [\beta] \in N_2\}$ is a new copy of N_2 and $R_2 = \{\overline{[\alpha]} \mid a \in \Sigma\}$ is a new copy of Σ. P' is specified by:

1. For $r \in R$,

$$P'(r) = \begin{cases} [A] \to \lambda & \text{if } P(r) = A \to \lambda, \\ [A] \to [B] & \text{if } P(r) = A \to B,\ A \in N,\ B \in V, \\ [A] \to [B][\beta] & \text{if } P(r) = A \to B\beta,\ A \in N,\ B \in V,\ \beta \in V^+. \end{cases}$$

2. For $\overline{[A\beta]} \in R_1$, $P'(\overline{[A\beta]}) = [A\beta] \to [A][\beta]$.
3. For $\overline{[a]} \in R_2$, $P'(\overline{[a]}) = [a] \to a$.

$C_2\colon \mathcal{G}_2 \to \mathcal{C}_2$ is the *canonical two forming* function.

For each $G \in \mathcal{G}_2$ there are a pair of externally fixed grammar functors, $F\colon \mathbf{S}(G) \to \mathbf{S}(C_2(G))$ and $H\colon \mathbf{S}(C_2(G)) \to \mathbf{S}(G)$. We proceed to define them.

(H1) For $a \in \Sigma$, $H(a) = a$.

(H2) For $[\beta] \in N'$, $H([\beta]) = \beta$.

(H3) For $r \in R$, $H(r) = r$.

(H4) For $\overline{[\beta]} \in R_1 \cup R_2$, $H(\overline{[\beta]}) = 1_\beta$, the identity derivation at the string $\beta \in V^*$.

The specifications (H1) and (H2) generate H on all the objects $(V')^*$. Part (H3) in combination with the data 2 in the specification of the canonical two forming function determine the way that H "removes brackets" from rewrite rules indexed by R. Part (H4) trivializes the "bracket adding" rules in R_1 and the "externalizing" rules in R_2. Since $\mathbf{S}(C_2(G))$ is a free sm-category and H is a sm-functor, H is completely determined by (H1)–(H4). H may be called the *bracket removal functor.*

To specify the *bracket adding functor* F: $\mathbf{S}(G) \to \mathbf{S}(C_2(G))$, the following notation is convenient. For $A \in N$, $e[A] = 1_{[A]}$, the identity derivation at $[A]$. For $a \in \Sigma$, $e[a] = (\overline{[a]}: [a] \to a)$, the externalizing rewrite rule. This defines e for every $[A] \in N_1$.

(F1) For $a \in \Sigma$, $F(a) = a$.

(F2) For $A \in N$, $F(A) = [A]$.

(F3) For $r \in R$ such that $P(r) = A \to \lambda$, $F(r) = r$.

(F4) For $r \in R$ such that $P(r) = A \to B$, $B \in V$, $F(r) = r \circ e[B]$.

(F5) For $r \in R$ such that $P(r) = A \to BC$, $B, C \in V$, $F(r) = r \circ (e[B] + e[C])$.

(F6) For $r \in R$ such that $P(r) = A \to A_1A_2 \cdots, A_n$, $n > 2$, $A_i \in V$,

$$F(r) = r \circ \bigcirc_{i=1}^{n-2} ([A_1] \cdots [A_i] + \overline{[A_{i+1} \cdots A_n]}) \circ \sum_{i=1}^{n} e[A_i].$$

Some explanation is in order. In (F4), if $P(r) = A \to a$ with $a \in \Sigma$, then $F(r) = [A] \xrightarrow{r} [a] \xrightarrow{\overline{[a]}} a$. In [F6], $P'(r) = [A] \to [A_1][A_2 \cdots A_n]$. The bracketed symbol $[A_2 \cdots A_n]$ must be rewritten to the fully bracketed string $[A_2] \cdots [A_n]$. This can only be accomplished via the rewriting rules $[A_{i+1} \cdots A_n]$: $[A_{i+1} \cdots A_n] \to [A_{i+1}][A_{i+2} \cdots A_n]$ with $n-2$ steps since the last step is $[A_{n-1}A_n]$: $[A_{n-1}A_n] \to [A_{n-1}][A_n]$. The resulting string is in N_1^*. The symbols of the form $[a]$, for $a \in \Sigma$, must be externalized and the parallel rewriting $\sum_{i=1}^n e[A_i]$ accomplishes this.

LEMMA 9. *FH*: $\mathbf{S}(G) \to \mathbf{S}(G)$ *is the identity functor on* $\mathbf{S}(G)$.

Proof. Let g be any derivation in $\mathbf{S}(G)$. Let $g = \bigcirc_{i=1}^m g_i$ be any decomposition of g into terms $g_i = \mu_i + r_i + \nu_i$. Since FH is a functor, $FH(g) = \bigcirc_{i=1}^m FH(g_i)$. As F fully brackets strings and H removes them, $FH(\mu_i) = \mu_i$ and $FH(\nu_i) = \nu_i$ for $1 \leqq i \leqq m$. The rewrite rules r_i must be treated by the cases corresponding to (F3) through (F6).

(i) If $P(r) = A \to \lambda$ then $FH(r) = r$.

(ii) If $P(r) = A \to B$ then $F(r) = r \circ e[B]$. $H(r \circ e[B]) = H(r) \circ H(e[B]) = r \circ 1_B = r$.

(iii) If $P(r) = A \to BC$ then $F(r) = r \circ (e[B] + e[C])$. $H(r \circ (e[B] + e[C])) = H(r) \circ (H(e[B]) + H(e[C])) = r \circ (1_B + 1_C) = r \circ 1_{BC} = r$.

(iv) If $P(r) = A \to A_1 \cdots A_n$, $F(r) = r \circ \bigcirc_{i=1}^{n-2} ([A_1] \cdots [A_i] + [A_{i+1} \cdots A_n]) \circ \sum_{i=1}^n e[A_i]$. Now $H(r) = r$, $H([A_1] \cdots [A_i] + [A_{i+1} \cdots A_n]) = A_1 \cdots A_i + 1_{A_{i+1} \cdots A_n} = 1_{A_1 \cdots A_n}$ for $1 \leqq i \leqq n-2$, and $H(\sum_{i=1}^n e[A_i]) = 1_{A_1 \cdots A_n}$. Composing the above derivations, $H(F(r)) = r$. □

For each symbol $X \in V'$, define a derivation $\eta(X)$ in $\mathbf{S}(C_2(G))$ as follows:

(η1) For $a \in \Sigma$, $\eta(a) = \underline{1_a.}$

(η2) For $a \in \Sigma$, $\eta[a] = \overline{[a]}$: $[a] \to a$.

(η3) For $A \in N$, $\eta[A] = 1_{[A]}$.

(η4) For $[A_1 \cdots A_n] \in N_2$,

$$\eta[A_1 \cdots A_n] = \bigcirc_{i=1}^{n-1} ([A_1] \cdots [A_{i-1}] + \overline{[A_i \cdots A_n]}) \circ \sum_{i=1}^{n} e[A_i].$$

Note that N_2 and R_2 are defined so that $\eta[A_1 \cdots A_n]$ does always exist. This is Gray and Harrison's construction, necessary and just sufficient to define C_2: $\mathcal{G}_2 \to \mathcal{C}_2$, so that nothing has been artificially introduced to construct η. Further note that (η2) and (η3) define $\eta(X)$, for $X \in N_1$, as $\eta(X) = e(X)$. On strings in $(V')^*$, η is defined by the usual monoid homomorphism recursion equations:

(η5) $\eta(\lambda) = 1_\lambda$,

(η6) $\eta(\alpha X) = \eta(\alpha) + \eta(X)$, $\alpha \in (V')^*$, $X \in V'$.

LEMMA 10. *Let* $\mathbf{S}' = \mathbf{S}(C_2(G))$. η *as defined in* (η1)–(η6) *is a natural transformation* η: $I_{\mathbf{S}'} \dot{\to} HF$.

Proof. We must show that for all derivations f: $\gamma \to \delta$ in $\mathbf{S}'$ that $f \circ \eta(\delta) = \eta(\gamma) \circ fHF$, where fHF means $F(H(f))$. Since $\mathbf{S}'$ is freely generated, it suffices to show that

$$\begin{tikzcd}
\alpha \arrow[r, "\eta(\alpha)"] \arrow[d, "r"'] & \alpha HF \arrow[d, "rHF"] \\
\beta \arrow[r, "\eta(\beta)"'] & \beta HF
\end{tikzcd}$$

commutes for all rules $r \in R'$. Since $R' = R \cup R_1 \cup R_2$ there are three cases.

Case 1. $r \in R$. There are four subcases corresponding to (F3) through (F6).

(i) $P'(r) = [A] \to \lambda$. $rHF = r$, so (η3) and (η5) give the commuting diagram.

(ii) $P'(r) = [A] \to [B]$, $B \in V$. $rHF = r \circ e[B]$ so $1_{[A]} \circ r \circ e[B] = r \circ e[B]$.

(iii) $P'(r) = [A] \to [B][C]$. $rHF = r \circ (e[B] + e[C])$ and again $1_{[A]} \circ r \circ (e[B] + e[C]) = r \circ (e[B] + e[C])$.

(iv) $P'(r) = [A] \to [A_1][A_2 \cdots A_n]$, $n > 2$.

$$rHF = r \circ \bigcirc_{i=1}^{n-2} ([A_1] \cdots [A_i] + \overline{[A_{i+1} \cdots A_n]}) \circ \sum_{i=1}^{n} e[A_i].$$

Now

$$\begin{aligned}
\eta([A_1][A_2 \cdots A_n]) &= \eta[A_1] + \eta[A_2 \cdots A_n] \\
&= e[A_1] + \bigcirc_{i=1}^{n-2} ([A_2] \cdots [A_i] + \overline{[A_{i+1} \cdots A_n]}) \circ \sum_{i=2}^{n} e[A_i] \\
&= \bigcirc_{i=1}^{n-2} ([A_1][A_2] \cdots [A_i] + \overline{[A_{i+1} \cdots A_n]}) \circ \sum_{i=1}^{n} e[A_i]
\end{aligned}$$

so again the diagram commutes.

Case 2. $r \in R_1, P'(r) = [A_1 \cdots A_n] \to [A_1][A_2 \cdots A_n]$ for some $n \geqq 2$. $rHF = 1_{X_1 \cdots X_n}$ where $X_i = [A_i]$ if $A_i \in N$ and $X_i = A_i$ if $A_i \in \Sigma$. The same argument as in (1.4) shows that $r \circ (\eta[A_1] + \eta[A_2 \cdots A_n]) = \eta[A_1 \cdots A_n]$.

Case 3. $r \in R_2$, $P'(r) = [a] \to a$ for some $a \in \Sigma$. $rHF = 1_a$. Since $\eta[a] = r$ and $\eta(a) = 1_a$, again the diagram commutes. □

THEOREM 11. *The data given above form an adjunction* $(H, F; \eta, =)$: $\mathbf{S}(C_2(G)) \rightharpoonup \mathbf{S}(G)$.

Proof. Lemmas 9 and 10 as the triangular identities are immediate. □

COROLLARY 12. *From MacLane* [22, pp. 88–93], $\mathbf{S}(G)$ *is isomorphic to a reflective subcategory of* $\mathbf{S}(C_2(G))$, *F is injective on objects, full and faithful. Hence* $C_2(G)$ *preserves the ambiguities in G. Since H is right-inverse to F, H is full and* $L(C_2(G)) = L(G)$. *An interpretation of either G or* $C_2(G)$ *gives, via the appropriate adjoint functor, the corresponding interpretation of the other grammar.*

The adjunction $(H, F; \eta, =)$: $\mathbf{S}(C_2(G)) \rightharpoonup \mathbf{S}(G)$ provides a proof that $C_2(G)$ is $LR(k)$ iff G is $LR(k)$, strengthening a result of Gray and Harrison [14]. The nicest way to the proof is via certain categories derived from syntax categories.

DEFINITION 3. Let $G = (N, \Sigma, P, S)$ be a cfg with syntax category $\mathbf{S}$. Let $T = \{\alpha + r + w \mid \alpha \in V^*,\ r \in P,\ w \in \Sigma^*\}$ be the set of "rightmost" terms in S. The *rightmost category of G*, $\mathbf{R}(G)$, is the smallest subcategory of $\mathbf{S}$ such that Σ^* is a subset of the objects of $\mathbf{R}(G)$ and T is a subset of the arrows of $\mathbf{R}(G)$.

Objects in $\mathbf{R}(G)$ are of the form $\alpha A w$ for $\alpha \in V^*$, $A \in N$, $w \in \Sigma^*$ or are in Σ^*. $\mathbf{R}(G)$ is not a strict monoidal category. For every derivation from the point $S, f\colon S \to \gamma$, the cocanonical representation of f is rightmost.

PROPOSITION 13. *Every derivation in* $\mathbf{R}(G)$ *is both monic and epic.*

Proof. Epicity (left cancellation) follows from G being of type 0. Monicity (right cancellation) follows from the uniqueness of the rightmost representations in $\mathbf{R}(G)$. □

With care to preserve rightmost representations, the adjunction $(H, F; \eta, =)$: $\mathbf{S}(C_2(G)) \rightharpoonup \mathbf{S}(G)$ provides an adjunction between $\mathbf{R}(C_2(G))$ and $\mathbf{R}(G)$. Here are the details.

The *bracketing functor B*: $\mathbf{R}(G) \to \mathbf{R}(C_2(G))$ is determined by the following data. Let $[\,\cdot\,]\colon V^* \to (V')^*$ be the monoid homomorphism carrying each $X \in V$ to $[X] \in V'$. B is given on objects $\alpha A w \in V^* N \Sigma^*$ by

$$B(\alpha A w) = [\,\cdot\,](\alpha A) + w$$

and is the identity on strings in Σ^*. Next we define a map ζ from some objects of $\mathbf{R}(C_2(G))$ to the arrows of $\mathbf{R}(C_2(G))$. Let $N_i = \{[A] \| A \in N\}$ and $N_e = \{[a] \| a \in \Sigma\}$. For $\alpha[A] \in N_1^* N_i$ and $\sum_{i=1}^n [a_i] \in N_e^*$ let $\zeta(\alpha[A] + \sum_{i=1}^n [a_i]) = \alpha[A] + \sum_{i=1}^n \overline{[a_i]}$ and $\zeta(\sum_{i=1}^n [a_i]) = \sum_{i=1}^n \overline{[a_i]}$. Define Z from the productions of G to the derivations in $\mathbf{R}(C_2(G))$ as:

$$Z(r) = \begin{cases} r \circ \bigcirc_{i=1}^{n-2} \left(\sum_{j=1}^{i} [X_j] + \overline{[X_{i+1} \cdots X_n]} \right) \circ \zeta([X_1] \cdots [X_n]) & \\ \qquad\qquad\qquad\qquad \text{if } P(r) = A \to X_1 \cdots X_n,\ n \geqq 2, \\ r \circ e[B] & \text{if } P(r) = A \to B, \\ r & \text{otherwise.} \end{cases}$$

Now B is defined on the derivations f in $\mathbf{R}(G)$ by letting $\bigcirc_{i=1}^{n} (\mu_i + r + w_i)$ denote the cocanonical representation of f and setting

$$B(f) = \bigcirc_{i=1}^{n} ([\,\cdot\,](\mu_i) + Z(r_i) + w_i).$$

It is apparent by the construction of the rightmost categories that B is a functor, but not a strict monoidal functor as rightmost categories are not strict monoidal.

Since the bracket removal functor H: $\mathbf{S}(C_2(G)) \to \mathbf{S}(G)$ preserves cocanonical representations, H restricts to a bracket removal functor from $\mathbf{R}(C_2(G))$ to $\mathbf{R}(G)$, also denoted by H.

PROPOSITION 14. *For suitable* η: $I_{\mathbf{R}(C_2(G))} \dot{\to} HB$, $(H, B; \eta, =)$: $\mathbf{R}(C_2(G)) \rightharpoonup \mathbf{R}(G)$ *is an adjunction.*

Proof. It suffices to define η so that the appropriate diagrams commute for each term $\mu + r + w$ in $\mathbf{R}(C_2(G))$, as the remaining details are similar to Theorem 11. Let $N_e = \{[a] \| a \in \Sigma\}$. For $[X_1 \cdots X_n] \in N_2$, define

$$e[X_1 \cdots X_n] = \bigcirc_{i=1}^{n-1} ([X_1] \cdots [X_{i-1}] + \overline{[X_i \cdots X_n]}) \circ \zeta([X_1) \cdots [X_n]).$$

For $\alpha \in (V')^*$, $X' \in N'$, $x \in \Sigma^*$ define

$$\eta(\alpha X' x) = \begin{cases} \eta(\alpha) + e(X') + 1_x & \text{if } c(e(X')) \in \Sigma^*, \\ 1_\alpha + e(X') + 1_x & \text{otherwise} \end{cases}$$

$$\eta(x) = 1_x. \qquad \square$$

COROLLARY 15. *B is full, faithful and injective on objects. H is full. B reflects and preserves limits.*

Rightmost categories are close to being a suitable space in which to consider *LR* parsing. It is more convenient to introduce an augmentation so that the objects are the partial strings so far scanned by the *LR* shift-reduce parser.

DEFINITION 4. Let $\mathbf{R}$ be a rightmost category. The *augmented category* $\mathbf{A}$ is given by:

(A1) The objects of $\mathbf{R}$ are the objects of $\mathbf{A}$.

(A2) (f, z): $\alpha \to \beta$ is an arrow of $\mathbf{A}$ whenever f: $\alpha \to \beta z$ is an arrow of $\mathbf{R}$ and $z \in \Sigma^*$.

(A3) $\mathbf{A}$ is the smallest category closed under the composition

$$(f_1, z_1) \circ (f_2, z_2) = (f_1 \circ (f_2 + z_1), z_2 + z_1).$$

$\mathbf{A}$ is a category since composition is associative, i.e.,

$$\begin{aligned} &((f_1, z_1) \circ (f_2, z_2)) \circ (f_3, z_3) \\ &\qquad = (f_1 \circ (f_2 + z_1), z_2 + z_1) \circ (f_3, z_3) \\ &\qquad = (f_1 \circ (f_2 + z_1) \circ (f_3 + z_2 + z_1), z_3 + z_2 + z_1) \\ &\qquad = (f_1 \circ ((f_2 \circ (f_3 + z_2)) + z_1), z_3 + z_2 + z_1) \\ &\qquad = (f_1, z_1) \circ (f_2 \circ (f_3 + z_2), z_3 + z_2) \\ &\qquad = (f_1, z_1) \circ ((f_2, z_2) \circ (f_3, z_3)). \end{aligned}$$

To ease the notation, arrows (f, λ) in $\mathbf{A}$ will frequently be written as f.

Let $\mathbf{R}_1$, $\mathbf{R}_2$ be rightmost categories and $(H, B; \eta, =)\colon \mathbf{R}_2 \rightharpoonup \mathbf{R}_1$ an adjunction. Define

$$H\colon \mathbf{A}_2 \to \mathbf{A}_1\colon (f, z) \mapsto (fH, zH),$$

$$B\colon \mathbf{A}_1 \to \mathbf{A}_2\colon (g, y) \mapsto (gB, yB),$$

$$\eta'(\alpha) = (\eta(\alpha), \lambda).$$

Proposition 16. *$(H, B; \eta', =)\colon \mathbf{A}_2 \rightharpoonup \mathbf{A}_1$ is an adjunction.*

This adjunction between the augmented categories $\mathbf{A}(C_2(G))$ and $\mathbf{A}(G)$ will be used to show that $C_2(G)$ is $LR(k)$ iff G is $LR(k)$. Hereafter we write η for η'.

Because of the adjunction, there is an isomorphism

$$\phi\colon \mathbf{A}_1(\alpha H, \beta) \cong \mathbf{A}_2(\alpha, \beta B)$$

for all objects α, β.

Let $\mathbf{A}_1 = \mathbf{A}(G)$, $\mathbf{A}_2 = \mathbf{A}(C_2(G))$. In this case the adjunction isomorphism specializes to

$$\phi\colon \mathbf{A}_1(S, \beta) \cong \mathbf{A}_2([S], \beta B)$$

which, along with the more general form, will be used subsequently.

Turning specifically to LR-ness, the $LR(k)$ property of a grammar can be characterized by the existence of certain limits in $\mathbf{A}$.

Definition 5. G is $LR(k)$ iff for all $\alpha, \gamma, \omega \in V^*$, $w \in \Sigma^k$, $z, z' \in \Sigma^*$, rules $r\colon A \to \beta$, $r'\colon A' \to \beta'$, if in $\mathbf{R}(G)$

$$f\colon S \to \alpha\beta wz = (S \xrightarrow{f'} \alpha A wz) \circ (\alpha + r + wz)$$

and

$$g\colon S \to \alpha\beta wz' = (S \xrightarrow{g'} \gamma A' \omega) \circ (\gamma + r' + \omega)$$

then $r = r'$, $\alpha = \gamma$, and $\omega = wz'$.

One may compare this definition with that of Hotz and Claus [20, p. 191].

Definition 6. Let $(f, z)\colon S \to \delta$ be an arrow in $\mathbf{A}(G)$, for some cfg G. *f has k-handle* $(\alpha + r + w)$ for rule $r\colon A \to \beta$, $w \in \Sigma^k$, if $\delta = \alpha\beta w$ and $(f, z) = (f', z) \circ (\alpha + r + w, \lambda)$. Further for $\alpha, \beta \in V^*$, $w \in \Sigma^k$, $\alpha\beta w$ is a *parsable prefix* with *k-parse-term* $(\alpha + r + w)$ if there is an (f, z) with k-handle $(\alpha + r + w)$. G is *locally* LR(k) at γ if γ is a parsable prefix with k-parse-term $(\alpha + r + w)$ and for all $(g, z)\colon S \to \gamma$, $(\alpha + r + w)$ is the k-handle of (g, z).

The following two propositions are the characterization of LR-ness as a set of limits in $\mathbf{A}(G)$.

Proposition 17. *G is $LR(k)$ iff G is locally $LR(k)$ at every parsable prefix.*

Proposition 18. *G is locally $LR(k)$ at γ iff γ is a parsable prefix with k-parse-term $t\colon \delta \to \gamma$ and for all $(f, z)\colon S \to \gamma$ there is a unique $(f', z)\colon S \to \delta$ such that*

$$\begin{tikzcd}
S \arrow[d, "{(f,z)}"'] \arrow[dr, dashed, "{(f',z)}"] & \\
\gamma & \delta \arrow[l, "t"]
\end{tikzcd}$$

commutes.

The limit characterization of LR-ness, together with the fact that $B\colon \mathbf{A}_1 \to \mathbf{A}_2$—as a full and faithful right adjoint—preserves and reflects limits, gives the proof that $C_2(G)$ is $LR(k)$ iff G is $LR(k)$. We begin with lemmata.

LEMMA 19. *If an arbitrary* cfg *G is locally $LR(k)$ at γ with k-parse-term $t\colon \delta \to \gamma$ then for all $y \in \Sigma^*$ and all $(f, z)\colon S \to \gamma y$ there is a unique $(f', z)\colon S \to \delta y$ such that*

$$\begin{tikzcd}
 & \delta y \arrow[d, "t+y"] \\
S \arrow[ur, dashed, "{(f',z)}"] \arrow[r, "{(f,z)}"'] & \gamma y
\end{tikzcd}$$

commutes.

Proof. The map $\Psi(\gamma, y)\colon \mathbf{A}(S, \gamma y) \to \mathbf{A}(S, \gamma)\colon (f, z) \to (f, yz)$ is injective for each γ and y. Thus if every arrow $S \to \gamma$ factors uniquely through t, every arrow $S \to \gamma y$ factors uniquely through $t + y$. □

LEMMA 20. *If $C_2(G)$ is $LR(k)$ then for all $\theta \in (V')^*\Sigma^*$ for which there exists $(f, z)\colon [S] \to \theta$ in $\mathbf{A}_2$, every $(g, z)\colon [S] \to \theta HB$ factors uniquely through $\eta(\theta)\colon \theta \to \theta HB$.*

Proof. If $\eta(\theta)$ is the identity, the conclusion is trivial. Otherwise, let $\bigcirc_{i=1}^n t_i$ be the cocanonical decomposition of $\eta(\theta)$. Let γ_n be the parsable prefix of θHB, with unique k-parse-term t'_n. $\theta HB = \gamma_n + y$ for some $y \in \Sigma^*$ since θHB is a right sentential form. By Lemma 19 (g, z) factors through $t'_n + y$ and $t_n = t'_n + y$. A straightforward induction from t_n to t_1 finishes the proof. □

COROLLARY 21. $\mathbf{A}_2([S], \theta) \cong \mathbf{A}_2([S], \theta HB)$ *for* $\theta \in (V')^*\Sigma^*$.

THEOREM 22. *If $C_2(G)$ is $LR(k)$ then G is $LR(k)$.*

Proof. The argument uses the adjunction between $\mathbf{A}_2 = \mathbf{A}(C_2(G))$ and $\mathbf{A}_1 = \mathbf{A}(G)$ to find, for each parsable prefix γ of G, certain limit diagrams in $\mathbf{A}_2$. Since $B\colon \mathbf{A}_1 \to \mathbf{A}_2$ reflects limits, the $\mathbf{A}_2$ diagrams reflect into limit diagrams in $\mathbf{A}_1$, showing that G is locally $LR(k)$ at γ.

Let γ be a parsable prefix of G with some k-parse-term $t\colon \delta \to \gamma$, $t = (\alpha + r + w)$. One has $B(t) = ([\,\cdot\,](\alpha) + r + w) \circ \eta(\theta)$ where $\theta = [\,\cdot\,](\alpha) + c(r) + w$ with $c(r)$ being the codomain of r in $C_2(G)$, and $\gamma = \theta H$ so that $\gamma B = \theta HB$. For any $(g, y)\colon [S] \to \gamma B$, Lemma 20 shows that there is a unique $(g', y)\colon [S] \to \theta$ for which $(g, y) = (g', y) \circ \eta(\theta)$. Write t' for $([\,\cdot\,](\alpha) + r + w)$. Since t is a k-parse-term, there are $(f, z)\colon S \to \gamma$ and $(f', z)\colon S \to \delta$ in $\mathbf{A}_1$ for which $(f, z) = (f', z) \circ t$. Then $B(f, z) = B(f', z) \circ t' \circ \eta(\theta)$. Since $C_2(G)$ is $LR(k)$, t' is the unique k-parse-term at θ.

Hence each $(g, y)\colon [S] \to \gamma B$ uniquely factors

$$\begin{tikzcd}
 & \delta B \arrow[d, "t'"] \\
{[S]} \arrow[ur, dashed, "{(g',y)}"] \arrow[r] \arrow[dr, "{(g,y)}"'] & \theta \arrow[d, "\eta(\theta)"] \\
 & \theta HB = \gamma B
\end{tikzcd}$$

through $t' \circ \eta(\theta)$ as shown in the diagram. Since B is full, every $(g, y)\colon [S] \to \gamma B$ is the image of some $(f, y)\colon S \to \gamma$. Since B reflects limits,

$$\begin{tikzcd}
 & \delta \arrow[dd, "t"] \\
S \arrow[ur, dashed] \arrow[dr, "{(f, y)}"'] & \\
 & \gamma
\end{tikzcd}$$

is a limit diagram in $\mathbf{A}_1$, indeed a limit diagram for every $(f, y)\colon S \to \gamma$ since B is faithful. Hence every $(f, y)\colon S \to \gamma$ factors uniquely through t. □

THEOREM 23 (Gray and Harrison). *If G is $LR(k)$ then $C_2(G)$ is $LR(k)$.*

Proof. We show that $C_2(G)$ is locally $LR(k)$ at every parsable prefix using a proof by rule type. The cases are (i) the rule is one of the $\overline{[a]}$ in R_2, (ii) the rule is in R, and (iii) the rule is one of the $\overline{[\beta]} \in R_1$.

(i) Let γ be a parsable prefix in $\mathbf{A}_2$ with 0-parse-term $(\alpha + \overline{[a]})$. By the construction of $C_2(G)$, it is clear that $C_2(G)$ is locally $LR(0)$ at γ. By Lemma 19, $C_2(G)$ is locally $LR(k)$ at γw for each $w \in \Sigma^k$ such that γw is a parsable prefix.

(ii) Let γ be a parsable prefix with k-parse-term $t = (\alpha + r + w)$ for $r \in R$. Let $(g, y)\colon [S] \to \gamma$ factor through t, $(g, y) = (g', y) \circ t$ for some g'. Since $(gHB, y) = (g, y) \circ \eta(\gamma)$, we have the diagram

$$\begin{tikzcd}
 & \delta \arrow[d, "t"] \\
{[S]} \arrow[ur, "{(g', y)}"] \arrow[r, "{(g, y)}"] \arrow[dr, "{(gHB, y)}"'] & \gamma \arrow[d, "\eta(\gamma)"] \\
 & \gamma HB
\end{tikzcd}$$

Since $r \in R$, by a trivial induction $\delta HB = \delta$.

Noting that $S = [S]H$ the pertinent adjunction isomorphisms are:

$$\phi\colon \mathbf{A}_2([S], \gamma HB) \cong \mathbf{A}_1(S, \gamma H),$$

$$\phi\colon \mathbf{A}_2([S], \delta HB) \cong \mathbf{A}_1(S, \delta H),$$

$$\phi\colon \mathbf{A}_2(\delta, \gamma HB) \cong \mathbf{A}_1(\delta H, \gamma H)$$

giving the commuting diagram

$$\begin{tikzcd}
 & \delta H \arrow[dd, "\phi(t \circ \eta(\gamma))"] \\
S \arrow[ur, "{\phi(g', y)}"] \arrow[dr, "{\phi(gHB, y)}"'] & \\
 & \gamma H
\end{tikzcd}$$

in $\mathbf{A}_1$. Now $\phi(t \circ \eta(\gamma)) = (\alpha H + r + w)$, so it is the unique k-parse-term for parsable prefix γH, since G is $LR(k)$. Since B—as right adjoint—preserves limits, each $B(f, z)$: $[S] \to \gamma HB$ factors uniquely through $t \circ \eta(\gamma) = B(\phi(t \circ \eta(\gamma)))$. Since $\eta(\gamma)$ is monic, $B(f, z)$ factors uniquely through $\eta(\gamma)$, $B(f, z) = (h, z) \circ \eta(\gamma)$ for some unique $(h, z): [S] \to \gamma$. Therefore $\mathbf{A}_2([S], \gamma) \cong \mathbf{A}_2([S], \gamma HB) \cong \mathbf{A}_1(S, \gamma H)$. Hence every (h, z): $[S] \to \gamma$ factors uniquely through t: $\delta \to \gamma$, so that $C_2(G)$ is locally $LR(k)$ at γ.

(iii) Let γ be a parsable prefix with k-parse-term $t: \delta \to \gamma$, $t = (\alpha + \overline{[\beta]} + w)$, $\overline{[\beta]} \in R_1$. Let (g, y): $[S] \to \gamma$ factor through t. By the construction of $C_2(G)$, there exists (g', y) such that

$$(g, y) = (g', y) \circ \left(\alpha + \left(r \circ \bigcirc_{i=1}^{k} ([X_1] \cdots [X_i] + \overline{[X_{i+1} \cdots X_n]})\right) + w\right)$$

for some $r \in R$ and in which $\beta = X_{k+1} \cdots X_n$. The proof in part (ii) then goes through to show that $C_2(G)$ is locally $LR(k)$ at γ. □

6. Notes on covering. This section discusses certain aspects of Gray and Harrison's thought-provoking paper [14]. Obviously, familiarity with it is assumed.

Gray and Harrison originally presented Theorem 23 (their Theorem 2.1) in terms of *covers*. The following proposition shows that we have correctly used functors in the previous section.

PROPOSITION 24. *Let F: $\mathbf{S}(G') \to \mathbf{S}(G)$ be an externally fixed, externally full grammar functor for* cfg's *G', G, such that for each rule $r' \in R'$, $F(r')$ is a rule of R or an identity (zero-length) derivation. Let $H' = \{r' \in R' | F(r') \in R\}$. Then (G', H') covers (G, P) under F; hence G' completely covers G.*

Since H: $\mathbf{S}(C_2(G)) \to \mathbf{S}(G)$ is full, we immediately obtain their Theorem 1.1: $C_2(G)$ completely covers G. We note in passing that the construction used in their Theorem 2.2 is functorial as well.

Furthermore, the concept of *weak Reynolds cover* introduced by them is actually nothing but an externally fixed grammar functor, first explicated in 1966 by Hotz [18]. Therefore the existence of H and F between $\mathbf{S}(C_2(G))$ and $\mathbf{S}(G)$ show that $C_2(G)$ and G weakly Reynolds cover each other, despite ¶2 in Gray and Harrison [14, p. 684].

These methods also show that part (b) of the proposition on page 682 of [14] is wrong. To say that (G', P') covers (G, P) under F says only that F is a functor taking rules in R' into rules in R. This doesn't suffice to show that $\mathbf{S}'(S', x)$ and $\mathbf{S}(S, x)$ have the same cardinality for each $x \in L(G) = L(G')$ as trivial examples show. One requires functor surjectivity.

Despite these minor problems, covers form a more powerful tool than grammar functors as their Theorem 1.2 [14] and the paper of Mickunas [23] demonstrate. Both of these concepts, central to the idea of parsing simulation, deserve further work. Somewhat related program simulation ideas are in Goguen [12].

7. Concluding remarks on categorical methods. Comparing the examples presented here with the cited literature shows that one can obtain stronger results for very little additional effort by applying categorical algebra. For example, Theorem 22 fell out essentially for free while obtaining the current proof of Theorem 23. Indeed, comparing the structure of the proof for Theorem 23 with Gray and Harrison's original proof shows that the current proof is essentially simpler. This is not surprising once one notices that Gray and Harrison have almost established the adjunction and almost proved the preservation of limits, general theorems on which we rely.

Using the tools of categorical algebra simplifies and sharpens many of the results in formal language theory. The reader may find that most of the effort in the examples presented is in setting up the requisite categories, functors and natural transformations. This is, I believe, typical of the applications of category theory. The categorical propositions are general and powerful; the application requires only finding the appropriate framework in which to apply the propositions as opposed to the traditional technique of creating proofs de novo. At least part of the purpose is the discovery of how the algebra of formal language syntax relates to other branches of mathematics. Since monoidal categories are ubiquitous, one finds many connections, especially to universal algebra (Benson [7], Benabou [5]).

Acknowledgment. I appreciate the thoughts provided by referees, resulting in clarifying the relationship of grammar functors to other notions of grammatical relationships.

REFERENCES

[1] A. V. AHO AND J. D. ULLMAN, *The Theory of Parsing, Translation, and Compiling. Volume I: Parsing*, Prentice-Hall, Englewood Cliffs, NJ, 1972.

[2] ———, *The Theory of Parsing, Translation, and Compiling. Volume II: Compiling*, Prentice-Hall, Englewood Cliffs, NJ, 1973.

[3] S. ALAGIĆ, *Natural state transformations*, J. Comput. System Sci., 10 (1975), no. 2, pp. 266–307.

[4] M. A. ARBIB AND E. G. MANES, *Arrows, Structures, and Functors: The Categorical Imperative*, Academic Press, New York, 1975.

[5] J. BENABOU, *Structures algébriques dans les categories*, Cahiers Topologie Géom. Différentielle, 10 (1968), no. 1, pp. 1–126.

[6] D. B. BENSON, *Syntax and semantics: A categorical view*, Information and Control, 17 (1970), pp. 145–160.

[7] ———, *Semantic preserving translations*, Math. Systems Theory, 8 (1974), no. 2, pp. 105–126.

[8] ———, *The basic algebraic structures in categories of derivations*, Information and Control, 28 (1975), pp. 1–29.

[9] E. BERTSCH, *An observation on relative parsing time*, J. Assoc. Comput. Mach., 22 (1975), pp. 493–498.

[10] ———, *Surjectivity of functors on grammars*, Math. Systems Theory, 9 (1976), no. 4, pp. 298–307.

[11] H. W. BUTTLEMANN, *Semantic directed translation of context free languages*, Amer. J. Computational Linguistics 1, Micro 7, 1974.

[12] J. A. GOGUEN, *On homomorphisms, correctness, termination, unfoldments, and equivalence of flow diagram programs*, J. Comput. System Sci., 8 (1974), pp. 333–365.

[13] J. A. GOGUEN, J. W. THATCHER, E. G. WAGNER AND J. B. WRIGHT, *Initial algebra semantics*, IBM Research Rep. RC 5701, Yorktown Heights, NY, 1976.

[14] J. N. GRAY AND M. A. HARRISON, *On the covering and reduction problems for context-free grammars*, J. Assoc. Comput. Mach., 19 (1972), pp 675–698.

[15] S. A. GREIBACH, *A new normal form theorem for context free phrase structure grammars*, Ibid., 12 (1965), pp. 42–52.

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

[17] H. Herrlich and G. Strecker, *Category Theory*, Allyn and Bacon, Boston, 1973.

[18] G. Hotz, *Eindeutigkeit und Mehrdeutigkeit formaler Sprachen*, Elektron. Informationsverarbeit. Kybernetik, 2 (1966), pp. 235–247.

[19] ———, *Strukturelle Verwandtschaften von Semi-Thue-Systemen*, Category Theory Applied to Computation and Control, Lecture Notes in Computer Science No. 25, Springer-Verlag, New York, 1975, pp. 174–179.

[20] G. Hotz and V. Claus, *Automaten-Theorie und Formale Sprachen III. Formale Sprachen*, Bibliographisches Institut, Mannheim, West Germany, 1972.

[21] D. E. Knuth, *Semantics of context free languages*, Math. Systems Theory, 2 (1968), pp. 127–145.

[22] S. MacLane, *Categories for the Working Mathematician*, Springer-Verlag, New York, 1971.

[23] M. D. Mickunas,*On the complete covering problem for $LR(k)$ grammars*, J. Assoc. Comput. Mach., 23 (1976), pp. 17–30.

[24] P. S. Peters, Jr, and R. W. Ritchie, *Context-sensitive immediate constituent analysis: Context-free languages revisited*, Math. Systems Theory, 6 (1973), pp. 324–333.

[25] W. J. Savitch, *How to make arbitrary grammars look like context-free grammars*, this Journal, 2 (1973), pp. 174–182.

[26] C. P. Schnorr, *Transformational classes of grammars*, Information and Control, 14 (1969), pp. 252–277.

[27] C. P. Schnorr and H. Walter, *Pullback-Konstruktionen bei Semi-Thue-Systemen*, Elektron. Informationsverarbeit. Kybernetik, 5 (1969), pp. 27–36.

[28] D. F. Stanat, *Approximation of weighted type 0 languages by formal power series*, Information and Control, 21 (1972), pp. 344–381.

[29] H. Walter, *Verallgemeinerter Pullback-Konstruktionen bei Semi-Thue-Systemen und Grammatiken*, Elektron. Informationsverarbeit. Kybernetiik, 6 (1970), pp. 239–254.

[30] ———, *Grammatik-und-Sprachfamilien, Teil 1, Diagrammergänzungssätze und Abschlusseigenschaften*, Tech. Rep. AFS-13 Informatik, Technische Hochschule Darmstadt, West Germany, 1975.

[31] W. A. Woods, *Context-sensitive parsing*, Comm. ACM, 13 (1970), pp. 437–455.

SIAM J. COMPUT.
Vol. 6, No. 3, September 1977

IMPLEMENTATION CORRECTNESS INVOLVING A LANGUAGE WITH goto STATEMENTS*

BRUCE D. RUSSELL†

Abstract. Two languages, one a simple structured programming language, the other a simple goto language, are defined. A denotational semantics is given for each language. An interpreter for the goto language is given and is proved correct with respect to the denotational semantics. A compiler from the structured to the goto language is defined and proved to be a semantically invariant translation of programs. The proofs are by computational induction.

Key words. compiler, interpreter, semantics, denotational semantics, implementation correctness, computational induction

1. Introduction. If a formal definition of the semantics of a programming language is to be useful, it is essential that the implementation of the language is consistent with the formal definition. It makes little sense to prove the correctness of a program, or improve a program with semantically invariant transformations, if the implementation of the language does not preserve the semantics. Further, while we may program with high level, structured programming languages these programs will probably be translated into the machine language of the machine and the machine language will then be interpreted by the hardware. The correctness of both the compiler into the machine language and the interpreter for the machine language is essential.

To illustrate these problems and to give examples of the kinds of proofs required we define a structured programming language and a goto language. A denotational semantics is given for both languages. An interpreter is defined for the goto language and its correctness is proved. A compiler from the structured to the goto language is defined and its correctness is also proved. The example languages are deliberately kept simple to facilitate understanding.

2. Method and notation. The denotational semantics for the two languages are specified, following Scott [11], Scott and Strachey [13], Strachey and Wadsworth [14], as a function from a set of syntactic objects called *programs* into some set of "meanings" or denotations, appropriate for the particular language. The *compiler* will be defined as a function from one syntactic set of programs, the source language, into another syntactic set of programs, the target language. The *interpreter* will be defined as a function from machine states into machine states.

In all these cases the following questions arise. Are there sets of objects with the sorts of properties we require, to model things like syntax, machine states and denotations? Are there functions that behave as the semantic, interpreter and compiler functions should? Finally, can we prove things about these functions? The affirmative answers to these questions are provided by using the domain construction techniques of Scott [10], [12] and the least fixed point approach to the solution of function equations.

* Received by the editors April 2, 1975, and in revised form March 25, 1976.

† Computer Science Department, University College, Dublin, Ireland. Now at Department of Computer and Information Science, Ohio State University, Columbus, Ohio 43210.

We assume that a domain is a set that has a partial ordering (denoted $\leqq$), a least element (denoted $\perp$) and in which every chain (totally ordered subset) has a least upper bound (denoted $\bigsqcup$). The simplest way to form a domain is to extend any set A with a single element $\perp$. The partial ordering is defined as $\perp \leqq a$ and $a \leqq a$ for all $a \in A$. We call this the *natural extension of* A. Some typical extensions are I the integers, T the truth values, ID the identifiers. We will also extend any function f from A to B to the natural extensions of A and B. We call this the *natural extension of* f and its value for $a \in A$ is $f(a)$, and for $\perp$ in the natural extension of A its value is $\perp$ in the natural extension of B. Relations will be similarly extended.

We also form domains from existing domains using the operations of disjoint union (denoted +), and Cartesian product (denoted ×). If we define a domain D as

$$D = D_1 + D_2 + \cdots + D_m$$

we have the following function:

$$\text{inspection-}i\text{:} \qquad D \rightarrow T$$

$$\text{inspection-}i(d) = \begin{cases} \text{true}, & d \in D_i, \\ \perp, & d = \perp, \\ \text{false} & \text{otherwise.} \end{cases}$$

We will write this as $d : D_i$.

We also have the function

$$\text{projection-}i\text{:} \qquad D \rightarrow D_i$$

$$\text{projection-}i(d) = \begin{cases} d, & d \in D_i, \\ \perp & \text{otherwise} \end{cases}$$

where $\perp$ is an element of D_i. We will write this as $d|D_i$.

If we define a domain D as

$$D = D_1 \times D_2 \times \cdots \times D_m$$

we have the following functions:

constructor : $D_1 \times D_2 \times \cdots \times D_m \rightarrow D$

constructor$(d_1, d_2, \cdots, d_m)$ = the $d \in D$ corresponding to $\langle d_1, d_2, \cdots, d_m \rangle$

destructor : $D \rightarrow D_1 \times D_2, \cdots, D_m$

destructor(d) = the $\langle d_1, d_2, \cdots, d_m \rangle$ corresponding to d.

The final way of forming a domain from existing domains is to form the function space from one domain into another. The functions we are interested in are those that are continuous. Since we have required that domains be chain closed we will insist that all functions over the domains are continuous in the sense that they preserve the limits of chains. More formally a function $f : D \rightarrow D$ is continuous iff for any chain $X \subseteq D$ $\mathbf{U}'f(X) = f(\mathbf{U}X)$. We will denote the set of all continuous functions from domain D to domain D' by $D \to D'$. If we define a partial ordering over $D \to D'$ by $f \leqq g$ iff $f(d) \leqq g(d)$ for all $d \in D$, then $D \to D'$ is also a domain.

We now turn our attention to continuous functions and a notation for writing them. First note that we have a large supply of continuous functions from natural extensions, all of which are continuous. Also the identity function and constant functions are continuous as well as the functions associated with the domain constructions + and ×. The composition of two continuous functions $(f \cdot g)$ is continuous, as are any abstractions written in a lambda calculus notation, for example, $\lambda x.f(x, y)$, $\lambda x.\lambda y.f(x, y)$, $\lambda f.f(x)$, etc. For a detailed development of this sort of notation for continuous functions see Scott [10] or Reynolds [9].

A very important continuous function is the conditional function (called cond). Its logical type is $T \times D \times D \to D$ and its definition is

$$\text{cond}(t, d_1, d_2) = \begin{cases} d_1, & t = \text{true}, \\ d_2, & t = \text{false}, \\ \bot, & t = \bot. \end{cases}$$

We will write this as **if** t **then** d_1 **else** d_2.

Another continuous function that we will make use of in what follows is the update function. Its logical type is

$$\text{update}: ((A \to B) \times A \times B) \to (A \to B))$$

and its definition is

$$\text{update}(f, a, b) = \lambda a'.\mathbf{if}\ a = a'\ \mathbf{then}\ b\ \mathbf{else}\ f(a').$$

The notation we will use for update(f, a, b) is $f[b/a]$.

Since most of the functions we use are defined by cases over a syntactic domain we make use of the **case** and **let** notation due to Burstall [2]. For example if we define the domain PROGRAM as

$$\begin{aligned} \text{PROGRAM} &= \text{ASSIGNMENT} + \text{COMPOUND} \\ \text{COMPOUND} &= \text{PROGRAM} \times \text{PROGRAM} \\ \text{ASSIGNMENT} &= \text{IDENTIFIER} \times \text{EXPRESSION} \end{aligned}$$

we may then write

```
cases                $p$:
assignment($x$, $e$)      : "some expression in $x$ and $e$"
compound($p1$, $p2$) : "some expression in $p1$ and $p2$"
```

for

```
if $p$:ASSIGNMENT then
  let assignment($x$, $e$) = $p$ in "some expression in $x$ and $e$"
else if $p$ : COMPOUND then
        let compound($p1$, $p2$) = $p$ in "some expression $p1$ and $p2$".
```

As an example of these notations let us define a domain LIST of linear lists over domain ATOM and define the usual functions of head, tail and cons.

$$\begin{aligned} \mathrm{LIST} &= \mathrm{NIL} + \mathrm{COMPOUND} \\ \mathrm{COMPOUND} &= \mathrm{ATOM} \times \mathrm{LIST} \end{aligned}$$

$$\begin{array}{l} \mathrm{cons}: \mathrm{ATOM} \times \mathrm{LIST} \to \mathrm{LIST} \\ \mathrm{cons}(a, e) = \mathrm{compound}(a, e) \end{array}$$

$$\begin{array}{l} \mathrm{head}: \mathrm{LIST} \to \mathrm{ATOM} \\ \mathrm{head}(e) = \\ \quad \mathbf{cases}\ e: \\ \quad\quad \mathrm{nil}(\ ): \bot \\ \mathrm{compound}(a, e'): a \end{array}$$

$$\begin{array}{l} \mathrm{tail}: \mathrm{LIST} \to \mathrm{LIST} \\ \mathrm{tail}(e) = \\ \quad \mathbf{cases}\ e: \\ \quad\quad \mathrm{nil}(\ ): \mathrm{nil}(\ ) \\ \mathrm{compound}(a, e'): e'. \end{array}$$

Finally if we define a function recursively it is the least fixed point of the associated functional that is intended. For example, by the definition $f(n) = \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n \times f(n-1)$ we mean the least fixed point of the functional

$$\lambda f. \lambda n. \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n \times f(n-1).$$

To prove the equivalence of two expressions involving recursively defined functions we may make use of Scott's rule of computational induction. Suppose $F(f)$, $G(g)$ are expressions in the recursively defined functions $f = t1$ and $g = t2$. Then to show $F(f) \equiv G(g)$ it is sufficient to show

$$\text{(i)} \quad F(\bot) \equiv G(\bot)$$

$$(ii) \quad F(f) \equiv G(g) \Rightarrow F(t1) \equiv G(t2)$$

so long as the F, G, f, g are continuous.

There are many detailed expositions of the least fixed point approach to the solution of function equations such as de Bakker and de Roever [1], Manna, Ness and Vuillemin [4], Reynolds [9], and Scott [12], among others.

We now have the tools we need to define domains, define functions over the domains and then prove things about these functions.

3. Denotational semantics for the two languages. The first language might be called an ALGOL-like, structured programming language. It consists of an "assignment" statement, an "if" statement and a "while" statement. Two programs may be put together (perhaps using ;) to form a compound program.

The semantics of such a language is given by specifying for each program in the language what its effect on the values of identifiers is. The values of the identifiers at any point in time is given by STORE = ID → VALUES and thus the semantics of a program is some element of STORE → STORE.

The formal definition follows:

Syntax

$$\begin{aligned}
p \in \mathrm{STRUCPROGRAM} &= \mathrm{ASSIGN}\\
&\quad + \mathrm{IF}\\
&\quad + \mathrm{WHILE}\\
&\quad + \mathrm{COMPOUND}\\
\mathrm{ASSIGN} &= \mathrm{ID} \times \mathrm{EXP}\\
\mathrm{IF} &= \mathrm{EXP} \times \mathrm{STRUCPROGRAM}\\
&\quad \times \mathrm{STRUCPROGRAM}\\
\mathrm{WHILE} &= \mathrm{EXP} \times \mathrm{STRUCPROGRAM}\\
\mathrm{COMPOUND} &= \mathrm{STRUCPROGRAM}\\
&\quad \times \mathrm{STRUCPROGRAM}\\
\mathrm{ex} \in \mathrm{EXP} &= \text{“usual expressions”}\\
x \in \mathrm{ID} &= \text{“usual identifiers”}
\end{aligned}$$

Semantics

$$\begin{aligned}
s \in \mathrm{STORE} &= \mathrm{ID} \to \mathrm{VALUES}\\
\mathrm{VALUES} &= \text{“set of language dependent values”}
\end{aligned}$$

$$\mathrm{sem}' : \mathrm{STRUCPROGRAM} \to (\mathrm{STORE} \to \mathrm{STORE})$$

$$\mathrm{eval} : \mathrm{EXP} \times \mathrm{STORE} \to \mathrm{VALUES}$$

$$\begin{aligned}
&\mathrm{sem}'(p)(s) =\\
&\qquad \textbf{cases } p:\\
&\qquad \mathrm{assign}(x, \mathrm{ex}) : s[\mathrm{eval}(\mathrm{ex}, s)/x]\\
&\qquad \mathrm{if}(\mathrm{ex}, p1, p2) : \textbf{if } \mathrm{eval}(\mathrm{ex}, s) \textbf{ then } \mathrm{sem}'(p1)(s) \textbf{ else } \mathrm{sem}'(p2)(s)\\
&\qquad \mathrm{while}(\mathrm{ex}, \mathrm{p1}) : \textbf{if } \mathrm{eval}(\mathrm{ex}, s) \textbf{ then } (\mathrm{sem}'(p) \cdot \mathrm{sem}'(p1))(s) \textbf{ else } s\\
&\qquad \mathrm{compound}(p1, p2) : (\mathrm{sem}'(p2) \cdot \mathrm{sem}'(p1))(s).
\end{aligned}$$

Let us examine sem′ clause by clause:

assign(x, ex): Evaluate the expression ex in the store s and make this the new value of x in the store.

if (ex, $p1$, $p2$): Evaluate the expression ex in the store s. If its value is true, then execute the then clause $p1$, otherwise execute the else clause $p2$.

while(ex, $p1$): Evaluate the expression ex in the store s. If it is false then do nothing to the store, otherwise execute the body of the loop $p1$ and then execute the entire loop p.

compound($p1$, $p2$): First execute $p1$ and then $p2$.

The second language might be called a FORTRAN-like, "goto" language. It consists of an "assignment" statement, an "ifnot" statement, a "goto" statement and a "program" statement. A program is an n-tuple of labeled statements with one extra terminating label at the end.

The "assignment" and "goto" statements are obvious enough. The "ifnot" tests a condition and if it is false then branches. A *program statement* is an n-tuple of labeled statements with a terminating label, where only branches within the statement are allowed. The use of this last statement may easily be eliminated from any program by making all labels in the program distinct. It is included for convenience.

Programs in this language give a meaning to every label that occurs in the program. The meaning, or semantics, of a label is the element of STORE →

STORE associated with the execution of the program starting at that label. If we let CONTINUATION = STORE → STORE and LABELENV = LABEL → CONTINUATION, then the semantics for the labels will be given by an element of LABELENV. The function sem will thus be of logical type PROGRAM → LABELENV. It will be defined by taking an everywhere undefined label environment and giving a value to each label that occurs in the program. The value given to a label is the semantics of the statement prefixed by that label.

The semantics of an individual statement in this language is a little more complicated. We cannot take the element of STORE → STORE that each statement denotes and simply compose them, since we may never execute the next statement if the first is a goto. To phrase it differently, once we have executed a given statement we then continue with the following statement or continue with a statement whose label is mentioned in a goto. Thus the type of the function semstatement, giving the semantics of individual statements, is STATEMENT → (LABELENV → (CONTINUATION → CONTINUATION)) where LABELENV=LABEL → CONTINUATION gives the semantics of the labels and CONTINUATION is the semantics of the program starting at the next sequential statement.

Syntax

$$\begin{aligned}
p \in \mathrm{PROGRAM} &= \mathrm{LABELEDSTATEMENT}^{+} \times \mathrm{LABEL}\\
\mathrm{LABELEDSTATMENT} &= \mathrm{LABEL} \times \mathrm{STATEMENT}\\
t \in \mathrm{STATEMENT} &= \mathrm{ASSIGN}\\
&\quad + \mathrm{IFNOT}\\
&\quad + \mathrm{GOTO}\\
&\quad + \mathrm{PROGRAM}\\
\mathrm{ex} \in \mathrm{EXP} &= \text{“usual expressions”}\\
x \in \mathrm{ID} &= \text{“usual identifiers”}\\
l \in \mathrm{LABEL} &= \text{“usual labels”}\\
\mathrm{ASSIGN} &= \mathrm{ID} \times \mathrm{EXP}\\
\mathrm{IFNOT} &= \mathrm{EXP} \times \mathrm{LABEL}\\
\mathrm{GOTO} &= \mathrm{LABEL}
\end{aligned}$$

Semantics

$$\begin{aligned}
s \in \mathrm{STORE} &= \mathrm{ID} \to \mathrm{VALUES}\\
\mathrm{VALUES} &= \text{“set of language dependent values”}\\
c \in \mathrm{CONTINUATION} &= \mathrm{STORE} \to \mathrm{STORE}\\
e \in \mathrm{LABELENV} &= \mathrm{LABEL} \to \mathrm{CONTINUATION}
\end{aligned}$$

$$\begin{aligned}
\mathrm{sem}: &\ \mathrm{PROGRAM} \to \mathrm{LABELENV}\\
\mathrm{semstatement}: &\ \mathrm{STATEMENT}\\
&\to (\mathrm{LABELENV} \to (\mathrm{CONTINUATION} \to \mathrm{CONTINUATION}))\\
\mathrm{eval}: &\ \mathrm{EXP} \times \mathrm{STORE} \to \mathrm{VALUES}.
\end{aligned}$$

We will occasionally wish to obtain the label on the first statement of a program. The function "firstlabel" will do this.

```
firstlabel : PROGRAM → LABEL
firstlabel($p$) =
        let ⟨$l1$: $t1$; · · · ; $ln$: $tn$; $ln+1$⟩ = $p$
        in $l1$
```

```
semstatement : STATEMENT
            → (LABELENV → (CONTINUATION → CONTINUATION))
semstatement($t$)($e$)($c$) =
      cases $t$:
   assign($x$, ex) : $c \cdot \lambda s\, .\, s[\text{eval}(\text{ex}, s)/x]$
     ifnot(ex, $l$) : $\lambda s.$ if eval(ex, $s$) then $c(s)$ else $e(l)(s)$
        goto($l$) : $e(l)$
   program($p$) : $c \cdot$ sem($p$) (firstlabel ($p$))
```

Let us examine semstatement clause by clause, where e is the labelenv specifying the continuation associated with each label and c is the continuation associated with the next sequential statement.

assign (x, ex): Modify the store as usual for an assignment statement and then go on to the next sequential instruction.

ifnot(ex, e): Evaluate the expression ex in the store and if true then go on to the next sequential instruction else go to the statement labeled l.

goto(l): Do nothing to the store and carry on with the statement labeled l.

program(p): Take the semantics of p isolated from any of the labels outside p. Apply the element of STORE → STORE given by the label on the first statement of p and then carry on with the next sequential instruction.

```
    sem : PROGRAM → LABELENV
sem($p$) =
let $\langle l1\colon t1; l2\colon t2; \cdots; ln\colon tn; ln+1\rangle = p$
in $\bot$[semstatement($t$1)(sem($p$))(sem($p$)($l$2))/$l$1]
    [semstatement($t$2)(sem($p$))(sem($p$)($l$3))/$l$2]
        ⋮
    [semstatement($tn$)(sem($p$)(sem($p$)($ln$ + 1))/$ln$]
    [$\lambda s . s/ln + 1$]
```

Now the definition of sem says we are to create a labelenv sem(p) where the value of a label li on a statement ti is given by taking the semantics of the statement ti, semstatement(ti), with the labelenv of the entire program, sem(p), and the continuation of the next sequential instruction sem(p)(li + 1). If it is the last statement in the program then the continuation of the next sequential instruction would be to do nothing or $\lambda s . s$.

4. An interpreter and its correctness. The first implementation model is an interpreter, called "imp", for the goto language. It is a version of the 3 component (program, instruction pointer, data) model so common in operational semantics. Since every statement in our program is labeled we use a label as an instruction pointer. The data component is just the same STORE we have been using so far.

The formal definition of imp is:

```
imp : PROGRAM × LABEL × STORE → STORE
imp(⟨$l$1: $t$1; $l$2: $t$2; ⋯ ; $ln$: $tn$; $ln$ + 1⟩, $li$, $s$) = if $li = ln + 1$ then $s$
```

```
else cases ti:
              assign(x, ex):imp(⟨ · · · ⟩, li + 1, s[eval(ex, s)/x])
                ifnot(ex, l):if eval(ex, s) then imp(⟨ · · · ⟩, li + 1, s)
                                             else imp(⟨ · · · ⟩, l, s)
                    goto(l):imp(⟨ · · · ⟩, l, s)
              program(p′):imp(⟨ · · · ⟩, li + 1, imp(p′, firstlabel(p′), s)).
```

If the label of the next instruction is the terminating label, then return the data component at this time as the result of the interpreter. Otherwise the definition clause by clause is:

assign(x, ex): Modify the store as usual and move the instruction pointer to the next sequential instruction.

ifnot(ex, l): Evaluate ex and if true move the instruction pointer to the next sequential instruction, else set the instruction pointer to l.

goto(l): Set the instruction pointer to l.

program(p'): Run the interpreter on program p' with the label on the first statement as the instruction pointer and the current store. Then run the interpreter with the instruction pointer at the next sequential instruction with the modified store.

We now prove our first correctness result showing that the interpreter imp run with program component p, instruction pointer l and data component s will give the same final data component as taking the semantics of program p, looking up the label l and applying the value of l to s.

THEOREM. *For all* p, s *and* l *in* p, $\mathrm{imp}(p, l, s) \equiv \mathrm{sem}(p)(l)(s)$.

Proof. Proceed by computational induction on imp and sem. Clearly $\bot(p, l, s) \equiv \bot(p)(l)(s)$. Let $\langle l1\colon t1; l2\colon t2; \cdots; ln\colon tn; ln+1\rangle = p$ and $li = l$.

If $l = ln + 1$, then:

R.H.S.

$\equiv (\lambda s\,.\,s)(s)$

$\equiv s$

$\equiv$ L.H.S.

Otherwise, by cases of ti corresponding to li

assign(x, ex):

R.H.S.

$\equiv (\mathrm{sem}(p)(li+1) \cdot \lambda s\,.\,s[\mathrm{eval}(\mathrm{ex}, s)/x])(s)$ (def. of semstatement)

$\equiv \mathrm{sem}(p)(li+1)(s[\mathrm{eval}(\mathrm{ex}, s)/x])$

$\equiv \mathrm{imp}(p, li+1, s[\mathrm{eval}(\mathrm{ex}, s)/x])$ (induction hypothesis)

$\equiv$ L.H.S.

ifnot(ex, l):

R.H.S

$\equiv (\lambda s\,.\,\textbf{if}\ \mathrm{eval}(\mathrm{ex}, s)\ \textbf{then}\ \mathrm{sem}(p)(li+1)(s)\ \textbf{else}\ \mathrm{sem}(p)(l)(s))(s)$

(def. of semstatement)

$\equiv \textbf{if}\ \mathrm{eval}(\mathrm{ex}, s)\ \textbf{then}\ \mathrm{sem}(p)(li+1)(s)\ \textbf{else}\ \mathrm{sem}(p)(l)(s)$

$\equiv \textbf{if}\ \mathrm{eval}(\mathrm{ex}, s)\ \textbf{then}\ \mathrm{imp}(p, li+1, s)\ \textbf{else}\ \mathrm{imp}(p, l, s)$ (induction hypothesis)

$\equiv$ L.H.S.

goto(l):
R.H.S.
$\equiv \mathrm{sem}(p)(l)(s)$ (def. of semstatement)
$\equiv \mathrm{imp}(p, l, s)$ (induction hypothesis)
$\equiv$ L.H.S.

program(p'):
R.H.S.
$\equiv ((\mathrm{sem}(p)(li+1)) \cdot \mathrm{sem}(p')(\mathrm{firstlabel}(p')))(s)$ (def. of semstatement)
$\equiv \mathrm{sem}(p)(li+1)(\mathrm{sem}(p')(\mathrm{firstlabel}(p'))(s))$
$\equiv \mathrm{sem}(p)(li+1)(\mathrm{imp}(p', \mathrm{firstlabel}(p'), s))$ (induction hypothesis)
$\equiv \mathrm{imp}(p, li+1, \mathrm{imp}(p', \mathrm{firstlabel}(p'), s))$ (induction hypothesis)
$\equiv$ L.H.S.

5. A compiler and its correctness. The second implementation model is a compiler, called "compiler", for the structured language. The target language is the goto language. The compiler is straightforward and its formal definition follows.

```
compiler: STRUCPROGRAM → PROGRAM
compiler(p) =
               cases p:
       assign(x, ex):  l1: assign(x, ex)

       if(ex, p1, p2):  l1: ifnot(ex, l4);
                        l2: compiler(p1);
                        l3: goto(l5);
                        l4: compiler(p2);
                        l5

       while(ex, p1):   l1: ifnot(ex, l4);
                        l2: compiler(p1);
                        l3: goto(l1);
                        l4

   compound(p1, p2):    l1: compiler(p1);
                        l2: compiler(p2)
```

The compiler is simple enough that we need not describe it clause by clause. Note that the labels $l1$, $l2$, $l3$, $l4$ and $l5$ are distinct. These are all the distinct labels we need since the semantics of a statement like $l2$: compiler($p1$) takes all the labels in compiler($p1$) in isolation.

The correctness of the compiler, with respect to the semantics of the source and target language, follows. We show the semantics of the source program to be equivalent to the semantics of the target program applied to the first label in the target program.

THEOREM. *For all* p', $\mathrm{sem}'(p') \equiv \mathrm{sem}(\mathrm{compiler}(p'))(\mathrm{firstlabel}(\mathrm{compiler}(p')))$.

Proof. Proceed by computational induction over sem′ and compiler. Clearly $\perp(p') \equiv \mathrm{sem}(\perp)(\mathrm{firstlabel}(\perp))$. In what follows, let $p = \mathrm{compiler}(p')$.

Cases of p':

assign(x, ex):

R.H.S.

$\equiv \mathrm{sem}(l1:\mathrm{assign}(x, \mathrm{ex}))(l1)$ (def. of compiler)

$\equiv \bot[\mathrm{semstatement}(\mathrm{assign}(x, \mathrm{ex}))(\mathrm{sem}(p))(\lambda s.s)/l1](l1)$ (def. of sem)

$\equiv (\lambda s.\, s)\cdot(\lambda s.\, s[\mathrm{eval}(\mathrm{ex}, s)/x])$ (def. of semstatement and lookup $l1$)

$\equiv \lambda s.\, s[\mathrm{eval}(\mathrm{ex}, s)/x]$

$\equiv$L.H.S.

if(ex, $p1$, $p2$):

R.H.S.

$\equiv \mathrm{sem}(l1\colon \mathrm{ifnot}(\mathrm{ex}, l4);$
$l2\colon \mathrm{compiler}(p1);$
$l2\colon \mathrm{goto}(l5);$
$l4\colon \mathrm{compiler}(p2);$
$l5)(l1)$ (def. of compiler)

$\equiv (\bot[\mathrm{semstatement}(\mathrm{ifnot}(\mathrm{ex}, l4))(\mathrm{sem}(p))(\mathrm{sem}(p)(l2))/l1]$
$[\mathrm{semstatement}(\mathrm{compiler}(p1))(\mathrm{sem}(p))(\mathrm{sem}(p)(l3))/l2]$
$[\mathrm{semstatement}(\mathrm{goto}(l5))(\mathrm{sem}(p))(\mathrm{sem}(p)(l4))/l3]$
$[\mathrm{semstatement}(\mathrm{compiler}(p2))(\mathrm{sem}(p))(\mathrm{sem}(p)(l5))/l4]$
$[\lambda s.\, s/l5])(l1)$ (def. of sem)

$\equiv (\bot[\lambda s.\mathbf{if}\ \mathrm{eval}(\mathrm{ex}, s)\ \mathbf{then}\ \mathrm{sem}(p)(l2)(s)\ \mathbf{else}\ \mathrm{sem}(p)(l4)(s)/l1]$
$[(\mathrm{sem}(p)(l3))\cdot \mathrm{sem}(\mathrm{compiler}(p1))(\mathrm{firstlabel}(\mathrm{compiler}(p1)))/l2]$
$[\mathrm{sem}(p)(l5)/l3]$
$[(\mathrm{sem}(p)(l5))\cdot \mathrm{sem}(\mathrm{compiler}(p2))(\mathrm{firstlabel}(\mathrm{compiler}(p2)))/l4]$
$[\lambda s.s/l5])(l1)$ (def. of semstatement)

$\equiv (\bot[\lambda s.\mathbf{if}\ \mathrm{eval}(\mathrm{ex}, s)\ \mathbf{then}\ \mathrm{sem}(p)(l2)(s)\ \mathbf{else}\ \mathrm{sem}(p)(l4)(s)/l1]$
$[(\mathrm{sem}(p)(l3))\cdot \mathrm{sem}'(p1)/l2]$
$[\mathrm{sem}(p)(l5)/l3]$
$[(\mathrm{sem}(p)(l5))\cdot \mathrm{sem}'(p2)/l4]$
$[\lambda s.s/l5])(l1)$ (induction hypothesis)

$\equiv \lambda s.\mathbf{if}\ \mathrm{eval}(\mathrm{ex}, s)\ \mathbf{then}\ \mathrm{sem}(p)(l2)(s)\ \mathbf{else}\ \mathrm{sem}(p)(l4)(s)$ (lookup $l1$)

$\equiv \lambda s.\mathbf{if}\ \mathrm{eval}(\mathrm{ex}, s)\ \mathbf{then}\ ((\mathrm{sem}(p)(l3))\cdot \mathrm{sem}'(p1))(s)$
$\mathbf{else}\ ((sem(p)(l5))\cdot \mathrm{sem}'(p2))(s)$ (def. of sem(p))

$\equiv \lambda s.\ \mathbf{if}\ \mathrm{eval}(\mathrm{ex}, s)\ \mathbf{then}\ ((\lambda s.\, s)\cdot \mathrm{sem}'(p1))(s)$
$\mathbf{else}\ ((\mathrm{sem}(p)(l5))\cdot \mathrm{sem}'(p2))(s)$ (def. of sem(p))

$\equiv \lambda s.\mathbf{if}\ \mathrm{eval}(\mathrm{ex}, s)\ \mathbf{then}\ \mathrm{sem}'(p1)(s)\ \mathbf{else}\ \mathrm{sem}'(p2)(s)$

$\equiv$L.H.S.

while(ex, p):

R.H.S.

$\equiv \mathrm{sem}(l1\colon \mathrm{ifnot}(\mathrm{ex}, l4);$
$l2\colon \mathrm{compiler}(p1);$
$l3\colon \mathrm{goto}(l1);$
$l4)(l1)$ (def. of compiler)

$\equiv (\bot[\mathrm{semstatement}(\mathrm{ifnot}(\mathrm{ex}, l4))(\mathrm{sem}(p))(\mathrm{sem}(p)(l2))/l1]$
$[\mathrm{semstatement}(\mathrm{compiler}(p1))(\mathrm{sem}(p))(\mathrm{sem}(p)(l3))/l2]$
$[\mathrm{semstatement}(\mathrm{goto}(l1))(\mathrm{sem}(p))(\mathrm{sem}(p)(l4))/l3]$
$[\lambda s.s/l4])(l1)$ (def. of sem)

$\equiv (\bot[\lambda\, s.\ \mathbf{if}\ \mathrm{eval}(\mathrm{ex},s)\ \mathbf{then}\ \mathrm{sem}(p)(l2)(s)\ \mathbf{else}\ \mathrm{sem}(p)(l4)(s)/l1]$

$$
\begin{array}{ll}
\quad [(\mathrm{sem}(p)(l3)) \cdot \mathrm{sem}(\mathrm{compiler}(p1))(\mathrm{firstlabel}(\mathrm{compiler}(p1)))/l2] & \\
\quad [\mathrm{sem}(p)(l1)/l3] & \\
\quad [\lambda s.s/l4])(l1) & \text{(def. of semstatement)} \\
\equiv (\perp[\lambda s.\ \mathbf{if}\ \mathrm{eval}(\mathrm{ex}, s)\ \mathbf{then}\ \mathrm{sem}(p)(l2)(s)\ \mathbf{else}\ \mathrm{sem}(p)(l4)(s)/l1] & \\
\quad [(\mathrm{sem}(p)(l3)) \cdot \mathrm{sem}'(p1)/l2] & \\
\quad [\mathrm{sem}'(p)/l3] & \\
\quad [\lambda s.s/l4])(l1) & \text{(induction hypothesis)} \\
\equiv \lambda s.\ \mathbf{if}\ \mathrm{eval}(\mathrm{ex}, s)\ \mathbf{then}\ \mathrm{sem}(p)(l2)(s)\ \mathbf{else}\ \mathrm{sem}(p)(l4)(s) & \text{(lookup } l1) \\
\equiv \lambda \mathrm{s}.\mathbf{if}\ \mathrm{eval}(\mathrm{ex}, s)\ \mathbf{then}\ ((\mathrm{sem}(p)(l3)) \cdot \mathrm{sem}'(p1))(s) & \\
\qquad\qquad\qquad \mathbf{else}\ \lambda s.\ s(s) & \text{(def. of sem}(p)) \\
\equiv \lambda s.\mathbf{if}\ \mathrm{eval}(\mathrm{ex}, s)\ \mathbf{then}\ (\mathrm{sem}'(p) \cdot \mathrm{sem}'(p1))(s) & \\
\qquad\qquad\qquad \mathbf{else}\ s & \text{(def. of sem}(p)) \\
\equiv \text{L.H.S.} &
\end{array}
$$

compound$(p1, p2)$:

$$
\begin{array}{ll}
\text{R.H.S.} & \\
\equiv \mathrm{sem}(l1\text{: compiler}(p1); & \\
\qquad l2\text{: compiler}(p2))(l1) & \text{(def. of compiler)} \\
\equiv (\perp[\mathrm{semstatement}(\mathrm{compiler}(p1))(\mathrm{sem}(p))(\mathrm{sem}(p)(l2))/l1] & \\
\quad [\mathrm{semstatement}(\mathrm{compiler}(p2))(\mathrm{sem}(p)(\lambda s.s)/l2])(l1) & \text{(def. of sem)} \\
\equiv (\perp[(\mathrm{sem}(p)(l2) \cdot \mathrm{sem}(\mathrm{compiler}(p1))(\mathrm{firstlabel}(\mathrm{compiler}(p1))/l1] & \\
\quad [(\lambda s.s) \cdot \mathrm{sem}(\mathrm{compiler}(p2))(\mathrm{firstlabel}(\mathrm{compiler}(p2)))/l2])(l1) & \\
 & \text{(def. of semstatement)} \\
\equiv (\perp[(\mathrm{sem}(p)(l2)) \cdot \mathrm{sem}'(p1)/l1] & \\
\quad [(\lambda s.s) \cdot \mathrm{sem}'(p2)/l2])(l1) & \text{(induction hypothesis)} \\
\equiv (\mathrm{sem}(p)(l2)) \cdot \mathrm{sem}'(p1) & \text{(lookup } l1) \\
\equiv ((\lambda s.s) \cdot \mathrm{sem}'(p2)) \cdot \mathrm{sem}'(p1) & \text{(def. of sem}(p)) \\
\equiv \mathrm{sem}'(p2) \cdot \mathrm{sem}'(p1) & \\
\equiv \text{L.H.S.} &
\end{array}
$$

6. Some related work. Work on the correctness of implementations, particularly compilers, includes the early results of McCarthy and Painter [5]. Subsequent to this, more ambitious languages have been attempted by Morris [8] and Milner and Weyhrauch [7], among others. The approach taken in all these cases is to use some form of structural induction over the syntax of the languages involved.

In McCarthy and Painter the approach is straightforward structural induction. Morris imposes an algebraic structure on the sets of source and target language denotations. This structure mirrors the syntactic structure of the source and target languages. He then uses a unique extension theorem of algebra to show that equivalence on the primitive syntactic structures leads to equivalence over the entire set of programs.

Milner and Weyhrauch follow the same path as Morris but they use a formal system (LCF, based on the work of Scott) to express their theorem and to carry out the proof. Because the formal system is quite primitive the proof is extremely long (approximately 1600 proof steps).

The equivalence of alternative interpreters (operational models) for a given language has also been studied. The approach taken by the Vienna group—see

Jones and Lucas [3]—is the so-called twin machine technique. The two alternative interpreters are combined into one, thus eliminating redundancy but maintaining distinct state components for those aspects of the interpreters that differ. The equivalence of the interpreters is then expressed as a property of the state of the twin machine. This property is shown to be true of initial states and then inductively for all states that arise in the execution of the interpreter.

A slightly different approach is that of McGowan and Wegner [6]. They give a mapping from the states of each interpreter into some common set. It is then shown that the image under these mappings of all initial states of both interpreters are equal. Further, they show inductively that the images are equal at various points in the computation and thus of the final states.

7. Conclusions. We have defined a denotational semantics for two simple programming languages. One is a goto or machine-like language and the other a structured or higher level language. An interpreter (an operational semantics) for the goto language has been defined and formally proved to be equivalent to the denotational semantics. A compiler from the structured to the goto language has also been defined and proved to preserve the semantics of the program translated.

These proofs have shown the equivalence of the more familiar and, certainly in the case of the goto, more intuitive implementation models to the denotational models. We have thus strengthened the claim that the denotational models provide the semantics they are intended to.

We have also shown that Scott's theory provides a satisfactory framework to pose these questions of equivalence and to carry through the proofs. With a suitable choice of notation the definitions and proofs are readable and the proofs shorter than in some of the related work.

Finally, this paper is another small step towards reliable implementations of languages. This reliability is extremely important since the correctness of all programs written in a language depends on the correctness of the implementation of the language. Further, most machine languages use the goto statement. The correctness of the implementations of all the various languages on such a machine depends ultimately on the correctness of the hardware interpreter of the goto language and the compilers that have the goto language as their target code. Hence, the results about the goto statement are particularly relevant.

There are many problems raised by this work that remain to be solved. One of the most interesting is the problem of generating implementation models from a denotational model automatically, thus guaranteeing correctness.

REFERENCES

[1] J. W. DE BAKKER AND W. P. DE ROEVER, *A calculus for recursive program schemes*, Mathematisch Centrum Rep. MR 131/72, Amsterdam, 1972.

[2] R. M. BURSTALL, *Proving properties of programs by structural induction*, Comput. J., 12 (1969), pp. 41–48.

[3] C. B. JONES AND P. LUCAS, *Proving correctness of implementation techniques*, Symposium on Semantics of Algorithmic Languages, E. Engler, ed., Lecture Notes in Mathematics 188, Springer-Verlag, New York, 1971, pp. 178–211.

[4] Z. MANNA, S. NESS AND J. VUILLEMIN, *Inductive methods for proving properties of programs*, Comm. ACM, 16 (1973), pp. 491–502.

[5] J. McCarthy and J. Painter, *Correctness of a compiler for arithmetic expressions*, Proc. Symp. of Appl. Math., American Mathematics Society, Providence, RI, 1967, pp. 33–41.

[6] C. McGowan and P. Wegner, *The equivalence of sequential and associative information structure models*, Proceedings of a Symposium on Data Structures in Programming Languages, SIGPLAN Notices 6, J. T. Tou and P. Wegner, eds., 1971, pp. 191–216.

[7] R. Milner and R. Weyhrauch, *Proving compiler correctness in a mechanized logic*, Machine Intelligence 7, B. Meltzer and D. Mitchie, eds., Edinburgh Press, Edinburgh, 1972, pp. 51–70.

[8] F. L. Morris, *Advice on structuring compilers and proving them correct*, Conf. Rec. of ACM Symp. on Principles of Programming Languages (Boston, 1973), pp. 144–152.

[9] J. C. Reynolds, *Notes on a lattice-theoretic approach to the theory of computation*, Dept. of Systems and Information Sci., Syracuse Univ., Syracuse, NY, 1972.

[10] D. Scott, *Outline of a mathematical theory of computation*, Proc. 4th Ann. Princeton Conf. on Information Sci. and Systems (Princeton, NJ, 1970), pp. 169–176.

[11] ———, *Mathematical concepts in programming language semantics*, AFIPS Conference Proceedings, Vol. 40, 1972, pp. 225–234.

[12] ———, *Data types as lattices*, Lecture notes, unpublished, Amsterdam, 1972.

[13] D. Scott and C. Strachey, *Towards a mathematical semantics for computer languages*, Computers and Automata, J. Fox, ed., John Wiley, New York, 1972, pp. 19–46.

[14] C. Strachey and C. P. Wadsworth, *Continuations, mathematical semantics for handling full jumps*, Oxford Univ. Tech. Monograph PRG–11, Oxford, England, 1974.

SIAM J. COMPUT.
Vol. 6, No. 3, September 1977

TWO-PROCESSOR SCHEDULING WITH START-TIMES AND DEADLINES*

M. R. GAREY AND D. S. JOHNSON†

Abstract. Given a set $\mathcal{T}=\{T_1, T_2, \cdots, T_n\}$ of tasks, each T_i having execution time 1, an integer start-time $s_i \geqq 0$ and a deadline $d_i > 0$, along with precedence constraints among the tasks, we examine the problem of determining whether there exists a schedule on two identical processors that executes each task in the time interval between its start-time and deadline. We present an $O(n^3)$ algorithm that constructs such a schedule whenever one exists. The algorithm may also be used in a binary search mode to find the shortest such schedule or to find a schedule that minimizes maximum "tardiness". A number of natural extensions of this problem are seen to be *NP*-complete and hence probably intractable.

Key words. multiprocessing systems, scheduling algorithms, *NP*-complete problems

1. Introduction. Since publication of the book *Theory of Scheduling* [4] by Conway, Maxwell, and Miller in 1967, considerable progress has been made in the mathematical analysis of abstract multiprocessing systems. One combinatorial model which is central to much of this work consists of a number m of identical, independent *processors*, a finite set $\mathcal{T}=\{T_1, T_2, \cdots, T_n\}$ of *tasks* to be executed, an *execution time* $\tau_i > 0$ for each $T_i \in \mathcal{T}$, and a *partial order* $<$ on $\mathcal{T}$. The partial order describes precedence constraints between the tasks, restricting allowable schedules to those in which, whenever $T_i < T_j$, the task T_j does not begin executing until T_i has been completed. The primary goal of scheduling is usually to minimize either the mean-time-in-system (mean flow time) or the maximum-time-in-system (maximum finishing time). Unfortunately these goals can be quite difficult to achieve and, in fact, most classes of scheduling problems appear to be computationally intractable [8], [10], [12], [15]. A notable exception to this state of affairs is the case of minimizing maximum finishing time when $m = 2$ and each $\tau_i = 1$, $1 \leqq i \leqq n$. Efficient scheduling algorithms for this case have been described in [3], [6], [7], [13]. These results have been extended in [9], which presents an efficient scheduling algorithm for the more complicated version of this problem in which each task $T_i \in \mathcal{T}$ also has a *deadline* $d_i > 0$ by which time its execution must be completed. In this paper we further extend these results to the situation in which each task $T_i \in \mathcal{T}$ has not only a deadline $d_i > 0$ but also an integer *start-time* s_i, $0 \leqq s_i \leqq d_i$, such that T_i must be executed entirely in the time interval $[s_i, d_i]$. We describe an $O(n^3)$ algorithm which determines whether there exists a schedule meeting all start-time, deadline, and precedence constraints and which constructs such a schedule if one exists.

In § 2 of this paper, we will describe the basic ideas behind the algorithm and show why it works. In § 3 we provide the details as to how it can be implemented to run in time $O(n^3)$. In § 4 we show how this basic feasibility algorithm can be used in iterative procedures to find schedules that minimize maximum finishing time or maximum tardiness. We also briefly examine the computational complexity of some related problems.

* Received by the editors January 26, 1976, and in revised form June 9, 1976.

† Bell Laboratories, Murray Hill, New Jersey 07974.

We conclude this section with a few preliminary definitions. For the sake of generality, we state them in terms of arbitrary m, $\{\tau_i\}$, and $\{s_i\}$.

A task T_i is called a *predecessor* of task T_j (and T_j is a *successor* of T_i) whenever there exists a sequence of tasks $T'_1, T'_2, \cdots, T'_k$, $k \geqq 1$, such that $T_i < T'_1 < T'_2 < \cdots < T'_k = T_j$. Given m, $\mathcal{T}$, $\{\tau_i\}$, $\{s_i\}$, $\{d_i\}$, and the partial order $<$, a *valid schedule* is a total function $\sigma : \mathcal{T} \to [0, \infty)$ which satisfies the following three properties:

(i) For all $t \in [0, \infty)$, $|\{T_i \in \mathcal{T} : \sigma(T_i) \leqq t < \sigma(T_i) + \tau_i\}| \leqq m$;
(ii) Whenever $T_i < T_j$, $\sigma(T_i) + \tau_i \leqq \sigma(T_j)$;
(iii) For each $T_i \in \mathcal{T}$, $\sigma(T_i) \geqq s_i$.

In plain language, the function σ assigns a starting time for execution to each task in $\mathcal{T}$, property (i) states that no more than m tasks are ever executed simultaneously, property (ii) ensures that the partial order is respected, and property (iii) ensures that the start-time constraints are not violated (although the deadlines may be). Note that the processing is assumed to be *nonpreemptive* in that once a task is initiated it continues executing until its completion. A valid schedule σ is said to *meet all the deadlines* if, for each $T_i \in \mathcal{T}$, $\sigma(T_i) + \tau_i \leqq d_i$. The *finishing time* ω for a valid schedule σ is given by $\omega = \max\{\sigma(T_i) + \tau_i : T_i \in \mathcal{T}\}$.

Unless stated otherwise, we shall assume henceforth that $m = 2$, each $\tau_i = 1$, and each s_i is an integer. Notice that, when minimizing maximum finishing time under these assumptions, there is no loss of generality in restricting consideration to schedules σ which map $\mathcal{T}$ into the nonnegative integers. Thus we may assume also that each deadline d_i is a positive integer.

2. The basic scheduling algorithm. In this section we describe the basic ideas behind our algorithm, which finds a valid schedule meeting all deadlines whenever one exists. The algorithm is similar to that of [9] in that it may be thought of as finding a priority list for directing the scheduling process. A *priority list L* is a permutation of the tasks in $\mathcal{T}$ which is used to define a valid schedule f in the following intuitive manner: Initially, all processors are idle. At any time t at which a processor is idle, the processor instantaneously scans L from the beginning and selects the first task T_k (if any) which may validly be executed, i.e., $s_k \leqq t$, all predecessors of T_k have been completed, and T_k itself has not yet been started. In case of a tie, T_k is assigned to the idle processor with lowest index and the remaining processors continue scanning the list. In our formal notation $\sigma(T_k)$ is set equal to that time t at which T_k is selected for execution by one of the processors. The reader should have no difficulty in specifying an $O(n^2)$ algorithm for computing σ from the list L.

Our algorithm will determine a specific priority list L for scheduling the tasks by this method. As in [9] the key idea involves a special modification of the task deadlines, having the property that a valid schedule meets all the modified deadlines if and only if it meets all the original deadlines. However the addition of task start-time constraints considerably complicates the necessary deadline modifications.

In order to state the lemma on which our deadline modifications are based, we require a few preliminary definitions. For any task T_i and integers s, d

satisfying $s_i \leqq s \leqq d_i \leqq d$, we define $S(i, s, d)$ to be the set of all tasks T_j $(j \neq i)$ which have $d_j \leqq d$ and either are successors of T_i or have $s_j \geqq s$. Let $N(i, s, d)$ denote the number of tasks in $S(i, s, d)$. We use $\lceil x \rceil$ to denote the least integer no less than x.

LEMMA 1. *For any task T_i and integers s, d satisfying $s_i \leqq s \leqq d_i \leqq d$, if $N(i, s, d) \geqq 2(d-s)$, then T_i must be completed by time $d - \lceil N(i, s, d)/2 \rceil$ in any valid schedule that meets all task deadlines.*

Proof. Suppose $N(i, s, d) \geqq 2(d-s)$ and let σ be any valid schedule that meets all task deadlines. We divide the proof into two cases, depending on whether $N(i, s, d)$ equals or exceeds $2(d-s)$.

First suppose $N(i, s, d) > 2(d-s)$. Since all tasks in $S(i, s, d)$ must be completed by time d and there are only two processors, operating nonpreemptively, there must be some task $T_j \in S(i, s, d)$ for which $\sigma(T_j) \leqq d - \lceil N(i, s, d)/2 \rceil < s$. Since $\sigma(T_j) < s$, the definition of $S(i, s, d)$ implies that T_j must be a successor of T_i. Hence T_i must be completed when T_j starts at time $\sigma(T_j)$ and the desired result follows.

Now suppose $N(i, s, d) = 2(d-s)$. Then we have $\lceil (N(i, s, d)+1)/2 \rceil = \lceil N(i, s, d)/2 \rceil + 1$. We parallel the previous argument using the set $S = S(i, s, d) \cup \{T_i\}$ instead of $S(i, s, d)$. Since all tasks in S must be completed by time d and there are only two processors, operating nonpreemptively, there must be some task $T_j \in S$ for which $\sigma(T_j) \leqq d - \lceil (N(i, s, d)+1)/2 \rceil = s-1$. Since $\sigma(T_j) < s$, the definition of $S(i, s, d)$ implies that either $T_j = T_i$ or T_j is a successor of T_i. In either case we have

$$\sigma(T_i) + 1 \leqq \sigma(T_j) + 1 \leqq s = d - \lceil N(i, s, d)/2 \rceil$$

as desired. □

The significance of Lemma 1 is that, when the described conditions are met, it gives an additional constraint on the latest possible finishing time for T_i. Thus, if $N(i, s, d) \geqq 2(d-s)$ and $d - \lceil N(i, s, d)/2 \rceil < d_i$, we may set d_i equal to $d - \lceil N(i, s, d)/2 \rceil$ without foreclosing any possible valid schedules that meet all task deadlines. That is, a valid schedule meets all the original task deadlines if and only if it meets the new set of deadlines obtained by making this single change to d_i. In fact, such modifications may be performed repeatedly until either no further modifications are possible or we have some $d_i < s_i + 1$, in which case no valid schedule can possibly meet all the deadlines. In a later section we shall describe an algorithm for performing these successive modifications in an organized and efficient manner.

Motivated by the preceding discussion, we will call the deadlines *internally consistent* whenever the following conditions hold for every task $T_i \in \mathcal{T}$:

1) $d_i \geqq s_i + 1$;
2) For every pair of integers s, d satisfying $s_i \leqq s \leqq d_i \leqq d$, if $N(i, s, d) \geqq 2(d-s)$, then $d_i \leqq d - \lceil N(i, s, d)/2 \rceil$.

Two basic facts which follow from internal consistency will prove useful.

FACT 1. *If the deadlines are internally consistent, then $T_i < T_j$ implies $d_i < d_j$.*

Proof. Suppose we had both $T_i < T_j$ and $d_i \geqq d_j$. Then T_j belongs to $S(i, d_i, d_i)$ and hence $N(i, d_i, d_i) > 2(d_i - d_i) = 0$. Property 2) of internal consistency then requires that we have $d_i \leqq d_i - \lceil N(i, d_i, d_i)/2 \rceil \leqq d_i - 1$, a contradiction. □

FACT 2. *If the deadlines are internally consistent, then $s \leqq d$ implies*

$$|\{T_i \in \mathcal{T}: s \leqq s_i \text{ and } d_i \leqq d\}| \leqq 2(d-s).$$

Proof. Suppose for some $s \leqq d$ the set $S=\{T_i \in \mathcal{T}: s \leqq s_i$ and $d_i \leqq d\}$ had $|S|>2(d-s)$. Let T_k be a task in S having the earliest start-time. Then $S-\{T_k\} \subseteq S(k, s_k, d)$ and hence $N(k, s_k, d) \geqq 2(d-s) \geqq 2(d-s_k)$. Property 2) of internal consistency then requires that

$$d_k \leqq d - \lceil N(k, s_k, d)/2 \rceil \leqq s.$$

But since $s_k \geqq s$, this implies that $s_k \geqq d_k$, contradicting property 1) of internal consistency. □

We now are prepared to state our main result.

THEOREM 1. *Let $L=(T_1, T_2, \cdots, T_n)$ be any priority list such that $d_i \leqq d_{i+1}$ for $1 \leqq i \leqq n-1$. If the deadlines are internally consistent, then the valid schedule defined by L meets all task deadlines.*

Proof. Suppose that the valid schedule σ constructed from L fails to meet the task deadlines. Since each $t_i=1$ and all start-times and deadlines are integers, σ assigns an integer starting time to each task. Let T_j be a task with minimum $\sigma(T_j)$ which fails to meet its deadline, i.e., $\sigma(T_j)+1>d_j$ and hence by integrality $\sigma(T_j) \geqq d_j$. Let s be the greatest integer time, $0 \leqq s \leqq \sigma(T_j)$, for which the set $P(s)=\{T_i \in \mathcal{T}: \sigma(T_i)=s-1$ and $d_i \leqq d_j\}$ satisfies $|P(s)|<2$. Defining $S=\{T_i \in \mathcal{T}: s \leqq \sigma(T_i)<\sigma(T_j)\} \cup \{T_j\}$, we observe that $|S|=2(\sigma(T_j)-s)+1$ and each $T_i \in S$ has $d_i \leqq d_j$. We divide the proof into two cases depending on whether $|P(s)|=0$ or $|P(s)|=1$.

First, suppose $|P(s)|=0$. By the definition of S, any tasks scheduled to begin execution at time $s-1$ must have followed all the tasks in S on the priority list L. Thus none of the tasks in S were ready to begin execution at time $s-1$, and any tasks that *were* executed at that time could not have been predecessors of any task in S (by Fact 1 about internally consistent deadlines). Therefore every task in S must have start-time exceeding $s-1$ and hence, by integrality, has start-time s or greater. (All these observations hold trivially if $s=0$). This implies that

$$S \subseteq \{T_i \in \mathcal{T}: s \leqq s_i \text{ and } d_i \leqq d_j\},$$

from which it follows that

$$\begin{aligned}|\{T_i \in \mathcal{T}: s \leqq s_i \text{ and } d_i \leqq d_j\}| &\geqq |S| \\ &= 2(\sigma(T_j)-s)+1 \geqq 2(d_j-s)+1 \\ &> 2(d_j-s).\end{aligned}$$

This last inequality contradicts Fact 2, proving the desired result when $|P(s)|=0$.

Now suppose $|P(s)|=1$. Let T_k be the single task in $P(s)$. Again, any other task scheduled to begin execution at time $s-1$ must have followed all tasks in S on the priority list L, and hence was not a predecessor of any task in S, while no task in S was ready to begin executing at time $s-1$. Therefore every task in S either has start-time at least s or is a successor of T_k. This implies that $S \subseteq S(k, s, d_j)$, from which it follows that

$$N(k, s, d_j) \geqq |S| = 2(\sigma(T_j) - s) + 1$$
$$\geqq 2(d_j - s) + 1 > 2(d_j - s).$$

From property (2) of internal consistency we then must have

$$d_k \leqq d_j - \lceil N(k, s, d_j)/2 \rceil \leqq d_j - (d_j - s + 1) = s - 1.$$

Therefore T_k failed to meet its deadline, contradicting the choice of T_j as the earliest such task. This proves the desired result when $|P(s)| = 1$, completing the proof. □

We now summarize our basic algorithm, which determines whether a valid schedule meeting all deadlines exists and, if so, constructs one.

Step 1. Successively modify the deadlines using Lemma 1 until either (a) the deadlines are internally consistent or (b) some $s_i \geqq d_i$. In case (b), report that no schedule exists and halt.

Step 2. Form the priority list L by sorting the tasks in order of nondecreasing modified deadlines.

Step 3. Compute the valid schedule defined by priority list L. By Theorem 1 it meets all the task deadlines. □

By our previous comments on the complexity of Step 3, and the fact that all Step 2 involves is a simple sorting process, we see that the number of operations required by this algorithm is $O(n^2)$ plus the number of operations required for Step 1. In the next section we describe a method for performing Step 1 using $O(n^3)$ operations.

3. A deadline modification algorithm. In this section we describe an algorithm for successively modifying the task deadlines using Lemma 1. A straightforward approach to doing this would repeatedly examine $N(i, s, d)$ for all appropriate values of i, s, and d, modifying deadlines when required, until either the deadlines are internally consistent or some task deadline no longer exceeds the corresponding start-time. Though this algorithm will certainly terminate with the correct result, its computation time is potentially rather large. One reason for this is that the same $N(i, s, d)$ may have to be considered many times, since modifying task deadlines can change previously examined $N(i, s, d)$ values. A second reason is that the number of integer values for s and d which must be considered may be quite large. We now describe how a more careful approach avoids these difficulties.

Our algorithm is structured as three nested loops, each selecting successive values of one of the three parameters i, s, and d. The outer loop selects values of d in decreasing order. For each d, the next loop selects values of i in increasing order, skipping those values of i for which $d_i > d$. Finally, for fixed i and d, the inner loop selects appropriate values of s in increasing order and modifies d_i if required by the value of $N(i, s, d)$.

We first observe that this loop structure for choosing i, s, and d allows each triple to be considered only once. This is, of course, desirable, but can be justified only if it insures that no possible deadline modification is missed. To see that this is the case, we first observe that the value of $N(i, s, d)$ and the new deadline that may be imposed on task T_i if $N(i, s, d) \geqq 2(d-s)$ do not depend on the value of d_i itself. Thus as long as the value of $N(i, s, d)$ remains unchanged, extra considerations of the triple (i, s, d) cannot lead to modifications that were not made the first time that triple was considered. Next, we observe that the value of $N(i, s, d)$ changes only when a task T_j with deadline $d_j > d$ has its deadline modified to be less than or equal to d. However, d_j can be so modified only when examining $N(j, s', d')$ for some $d' \geqq d_j > d$. Considering values of d in decreasing order insures that all such modifications affecting the value of $N(i, s, d)$ have been made before $N(i, s, d)$ is examined. Thus we never need to consider a triple (i, s, d) more than once when using our loop structure.

Our next step is to bound the *number* of triples considered. If our algorithm is to be $O(n^3)$, we clearly can consider only $O(n^3)$ such triples, but at present we only have a bound of (number of d's considered) $\cdot$ (number of i's considered for each d) $\cdot$ (number of s's considered for each i and d). The middle factor is at most n, since there are only n tasks, but the remaining factors might be considerably larger, since individual tasks might originally have start-times and/or deadlines which are much larger than n.

We first examine the parameter s, and show that, for fixed i and d, the inner loop need never consider more than $n+1$ possible values for s. In particular, we show that the only values for s that need to be considered are those integers s, $s_i \leqq s \leqq d_i$, which are task start-times or d_i itself. Suppose there is some s not of this form for which $N(i, s, d)$ requires that d_i be modified. Letting s' be the minimum of d_i and the least start-time exceeding s, we have $N(i, s', d) = N(i, s, d) \geqq 2(d-s) > 2(d-s')$. Thus the same modification forced by $N(i, s, d)$ will be forced by $N(i, s', d)$ and s' belongs to our more restricted set of choices. Furthermore, once d_i has been modified, no more choices for s need be considered for these fixed values of i and d. To see this, let s^* denote the least value of s such that $N(i, s^*, d)$ forces d_i to be modified. Since the modified value of d_i must be less than or equal to s^* and since values for s were selected in increasing order, it follows that all values for s which remain relevant (s such that $s_i \leqq s \leqq s^*$) have already been considered. Thus, for fixed i and d, at most $n+1$ values for s need be considered and (a remark for future reference) at most one deadline modification occurs.

To show that at most n values of d need be considered in the outer loop involves a more complicated argument. The basic idea is that essentially every value of d considered will be guaranteed to be a final deadline for some one of the n tasks. In order to insure this, however, we need to be a bit more careful in the algorithm than we have indicated so far.

First, it will be useful to have the property that whenever T_j is a predecessor of T_i, $d_j \leqq d_i$. Note that reducing d_j so that this is the case will never preclude any possible valid schedules that meet the original deadlines. (In fact we could even reduce d_j to $d_i - 1$, but the weaker condition is sufficient for our purposes and is easier to maintain.) Some preprocessing is required to make this property hold initially; we postpone a detailed explanation of how this is done. Then, whenever a deadline d_i is modified in the algorithm, we merely need to examine each predecessor T_j of T_i and set d_j equal to the smaller of its current value and the new value for d_i. This ensures that the desired property will hold throughout execution of the algorithm. (Note also that these extra modifications do not affect the property that permitted us to consider each $N(i, s, d)$ at most once, namely, that no deadline is ever modified when considering a value of d less than that deadline.)

Having the abovementioned property, we always select the next value of d to be the largest current task deadline which is less than all previously selected values of d. This immediately insures that no required modifications will be overlooked, since if d' is any integer between d and the previous value for d, we have $N(i, s, d) = N(i, s, d')$ for all appropriate i and s. Now suppose that, after considering all values of i and s for some d, no task remains with deadline d. We show that this implies no valid schedule can possibly meet all the deadlines. Let j be the largest task index such that, when d was selected, T_j had deadline d and had no successors with deadline d (at least one such task must exist). By our property for deadlines, T_j also has no successor with deadline less than d. Thus $S(j, s, d)$ contains no successors of T_j, $s_j \leqq s \leqq d$. Since d_j was modified, there must have been some s such that $N(j, s, d) \geqq 2(d-s)$. By our choice of T_j and the above remark, when $N(j, s, d)$ was examined, every task in $S(j, s, d)$ had start-time s or larger and either had deadline less than or equal to $d-1$ or else was a predecessor of T_j with deadline d. Thus all tasks in $S(j, s, d)$ must in fact be executed in the time interval $[s, d-1]$ in any valid schedule that meets all the deadlines. However, by choice of s, we have $N(j, s, d) \geqq 2(d-s) > 2(d-1-s)$, and so $S(j, s, d)$ contains more tasks than can *possibly* be executed in the time interval $[s, d-1]$. Hence no valid schedule can meet all the deadlines. Therefore, if we finish processing a deadline d and have no task left with that deadline, we know that no such valid schedule can exist. Hence, if we terminate whenever this situation arises, we will not be overlooking any modification which can lead to a set of internally consistent deadlines. Moreover, by terminating in this fashion, we insure that at most n values of d need be considered.

The above arguments show that at most $O(n^3)$ triples (i, s, d) need be considered. It remains to be shown that the various costs associated with preprocessing and with updating the values of i, s, d and $N(i, s, d)$ are also $O(n^3)$. In order to do this, we must provide more specific details on how the algorithm is to be implemented.

First we describe and discuss the preprocessing that must be done. The first step is to sort and re-index the tasks so that $s_1 \leqq s_2 \leqq \cdots \leqq s_n$. This takes time $O(n \cdot \log n)$. Next we compute the transitive closure of the partial order so that, in constant time, we can determine whether or not T_i precedes T_j, $1 \leqq i, j \leqq n$. This can be accomplished with $O(n^{2.81})$ operations using [5] or $O(n^3)$ operations using any of [2], [14], [16]. Then we perform the preliminary deadline modifications to ensure that $d_i \leqq d_j$ whenever T_i precedes T_j. This can be done in $O(n^2)$ operations by working "backwards" in the partial order, examining a task T_i only after examining all its successors and then setting d_i to the minimum value in $\{d_i\} \cup \{d_j : T_j \text{ is a successor of } T_i\}$. Finally we initialize the variable d to a value that exceeds the largest task deadline. The algorithm then proceeds as follows:

Step 1. If any task T_i has $d_i \leqq s_i$, halt (no schedule is possible). If no task has a deadline less than d, halt (the current deadlines are internally consistent). Otherwise set d to the largest task deadline less than d and set i to the least task index for which d_i is less than or equal to the new value of d.

Step 2. Scan the task list to compute $N(i, s_i, d)$. Set COUNT$\leftarrow N(i, s_i, d)$, $s \leftarrow s_i$, and set $k \leftarrow$ least j such that $s_j = s_i$.

Step 3. If COUNT$\geqq 2(d-s)$ and $d_i > d - \lceil \text{COUNT}/2 \rceil$, set $d_i \leftarrow d - \lceil \text{COUNT}/2 \rceil$ and, for each predecessor T_j of T_i whose deadline exceeds the new d_i, set $d_j \leftarrow d_i$.

Step 4. If $s \geqq d_i$, go to Step 5. Otherwise increment k by 1 until either $k > n$ or $s_k > s$. During this scan, subtract 1 from COUNT for each T_j (original $k \leqq j <$ new k) which is not a successor of T_i and which satisfies $d_j \leqq d$, $s_j = s$, and $j \neq i$. If $k = n+1$ or $s_k > d_i$, set $s \leftarrow d_i$ and go to Step 3. Otherwise set $s \leftarrow s_k$ and go to Step 3.

Step 5. Find the least $j > i$ such that $d_j \leqq d$. If such a j exists, set $i \leftarrow j$ and go to Step 2. If no such j exists and some task has current deadline d, go to Step 1. Otherwise halt (no schedule is possible).

Step 1 checks two termination conditions, both of which can be verified in $O(n)$ operations by a simple scan through all the tasks. If both conditions fail, it then selects the next value for d and the first relevant value of i for d, again accomplished easily by simple scans in $O(n)$ operations. Since, by our previous comments, at most n values of d will be selected, this step will be entered at most $n+1$ times, for a total contribution of at most $O(n^2)$ operations.

Step 2 computes $N(i, s_i, d)$, stores it in COUNT, initializes s to s_i, and initializes the variable k which will be used in updating COUNT as s changes. All of these can again be accomplished by simply scanning through all n tasks, with a constant number of operations for each task (for instance, to determine whether it meets the membership conditions for $S(i, s_i, d)$). Thus each entry of this step uses $O(n)$ operations. Because it is entered at most n times for each d, always with a new value for i, the total contribution to the algorithm is at most $O(n^3)$ operations.

Step 3 checks the internal consistency conditions for $N(i, s, d) = \text{COUNT}$ and, if necessary, modifies the appropriate deadlines. This step is entered at most $O(n^3)$ times, once for each choice of the three parameters i, s, and d. It requires only constant time unless deadline modification is necessary, in which case $O(n)$ operations may be required. However, since d_i is modified at most once for a particular value of d, those $O(n)$ modification operations are required at most n^2 times. Thus this step contributes a total of at most $O(n^3)$ operations.

Step 4 first checks whether all relevant values of s have been considered for the current i (i.e., either d_i has been modified or the last s was equal to d_i). If not, it continues scanning the task list from T_k to find the next relevant s. During this scan it continually updates COUNT, subtracting 1 for each task which belonged to the previous $S(i, s, d)$ but not the new $S(i, s, d)$. After resetting s, it returns to Step 3. This step is entered at most $O(n^3)$ times since it is entered only through Step 3. However, by using the variable k to resume scanning the task list from where it left off, it only scans the task list once for each choice of i and d. It follows that Step 4 contributes a total of at most $O(n^3)$ operations to the algorithm.

Finally, Step 5 selects the next relevant value of i, if any, and returns to Step 2. If all values of i for this d have been checked and some task deadline remained equal to d, it goes to Step 1 to determine the next d. Otherwise, by our previous discussion, we know that no valid schedule can possibly meet all the deadlines and the algorithm halts. Since Step 5 is entered at most once for each choice of i and d, it is entered a total of at most $O(n^2)$ times. The scanning of the tasks is a process which occurs only once for each d, although it is interrupted each time i is updated. Thus this step contributes a total of at most $O(n^2)$ operations.

From our discussion preceding the algorithm, the reader should have little difficulty in verifying that the algorithm works properly. It terminates either with a set of internally consistent deadlines equivalent to the original deadlines, or with the conclusion that no valid schedule can meet all the deadlines. Furthermore, since each step contributes at most $O(n^3)$ operations to the total procedure, and all required preprocessing can be done in at most $O(n^3)$ operations, the algorithm takes total time $O(n^3)$ as claimed. Although we know of no method for doing this faster than proportional to n^3, we note that the algorithm as described could probably be improved by a constant factor in a careful implementation. We chose not to incorporate such details into the description of our algorithm since that would have served primarily to further complicate an already complicated algorithm, without substantially improving its performance.

4. Conclusion. The algorithm described in §§ 2 and 3 is designed only to test for feasible schedules and to generate such a schedule whenever one exists. Given that feasible schedules exist, however, one might wish to find such schedules that, for example, minimize maximum finishing time. This can be done using our algorithm in a simple iterative procedure as follows.

First observe that, for any integer D, we can decide whether there exists a valid schedule that meets all deadlines and has maximum finishing time at most D, by applying our algorithm after setting equal to D all deadlines that exceed D. The least possible maximum finishing time can then be found using a binary search on D. Since the only values of D that need be considered are those integers that exceed the largest start-time by no more than n, this involves only $O(\log n)$ applications of our basic algorithm, and the complete procedure requires $O(n^3 \log n)$ operations. In case all start-times or all deadlines are the same, the simpler procedure described in [9] can be used.

A similar approach, using binary search, can be used to find a valid schedule that minimizes maximum tardiness (the *tardiness* of a task in a schedule is the maximum of zero and its finishing time minus its deadline), in case no valid schedule meeting all the deadlines is possible. To check whether a valid schedule exists with maximum tardiness D or less, merely replace each deadline d_i by $D+d_i$ and apply our basic algorithm.

Unfortunately, the problem of minimizing the *number* of tardy tasks, even if we have only one processor and all start-times (or all deadlines) are the same, is *NP*-complete [9] and hence probably computationally intractable (see [1], [11], [12] for comprehensive treatments of "*NP*-completeness"). A number of other simple generalizations of our scheduling problem are also *NP*-complete.

First let us consider relaxing the constraint that all $\tau_i = 1$ by allowing $\tau_i \in \{1, 2\}$. In this case, with all $s_i = 0$ and all $d_i = D$, Ullman [15] has shown that the problem of deciding whether there exists a valid schedule meeting all deadlines is

NP-complete. If we further relax the constraint on task times to allow them to be arbitrary integers, then the problem of deciding whether there exists a valid schedule meeting all deadlines is *NP*-complete even for one processor and no precedence constraints. This result has not appeared previously but can be proved easily from the *NP*-complete 3-PARTITION problem [8], [10]. The 3-PARTITION problem is, given $3n$ integers $a_1, a_2, \cdots, a_{3n}$ and an integer B such that each a_i satisfies $B/4 < a_i < B/2$, to determine whether the $\{a_i\}$ can be partitioned into n 3-element sets which each sum exactly to B. It follows, as in [10], that this scheduling problem is *NP*-complete even if input size is measured by the sum of the task execution times, rather than the number of bits required to encode them. This means that any algorithm that always finds the desired schedule, if it exists, will probably require time exponential in the sum of execution times rather than just exponential in the number of tasks, a significant difference when tasks with large execution times are present.

Returning now to our original problem, suppose we relax the constraint that all start-times must be integers. (Allowing arbitrary rational start-times is equivalent to allowing all tasks to have arbitrary, but identical, integer execution times). In this case little is yet known. In fact, the complexity of determining the existence of valid schedules meeting all deadlines for the case of one processor, unit execution times, no precedence constraints, and arbitrary start-times and deadlines is still open. (In the very special case with all s_i multiples of $\frac{1}{2}$, we *can* give a polynomial-time algorithm).

Returning again to our original problem with integer start-times (and hence integer deadlines), consider the effect of increasing the number of processors. If the number of processors is arbitrary, the problem is *NP*-complete even with all $s_i = 0$ and all $d_i = D$ [15]. It is not known whether this problem is *NP*-complete for any *fixed* number m of processors, although $m = 2$ is the largest value for which a polynomial time algorithm is known. Our algorithm *can* be used to solve a significant special case of the 3-processor problem (with all $s_i = 0$ and all $d_i = D$) in which we ask whether there exists a valid schedule with maximum finishing time D, where D is the length of the longest chain ("critical path") in the partial order. Choosing one such chain, there is no loss of generality in assigning all tasks in the chain, in order, to the third processor. Then all remaining tasks must be executed on the remaining two processors, with the precedence constraints between these tasks and the tasks on the third processor serving merely to impose individual integer start-times and deadlines on the remaining tasks. Hence we are reduced to precisely the problem our algorithm was designed to solve. The solvability of this special case is particularly interesting because the authors (and others) have made numerous attempts to prove the general 3-processor problem *NP*-complete by proving that this special case was *NP*-complete. If, as is generally believed, the *NP*-complete problems are intractable, this particular approach was doomed to fail.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974, Chap. 10.

[2] 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.

[3] E. G. COFFMAN AND R. L. GRAHAM, *Optimal scheduling for two-processor systems*, Acta. Informatica, 1 (1972), pp. 200–213.

[4] R. W. CONWAY, W. L. MAXWELL AND L. W. MILLER, *Theory of Scheduling*, Addison-Wesley, Reading, MA, 1967.

[5] M. J. FISCHER AND A. R. MEYER, *Boolean matrix multiplication and transitive closure*, 12th Ann. IEEE Symp. on Switching and Automata Theory, East Lansing, MI, 1971, pp. 129–131.

[6] M. FUJII, T. KASAMI AND K. NINOMIYA, *Optimal sequencing on two equivalent processors*, SIAM J. Appl. Math., 17 (1969), pp. 784–789.

[7] ———, *Erratum*, Ibid., 20 (1971), p. 141.

[8] M. R. GAREY AND D. S. JOHNSON, *Complexity results for multiprocessor scheduling under resource constraints*, this Journal, 4 (1975), pp. 397–411.

[9] ———, *Scheduling tasks with nonuniform deadlines on two processors*, J. Assoc. Comput. Mach., 23 (1976), pp. 461–467.

[10] M. R. GAREY, D. S. JOHNSON AND R. SETHI, *Complexity of flowshop and jobshop scheduling*, Math. Operations Res., 1 (1976), pp. 117–129.

[11] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. M. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.

[12] ———, *On the complexity of combinatorial problems*, Networks, 5 (1975), pp. 45–68.

[13] Y. MURAOKA, *Parallelism, exposure and exploitation in programs*, Ph.D. thesis, Computer Sci. Dept., Univ. of Illinois, 1971.

[14] P. PURDOM, *A transitive closure algorithm*, BIT, 10 (1970), pp. 76–94.

[15] J. D. ULLMAN, *NP-complete scheduling problems*, J. Comput. System Sci., 10 (1975), pp. 384–393.

[16] S. WARSHALL, *A theorem on Boolean matrices*, J. Assoc. Comput. Mach., 9 (1962), pp. 11–12.

SIAM J. COMPUT.
Vol. 6, No. 3, September 1977

A 2.5n-LOWER BOUND ON THE COMBINATIONAL COMPLEXITY OF BOOLEAN FUNCTIONS*

WOLFGANG J. PAUL†

Abstract. Consider the combinational complexity $L(f)$ of Boolean functions over the basis $\Omega=\{f|f:\{0,1\}^2\rightarrow\{0,1\}\}$. A new method for proving linear lower bounds of size $2n$ is presented. Combining it with methods presented in Savage [13, (1974)] and Schnorr [18, (1974)], we establish for a special sequence of functions $f_n:\{0,1\}^{n+2\log(n)+1}\rightarrow\{0,1\}$: $2.5n\leqq L(f)\leqq 6n$. Also a trade-off result between circuit complexity and formula size is derived.

Key words. combinational complexity, circuit size

1. Introduction. The interest in lower bounds for the combinational complexity of Boolean functions stems from two facts:

(i) practical interest: the hardware of a computer consists largely of switching networks;

(ii) theory of algorithms: proving lower bounds for the run time of algorithms has been reduced to the problem of proving lower bounds for the combinational complexity of Boolean functions [2], [15], [17]. However, for this purpose lower bounds of size greater than $O(n\log(n))$ would be needed.

Several functions with exponential combinational complexity are known [8], [20]. If one measures the complexity of the definition of a sequence of functions $f_n:\{0,1\}^n\rightarrow\{0,1\}$ by the complexity of a Turing machine which accepts $\{x|f_{\text{length}(x)}(x)=1\}$, then all functions which can presently be shown to have exponential complexity are defined by Turing machines of exponential tape complexity.

The proof relies on the fact that (for reasonable functions $s:\mathbb{N}\rightarrow\mathbb{N}$) a Turing machine with tape complexity $s(n)\log s(n)$—or a formal system with equivalent descriptive power—can construct an optimal circuit with complexity $s(n)$.

There is however a great interest in nonlinear lower bounds for functions which are definable by Turing machines with polynomial time complexity or even with complexity around $n\log n$ (e.g., multiplication). Also, if by proving a lower bound on circuit size one wishes, say, to settle the $P=NP?$ question [1], [4] in a negative sense, one has to prove a lower bound which grows faster than any polynomial for a sequence of functions definable in NP. The best lower bounds known for sequences of functions definable in NP and relative to the full basis of 2-input 1-output gates are of size $2n$ [18].

We first list some basic notations and facts. In Theorem 1, with known techniques a $2n$-lower bound is derived for a function which describes reading in a storage of n cells of 1 bit each.

To prove a lower bound better than $2n$, new methods are needed which—unlike the known ones—assert the existence of gates which are not necessarily near the inputs of the circuit. Theorem 2 is proven with such a method. Applied alone, the method yields $2n$-lower bounds.

* Received by the editors April 28, 1975, and in final revised form June 1976.

† Department of Computer Science, Cornell University, Ithaca, New York 14853, and Universität Saarbrücken, Saarbrücken, West Germany. This research was supported by DAAD (German Academic Exchange Service) under Grant 430/402/563/5.

In Theorem 3, extension of the known techniques together with the new method are applied to prove a $2.5n$-lower bound. This proof turns out to be considerably more difficult than proofs of $2n$-lower bounds.

In the last section we point out a relation to Neciporuk's test [9] and prove a trade-off result between formula size and circuit complexity.

2. Basic notations and facts. Let $K=\{0, 1\}$.

$\Omega=\{f|f: K^2 \to K\}$ is the set of *basic operations.* A *switching circuit* N is a directed, acyclic graph, each node having indegree 0 or 2. A node with indegree 0 is an *input node.* A node with outdegree 0 is an *output node.* $X_n=\{x_1, \cdots, x_n\}$ is the set of input nodes of N.

Each node with indegree 2 is labeled by an $f \in \Omega$. Associate with each node C of N a function $\text{res}_N(C)$: $K^n \to K$ in the obvious way. In the same way associate functions $\text{res}_N(e)$ with the edges e of N. If it is clear, which circuit is meant, the subscript N in $\text{res}_N(\cdot)$ is dropped.

For f: $K^s \to K$, *N computes f* if there is a node n in N such that $\text{res}(n)=f$. Define the *combinational complexity* of f by $L(f)=\min\{\#$ nodes in N which are no input nodes $|$ N computes $f\}$.

Assigning values 0 or 1 to the variables of a subset $S \subset X_n$ allows us to concentrate on subfunctions as well as on the subcircuit of N, which consists only of paths from an $x_i \in S$ to a terminal node.

A partial assignment (P.A.) is a set of assignments $\alpha=\{x_i:=c_i | c_i \in K, i \in J \subseteq [1:n]\}$.

Given f: $K^n \to K$, a P.A. α implies in a natural way a subfunction f_A : $K^{n-\#\alpha} \to K$ of f. Functions res $(C)_\alpha$ will be of special interest.

A path $(n \Rightarrow n')$ is a path from node n to node n'. Let p be a path $(n \Rightarrow n')$. Denote by $[n, n']_p$ the set of nodes of p. If there is only one path $(n \Rightarrow n')$, the subscript p is dropped. Define

$$[n, n')_p=[n, n']_p-\{n'\},$$

$$(n, n']_p=[n, n']_p-\{n\},$$

$$(n, n')_p=[n, n']_p-\{n, n'\}.$$

For sets of nodes S, S' define

$$\exists \text{ path } (S \Rightarrow S')\text{: } \exists n \in S, n' \in S' : \exists \text{ path } (n \Rightarrow n').$$

Example. In Fig. 1, $\exists$ path $(A \Rightarrow C)$; $\exists$ path $([A, C] \Rightarrow [D, F])$.

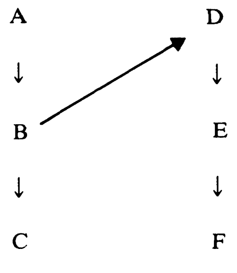


FIG. 1

Denote by $\wedge$, $\vee$, $\oplus$ and $\neg$ logical AND, OR, +mod 2, and negation, respectively.

For Boolean variables or functions x and $a \in K$, denote by

$$x^a = \begin{cases} x & \text{if } a = 1, \\ \neg x & \text{if } a = 0. \end{cases}$$

We sum up some simple facts which will be used later, often without being mentioned explicitly.

LEMMA 1.

(i) $(\neg a)\oplus b = a\oplus(\neg b) = \neg(a\oplus b)$.

(ii) *In an optimal circuit all labelings of nodes are of the form*

$$(y_1^a \wedge y_2^b)^c \qquad (\text{AND-}\textit{type gate})$$

or

$$(y_1 \oplus y_2)^d \qquad (\oplus\text{-}\textit{type gate}) \qquad (a, b, c, d \in K).$$

(All other gates compute trivial functions and can be "absorbed" into adjacent nontrivial gates.)

So one can consider an optimal circuit as consisting of $\wedge$, $\oplus$ and $\neg$, where L counts only the number of $\wedge$'s and $\oplus$'s. A function of two variables, which can be computed by an AND-type gate ($\oplus$-type gate) we call an AND-like function ($\oplus$-like function).

(iii) *No* AND-*like function is* $\oplus$-*like and vice versa.*

(iv) *An optimal circuit for a single function f has exactly one output node t with* res $(t) = f$.

From now on the letter t stands always for the output node of the circuit under consideration.

(v) *If one output of a gate is constant, independent of the assignments of values to the input nodes, this gate can be eliminated.*

(vi) *In an optimal circuit no two edges, starting from one node, go into the same node.*

(vii) *If two functions f and g are equal, then* $f_\alpha = g_\alpha$ *for all P.A.'s* α.

In the various upcoming figures the symbolism of Fig. 2 will be used. Occasionally branching points of splitting paths will be treated like nodes without cost. Negations will not be drawn in figures.

3. Two 2*n*-lower bounds. The first theorem has a practical aspect. It gives an answer to the question: how much does it cost to read from storage? On the other hand, its proof introduces some techniques, which will be used later.

For $\mathbf{a} = a_1 \cdots a_r \in K^*$ denote by

$$(\mathbf{a}) = \text{binary number represented by } \mathbf{a} \ +1.$$

THEOREM 1. *Define* f_n: $K^{n+\log(n)} \to K$ *by*:

$$f_n(a_1, \cdots, a_{\log(n)}, x_1, \cdots, x_n) = x_{(\mathbf{a})}.$$

Then $2n-2 \leqq L(f_n) \leqq 3n-3$.

Proof. We first prove the lower bound. Define for $1 \leqq s \leqq n$ the statement E_s:

E_s: For any function f: $K^{n+\log n} \to K$ with the property $[\exists S \subseteq [1:n],\ \# S = s$

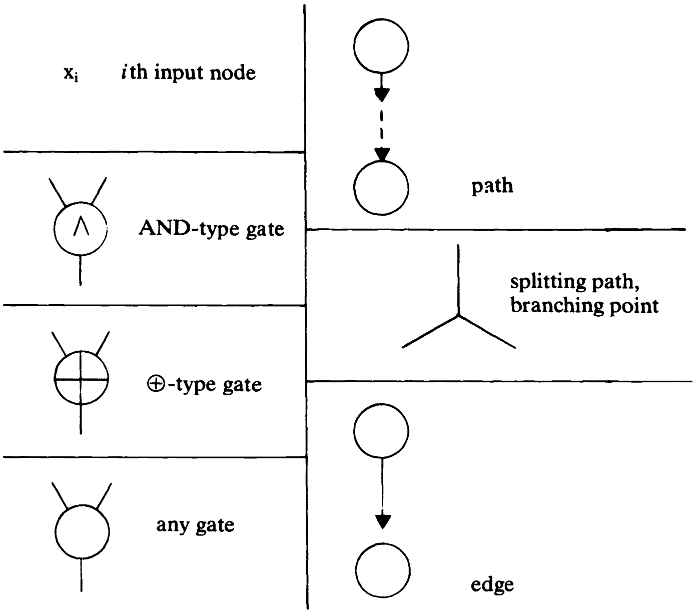


FIG. 2

such that for all $\mathbf{a}$ with $(\mathbf{a}) \in S$: $f(\mathbf{a}, x_1, \cdots, x_n) = x_{(\mathbf{a})}]$ holds: $2s-2 \leqq L(f)$.

E_1 is trivially true. We show that E_s holds if E_{s-1} holds.

Consider an optimal realization N of a function f as defined in E_s:

Case 1. $\exists i \in S$: outdegree(input node x_i) $\geqq 2$. Thus setting x_i :=const. eliminates at least two gates. The new circuit N obtained by elimination of the gates computes a function f' for which E_{s-1} applies with $S' = S\backslash\{i\}$. If N was cheaper than $2s-2$, then N' is cheaper than $2(s-1)-2$, thus contradicting E_{s-1}.

Case 2. $\exists i \in S$: outdegree(input node x_i) = 1 and the edge from x_i goes into an AND-type gate C, i.e., res $(C) = (x_i^a \wedge y^b)^c$ with $a, b, c \in K$ and y some function of the other input variables. Claim: $C \neq t$ (recall that t was defined to be the terminal node). As $s \geqq 2$, $\exists j \in S$ such that $j \neq i$. Fix the variables in $\mathbf{a}$ such that $(\mathbf{a}) = j$. Call this P.A. α. By definition of f, $f_\alpha = \text{res}\,(t)_\alpha = x_j$. But setting $x_i := (\neg a)$ yields $(\text{res}\,(C)_\alpha)_{x_i := (\neg a)} = (\neg c)$ independent of x_j. Hence $C \neq t$. Fix $x_i := (\neg a)$. This eliminates C and its successor.

Case 3. $\exists i \in S$: outdegree(input node x_i) = 1 and the edge from x_i goes into a $\oplus$-type gate C. For the P.A. α of Case 2, res $(t)_\alpha$ is independent of x_i. But res $(C)_\alpha$ depends on x_i, hence $C \neq t$.

Consider Fig. 3. Identify x_i with y. In the new circuit N', $\text{res}_{N'}(C) = \text{const}$. This eliminates C and its successor. $\text{res}_{N'}(t)$ is again a function in the sense of E_{s-1} with $S' = s\backslash\{i\}$.

This completes Case 3 and the proof of the lower bound.

In order to obtain the upper bound first observe that if n is not a power of two, f_n is not completely specified. We give a realization of a function which takes the same values as f_n where f_n is specified.

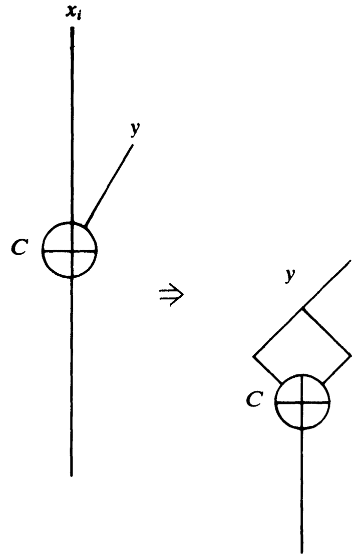


FIG. 3

Let $n=n_1+n_2$ where n_1 is a power of two and $2n_1\geqq n$. Let $k=\lceil\log(n_2)\rceil$. Then

$$f_n(a_1,\cdots,a_{\log(n)},x_1,\cdots,x_n)$$
$$=((\neg a_1)\wedge f_{n_1}(a_2,\cdots,a_{\log(n)},x_1,\cdots,x_{n_1}))$$
$$\vee(a_1\wedge f_{n_2}(a_{\log(n)-k+1},\cdots,a_{\log(n)},x_{n_1+1},\cdots,x_n)).$$

Thus $L(f_n)\leqq L(f_{n_1})+L(f_{n_2})+3$. Also $L(f_1)=0$. By induction follows $L(f_n)\leqq 3n_1-3+3n_2-3+3=3n-3$. □

While the argument in the proof of Theorem 1 was limited to the top of the circuit (i.e., near the input nodes), the next proof establishes the existence of branching points, which may be deep in the circuit.

THEOREM 2. *If for $f: K^n\to K$ $\exists S\subseteq[1:n]$ such that $[\forall i,j\in S, i\neq j\ \exists$ P.A.'s α^1_{ij}, α^2_{ij} which fix all variables except x_i and x_j such that*

$$f_{\alpha^1_{ij}}(x_i,x_j)=(x_i^a\wedge x_j^b)^c,$$
$$f_{\alpha^2_{ij}}(x_i,x_j)=(x_i\oplus x_j)^d\qquad (a,b,c,d\in K)],$$

then $L(f)\geqq 2(\# S)-2$.

We first prove two lemmas.

LEMMA 2. *Let G be a directed acyclic graph such that:*

(i) *G has n input nodes;*

(ii) *G has m output nodes;*

(iii) *Any interior node (i.e., which is not an input node) has indegree $\leqq 2$;*

(iv) *G contains p nodes with outdegree $\geqq 2$.*

Then G has at least $n-m+p$ interior nodes.

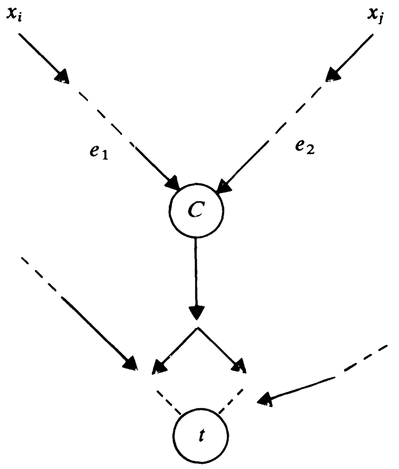


FIG. 4

Proof. Begin to construct G from the input nodes by inserting node after node. In the beginning one has n "wires hanging down" from the input nodes. One interior node decreases the "number of down hanging wires" only by one. The number is increased during the construction by at least p. In the end it is m. Hence G contains at least $n+p-m$ interior nodes. □

Lemma 2 will be applied several times to certain subgraphs of circuits. The next lemma helps to establish the existence of branching points in circuits for functions f, to which the hypothesis of Theorem 2 applies:

LEMMA 3. $\forall i, j \in S$, $i \neq j$: *before a path* $(x_i \Rightarrow t)$ *meets a path* $(x_j \Rightarrow t)$, *one of these paths splits.*

Proof. Suppose the lemma is false for i and j. Consider Fig. 4. Let C be the node where the paths meet. Let e_1, e_2 be the input wires of C. Let res (t) be the function computed by the circuit. We show: if C is an AND-type gate, then all subfunctions of res (t), which depend only on x_i and x_j are AND-like or constant. If C is a $\oplus$-type gate, all such subfunctions are $\oplus$-like or constant. But f has subfunctions of both types. More precisely:

Take any P.A. α which fixes all inputs except x_i, x_j. As—by assumption—neither path $(x_i \Rightarrow C)$ nor path $(x_j \Rightarrow C)$ splits:

$$\text{res}\,(e_1)_\alpha = x_i^{a_1} \text{ or } 0 \text{ or } 1,$$

$$\text{res}\,(e_2)_\alpha = x_j^{a_2} \text{ or } 0 \text{ or } 1,$$

$$\text{res}\,(t)_\alpha = \text{res}\,(C)_\alpha^{a_3} \text{ or } 0 \text{ or } 1.$$

Suppose C is an AND-type gate. Then

$$\text{res}\,(t)_\alpha = (x_i^{a_4} \wedge x_j^{a_5})^{a_6} \text{ or } 0 \text{ or } 1 \neq f_\alpha$$

by Lemma 1.3.

In the same way one excludes: C is a $\oplus$-type gate. □

Proof of Theorem 2. There are $\# S-1$ input nodes x_i, $i \in S$, such that any path $(x_i \Rightarrow t)$ splits. Suppose that this is false. Then there are $i_0, j_0 \in S$ such that there is only one path $(x_{i_0} \Rightarrow t)$ and one path $(x_{j_0} \Rightarrow t)$. This contradicts Lemma 3. Hence on $\# S-1$ paths $(x_i \Rightarrow t)$ *there are first* nodes D_i with outdegree $(D_i) \geqq 2$. Again by Lemma 3 these are mutually distinct. Now the theorem follows from Lemma 2. □

That the branching points need not be directly under the input nodes is shown in Fig. 5.

Example. For any $c \in [1:n-1]$ and any

$$f_c\colon K^n \to K \quad \text{such that} \quad f_c(x_1, \cdots, x_n) = 1 \text{ if } \sum x_i = c,$$

the following is true:

$$L(f_c) \geqq 2n-2.$$

4. A 2.5*n*-lower bound. Notation:

$$\mathbf{a}_1 = a_1, \cdots, a_{\log(n)},$$

$$\mathbf{a}_2 = a_{\log(n)+1} \cdots a_{2\log(n)} \qquad (a_i \in K).$$

THEOREM 3. *For* $f\colon K^{n+2\log(n)+1} \to K$,

$$f(a_1, \cdots, a_{2\log(n)}, q, x_1, \cdots, x_n) = \begin{cases} x_{(\mathbf{a}_1)} \wedge x_{(\mathbf{a}_2)} & \textit{if } q=1, \\ x_{(\mathbf{a}_1)} \oplus x_{(\mathbf{a}_2)} & \textit{if } q=0, \end{cases}$$

$$2.5n-2 \leqq L(f) \leqq 6n+C \qquad (C \in \mathbb{N} \textit{ independent of } n).$$

Proof. Note that each of Theorem 1 and Theorem 2 implies $2n$-lower bounds for f. The upper bound is an easy consequence of Theorem 1. As in the proof of Theorem 1, we first make f independent of all inputs x_i, which allows us to

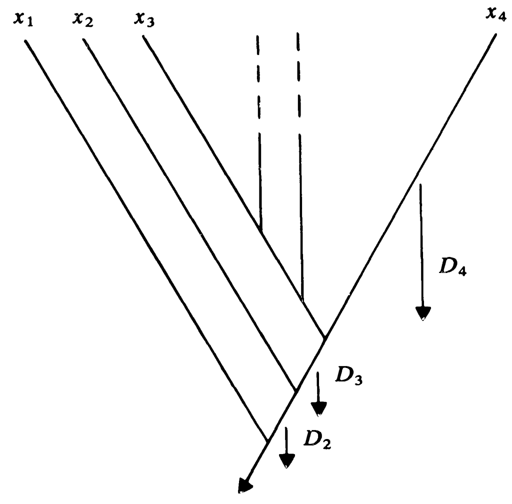


FIG. 5

eliminate 3 gates (see Cases I–IV). When that is no longer possible, we will know quite exactly how the "top" of the circuit looks. For the remaing s inputs, we prove the existence of $5s/2-2$ gates without an inductive argument. Define for $1 \leqq s \leqq n$ the statement E_s:

E_s: For any function $f\colon K^{n+2\log(n)+1} \to K$ with the property

$$\Big[\exists S \subseteq [1\colon n],\ \# S = s \text{ such that for } \mathbf{a}_1, \mathbf{a}_2 \text{ with } (\mathbf{a}_1), (\mathbf{a}_2) \in S\colon$$

$$f(\mathbf{a}_1, \mathbf{a}_2, q, x_1, \cdots, x_n) = \begin{cases} x_{(\mathbf{a}_1)} \wedge x_{(\mathbf{a}_2)} & \text{if } q = 1, \\ x_{(\mathbf{a}_1)} \oplus x_{(\mathbf{a}_2)} & \text{if } q = 0 \end{cases}\Big]$$

holds, $2.5s - 2 \leqq L(f)$.

E_1 is trivially true. To see that E_{s-1} implies E_S, consider an optimal realization N of a function f as defined in E_S:

Case I: $\exists i \in S$ such that outdegree(input node x_i) $\geqq 3$. This is similar to Case 1 in the proof of Theorem 1.

Case II: $\exists i \in S$ such that outdegree(input node x_i) $= 2$ and one of the edges e_1, e_2 going out from x_i goes into an AND-type gate. This is almost like Case 2 in the proof of Theorem 1. A small difficulty occurs in the case of Fig. 6.

If the edge $(g_1 \to g_2)$ does not split, N is not optimal. Hence it splits. Now the method of Case 2 of the proof of Theorem 1 work, no matter if g_1 or g_2 is the AND-type gate.

Case III. $i \in S$: outdegree(input node x_i) $= 2$ and both edges from x_i go into $\oplus$-type gates A_1 and A_2. Consider Fig. 7. Let B (C) be the first place on a path $(A_1 \Rightarrow t)$ (on a path $(A_2 \Rightarrow t)$) which is an AND-type gate or a branching point. The existence of B or C will be shown in a moment. Denote by $f_1, \cdots, f_s$ $(g_1, \cdots, g_t)$ the functions associated with the other input wires to this path. By definition of B

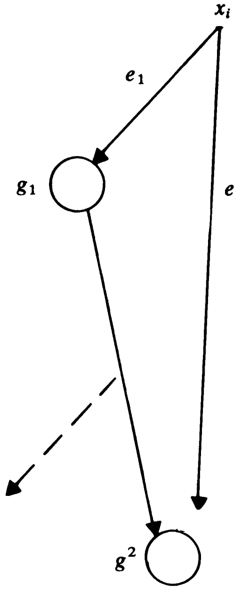


FIG. 6

and C, there is only one path $(A_1 \Rightarrow B)$ and only one path $(A_2 \Rightarrow C)$. Because of Lemma 1 (part (i)) one can assume that all gates in $[A_1, B']$ and in $[A_2, C']$ are $\oplus$-gates (all occurring negations can be removed by possibly negating f_1 or g_1 or both).

LEMMA 4. $[A_1, B'] \cap [A_2, C'] = \varnothing$.

Proof. Suppose not. Then the situation is like that in Fig. 8. Hence

$$\text{res}\,(D) = f_1 \oplus \cdots \oplus g_q \oplus x_i \oplus x_i = f_1 \oplus \cdots \oplus g_q.$$

As no other path goes out of $[A_1, D)$ and $[A_2, D)$, $f_1, \cdots, g_q$ are independent of x_i. Hence res (D) is independent of x_i. By the same reasoning, res (t) depends on x_i only via res (D). Hence res (t) is independent of x_i. But as $i \in S$, f depends on x_i.

COROLLARY 1. *B' and C' are different.*

COROLLARY 2. *B or C exists.*

Proof. Suppose not; then $B' = C' = t$, contradicting Lemma 4.

LEMMA 5. *If B and C exist*:

$$(\neg \exists \, \textit{path}\, (B \Rightarrow [A_2, C'])) \vee (\neg \exists \, \textit{path}\, (C \Rightarrow [A_1, B']));$$

if without loss of generality only C exists:

$$\neg \exists \, \textit{path}\, (B' \Rightarrow [A_2, C']).$$

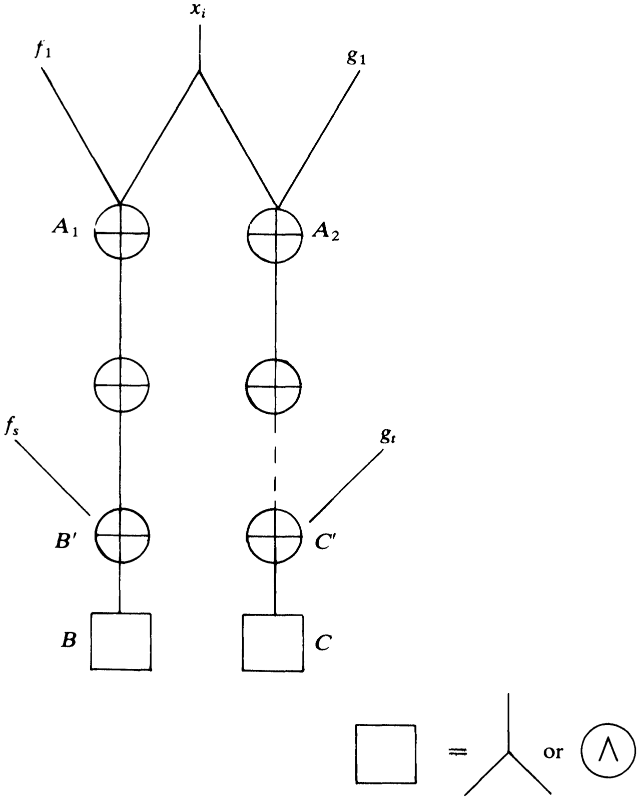


FIG. 7

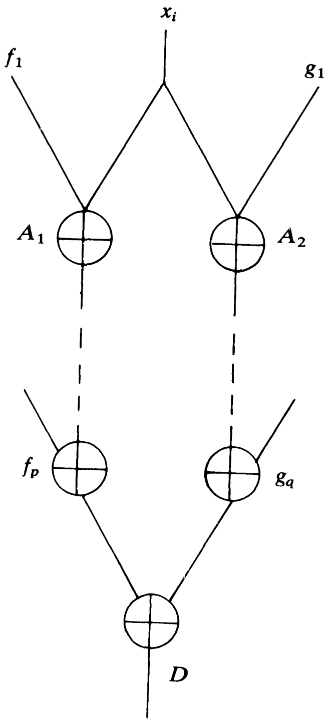


FIG. 8

Proof. If B and C and both paths exist, the graph N contains a cycle.

If only C exists, the existence of a path $(B' \Rightarrow [A_2, C'])$ implies the existence of B.

From now on without loss of generality C exists and if B exists, then $\neg\exists$ path $(B \Rightarrow [A_2, C'])$.

LEMMA 6. $C \neq t$.

Proof. Suppose $C = t$. Then C is an AND-type gate and $\exists$ path $(A_1 \Rightarrow C)$. This implies the existence of B. Hence if B does not exist, the lemma is true. If B exists, proceed as follows: Since $\# S \geqq 2$, $\exists j \in S$, $j \neq i$. Fix $\mathbf{a}_1$, $\mathbf{a}_2$ such that $(\mathbf{a}_1) = (\mathbf{a}_2) = j$, $q = 1$. Fix all other variables except x_i and x_j arbitrarily. Call this P.A. α. Then $f_\alpha(x_i, x_j) = x_j$ independent of x_i. We prove that the output res $(t)_\alpha$ still depends on x_i and that, by fixing x_i to the right constant, one can produce a false output:

$$\neg\exists \text{ path } (B \Rightarrow [A_2, C']) \quad \text{implies} \quad g_1, \cdots, g_t \text{ do not depend on } x_i.$$

Hence

$$res_\alpha\ (C') = x_i \oplus h(x_j)$$

where h is some Boolean function of one variable. Furtnermore because C is an AND-type gate:

$$\text{res}_\alpha\ (C) = (\text{res}_\alpha\ (C')^a \wedge y(x_i, x_j)^b)^c$$

for some constants $a, b, c \in K$ and some Boolean function y. Fix $x_j := c$ and then

$x_i := d$ such that $(x_i \oplus h(c))^a = 0$. Then

$$\mathrm{res}_\alpha \,(C)(d, c) = (0 \wedge y(d, c)^b)^c = 0^c = (\neg c) \neq f_\alpha(d, c).$$

Elimination step. Change the circuit N. Compute $(g_1 \oplus \cdots \oplus g_t)^a$ for some $a \in K$ and connect the result with x_i. This increases the costs by $t-1$. The new graph N' is cycle free because $\neg\exists$ path $(B \Rightarrow [A_2, C'])$. But now $\mathrm{res}_{N'}\,(C') = \neg a$. Hence delete $[A_2, C']$, decreasing the costs by t. If C is a branching point, this eliminates at least two more gates. If C is an AND-type gate, choose a such that $\mathrm{res}_{N'}\,(C) = \mathrm{const}$. This eliminates C and a successor which must exist by Lemma 6.

Case IV. $\exists i \in S$: outdegree(input node x_i) = 1 and the edge from x_i goes into a $\oplus$-type gate g. This case is slightly easier than Case III: treat g and the $\oplus$'s immediately following it like $[A_2, C']$ and don't worry about $[A_1, B']$.

If none of the Cases I–IV are applicable, we cannot eliminate enough gates by decreasing S. But we know: $\forall i \in S$: input node x_i has exactly one outgoing edge. This goes into an AND-type gate. Call this gate G_i. Define $\mathbf{G} = \{G_i \mid i \in S\}$.

LEMMA 7. *$\forall i, j, i \neq j$: $G_i \neq G_j$.*

Proof. The proof follows immediately from Lemma 3. □

DEFINITION. A path $n \to n_1 \to \cdots \to n_t \to n'$ is *free* if $\forall i$: $n_i \notin \mathbf{G}$. A superscript f in expressions like $[n, n']^{\mathrm{f}}$ will denote that the path, which defines this set of nodes, is free.

LEMMA 8. *$\forall i \in S$: $\exists$ free path $(G_i \Rightarrow t)$.*

Proof. Suppose for $i \in S$: $\neg\exists$ free path $(G_i \Rightarrow t)$. Then all paths $(G_i \Rightarrow t)$ go through a G_j, $j \neq i$ (Fig. 9).

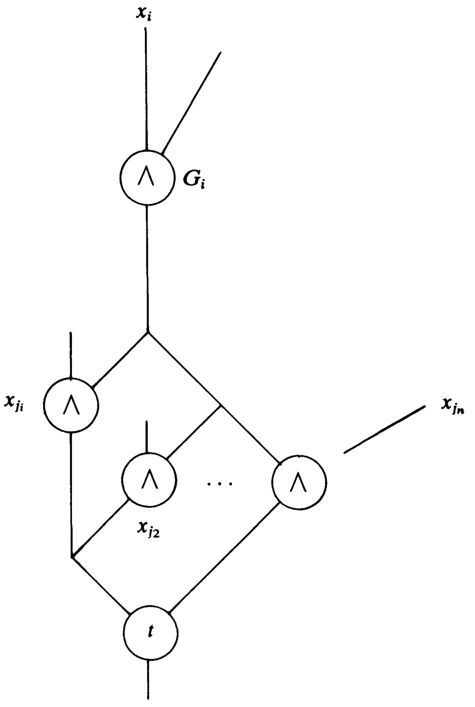


FIG. 9

Fix all variables except x_i such that $(\mathbf{a}_1)=(\mathbf{a}_2)=i$, $q=1$ and such that by the induced P.A. α, res $(G_j)_\alpha=$ const. for all $j\neq i$. Then $f_\alpha=x_i$ but, as there is no free path $(G_i\Rightarrow t)$: res $(t)_\alpha$ independent of x_i. □

DEFINITION. Let p be a free path $(n\Rightarrow n')$. p splits into a free path to n'' if $\exists C\in[n,n']_p$, $D\notin[n,n']_p\cup\mathbf{G}$, such that there is an edge from C to D and a free path $(D\Rightarrow n'')$.

DEFINITION. $\mathbf{E}=\{e\mid\exists i$ such that e is node on a free path $(G_i\Rightarrow t)$, $e\notin\mathbf{G}\}$.

Clearly $E\cap G=\emptyset$. If we would show that there are $\#S-1$ nodes in E where a free path $(G_i\Rightarrow t)$ splits into another free path to t, an application of Lemma 2 to determine the size of E would yield a $3n$-lower bound. But Fig. 10 shows that it might be the case that no free path $(G_i\Rightarrow t)$ splits into another free path to t.

By placing gates on places like α, whose other input is an appropriate function of the a_i's and q, one would still be able to compute $x_i\wedge x_j$ as well as $x_i\oplus x_j$ for all i and j. But such gates contribute to the costs. Though we are not able to locate these gates exactly in the circuit, we can find a certain number, say n, of distinct places, where paths to such gates start. Then, we will conclude, there must be at least $n/2$ such gates (as one gate has only two inputs).

The following lemma is the counterpart to Lemma 3.

LEMMA 9. *Let x_i, x_j $(i,j\in S)$ be such that a free path $(x_i\Rightarrow t)$ and a free path $(x_j\Rightarrow t)$ meet before any of them splits into another free path to t. Let C be the node where these paths meet; then*

(i) *either $\exists$ free path $(G_i\Rightarrow G_j)$ or $\exists$ free path $(G_j\Rightarrow G_i)$;*

(ii) *each such path goes through a gate g, $g\notin[G_i,C]^{\mathrm{f}}\cup[G_j,C]^{\mathrm{f}}$.*

Proof. Consider Fig. 11. First note that by the hypothesis $\neg\exists$ free path $([G_i,C)^{\mathrm{f}}\Rightarrow(G_j,C)^{\mathrm{f}})$ or vice versa.

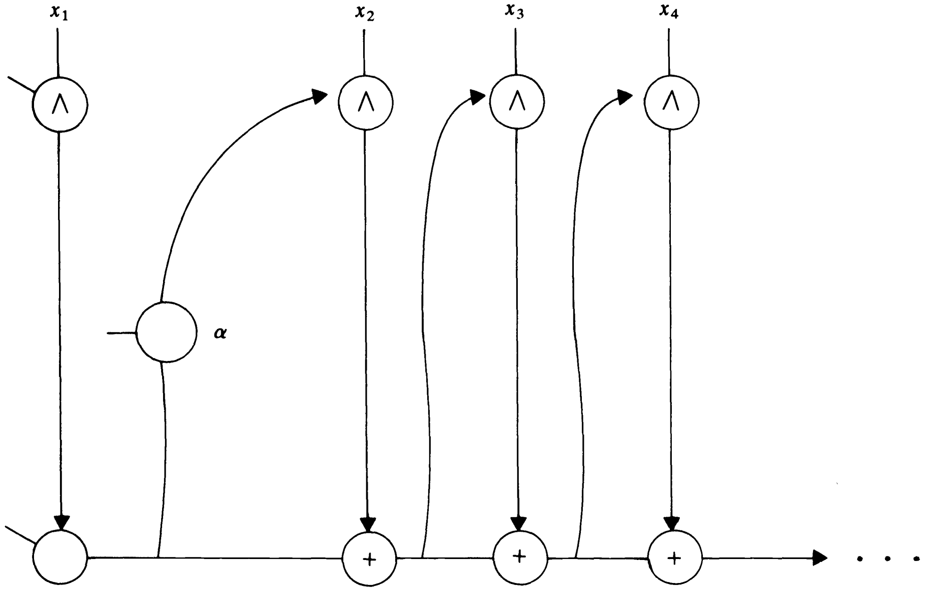


FIG. 10

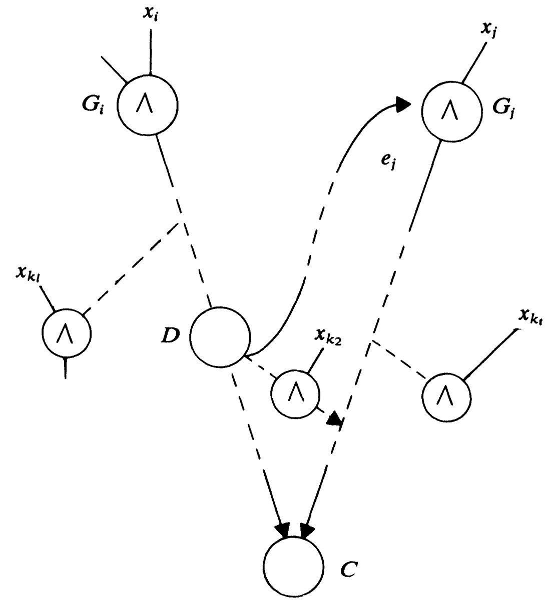


FIG. 11

Fix all variables except x_i, x_j and q such that $(\mathbf{a}_1)=i$, $(\mathbf{a}_2)=j$ and such that by the induced P.A. α: res $(G_k)_\alpha=$ const. for all $k\notin\{i,j\}$.

(i) Both types of paths cannot be present, because then the graph would contain a cycle. Suppose no path $(G_i\Rightarrow G_j)$ and no path $(G_j\Rightarrow G_i)$ is present or all such paths are not free. Then exclude, as in the proof of Lemma 3, the cases $C=$ AND-type gate and $C=\oplus$-type gate.

(ii) Without loss of generality $\exists$ free paths $(G_i\Rightarrow G_j)$. Suppose one such path contains no gate $g\notin[G_i,C]^{\mathrm{f}}\cup[G_j,C]^{\mathrm{f}}$. Denote by D the last node this path has in common with the free path $(G_i\Rightarrow C)$ (the first candidate for D is G_i). Then there is an edge e_j from D to G_j. Fix q :=0.

As $(\mathrm{res}(t)_\alpha)_{q\,:=0}$ depends only on x_i and x_j,

$$(\mathrm{res}(D)_\alpha)_{q:=0}=x_i^a\text{ or }0\text{ or }1.$$

If $(\mathrm{res}(D)_\alpha)_{q:=0}=0$ or 1, then

$$(\mathrm{res}(t)_\alpha)_{q:=0}\quad\text{is independent of }x_i.$$

Hence $(\mathrm{res}(D)_\alpha)_{q\,:=0}=x_i^a$. Now fix x_i :=c such that

$$(\mathrm{res}(G_j)_\alpha)_{\{q:=0,\,x_i:=c\}}$$

is independent of x_j. Hence $(\mathrm{res}(t)_\alpha)_{\{q:=0,x:=c\}}$ is independent of x_j. But $(f_\alpha)_{\{q:=0,x_i:=c\}}=x_j\oplus c$.

DEFINITION. For $i,j\in S$: iRj: $\Leftrightarrow$free path $(G_i\Rightarrow t)$ and free path $(G_j\Rightarrow t)$ meet before any of them splits into another free path to t.

LEMMA 10. *R is an equivalence relation.*

R partitions S into, say, k groups: $S=S_1\cup\cdots\cup S_k$. The free paths $(G_i\Rightarrow t)$ of the group r meet in a point t_r before splitting into other free paths to t. So the free paths $(G_i\Rightarrow t)$ for $i\in S_r$ form a tree with root t_r. Call this tree the *free tree* of group r.

If $S=\{i\}$, then we consider the node G_i as the free tree as well as $G_i=t_r$.

Let $\mathbf{A}=\{g|g$ is a node in the circuit and $\neg\exists$ free path $(g\Rightarrow t)\}$. Clearly by Lemma 8 and by the definition of $\mathbf{E}$:

$$\mathbf{G}\cap\mathbf{A}=\mathbf{E}\cap\mathbf{A}=\varnothing.$$

Let $T=\{i\in S|\exists r$ such that $i\in S_r$ and $\#S_r\geqq 2\}$, $k=\#\{r|\#S_r\geqq 2\}$.

LEMMA 11. $\#\mathbf{A}\geqq(\#T-k)/2$.

Proof. We claim: on any free tree with n leaves $(n\geqq 2)$, there are at least $n-1$ nodes D such that a free path $(D\Rightarrow\mathbf{A})$ leaves the free tree in D. As for different trees such sets of nodes D are disjoint, summing over all free trees one finds at least $\#T-k$ paths going into $\mathbf{A}$; hence $\#\mathbf{A}\geqq(\#T-k)/2$.

To prove the claim, observe that in the free tree of group r there is a node F such that a free path $(G_i\Rightarrow t)$ and free path $(G_j\Rightarrow t)$ meet in F; $i,j\in S_r$ but $\neg\exists l\in S_r$ such that a free path $(G_l\Rightarrow t)$ goes through F. Apply to i and j Lemma 9. This implies the existence of one node D—without loss of generality, $D\in(G_i,F)^{\mathrm{f}}$—a gate $g\notin[G_i,F]^{\mathrm{f}}\cup[G_j,F]^{\mathrm{f}}$ and a path $(D\Rightarrow g)$. If there was a free path $(g\Rightarrow t)$, then $=\neg i\,Rj$. Hence $g\in\mathbf{A}$.

Now mark $[G_i,F)$ in the free tree and repeat the above argument. As the unmarked paths are disjoint from the marked ones, all new nodes D are disjoint from the old ones.

Repeating this argument $n-1$ times proves the claim. □

Proof of Theorem 3. Consider free paths $(t_r\Rightarrow t)$ and suppose there is a free path $(t_r\Rightarrow t)$ and a free path $(t_{r'}\Rightarrow t)$ which both don't split into another free path to t. Then there are two cases:

(i) The free path $(t_r\Rightarrow t)$ meets the free path $(t_{r'}\Rightarrow t)$. Then by definition of R and as none of the paths splits into another free path to t, neither t_r nor $t_{r'}$ is root of a free tree.

(ii) There is a free path $(t_r\Rightarrow t_{r'})$ or a free path $(t_{r'}\Rightarrow t_r)$. Then t_r or $t_{r'}$ is not a root of a free tree.

Hence all but one of the free paths $(t_r\Rightarrow t)$ must split into another free path to t.

This implies the existence of a set B' of nodes, in which free paths $(t_r\Rightarrow t)$ split into other free paths to t. Observe that there are $s-\#T+k$ roots t_r of free trees.

As in the proof of Theorem 2 one concludes that these must be $s-\#T+k-1$ first such nodes on free paths $(t_r\Rightarrow t)$.

Apply Lemma 2 to the graph which consists of input nodes G, interior nodes E, and the free paths between them. By Lemma 7 it has s input nodes. t is the output node. Hence

$$\#\mathbf{E}\geqq 2s-\#T+k-2.$$

Finally by Lemma 7 and 11:

$$L(f) \geqq \# \boldsymbol{G} + \# \boldsymbol{E} + \# \boldsymbol{A} \geqq 3s - \frac{\# T - k}{2} - 2.$$

As $\# T \leqq s$ the theorem follows. □

5. A trade-off result.

Definition. A *tree network* is a circuit with the property that all interior nodes (i.e., which are not input nodes) have outdegree 1.

Tree networks are isomorphic to Boolean formulas. For a switching function f define

$$T(f) = \min \{\# \text{ interior nodes in a tree network } N | N \text{ computes } f\}.$$

T is called the (minimal) formula size. If one allows only AND-like connectives in circuits and tree networks, the minimal formula size of the parity function $x_1 \oplus \cdots \oplus x_n$ is $n^2 - 1$ [5] and the circuit complexity is $3n - 3$ [18]. We prove a similar result for the case in which $\oplus$-like connectives are also allowed in formulas.

Notation. Let N be a circuit, X the set of input variables and $S \subset X$. A node C in N is called S-*node* if there are nodes C_1, C_2 in N, $C_1 \neq C_2$ (possibly $C_1, C_2 \in S$) such that there is an edge $(C_1 \Rightarrow C)$, an edge $(C_2 \Rightarrow C)$, a path $(S \Rightarrow C_1)$ if $C_1 \notin S$ and a path $(S \Rightarrow C_2)$ if $C_2 \notin S$.

The following lemma is proved by a method very similar to the proof of Lemma 3.

Lemma 12. *Let f: $K^n \to K$ be a switching function, X the set of input variables, $S \subset X$ and P.A.'s $\alpha_1, \cdots, \alpha_M$ which fix all variables in $X \backslash S$ such that*

$$f_{\alpha_i} \neq f_{\alpha_j} \quad \text{for } i \neq j \qquad (1 \leqq i, j \leqq M).$$

Let N be a circuit for f. Then

(i) $P_s := \#\{C | C \text{ is an } S\text{-node in } N\} \geqq (\log(M))/4 - 1$.

(ii) (Neciporuk) *If N is a tree network, then*

$$\sum_{j \in S} \text{outdegree } x_j \geqq \frac{\log(M)}{4} - 1.$$

Proof. (ii) follows from (i) and the definition of a tree network.

(i) Let N' be the subcircuit of N which consists only of the paths $(S \Rightarrow t)$. In N' the nodes with indegree 2 are exactly the S-nodes of N.

Let α be a P.A. which fixes all variables in $X \backslash S$. Then res $(n)_\alpha$ = const. for all nodes not in N'.

Let A be an S node or $A \in S$, let B be an S-node or $B = t$, and suppose $\exists$ path $(A \Rightarrow B)$ which contains no other S-node. Let e be the edge on this path which goes into B. We call such a path $(A \Rightarrow B)$ a *programmable edge of* N'. Why this name?

Consider how res $(e)_\alpha$ depends on res $(A)_\alpha$ for different P.A.'s α. No matter how many gates are on this path, one can "program"—by choice of α—this dependence in at most 4 ways:

$$\operatorname{res}(e)_\alpha = \operatorname{res}(A)_\alpha \text{ or } \neg \operatorname{res}(A)_\alpha \text{ or } 0 \text{ or } 1.$$

(Compare with the proof of Lemma 3.)

N' contains at most $2P_S+1$ programmable edges. Hence there are at most 4^{2P_S+1} different subfunctions f_α which can be "programmed" by different P.A.'s α.

By the hypothesis: $4^{2P_S+1} \geqq M$. This proves the lemma. □

We apply Lemma 11 to the function $f\colon K^n \to K$, defined as follows: let $n = \log(s) - \log\log(s) + 2s$ ($s = 2^{2^\tau}$, $\tau \in N$).

Partition the input variables into

$$\{a_1, \cdots, a_{\log(s)-\log\log(s)}\}, \{x_1, \cdots, x_s\}, \{y_1, \cdots, y_s\}.$$

Denote by **a** the string

$$a_1, \cdots, a_{\log(s)-\log\log(s)}.$$

Denote by **b** the string

$$x_{(\mathbf{a})\log(s)+1} \cdots x_{((\mathbf{a})+1)\log(s)}.$$

Define $f(a_1, \cdots, y_s) = y_{(\mathbf{b})}$.

THEOREM 4. (i) $T(f) \geqq O(n^2/\log(n))$. (ii) $L(f) = O(n)$.

Proof. (i) For $0 \leqq i \leqq s/\log(s) - 1$ define

$$S_i = \{x_{i\cdot\log(s)+1}, \cdots, x_{(i+1)\log(s)}\}.$$

Fix **a** such that $(\mathbf{a}) = i$ and fix the x_j, $j \notin S_i$, arbitrarily. Now each of the 2^s different assignments for the variables in $y_1, \cdots, y_s$ produces a different subfunction of f which depends only on the variables in S_i.

Applying Lemma 12 (ii) yields

$$\forall i \sum_{x_j \in S_i} \operatorname{outdegree}(x_j) \geqq \frac{s}{4} - 1.$$

As $S_i \cap S_j = \emptyset$ for $i \neq j$:

$$T(f) \geqq \sum_i \sum_{x_j \in S_i} [\operatorname{outdegree}(x_j)]$$

$$\geqq \left(\frac{s}{4} - 1\right)(s/\log(s)) = O(n^2/(\log(n)).$$

(ii) Realize f in a circuit as follows: for $1 \leqq k \leqq \log(s)$ realize the functions

$$f_k\colon K^{s/\log(s)+\log(s)-\log\log(s)} \to K$$

$$f_k(\mathbf{a}, x_k, x_{\log(s)+k}, x_{2\log(s)+k}, \cdots, x_{s-\log(s)+k}) = x_{(\mathbf{a})\log(s)+k}.$$

By Theorem 1 each f_k can be realized with cost $3 \cdot s/\log(s)$.

Let $\{c_1, \cdots, c_{\log(s)}\}$ be a set of new variables. Denote by **c** the string $c_1 \cdots c_{\log(s)}$. Construct a circuit for

$$f'\colon K^{s+\log(s)} \to K\colon f'(\mathbf{c}, y_1, \cdots, y_s) = y_{(\mathbf{c})}.$$

By Theorem 1 f' has cost $\leqq 3s$. For $1 \leqq k \leqq \log(s)$ connect the terminal node of the circuit f_k with the input c_k of f'. This gives a realization of f with cost $6s$ and the theorem is proven. □

Acknowledgments. I want to thank Professor John Hopcroft, Dr. Peter van Emde Boas, and Dany Paul for encouragement, help and discussion. Zvi Galil pointed out to me how to improve the original $2.25n$-lower bound of Theorem 3 [11] into $2.5n$.

REFERENCES

[1] S. A. Cook, *The complexity of theorem proving procedures*, Proc. 3rd ACM-STOC, 1971, pp. 151–158.

[2] M. J. Fischer, *Lectures on network complexity*, presented at the University of Frankfort, 1974.

[3] G. Hotz, *Schaltkreistheorie*, de Gruyter, Berlin, 1974.

[4] R. M. Karp, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatchers, eds., Plenum Press, New York, 1972, pp. 85–104.

[5] Khrapchenko, *On the complexity of the linear function in the class of* Π-*circuits*, Mat. Zamertki, 9 (1971), no. 1, pp. 35–40. (In Russian.)

[6] O. B. Lupanov, *Ueber den Schaltaufwand bei der Realisierung logischer Funktionen*, Probleme der Kybernetik, 3 (1957).

[7] K. Melhorn and Z. Galil, *Monotone switching circuits and Boolean matrix product*, preprint, 1974.

[8] A. Meyer, 6.853 *Lecture notes*, Dept. of Electrical Engineering, Mass. Inst. of Tech., Cambridge, MA, 1974.

[9] E. I. Neciporuk, *A Boolean function*, Soviet Math. Probl., 7 (1966), pp. 999–1000.

[10] M. S. Paterson, *Complexity of monotone networks for Boolean matrix product*, Tech. Rep., Univ. of Warwick, England, 1974.

[11] W. Paul, *2.25N-lower bound on the combinatorial complexity of Boolean functions*, Tech. Rep. TR 74-222, Cornell Univ, Ithaca, NY, 1974.

[12] V. Pratt, *The power of negative thinking in multiplying Boolean matrices*, 6th Ann. Symp. on Theory of Computing (SIGACT), 1974.

[13] J. E. Savage, *The complexity of computing*, draft, 1974.

[14] J. E. Savage and E. A. Lamagna, *Combinational complexity of some monotone functions*, 15th Ann. Symp. on Switching and Automata Theory (SWAT), 1974.

[15] J. E. Savage, *Computational work and time on finite machines*, J. Assoc. Comp. Mach., 19 (1974), pp. 660–674.

[16] C. E. Shannon, Bell System Tech. J., 28 (1949), no. 1.

[17] C. P. Schnorr, *Lower bounds for the product of time and space requirements of Turing machine computations*, Proc. MFCS, High Tatras, Poland, Sept. 3–8, 1973, pp. 153–163.

[18] ———, *Zwei lineare untere Schranken fuer die Komplexitaet Boolescher Funktionen*, Computing, 13 (1974), pp. 155–171.

[19] ———, *The combinational complexity of equivalence*, *GI-Fachtagung Automatentheorie und Formale Sprachen*, *Kaiserslautern*, Springer Lecture Notes, Springer-Verlag, Berlin, 1974.

[20] L. J. Stockmeyer, *The complexity of decision problems in Automata theory and Logic*, Project MAC TR 133, Mass. Inst. of Tech, Cambridge, MA, 1974.

[21] V. Strassen, *Berechnung und Programm 1*, Acta Informatica (1972), pp. 320–335.

SIAM J. COMPUT.
Vol. 6, No. 3, September 1977

ON RESOLUTION WITH CLAUSES OF BOUNDED SIZE*

ZVI GALIL†

Abstract. Several procedures based on (not necessarily regular) resolution for checking whether a formula in CF3 is contradictory are considered. The procedures use various methods of bounding the size of the clauses which are generated. The following results are obtained:

1. All of the proposed procedures which are forced to run in polynomial time do not always work—i.e., they do not identify all contradictory formulas.

2. Those which always work must run in exponential time. The exponential lower bounds for these procedures do not follow directly from Tseitin's lower bound for regular resolution since these procedures also allow nonregular resolution trees.

Key words. resolution proofs, regular resolution, bounded resolution

1. Introduction. We assume that there is an unbounded set $\{x_1, x_2, \cdots, y, \cdots\}$ of *variables*. A *literal* is either a variable or the complement of a variable, denoted x and $\bar{x}$ respectively (or x^1 and x^0). A *clause* is a finite set of literals $(l_1, \cdots, l_k)$ such that no variable appears more than once, and is denoted by $l_1 \vee l_2 \vee \cdots \vee l_k$. A *formula* (in conjunctive form (CF)) is a finite set of clauses. A *valuation* (or *assignment*) is a function v from the set of literals to $\{0, 1\}$. (Zero is to be thought of as meaning false and one as meaning true.) The *value* (relative to v) of the literal x ($\bar{x}$) is $v(x)$ $(1-v(x))$. The value of the clause $l_1 \vee l_2 \vee \cdots \vee l_k$ is maximum of the values of the l_i's. (By convention if $k=0$ we denote the clause by Λ, and define its value to be 0.) The value of a formula is 1 if the values of all its clauses are 1, and 0 otherwise.

A formula is *satisfiable* is there is some valuation for which the value of the formula is 1, and *unsatisfiable* (*contradictory*) otherwise. It is well known that any 0, 1 valued function of k arguments may be represented by a formula in CF with variables $\{x_1, \cdots, x_k\}$. Cook [2] has observed that any formula (not necessarily in CF) of length n can be transformed, using additional variables, into another formula of length $O(n)$ in CF with at most three literals per clause, and that the second formula is satisfiable if and only if the original formula is satisfiable. (This also appears implicitly in [8] and explicitly in [1].) We denote the latter by CF3.

In this paper we are interested in the following problem: given a formula in CF (i.e. a set of clauses), it is contradictory? The dual problem is the well-known "tautology problem". The existence of a good algorithm for either problem is closely related to the "$P=NP$?" problem [2].

Two clauses C_1 and C_2 are said to *conflict* if there are literals in C_1 which appear complemented in C_2. If C_1 and C_2 do not conflict, then we denote by $C_1 \vee C_2$ the clause which contains all literals in C_1 and C_2. The clauses C_1 and C_2 are said to *clash* if there is exactly one literal in C_1 which appears complemented in C_2. If $C_1 \vee x$ and $C_2 \vee \bar{x}$ clash, then their *resolvent* is the clause $C_1 \vee C_2$. We shall say that $C_1 \vee C_2$ is obtained from $C_1 \vee x$ and $C_2 \vee \bar{x}$ by applying the *resolution rule*, or by *annihilating* x.

* Received by the editors October 6, 1975, and in revised form April 15, 1976.

† Computer Sciences Department, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598. This paper is based on parts of the author's Ph.D. thesis at Cornell University [3]. This work was supported in part by the Office of Naval Research under Grant N00014-67-A-0077-0021.

A *resolution proof* of unsatisfiability of a set of clauses S is a sequence of clauses $C_1, \cdots, C_k$ such that $C_k = \Lambda$ and for $0 \leqq i \leq k-1$, C_{i+1} is the resolvent of some pair from $S \cup \{C_1, \cdots, C_i\}$.

Robinson [6] introduced the resolution principle and showed that a set of clauses is unsatisfiable if and only if there is a resolution proof of its unsatisfiability.

A (*resolution*) *proof tree for a clause C* (*using S*) is a binary tree T whose vertices contain clauses in such a way that (i) C appears at the root of T; (ii) every leaf of T contains one of the clauses in S, and (iii) the clause at any vertex which is not a leaf is the resolvent of the clauses at its sons.

A *resolution proof tree* (*using S*) is a proof tree for the empty clause using S. The complexity of a proof tree T, $N(T)$, is the number of distinct clauses which appear on vertices of T. Note that a proof tree T using S is just a way to represent a resolution proof of the unsatisfiability of S, and that $N(T)$ is actually the length of the resolution (straight line) proof defined above.

The (*unrestricted*) *resolution procedure* is the procedure that nondeterministically constructs a resolution proof tree using the given set of clauses S. It can be viewed as representing the class of deterministic procedures which use various heuristics for choosing the clauses to be resolved. The complexity of the resolution procedure is not known at present.

For a proof tree T using S and vertex i in T let C_i be the clause at i and T_i the subtree with root i. Obviously T_i is a proof tree for C_i using S. The root of a proof tree (for Λ) will always be denoted by 1. The height of T is the length of the longest path in T (i.e. between its root and one of the leaves).

Let T be a proof tree for some clause C. T is *regular* [8] if there is no vertex i in T such that C_i contains some literal which is annihilated in T_i. Note that a proof tree T is regular if and only if no path from a leaf to the root of T contains the annihilation of any variable more than once. *Regular resolution* is the procedure which nondeterministically constructs a regular resolution proof tree using S.

One should note that a set of clauses is unsatisfiable if and only if it has a regular proof tree. The "if" part is obvious. The "only if" part holds because the inductive proof of the completeness of resolution happens to construct a regular tree.

In [8], Tseitin introduced regular resolution. He described a method of associating contradictory sets of clauses with graphs. Tseitin investigated what happens in the graph when we walk along a regular proof tree using a set of clauses which corresponds to it. This enabled him to show, after choosing specific set of graphs, that regular resolution is exponential. Unfortunately his proof is formal, unintuitive, and does not contain most of the details. In [3] we give a simplified, hopefully intuitive, and detailed proof for a slightly improved version of Tseitin's main result.

We shall use Tseitin's method to generate contradictory sets of clauses. We shall apply it to a different set of graphs. Similarly to Tseitin we shall investigate what happens in the graph when we walk along a (not necessarily regular) proof tree using a set of clauses which corresponds to it. To that end, (i) we shall prove a correspondence between clauses on a regular tree and connected subgraphs of the given graph; and (ii) we shall prove an assertion made by Tseitin but without proof. These will enable us to prove our main result—a cn lower bound on the size of clauses in resolution trees. (n is the number of variables.)

We shall extensively use sets of clauses corresponding to graphs. We present below the notation which we use for dealing with graphs. All graphs will be connected, undirected, and without self loops. Thus a *graph* is a pair $\bar{G}=(\bar{V},\bar{E})$, where $\bar{V}$ is its set of vertices $(u, v, w \cdots)$ and $\bar{E}$ its set of edges (where an edge is an unordered pair (u, v), $u, v \in \bar{V}$, $u \neq v$). A graph $G=(V,E)$ is a *subgraph* of $\bar{G}$ if $V \subseteq \bar{V}$ and $E \subseteq \bar{E}$. (In the literature G is sometimes called a partial subgraph of $\bar{G}$.)

Let $G=(V,E)$ be a subgraph of $\bar{G}$. The *boundary of* G, F (or F_G when G is not clear from the context) is the set of edges of $\bar{G}$ which are incident with at least one vertex of G and do not belong to E. The *exterior boundary* (*interior boundary*) of G, F^{ext} (F^{int}), consists of those edges in the boundary of G which are incident with exactly one vertex (two vertices) of G. A graph $G=(V,E)$ is *bipartite* if $V=V_1 \cup V_2$, $V_1 \cap V_2=\phi$ and $E \subseteq \{(u,v) \mid u \in V_1, v \in V_2\}$. V_1 and V_2 will be called the two *sides* of G. A vertex is of *degree* k if there are exactly k edges incident with it. A graph is of *degree* k if all its vertices are of degree at most k.

The outline of the paper is as follows: In § 2 we introduce several versions of bounded resolution and iterated bounded resolution. We ask several questions about them and prove that all the versions of bounded resolution are equivalent. In § 3 we describe Tseitin's ingenious construction of sets of clauses associated with graphs. In § 4 we prove two general lemmas which will be used in the sequel. Although we have not been able to show that the unrestricted resolution procedure has an exponential lower bound, we believe that such lemmas as those in § 4 might be used to prove this result. In § 5 we describe a specific set of graphs and prove some of its properties. We shall use all these in § 6, where we prove the main theorem. Some easy corollaries of the main theorem settle the questions asked in § 2.

2. Bounded resolution. One way to force resolution procedure to run in polynomial time is to generate clauses only up to size k, where k is a given constant. (Note that there are at most $3^k\binom{n}{k} \leqq 5n^k$ distinct clauses of size up to k over n variables.) This approach was suggested to us by J. Hopcroft and this paper investigates the validity and complexity of several procedures which use this approach. We assume that the inputs are formulas in CF3. Otherwise, if one allows all formulas in CF, one cannot restrict attention to clauses of size $\leqq k$. For example, consider $S=\{x_1 \vee x_2 \vee \cdots \vee x_m, \bar{x}_1, \bar{x}_2, \cdots, \bar{x}_m\} m>k$. S is contradictory, but if we delete the first clause from S we get a satisfiable set of clauses.

We consider three versions of *bounded resolution* (BR): we assume that there is an integer k such that the size of each clause generated is bounded by k. All three versions use an operation RESOLVE(k) which *adds* to the current set of clauses α all those clauses of size $\leqq k$ obtained from two of the clauses in α by the resolution rule. Two of the versions use the operation EXPAND(k) which replaces each clause C of size less than k by all terms $\hat{C}$ of size k such that $C \subseteq \hat{C}$ (i.e. C *subsumes* $\hat{C}$). Let ALL(k) be a predicate which is true if and only if all terms of size k are present. The three routines are:[1]

[1] The numbers 1, 2, 3 denote positions in the programs. We shall refer to them below.

```
BR1(k):  repeat
           begin
             EXPAND(k)
             RESOLVE(k)
           end
         until no new clauses are generated;
      1
         if ALL(k) then output "unsatisfiable"
                   else output "satisfiable"

BR2(k):  repeat
           RESOLVE(k)
         until no new clauses are generated;
      2
         EXPAND(k);
      3
         if ALL(k) then output "unsatisfiable"
                   else output "satisfiable"

BR2(k):  repeat
           RESOLVE(k)
         until no new clauses are generated;
         if the empty clause is present
           then output "unsatisfiable"
           else output "satisfiable"
```

The three procedures discard clauses of size $>k$. BR1(k) was suggested by J. Hopcroft. It uses only clauses of size k exactly: Whenever a shorter clause is generated it is immediately replaced by all clauses of size k which are subsumed by it. BR2(k) is obtained from BR1(k) by a small modification. It allows also clauses shorter than k and the expansion to size k is executed only once—at the end. The third alternative (BR3(k)) is not to expand at all. Note that the empty clause is generated if and only if there exists a resolution proof tree with clauses of size $\leqq k$. Note also that the number of clauses generated by the three procedures is not uniform. But they all generate $O(n^k)$ distinct clauses, where n is the number of variables. Hence they all run in polynomial time. We shall say that a procedure works correctly if it outputs "satisfiable" iff the input is satisfiable. For each of these procedures we would like to know:

(I) Does the procedure work correctly for some constant integer k? and if not,

(II) Does it work correctly for $k = k(m)$, some slowly growing function of m, where m is the size of the input? (Obviously $k = n$ will do.)

A positive answer to (I) would imply $P = NP$. A positive answer to (II) would yield a subexponential algorithm for the tautology problem (and hence to all P-complete problems) provided $k(m)$ grows asymptotically slower than m^ε for every ε.

If k is not large enough, these procedures might give a wrong answer—they might decide that the input is satisfiable while it actually is unsatisfiable. The possibility of such an error can be detected. Obviously, if upon termination of

BR1 (BR2, BR3) all terms of size k are present (the empty clause is present), the input is unsatisfiable. Otherwise, if a new term can be generated (by the resolution rule) which is larger than k and does not contain any of the present clauses, one cannot conclude that the input is satisfiable, but rather that the bound k is too small. This leads to the *iterated bounded resolution* (IBR): IBR1 (IBR2, IBR3) is simply iterating BR1(k) (BR2(k), BR3(k)) with increasing k if the case mentioned above occurs.

These iterated bounded resolution procedures always give the correct answer. The question about them is:

(III) What is the time bound for the iterated bounded resolution procedures?

Example 2.1. Consider

$$\beta_{5,3}=\{x_1\vee x_2\vee x_3),(x_1\vee x_2\vee x_4),\cdots,(x_3\vee x_4\vee x_5),$$
$$(\bar{x}_1\vee\bar{x}_2\vee\bar{x}_3),(\bar{x}_1\vee\bar{x}_2\vee\bar{x}_4),\cdots,(\bar{x}_3\vee\bar{x}_4\vee\bar{x}_5)\}.$$

One can see immediately that $\beta_{5,3}$ is unsatisfiable, that BR(3) would not work and that BR(4) will work (where BR is any one of the versions).

Our attempts to answer questions (I)–(III) by first considering simple examples have failed. The seemingly simple question, "Is the bound $k=4$ sufficient for $n=6$?" (where n is the number of variables) was answered by J. Simon [7]. However, the proof is complicated and does not generalize.

Example 2.2. Let $\beta_{n,l}$ be a formula in CF over n variables which contains all clauses of size l such that the variables in any term are either all complemented or all noncomplemented. For every $n>0$ and $l\leqq\lceil n/2\rceil$,[2] $\beta_{n,l}$ is unsatisfiable since every assignment has either l zeros or l ones, and thus at least one of the clauses must have the value 0. Consider $\beta_{n,l}$, $l\leqq\lceil n/2\rceil$. Obviously a bound $k\geqq 2l-2$ is necessary for BR(k) to work for $\beta_{n,l}$. However, after using the usual trick of converting $\beta_{n,l}$ to CF3 it is no longer clear that a smaller bound would not suffice. Even if this method worked it would have settled question (I), but not questions (II) and (III), since the input is of length $m\approx\binom{n}{l}$ and the number of distinct clauses generated is $<l2^{2l}\binom{n}{2l-2}<m^4$ since one can show that the bound $k=2l-2$ suffices.

In Theorem 2.1, we prove the equivalence of BR1, BR2, and BR3. This will enable us to speak about BR which will stand for any one of them.

THEOREM 2.1. *For a fixed k, all proposed bounded resolution versions give the same answer to any given input.*

Proof. First we show that BR1 and BR2 are equivalent. Let term$_1$, term$_2$ and term$_3$ be the clauses present at points 1, 2 and 3 in BR1 and BR2. One shows that (i) for very $\hat{C}$ in term$_1$ there is $C\in$ term$_2$ such that $C\subseteq\hat{C}$; and (ii) for every C in term$_2$ every $\hat{C}$ of size k such that $C\subseteq\hat{C}$ is in term$_1$. This will imply (by definition of EXPAND(k)) that term$_1$ = term$_3$ and will complete the proof. Parts (i) and (ii) may be proved by induction on the depth of the resolution.

[2] $\lceil n/2\rceil$ is the smallest integer greater than or equal to $n/2$.

Now we show that BR2 and BR3 are equivalent. We prove that $\Lambda \in \text{term}_2$ if and only if ALL(k) is true for term$_3$. Since up to point 2 BR2 and BR3 are the same, this completes the proof.

If $\Lambda \in \text{term}_2$, then ALL($k$) is true for term$_3$ by definition of EXPAND(k). To prove the converse we need the following definition: A set of terms satisfies ALL$'(l)$ if, for every clause $\hat{C}$ of size l, there is a term C in the set such that $C \subseteq \hat{C}$. If ALL(k) is true for term$_3$, than ALL$'(k)$ is true for term$_2$. One then shows that if ALL$'(i)$, $i > 0$, is true for a set of clauses, then ALL$'(i-1)$ is true for the set obtained by applying RESOLVE(k). But RESOLVE(k) does not change term$_2$. Thus ALL$'(0)$ must be true for term$_2$, i.e. $\Lambda \in \text{term}_2$. For details see [3]. □

In our main theorem we shall prove a cn lower bound on the size of clauses generated by resolution proofs. An immediate corollary will settle questions (I) and (II) above in the negative. Another corollary will show that the time bound for IBR $\geqq 2^{cn}$ (answering question (III)). Note that all known procedures that check whether a given set of clauses is contradictory are exponential. We do not know how IBR performs in comparison to the other procedures. Note also that BR(k) can still be used as a heuristic for showing that a set of clauses is contradictory (but not for showing that it is satisfiable). Thus, our main theorem gives examples where using this heuristic fails. We do not know how BR(k) compares with other heuristics for the same problem.

3. Sets of clauses corresponding to graphs. In this section we describe Tseitin's method of associating contradictory sets of clauses with graphs. The definitions and results are due to Tseitin.

Let $G = (V, E)$ be a given undirected graph without loops (i.e. there is no edge (v, v)). Let l be a labeling which assigns to every vertex v a label $\varepsilon_v \in \{0, 1\}$, the *parity of* v, and each edge a literal in such a way that no two edges are labeled by a literal corresponding to the same variable. $G(l)$ will stand for the labeled graph.

Consider a vertex v in V. Assume that v has degree k and $z_1, z_2, \cdots, z_k$ are the literals which label the edges incident with v. A clause C corresponds to the vertex v if and only if

$$C = z_1^{a_1} \vee \cdots \vee z_k^{a_k}$$

and the number of complemented z_i's is of parity opposite to the parity of v i.e.

$$\sum_{i=1}^{k} (1 - a_i) \neq \varepsilon_v$$

which will be referred to as the *parity condition with respect to* v.[3] Note that there are exactly 2^{k-1} distinct clauses which correspond to v.

LEMMA 3.1. *The assignment δ given by $z_1 = \delta_1, \cdots, z_k = \delta_k$ satisfies all the clauses corresponding to v if and only if $\sum_{i=1}^k \delta_i = \varepsilon_v$.*

Proof. If $\sum_{i=1}^k \delta_i \neq \varepsilon_v$, then $C = z^{1-\delta_1} \vee \cdots \vee z_k^{1-\delta_k}$ corresponds to v and is not satisfied by the assignment δ.

[3] Throughout this paper, $\sum$ and $+$ denote sum modulo 2.

Assume $\sum_{i=1}^{k} \delta_i = \varepsilon_v$ and let $C = z_1^{a_1} \vee \cdots \vee z_k^{a_k}$ be an arbitrary clause corresponding to v. The only assignment which does not satisfy C is $z_1 = 1 - a_1, \cdots, z_k = 1 - a_k$. Since $\sum_{i=1}^{k}(1-a_i) \neq \varepsilon_v$ it must differ from the assignment δ. Thus δ must satisfy C. □

We denote by $\alpha(G(l))$ the set of all clauses which correspond to all vertices in G (l is the labeling).

Claim. $\alpha(G(l))$ is unchanged if we apply the following transformations to the labeling:

Transformation 1. Change exactly one edge label (from z to $\bar{z}$) and simultaneously change the vertex labels of its endpoints.

Transformation 2. Change all edge labels along a path from vertex u to vertex v and simultaneously change ε_u and ε_v.

Transformation 2 is a sequence of applications of Transformation 1. The latter does not affect $\alpha(G(l))$ since if a clause C corresponds to a vertex w under the old labeling it does so under the new labeling: for $w \notin \{u, v\}$ this is obvious since nothing has changed; for $w = u$ $(w = v)$ the change of ε_u (ε_v) and of z to $\bar{z}$ implies a double change in parity. Hence C satisfies the parity condition with respect to u (v) under the new labeling.

Let $\varepsilon(G(l)) = \sum_{v \in V} \varepsilon_v$.

LEMMA 3.2. *For a connected graph G and a labeling l, $\alpha(G(l))$ is satisfiable if and only if $\varepsilon(G(l)) = 0$.*

Proof. Assume $\varepsilon(G(l))$ is satisfiable. Using Lemma 3.1 and summing (mod 2) all ε_v's we get $\varepsilon(G(l)) = 0$, since every δ_i appears twice (once for each endpoint of the edge labeled by z_i).

Assume $\varepsilon(G(l)) = 0$. Using Transformation 2 we can set all vertex labels to 0. If the new edge labels are $z_1', \cdots, z_n'$ the assignment $z_i' = 0$ for $1 \leqq i \leqq n$ satisfies $\alpha(G(l))$ (by Lemma 3.1). □

From now on we consider a fixed connected graph $\bar{G} = (\bar{V}, \bar{E})$ with a fixed labeling l such that $\varepsilon(\bar{G}(l)) = 1$. Thus $\alpha(\bar{G}(l))$ is unsatisfiable. We will refer to $\alpha(\bar{G}(l))$ as the original set of clauses. We will identify the edges and their labeling if no confusion will arise.

4. Two general lemmas.

In this section we prove two general lemmas: (i) the correspondence lemma, which states the correspondence between operations on a graph $\bar{G}$ and a regular derivation using the clauses corresponding to $\bar{G}$, (ii) the regularity lemma, which states that if we can prove a given clause C by a resolution tree, then we can prove a clause C', $C' \subseteq C$, by a regular tree. The latter was stated without proof by Tseitin. These two lemmas will be used below for proving the main theorem. We still hope that by using these lemmas one will be able to prove an exponential lower bounds for the unrestricted resolution procedure.

Let $G = (V, E)$ be a connected subgraph of $\bar{G}$. A clause C is *associated with* G if its variables are all the labels of the edges in the boundary of G. If C is associated with G, then the *parity of* C, $\delta(C)$, is the parity of the number of complemented variables in C which correspond to the exterior boundary of G, i.e. if $C = z_1^{a_1} \vee \cdots \vee z_k^{a_k}$, then $\delta(C) = \sum_{z_i \in F^{\text{ext}}}(1 - a_i)$. The *parity of* G, $\varepsilon(G)$, is defined to be the sum modulo 2 of the vertex labels of G. C is a *regular clause associated with* G if C is associated with G and the parity of C is opposite the parity of G, i.e.

$\delta(C) \neq \varepsilon(G)$. C is a *regular clause* if there is a connected subgraph G of $\bar{G}$ such that C is a regular clause associated with G.

Example. Consider the subgraph $G=(V, E)$ of $\bar{G}$ in Fig. 4.1. V consists of the circles and E consists of the dotted lines. Assume that all vertex labels are 0; thus $\varepsilon(G)=0$, $F^{\text{int}}=\{a\}$ and $F^{\text{ext}}=\{x, y, z\}$. Let $C_1=a \vee x \vee y \vee z$, $C_2=\bar{a} \vee x \vee y \vee z$ and $C_3=a \vee x \vee y \vee \bar{z}$. All three clauses are associated with G, but only C_3 is a regular clause since $\delta(C_3)=1$ while $\delta(C_1)=\delta(C_2)=0$.

One can immediately make the following observation: If C is one of the original clauses which corresponds to a vertex v in $\bar{G}$, then C is a regular clause associated with the subgraph $G=(v, \phi)$ consisting of an isolated vertex. (We identify the vertex v and the singleton $\{v\}$.)

The next lemma proves that if T is a regular proof tree for a clause C, then C is a regular clause. The lemma specifies exactly the connected subgraph G with which C is associated.

LEMMA 4.1 (the correspondence lemma). *If T is a regular proof tree for a clause C, then C is a regular clause associated with some connected subgraph $G=(V, E)$ of $\bar{G}$ where $V=\{v \mid v \in \bar{V}$ and a clause corresponding to v appears on a leaf of $T\}$ and $E=\{x \mid x$ is annihilated in $T\}$.*

Proof. Proof is by induction on the height h of T. The basis of the induction ($h=0$) follows immediately from the observation preceding the lemma.

Induction step. Let T be the tree in Fig. 4.2 and assume $x \vee C_1 (\bar{x} \vee C_2)$ is a regular clause associated with $G_1 (G_2)$ which satisfies the hypothesis of the lemma.

Without loss of generality there are four cases. (For $i=1, 2$ $G_i=(V_i, E_i)$, F_i^{ext}, F_i^{int} are defined in the obvious way.)

1. $G_1=G_2$ (the subgraphs are identical).
2. $V_1 \cap V_2=\phi$ (they are vertex disjoint).
3. $V_1=V_2$ and there is $z \in E_1-E_2$.
4. $V_1 \cap V_2 \neq \phi$ and there is $u \in V_1-V_2$.

Claim. Cases 3 and 4 are not possible.

Proof. We show that in both cases $E_1 \cap F_2 \neq \phi$, i.e. there is a variable z annihilated in T_1 such that either z or $\bar{z}$ appears in $C_2 \vee \bar{x}$—a contradiction since T is regular. For case 3 it is obvious that $z \in E_1 \cap F_2$. For case 4 let $v \in V_1 \cap V_2$. G_1 is connected; thus there is a path in G_1 from u to v. There must be an edge z on this path such that one of its endpoints is in V_2 and one is not. Obviously $z \in E_1 \cap F_2$. □

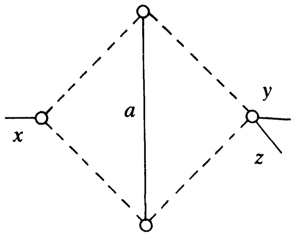


FIG. 4.1. *A subgraph G of G and all edges of G which are incident with its vertices*

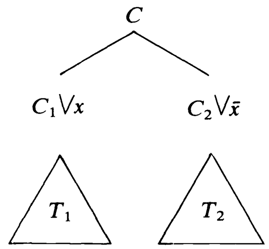


FIG. 4.2

To complete the proof of the lemma we now consider cases 1 and 2.

Case 1. $G_1 = G_2$ and thus $F_1 = F_2$. $x \vee C_1$ and $\bar{x} \vee C_2$ consist of the same variables and have exactly one conflict (x vs. $\bar{x}$); thus $C_1 = C_2 = C$. Since the parity of $C_1 \vee x$ and $C_2 \vee \bar{x}$ both differ from the parity of $G_1 = G_2$, they must be the same. Hence $x \in F_1^{\text{int}} = F_2^{\text{int}}$. Let $G = (V_1, E_1 \cup x)$. It is easy to check that the pair (C, G) satisfies the induction hypothesis. The parity of G differs from the parity of C since $\varepsilon(G) = \varepsilon(G_1)$ and $\delta(C) = \delta(C_1 \vee x)$.

Case 2. $V_1 \cap V_2 = \phi$. Let $G = (V_1 \cup V_2, E_1 \cup E_2 \cup x)$. Obviously, G is connected, C is associated with G, $E = E_1 \cup E_2 \cup x$ consists exactly of the variables annihilated in T and $V = V_1 \cup V_2$ consists exactly of those vertices which correspond to the clauses on the leaves of T. To complete the proof that the pair (C, G) satisfies the induction hypothesis we now show that G has parity opposite to that of C. Since $\delta(C_1 \vee x) \neq \varepsilon(G_1)$ and $\delta(C_2 \vee x) \neq \varepsilon(G_2)$, $\delta(C_1 \vee x) + \delta(C_2 \vee \bar{x}) = \varepsilon(G_1) + \varepsilon(G_2) = \varepsilon(G)$. We show below that

$$(1) \qquad \delta(C_1 \vee x) + \delta(C_2 \vee \bar{x}) \neq \delta(C)$$

which will imply that $\delta(C) \neq \varepsilon(G)$ and will complete the proof of the lemma.

To prove (1), recall that if $C = z_1^{a_1} \vee \cdots \vee z_r^{a_r}$, then $\delta(C) = \sum_{z_i \in F^{\text{ext}}} (1 - a_i)$. But $F_1^{\text{ext}} \cap F_2^{\text{ext}} \equiv x \cup F_{12}$ and $F_{12} \subseteq F^{\text{int}}$. (See Fig. 4.3.) In $\delta(C_1 \vee x) + \delta(C_2 \vee \bar{x})$ the terms corresponding to F^{ext} appear exactly once, the terms for F_{12} appear exactly twice (hence they can be dropped) and since x appears once as x and once as $\bar{x}$, $\delta(C_1 \vee x) + \delta(C_2 \vee \bar{x}) = \delta(C) + 0 + 1$ and (1) follows. □

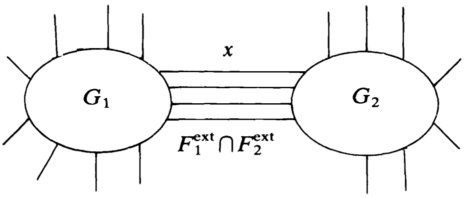


FIG. 4.3

We now show that every clause in every proof tree contains a regular clause.

LEMMA 4.2 (the regularity lemma). *Let T' be a proof tree for the clause C'. Then there is a $C \subseteq C'$ and a regular proof tree T for C.*

Proof. Proof is by induction on the height of T'. The basis of the induction is trivially true.

Induction step: Let T' be the tree shown in Fig. 4.4. Apply the induction hypothesis to T_1' (T_2') to obtain a regular tree T_1'' (T_2'') which proves $\hat{C}_1(\hat{C}_2)$ with $\hat{C}_1 \subseteq x \vee C_1'(\hat{C}_2 \subseteq \bar{x} \vee C_2')$.

If $x \to \hat{C}_1$ ($\bar{x} \notin \hat{C}_2$) then $T = T_1''$ ($T = T_2''$) and $C = \hat{C}_1$ ($C = \hat{C}_2$) satisfy the hypothesis of the lemma. Otherwise, let $\hat{C}_1 = C_1'' \vee x$ and $\hat{C}_2 = C_2'' \vee \bar{x}$ and let T'' be the tree shown in Fig. 4.5. T'' need not be regular because a literal which appears in $C_1'' \vee x$ ($C_2'' \vee \bar{x}$) might be annihilated in T_2'' (T_1''). So let

$$P = \{z \mid z \in C_1'', z \text{ is annihilated in } T_2''\}$$
$$\cup \{z \mid z \in C_2'', z \text{ is annihilated in } T_1''\}.$$

If $P = \phi$ then T'' is regular and $T = T''$ and $C = C''$ will do. Otherwise, we eliminate in T'' all annihilations of z's in P by deleting corresponding subtrees. T'' decomposes into a collection of separate subtrees and we construct from them a new tree T by performing all possible annilations.

More precisely, we construct T by the following inductive process: If T_i'' consists of a single vertex, then $T_i = T_i''$. If T_i'' is the tree shown in Fig. 4.6, then

(i) If $z \in P$ ($\bar{z} \in P$), then $T_i = T_{i_1}$ ($T_i = T_{i_2}$), i.e., the other subtree is deleted to

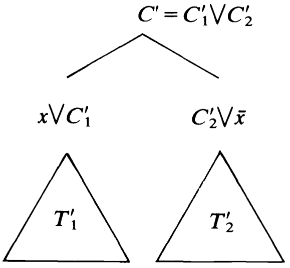


FIG. 4.4

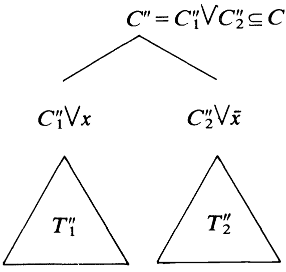


FIG. 4.5

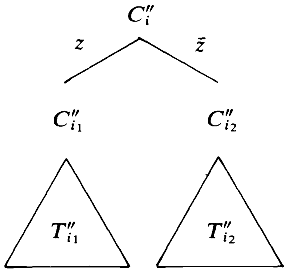


FIG. 4.6

prevent annihilation; otherwise

(ii) If $z \notin C_{i_1}$ $(\bar{z} \notin C_{i_2})$, then $T_i = T_{i_1}$ $(T_i = T_{i_2})$, i.e., the other subtree is deleted since we do not need to annihilate z in the new tree; otherwise

(iii) T_i is obtained by combining T_{i_1} and T_{i_2} (i.e., by resolving C_{i_1} and C_{i_2}).

Let $T = T_1$ (where 1 is the root) and $C = C_1$. For i in T, an easy induction on the height of T_i proves that

(a) T_i is regular,

(b) $C_i \subseteq C_i'' \vee P$. (Recall that a clause is a set of literals.)

Thus T is regular and $C \subseteq C'' \subseteq C'$, since $P \subseteq C''$. Hence T and C satisfy the required properties. □

Example 4.1. Figure 4.7 illustrates the induction step in the proof of Lemma 4.3. Although the two main subtrees of T' are regular T' itself it not regular

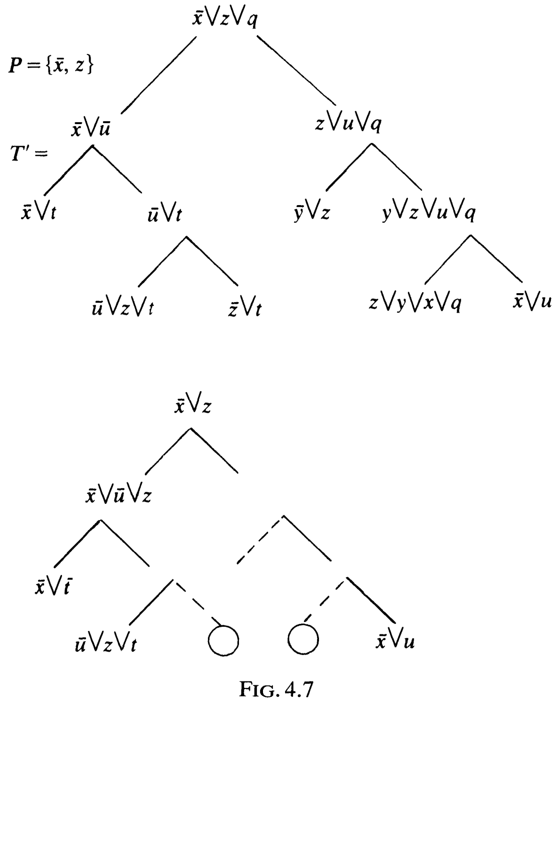


FIG. 4.7

$(P \neq \phi)$. The circled subtrees are eliminated to prevent annihilation of literals in P ((i) above). The third subtree is eliminated because y does not appear in the corresponding clause ((ii) above). As expected (a) and (b) above hold. Note that $\bar{x} \vee \bar{u} \vee z$ is longer than its corresponding clause in $T'(\bar{x} \vee \bar{u})$.

Tseitin used the regularity lemma to show that every nonregular proof tree T can be transformed to a smaller regular tree T'. However T' might have many more distinct clauses than T. Hence, the regularity lemma and the fact that regular resolution is exponential do not imply that resolution is exponential.

5. The specific collection of graphs. We consider a graph $\bar{G} = (\bar{V}, \bar{E})$ which depends on a parameter m and hence it actually represents a collection of graphs. When speaking on constants (like c, d below) we shall mean constants with respect to m. $\bar{G}$ is defined by a transformation on a graph $H_m = (\tilde{V}, \tilde{E})$ which was introduced by Margulis [5]. We do not describe H_m below. We only mention some of its properties. H_m is a bipartite graph of degree 5, such that each of its sides contains m^2 vertices. Theorem 5.1 below is a rewording of a special case of Margulis' main theorem. (Namely, taking $\alpha = \frac{1}{2}$ in Theorem 2.3 in [5].)

THEOREM 5.1. *There is a constant $d > 1$, such that if $\hat{V}_1$ is contained in one side of H_m $|\hat{V}_1| \leqq m^2/2$, and $\hat{V}_2$ consists of all the vertices in the other side of H_m which are connected to vertices of $\hat{V}_1$ by an edge, then $|\hat{V}_2| \geqq d|\hat{V}_1|$.*

COROLLARY 5.1. *There is a constant $c > 0$, such that if $G = (V, E)$ is a subgraph of H_m and $m^2/4 \leqq |V| < m^2/2$, then $|F_G^{\text{ext}}| \geqq cm^2$.*

Proof. Assume $V = V_1 \cup V_2$ and $V_1(V_2)$ consists of vertices in the first (second) side of H_m. Without loss of generality $|V_1| \geqq |V_2|$. Let $\hat{V}_1 = V_1$ and $\hat{V}_2$ be as in Theorem 5.1. It follows that $|\hat{V}_2| \geqq d|\hat{V}_1|$. Hence,

$$|F_G^{\text{ext}}| \geqq |\hat{V}_2| - |V_2| \geqq (d-1)|V_1| \geqq \frac{d-1}{2}|V| \geqq \frac{d-1}{8}m^2. \qquad \square$$

Using a trick which was introduced by Kirkpatrick [4], we transform H_m to a graph of degree 3 to obtain $\bar{G}$: if v is a vertex in H_m of degree l $(3 \leqq l \leqq 5)$ it is represented in $\bar{G}$ by a cycle of l vertices $v^{(1)}, \cdots, v^{(l)}$. If (u, v) is the ith (jth) edge emerging from u (v) in H_m, it is represented by the edge $(u^{(i)}, v^{(j)})$ in $\bar{G}$. (See Fig. 5.1.)

Let G be a connected subgraph of $\bar{G}$. $\tilde{G}$ is the *projection* of G in H_m if its vertices and edges correspond to those of G. ($\tilde{G}$ is obtained from G by shrinking the cycles which were introduced in the definition of $\bar{G}$.) Obviously $\tilde{G}$ is a connected subgraph of H_m.

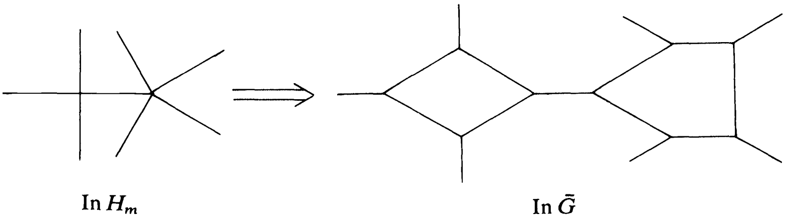


FIG. 5.1. *The transformation from H_m to $\bar{G}$*

We take a labeling l such that $\varepsilon(\bar{G}(l)) = 1$, and $\alpha(\bar{G}(l))$ is our set of clauses. (It is contradictory by Lemma 3.2). Obviously $n = O(m^2)$, and $\alpha(\bar{G})$ contains $O(n)$ clauses of size 3.

LEMMA 5.1. *Let T be a tree such that each vertex i in T is labeled by $G_i = (V_i, E_i)$, a connected subgraph of $\bar{G}$. Assume that* (i) $G_1 = \bar{G}$, (ii) $|V_j| = 1$ *if j is a leaf and* (iii) $V_i \subseteq V_{i_1} \cup V_{i_2}$ *if i_1 and i_2 are the sons of i. Then, there is a vertex i in T, such that $|F_{G_i}^{\text{ext}}| \geqq cn$ for some $c > 0$.*

Proof. Let $\tilde{G}_i = (\tilde{V}_i, \tilde{E}_i)$ be the projection of G_i in H_m. By (iii) we can go along a path from the root 1 to a leaf, choosing the son j' of j with $|\tilde{V}_{j'}| \geqq |\tilde{V}_j|/2$. So $|\tilde{V}_j|$ decreases from $2m^2$ to 1 and never decreases by more than half in any one step. Thus, there must be an i on the path with $m^2/4 \leqq |\tilde{V}_i| < m^2/2$. By Corollary 5.1, $|F_{\tilde{G}_i}^{\text{ext}}| \geqq cm^2 \geqq c'n$. But if any one of the edges which is incident with v is in $F_{\tilde{G}_i}^{\text{ext}}$, then at least one edge which is incident with some $v^{(j)}$ is in $F_{G_i}^{\text{ext}}$. Hence $|F_{\tilde{G}_i}^{\text{ext}}| \geqq |F_{\tilde{G}_i}^{\text{ext}}|/5 \geqq c''n$. □

6. The main theorem.

THEOREM 6.1. *For infinitely many $n \geqq 1$, there are unsatisfiable formulas α_n in* CF3 *over n variables which contain $O(n)$ clauses such that every resolution tree using α_n contains clauses of size $> cn$ for some $c > 0$.*

Note that a theorem similar to Theorem 6.1 concerning regular resolution follows from Lemma 5.1 and the correspondence lemma. By applying the regularity lemma to a given proof tree we obtain a regular proof tree. But the latter can contain longer clauses (as in Example 4.1) and thus Theorem 6.1 does not follow immediately from the regularity lemma. Yet, the theorem does follow from careful application of the regularity lemma and its proof.

Proof. Let α_n be the set of clauses which corresponds to the graph $\bar{G}$, and let T' be an arbitrary resolution proof tree using α_n. By the regularity lemma we can associate with each vertex i in T' a regular clause $C_i \subseteq C_i'$. Let $G_i = (V_i, E_i)$ be the subgraph of $\bar{G}$ which correspond to C_i by the correspondence lemma. ($V_i = \{v \mid$ a clause corresponding to v appears on a leaf of T_i, the regular tree for $C_i\}$.) Consider T' and the subgraphs $\{G_i\}$. By the proof of the regularity lemma, the conditions for Lemma 5.1 hold. By the conclusion of Lemma 5.1, there is a vertex i with $|F_i^{\text{ext}}| \geqq cn$. Hence, $|C_i'| \geqq |C_i| \geqq |F_i^{\text{ext}}| \geqq cn$. □

Theorem 6.1 answers questions (I) and (II) about the bounded resolution procedures:

COROLLARY 6.1. BR(k) *works for α_n only if $k \geqq cn$.*

It also answers question (III) about that iterated bounded resolution procedures:

COROLLARY 6.2. *The run-times of* IBR1, IBR2 *and* IBR3 *are all greater than 2^{cn}.*

Proof. Corollary 6.1 for BR3 follows from Theorem 6.1. But by Theorem 2.1, the same holds for BR1 and BR2. Corollary 6.2 is obvious for IBR1 and IBR2 since 2^{cn} is a lower bound for the number of distinct clauses of size cn. Also, it is easy to see that if IBR3 generates a regular clause associated with some subgraph G, it must generate all regular clauses associated with G. Since a G with $|F_G^{\text{ext}}| > cn$ must exist, at least 2^{cn} distinct clauses are generated by IBR3. □

7. Concluding remarks. For three literals x, y, z and a Boolean function f over 2 variables, let $\alpha(f, z, x, y)$ be the set of clauses over x, y, and z which is equivalent to $z \equiv f(x, y)$. One can verify that $\alpha(f, z, x, y)$ exists for all 16 possible choices of f.

Example.

$$\alpha(\vee, z, x, y)=\{\bar{x}\vee z), (\bar{y}\vee z), (\bar{z}\vee x\vee y)\}$$

$$\alpha(+, z, x, y)=\{(\bar{x}\vee y\vee z), (x\vee \bar{y}\vee z), (x\vee y\vee \bar{z}), (\bar{x}\vee \bar{y}\vee \bar{z})\},$$

$$\alpha(U_1, z, x, y)=\{(\bar{z}\vee x), (\bar{x}\vee z)\} \quad \text{where } U_1(x, y)=x.$$

The *extension rule* [8] is applied by choosing f, x and y, creating a new variable z, and adding $\alpha(f, z, x, y)$.

We saw that allowing only clauses of size up to a certain constant may prevent the procedure from working correctly. The question which naturally arises is: What happens if we consider bounded resolution with extension. The following theorem answers this question. Its proof appears in the Appendix.

THEOREM 7.1. *Let T be a resolution proof tree (with or without extension). Let $N=N(T)$ be the number of distinct clauses, and m the number of distinct variables in T. Suppose the original set of clauses has at most l literals per clause. Then, using extension, we can transform T into a proof tree T' with clauses of size $\leqq l$, T' contains at most $O(Nm)$ distinct clauses.*

Hence if the input S is in CF3, then every resolution proof tree using S can be transformed into another tree which uses extension and with clauses of size $\leqq 3$. The latter, however, does not necessarily contain a number of clauses which is bounded by a polynomial in the number of the variables in S, since the number of variables which are introduced by the extension rule may be very large.

We conclude by considering Fig. 7.1 which considers four related problems: Problem 1—showing a cn lower bound on the size of clauses in regular resolution. It follows directly from the correspondence lemma and Lemma 5.1. Problem 2—the analogous problem for unrestricted resolution was the subject of this paper. Problem 3—showing exponential lower-bound for regular resolution was settled by Tseitin. Problem 4—getting a lower-bound on the number of distinct clauses in resolution is still open. We hope that the techniques used to solve Problems 2 and 3 will help to settle Problem 4.

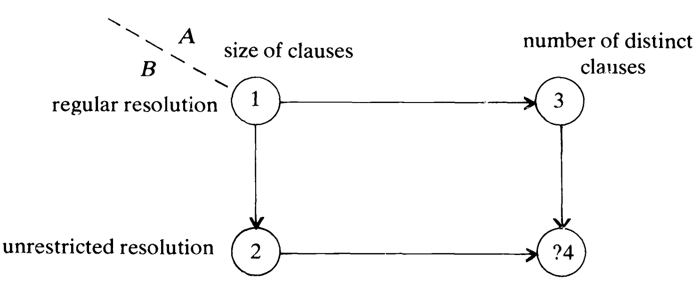
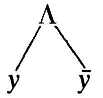


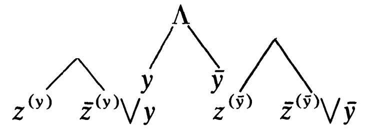
FIG. 7.1. *Lower bounds on A using B*

Appendix.

Proof of Theorem 7.1. Without loss of generality $l \geqq 3$. For every clause C in T we have a new variable $z^{(C)}$ such that $z^{(C)} \equiv C$. Using additional variables, we show that if E is derived from C and D in T, then given trees for $z^{(C)}$ and $z^{(D)}$, we can derive $z^{(E)}$. We also indicate how to derive $z^{(C)}$ for every input clause C. If the top of T looks like



then it follows by induction from the claim above that we can derive $z^{(y)}$ and $z^{(\bar{y})}$. But $z^{(y)} \equiv y$ and $z^{\bar{y}} \equiv \bar{y}$, and using the available clauses expressing these facts we get the top of T':



Since we shall use only clauses of size $\leqq l$ this will complete the construction of T'.

Let $x_1, x_2, \cdots, x_m$ be the variables used in T. Assume $C = y_{i_1} \vee \cdots \vee y_{i_r}$ where $1 \leqq i_1 < \cdots < i_r \leqq m$. Let $C_j \equiv y_{i_1} \vee \cdots \vee y_{i_j}$ $(C_r = C)$. Thus,

(A.1a) $$z^{(C_1)} \equiv y_{i_1}$$

and

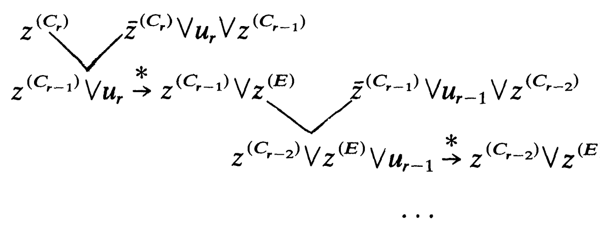
(A.1b) $$z^{(C_j)} \equiv z^{(C_{j-1})} \vee y_{i_j}, \qquad 1 < j \leqq r.$$

As in the example in § 7, we add the clauses corresponding to (A.1) and call them the *additional clauses.*

We now show how to obtain $z^{(E)}$ from $z^{(C)}$ and $z^{(D)}$. Let $C = u_1 \vee \cdots \vee u_r$, $D = v_1 \vee \cdots \vee v_s$, x is u_p, $\bar{x}$ is v_q. Assume $\bar{z} \vee z'$, is an additional clause. $C \vee z \rightarrow C \vee z'$ will stand for deriving $C \vee z'$ from $\bar{z} \vee z'$ and $C \vee z$, and let $\overset{*}{\rightarrow}$ be the transitive closure of $\rightarrow$. Note that if $w \in E$, then $w \vee C' \overset{*}{\rightarrow} z^{(E)} \vee C'$: If w is the jth literal in E then,

$$w \vee C' \rightarrow z^{(E_j)} \vee C' \rightarrow z^{(E_{j+1})} \vee C' \rightarrow \cdots \rightarrow z^{(E)} \vee C'.$$

(The clauses $\bar{w} \vee z^{(E_j)}$, $\bar{z}^{(E_j)} \vee z^{(E_{j+1})}, \cdots$ are additional clauses.)


$z^{(C_r)}$
$\bar{z}^{(C_r)} \vee u_r \vee z^{(C_{r-1})}$
$z^{(C_{r-1})} \vee u_r \overset{*}{\rightarrow} z^{(C_{r-1})} \vee z^{(E)}$
$\bar{z}^{(C_{r-1})} \vee u_{r-1} \vee z^{(C_{r-2})}$
$z^{(C_{r-2})} \vee z^{(E)} \vee u_{r-1} \overset{*}{\rightarrow} z^{(C_{r-2})} \vee z^{(E)}$
$\cdots$

since $u_r, u_{r-1} \cdots \in E$.

Thus, by repeating the above we derive $z^{(C_{p-1})}\vee z^{(E)}\vee x$ and $z^{(D_{q-1})}\vee z^{(E)}\vee \bar{x}$. We resolve them to get $z^{(C_{p-1})}\vee z^{(D_{q-1})}\vee z^{(E)}$. If we continued as before we would get clauses of size 4. So, we define a new variable $\hat{z}\equiv z^{(D_{q-1})}\vee z^{(E)}$. We use the new additional clauses to derive $z^{(C_{p-1})}\vee z^{(D_{q-1})}\vee z^{(E)}\overset{*}{\to} z^{(C_{p-1})}\vee \hat{z}$. Then as before we continue the decomposition to get $z^{(C_{p-2})}\vee z^{(E)}\vee \hat{z}$; then "substituting" back $\hat{z}$ we get

$$\begin{array}{ccc} z^{(C_{p-2})}\vee z^{(E)}\vee \hat{z} & & \bar{\hat{z}}\vee z^{(E)}\vee z^{(D_{q-1})} \\ & \diagdown\;\diagup & \\ & z^{(C_{p-2})}\vee z^{(E)}\vee z^{(D_{q-1})}. & \end{array}$$

This process (using new $\hat{z}$'s) is repeated. The reader can easily see that $z^{(E)}$ is obtained at the end.

For an input clause C one can easily verify that $z^{(C)}$ can be derived using C and the additional clauses ($C \overset{*}{\to} z^{(C)}$).

Obviously, all the clauses on the tree T' are of size $\leqq l$ and $N(T')=O(m\cdot N)$. □

Acknowledgments. I wish to thank my thesis advisor Professor J. Hopcroft and Professor J. Hartmanis for their encouragement, advice and suggestions. Professor Hopcroft has also suggested BR1 of § 2. Thanks also to J. Simon and W. Paul for many discussions, and to P. van Emde Boas and M. Solomon for help with the presentation. I am grateful to N. Pippenger for advising me on Margulis' paper, which helped me to obtain a higher bound in Theorem 6.1. Finally, I would like to thank an anonymous referee for many suggestions and for simplifying considerably a previous version of the proof of the main theorem.

REFERENCES

[1] M. BAUER, D. BRAND, M. J. FISCHER, A. R. MEYER AND M. S. PATERSON, *A note on disjunctive form tautologies*, SIGACT News, 5 (1973), April, pp. 17–20.

[2] S. A. COOK, *The complexity of theorem proving procedures*, Proc. 3rd ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1971, pp. 151–158.

[3] Z. GALIL, *The complexity of resolution procedures for theorem proving in the propositional calculus*, Ph.D. thesis, Tech. Rep. 75-239, Cornell University, Ithaca, NY, May 1975.

[4] D. G. KIRKPATRICK, *Topics in the complexity of combinatorial algorithms*, Ph.D. thesis, Dept. of Computer Sci., Tech. Rep. 74, Univ. of Toronto, December 1974.

[5] G. A. MARGULIS, *Explicit construction of concentrators*, Problems of Information Transmission 9, 1973, pp. 325–332.

[6] J. A. ROBINSON, *A machine oriented logic based on the resolution principle*, J. Assoc. Comput. Mach., 12 (1965), pp. 23–41.

[7] J. SIMON, personal communication.

[8] G. S. TSEITIN, *On the complexity of derivations in the propositional calculus*, Structures in Constructive Mathematics and Mathematical Logic, Part II, A. O. Slisenko, ed., Consultants Bureau, New York, 1970, pp. 115–125.

SIAM J. COMPUT.
Vol. 6, No. 3, September 1977

THE TAPE COMPLEXITY OF SOME CLASSES OF SZILARD LANGUAGES*

Y. IGARASHI†

Abstract. In this paper we discuss lower and upper bounds for the tape complexity of Turing machines which recognize some classes of Szilard languages. The main results are as follows: log n is the optimal tape bound for on-line deterministic Turing machines which recognize Szilard languages of context-free grammars, and it is also the optimal tape bound for off-line deterministic Turing machines which recognize leftmost Szilard languages of phrase structure grammars. The Szilard language of an arbitrary phrase structure grammar is a deterministic context-sensitive language.

Key words. associate language, computational complexity, context-free grammar, derivation language, label language, phrase structure grammar, Szilard language, tape-complexity, Turing machine

Introduction. There seems to be no agreement on the precise name of the concept which is discussed in this paper. The term "Szilard language" is used in [6] and [14], and in [3], [10] and [12] the terms "label language", "associate language" and "derivation language" are respectively used as synonyms for "Szilard language".

There are several interesting observations to make about Szilard languages. For example, the study of grammars with control sets [4] can be considered as a study of Sz(G) ∩ C, where G is a grammar, Sz(G) is the Szilard language of G and C is a control set. Another interesting example, which is relevant to many areas in theoretical computer science, is the reachability problem for vector-addition systems. Partial solutions to this problem have been obtained [2], [8], but the general problem which was proposed by Karp and Miller [7] is still unsolved. The reachability problem can be reduced to the decision problem of the emptiness for Sz(G) ∩ R, where G is a context-free grammer and R is a regular set. We can also consider Szilard languages to be useful tools in the study of extensions of context-free grammars such as matrix grammars [1], [6], [14] and programmed grammars [13], [14]. Some properties of Szilard languages have been already studied [3], [6], [10], [12], [14]. However, no references on their computational complexity seem to exist. It would therefore seem that further investigation of Szilard languages from a computational complexity viewpoint is required.

In this paper we describe the following results:

1) log n is the optimal tape bound for on-line deterministic Turing machines which recognize Szilard languages of context-free grammars, and it is also the optimal tape bound for off-line deterministic Turing machines which recognize leftmost Szilard languages of phrase structure grammars.
2) n is the optimal tape bound for on-line deterministic Turing machines which recognize leftmost Szilard languages of context-free grammars, and it is also the optimal tape bound for on-line deterministic Turing machines which recognize leftmost Szilard languages of phrase structure grammars.

(The above n and log n are also the optimal bounds for the corresponding problems for nondeterministic Turing machines.)

*Received by the editors June 16, 1975, and in revised form July 22, 1976.

†Centre for Computer Studies, University of Leeds, Leeds, LS2 9JT, England.

3) The Szilard language of an arbitrary phrase structure grammar is a deterministic context-sensitive language.

1. Preliminaries. In this section, we shall describe the basic definitions and terminology necessary for an understanding of this paper. It is assumed that the reader is familiar with the fundamental concepts of formal languages, automata theory and computational complexity [5], [14].

Let S^* be the set of all finite sequences of elements from the set S including the empty word λ, and $S^+ = S^* - \{\lambda\}$. If u and v are strings, then uv denotes the concatenation of u and v, and $\varnothing$ denotes the empty set.

DEFINITION 1. A *phrase structure grammar* (abbreviated PSG) is a quadruple $G = (V_N, V_T, P, \sigma)$, where

1. V_N is a finite set of nonterminals,
2. V_T is a finite set of terminals,
3. P is a finite set of productions whose forms are $u \rightarrow V$, with u in V_N^+ and v in $(V_N \cup V_T)^*$, and
4. $\sigma \in V_N$ is the initial symbol.

DEFINITION 2. Let $G = (V_N, V_T, P, \sigma)$ be a PSG. For w and y in $(V_N \cup V_T)^*$, write $w \Rightarrow^{(G)} y$ (or $w \Rightarrow y$ when G is understood) if there exist z_1, z_2, u and v such that $w = z_1uz_2$, $y = z_1vz_2$ and $u \rightarrow v \in P$. If either $w = y$ or there exist $w_0, \cdots, w_r$ such that $w_0 = w$, $w_r = y$ and $w_i \Rightarrow w_{i+1}$ for each i, the sequence $w_0, \cdots, w_r$ is called a *derivation* from w to y, and is denoted by $w_0 \Rightarrow_{q_1} \cdots \Rightarrow_{q_r} w_r$ (or $w_0 \Rightarrow \cdots \Rightarrow w_r$) or $w_0 \overset{*}{\Rightarrow}_\alpha w_r$ (or $w_0 \overset{*}{\Rightarrow} w_r$), where q_i $(1 \leqq i \leqq r)$ is a production applied to derive w_i from w_{i-1}, and $\alpha = q_1 \cdots q_r$. α is called the *associate word* of the derivation. A string β is called a *sentential form* (abbreviated SF) if $\sigma \overset{*}{\Rightarrow} \beta$. The subset of V_T^*, $L(G) = \{x \in V_T^* | \sigma \overset{*}{\Rightarrow} x\}$ is called a *phrase structure language* (abbreviated PSL). The subset of P^*, $\mathrm{Sz}(G) = \{\alpha \in P^* | \sigma \overset{*}{\Rightarrow}_\alpha w, w \in V_T^*\}$ is called the *Szilard language of G.*

DEFINITION 3. Let $G = (V_N, V_T, P, \sigma)$ be a PSG, and let q be a production $u \rightarrow v$. Then $xuy \Rightarrow_q^{(G,L)} xvy$ (or $xuy \Rightarrow_q^L xvy$ when G is understood, or $xuy \Rightarrow^L xvy$ when G and q are understood) if x is in V_T^*. The relation $\overset{*}{\Rightarrow}_\alpha^{(G,L)}$ (or $\overset{*}{\Rightarrow}_\alpha^L$ when G is understood, or $\overset{*}{\Rightarrow}^L$ when G and α are understood) is defined in the same manner as $\overset{*}{\Rightarrow}$, and is called a leftmost derivation. $\mathrm{Sz}_{\mathrm{left}}(G) = \{\alpha | \sigma \overset{*}{\Rightarrow}_\alpha^L w, w \in V_T^*\}$ is called the *leftmost Szilard language of G.* $xuy \Rightarrow_q^{(G,R)} xvy$ (or $xuy \Rightarrow_q^R xvy$ when G is understood, or $xuy \Rightarrow^R$ when G and q are understood) if y is in V_T^*. The relation $\overset{*}{\Rightarrow}_\alpha^{(G,R)}$ (or $\overset{*}{\Rightarrow}_\alpha^R$ when G is understood, or $\overset{*}{\Rightarrow}^R$ when G and α are understood) is defined in the same manner as $\overset{*}{\Rightarrow}$, and is called a *rightmost derivation.* $\mathrm{Sz}_{\mathrm{right}}(G) = \{\alpha | \sigma \overset{*}{\Rightarrow}_\alpha^R w, w \in V_T^*\}$ is called the *rightmost Szilard language of G.*

In the following, rightmost Szilard languages will not be described because each result on leftmost Szilard languages can be restated for right most Szilard languages.

We do not describe the definition of a Turing machine (abbreviated TM) in this paper. The reader is refered to [5] for formal definitions and detailed descriptions of several types of Turing machines. We use the following abbreviations:

OFF-DSPACE$(f)=\{L|L$ is a language accepted by an off-line deterministic TM which operates within tape bound $f\}$.

OFF-NSPACE$(f)=\{L|L$ is a language accepted by an off-line nondeterministic TM which operates within tape bound $f\}$.

ON-DSPACE $(f)=\{L|L$ is a language accepted by an on-line deterministic TM which operates within tape bound $f\}$.

ON-NSPACE $(f)=\{L|L$ is a language accepted by an on-line nondeterministic TM which operates within tape bound $f\}$.

The base of the logarithm is immaterial to our discussion. It is convenient to define $\log n = \lceil \log_2 n \rceil$, where $\lceil r \rceil$ is the least positive integer not less than r. If $\sup_{n\to\infty} (f(n)/n) > 0$, ON-DSPACE$(f)$ = OFF-DSPACE and ON-NSPACE(f) = OFF-SPACE(f). Therefore if $\sup_{n\to\infty} (f(n)/n) > 0$, we shall write DSPACE$(f)$ instead of OFF-DSPACE(f) or ON-DSPACE(f) and NSPACE(f) instead of OFF-NSPACE(f) or ON-NSPACE(f).

2. Tape requirements of on-line TM's. Interesting observations on Szilard languages have been made in several papers [3], [6], [10], [12]. The reader is refered to Salomaa [14] which contains a comprehensive description of known facts about Szilard languages [14, pp. 185–186]. For example, it is known that there is a context-free language (abbreviated CFL) L such that no grammar for L has a context-free Szilard language [14]. Stearns, Hartmanis and Lewis [15] showed that if $\sup_{n\to\infty} (f(n)/(\log n)) = 0$, any member of ON-DSPACE(f) is a regular set. Therefore $\log n$ is a lower bound on f such that ON-DSPACE(f) includes the set of all Szilard languages of context-free grammars (abbreviated CFG's), because there is a CFG G such that Sz(G) is not a CFL [10] [14]. We shall shown that $\log n$ is the optimal bound on such functions.

THEOREM 1. *The class of Szilard languages of CFG's is included in* ON-DSPACE$(\log n)$.

Proof. Let $G = (V_N, V_T, P, \sigma)$ be an arbitrary CFG, where $V_N = \{x_1, \cdots, x_k\}$ and $\sigma = x_1$. Let M be a TM which operates in the following way. Let $\beta = b_1 \cdots b_n$ be an input string to M, and let $b_i = (x_{t(i)} \to \gamma_i)$ and $m_i(j)$ be the number of occurrences of x_j in γ_i $(i = 1, \cdots, n; j = 1, \cdots, k)$. The construction of M is based on keeping track of the number of nonterminals in SF's, i.e. the Parikh-vector (in binary form) of SF's. Before starting the simulation of a derivation associated with β, M constructs k registers. X_j is used to store the number of occurrences of x_j in the SF at each stage of the derivation in binary form $(j = 1, \cdots, k)$. M operates as follows:

1. Set i, X_1 to 1, and $X_2, X_3, \cdots, X_k$ to 0.
2. Read b_i. If $X_{t(i)} = 0$, then go to 7.
3. For each j such that $X_j \neq X_{t(i)}$, set X_j to $X_j + m_i(j)$, and set $X_{t(i)}$ to $X_{t(i)} + m_i(t(i)) - 1$.
4. If $i \neq n$, then set i to $i+1$ and go to 2.
5. If there exists j such that $X_j \neq 0$, then go to 7.
6. M halts and accepts β.
7. M halts and rejects β.

The entire operation of M described above is deterministic and M recognizes $\mathrm{Sz}(G)$. Since the range of numbers stored in the registers $X_1, \cdots, X_k$ is from 0 to n, the working tape capacity used by M is at most $c \log n$, where c is a constant independent of n Therefore $\mathrm{Sz}(G)$ is in ON-DSPACE($\log n$). Q.E.D.

The result given by Stearns, Hartmanis and Lewis [15] also holds for the nondeterministic case, i.e. any member of ON-NSPACE(f) is regular if $\sup_{n\to\infty} (f(n)/\log n) = 0$ (the proof is essentially the same as the proof for the deterministic case.). Therefore $\log n$ is also the optimal tape bound for on-line nondeterministic TM's which recognize Szilard languages of CFG's.

The result in Theorem 1 also holds for Szilard languages of λ-free context-free programmed grammars under the free interpretation. The same construction as M in the proof of Theorem 1, together with a checking facility in the application of the productions, does not require more than $\log n$ working tape capacity.

We shall now look at leftmost Szilard languages. Let $G_1 = (V_N, V_T, P, \sigma)$, where $V_N = \{\sigma, x_1, x_2\}$, $V_T = \emptyset$, $P = \{a_1 = (\sigma \to \sigma x_1), a_2 = (\sigma \to x_2), c = (\sigma \to \lambda), \hat{a}_1 = (x_1 \to \lambda), \hat{a}_2 = (x_2 \to \lambda)\}$. Let h_1 be a homomorphism such that $h_1(a_1) = \hat{a}_1$ and $h(a_2) = \hat{a}_2$. Then $\mathrm{Sz}_{\mathrm{left}}(G_1) = \{xch_1(x^R) | x \in \{a_1, a_2\}^+\}$, where x^R is the reversal of x. $\mathrm{Sz}_{\mathrm{left}}(G_1)$ is essentially a palindrome language. It is well known that the palindromes take at least linear space to be recognized by on-line TM's [5]. It is apparent that for any PSG G, $\mathrm{Sz}_{\mathrm{left}}(G)$ is in ON-DSPACE(n). We therefore have the next theorem.

THEOREM 2. *The class of all leftmost Szilard languages of PSG's is included in* ON-DSPACE(n), *but is not included in* ON-NSPACE(f) *if* $\sup_{n\to\infty}(f(n)/n) = 0$.

Since G_1 is a CFG, n is also the optimal tape bound for both on-line deterministic and on-line nondeterministic TM's which recognize leftmost Szilard languages of CFG's.

3. Tape requirements of off-line TM's. We shall devote this section to a proof of the following theorem:

THEOREM 3. *Let G be an arbitrary PSG. Then* $\mathrm{Sz}_{\mathrm{left}}(G)$ *is in* OFF-DSPACE($\log n$).

It is well known that the palindromes take at least $\log n$ space to be recognized by off-line TM's. $\mathrm{Sz}_{\mathrm{left}}(G_1)$ which was given in § 2 is essentially a palindrome language. Therefore the above theorem implies that $\log n$ is the optimal tape bound for off-line deterministic TM's which recognize leftmost Szilard languages of PSG's. It is known that any $\mathrm{Sz}_{\mathrm{left}}(G)$ is a CFL [10], [14]. The best upper bound on off-line deterministic tape-complexity of arbitrary CFL's presently known is $(\log n)^2$ [15]. However we suspect that large subclasses of CFL's are recognizable in less than $(\log n)^2$ tape bound. In fact, some subclasses of CFL's have been shown to be recognized in $\log n$ tape bound [9], [16]. The tape-complexity of off-line deterministic TM's for the class of leftmost Szilard languages of PSG's is now here improved to $\log n$ also.

Proof of Theorem 3. Let G be an arbitrary PSG, and let M be an off-line TM which simulates derivations of G in the following manner: Let $\beta = b_1 \cdots b_n$ be an input sequence to M, where each b_i $(1 \leqq i \leqq n)$ is a production of G. There is at most one leftmost derivation associated with β. Let $b_k = (\theta(k) \to \alpha(k))$. If γ is a string, $|\gamma|$ is the length of γ and $\|\gamma\|$ is the number of nonterminals in γ. Suppose that $\theta(m) = A_1 \cdots A_{|\theta(m)|}$, where each A_i $(1 \leqq i \leqq |\theta(m)|)$ is a nonterminal. In order to check whether the left side of the production b_m $(1 < m \leqq n)$ is the leftmost nonterminal substring of the SF associated with $b_1 \cdots b_{m-1}$, M counts the following numbers (the test for b_1 can be easily done by using only internal memory of M):

(i) The number of nonterminals to the left of A_R (which should be $R-1$ for each R $(1 \leqq R \leqq |\theta(m)|)$ if b_m can be the mth production of the leftmost derivation).

(ii) The number of symbols to the right of A_R (which should be one smaller than the number of such symbols to the right of the same number for A_{R-1} $(2 \leqq R \leqq |\theta(m)|)$ if b_m can be the mth production of the leftmost derivation).

For each R $(1 \leqq R \leqq |\theta(m)|)$ M counts the number indicated in (i) above, and successively determines whether the first, second, third, $\cdots$ occurrence of A_R generated satisfied the condition in (i). For each R $(1 \leqq R \leqq |\theta(m)|)$, if the condition indicated in (i) is satisfied, then M counts the number indicated in (ii) and determines whether the condition indicated in (ii) is satisfied. If both the conditions indicated in (i) and (ii) are satisfied for all R $(1 \leqq R \leqq |\theta(m)|)$, then $\theta(m)$ is the leftmost substring of the SF associated with $b_1 \cdots b_{m-1}$ and b_m can be applied as the mth production of the leftmost derivation.

We shall simply describe an algorithm to count the number indicated in (i) for the first, second, third, $\cdots$ occurrences of A_R generated for an arbitrary R $(1 \leqq R \leqq |\theta(m)|)$. Then an algorithm to count the number indicated in (ii) will be informally described. The reader can then easily construct a complete algorithm by using the algorithms indicated below sequentially. In the followng algorithm, $n_{A_R}(k)$ denotes the number of occurrences of the nonterminal A_R in $\alpha(k)$, I keeps track of which occurrence of A_R generated is checked, J counts the number of nonterminals to the left of the Ith occurrence of A_R generated, L counts the number of occurrences of A_R generated, k counts the index of the production considered in the input sequence $b_1 \cdots b_m$, and m is the index of the production that we want to check.

```
1. Set $I$ to 0.
2. Set $I$ to $I+1$, $J$ and $L$ to 0, and $k$ to 1.
3. If $I > L + n_{A_R}(k)$, then go to 6; else if $I \leqq L$, then go to 5.
4. Set $J$ to the number of nonterminals in $\alpha(k)$ to the left of the $(I-L)$th
   occurrence of $A_R$ in $\alpha(k)$, and go to 6.
5. If $J < |\theta(k)|$, then go to 2; else set $J$ to $J + \|\alpha(k)| - |\theta(k)|$.
6. Set $L$ to $L + n_{A_R}(k)$, and $k$ to $k+1$. If $k < m$, then go to 3.
7. If $I > L$, then go to 9; else if $J = R - 1$, then go to 8; else go to 2.
8. The $I$th occurrence $A_R$ is the $R$th leftmost nonterminal in the SF
   associated with $b_1 \cdots b_{m-1}$.
9. $A_R$ is not the $R$th leftmost nonterminal in the SF associated with
   $b_1 \cdots b_{m-1}$ (i.e. $\beta$ is rejected).
```

All the above operations are deterministic, and the range numbers stored in the registers is from 0 to cn, where c is a constant independent of n. Therefore the tape bound for counting the number indicated in (i) is log n. We now show that the number indicated in (ii) can also be counted in log n tape bound. The argument is an induction on m. When $m = 1$, the number should be 0 if $\theta(1)$ is the initial symbol. Suppose that for each k $(1 \leqq k < m)$ there exists the leftmost derivation associated with $b_1 \cdots b_k$, and that M can count each number indicated in (ii) for each b_k $(1 \leqq k < m)$ in log n tape bound. From the above algorithm for counting the number indicated in (i), M can compute I in log n tape bound such that the Ith occurrence of A_R is the Rth nonterminal from the leftmost of the SF associated with $b_1 \cdots b_{m-1}$ (we suppose that for m and each R $(1 \leqq R \leqq |\theta(m)|)$ the condition indicated in (i) is satisfied). Then by checking $b_1 \cdots b_{m-1}$ of the input sequence, M knows which production has produced the Ith occurrences of A_R. Say that b_j has produced the Ith occurrence of A_R, and that the number of symbols to the right of the A_R in $\alpha(j)$ is Q_1. From the induction hypothesis, M can count the number of symbols to the right of the $\theta(j)$ rewritten in the SF associated with $b_1 \cdots b_{j-1}$ in log n tape bound. Say the number is Q_2. Then the number of symbols to the right of the Ith occurrence of A_R in the SF associated with $b_1 \cdots b_{m-1}$ is $Q_1 + Q_2$. All these operations are deterministic, and can be performed in log n tape bound.

If there exists the leftmost derivation associated with β, M checks whether the final SF is a terminal string. The algorithm for this computation is essentially the same as the algorithm in the proof of Theorem 1. This completes the proof. Q.E.D.

In this paper we employ a specific form of leftmost derivability which we defined in Definition 3. The definition seems to be most commonly used. The method of counting numbers recursively which we used in the above proof plays an essential role to derive the result of log n tape bound. This method seems to be widely applicable to similar problems. One may show that the above result holds for any interpretation of leftmost derivability by using this counting method with minor modification.

4. A tape bound on Szilard languages of PSG's. In this section, we shall show that the Szilard language of an arbitrary PSG is a deterministic context-sensitive language. Let $G = (V_N, V_T, P, \sigma)$ be an arbitrary PSG. We can construct a single tape nondeterministic TM M which recognizes Sz(G) in time-complexity n^2 as follows: Let $a_1 \cdots a_n$ be an input sequence to M, where each a_i $(1 \leqq i \leqq n)$ is in P. M simulates nondeterministically a derivation associated with $a_1 \cdots a_n$ in the following manner: Suppose that M has simulated the derivation up to the $(i-1)$st step and the ith SF of the derivation is on the tape of M. Then M guesses the position in which the ith production a_i is applied to the SF by moving its head to the position, and replaces the substring which corresponds to the right side of a_i by the left side of a_i. The rewritten part may be shortened, or may be lengthened. If so, M shifts appropriately the nonrewritten part of the SF so that the SF is written on the tape without overlaps or gaps of any parts of it. It is obvious that all these operations can be performed in at most cn steps, where c is a constant independent of n. If M cannot rewrite at the position which M guessed, M halts and rejects the input sequence. If M could perform the rewriting and $i < n$, then M starts the simulation of the next production a_{i+1}. If M could perform the rewriting and $i = n$, and if the SF of the derivation is a terminal string, then M halts and accepts the input sequence. Otherwise, the input sequence is rejected. From the above explanation, the total computation time to decide whether $a_1 \cdots a_n$ is a member of Sz(G) is nondeterministically at most cn^2 steps. Therefore from Paterson's result [11, Thm. 1], there exists a deterministic TM which recognizes Sz(G) in tape-complexity n. Then we have the next theorem.

THEOREM 4. *The Szilard language of an arbitrary PSG is a deterministic context-sensitive language.*

It is well known that the Szilard language of an arbitrary PSG is a context-sensitive language [10], [14]. However, it seems that no published literature includes the above result. In this view, it may be well that the above theorem is described here.

Acknowledgments. The author would like to thank Dr. Leslie G. Valiant for his helpful discussions. He is also grateful to the referees for their useful suggestions in preparing this revision. Theorem 4 was suggested by one of the referees.

REFERENCES

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

[2] S. CRESPI-REGHIZZI AND D. MANDRIOLI, *A decidability theorem for a class of vector-addition systems*, Information Processing Lett. 3 (1975), pp. 78–80.

[3] A. C. FLECK, *An analysis of grammars by their derivation sets*, Information and Control, 24 (1974), pp. 389–398.

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

[5] J. E. HOPCROFT AND J. D. ULLMAN, *Formal Languages and their Relation to Automata*, Addison-Wesley, Reading, MA, 1969.

[6] M. HÖPPNER, *Ueber den Zusammenhang von Szilard Sprachen und Matrixgrammatiken*, Institut für Informatik, Hamburg, W. Germany, 1974.

[7] R. M. KARP AND R. E. MILLER, *Parallel program schemata*, J. Comput. System Sci., 3 (1969), pp. 147–195.

[8] J. VAN LEEUWEN, *A partial solution to the reachability-problem*, Conf. Record of 6th Ann. ACM Symp. on Theory of Computing, 1974, pp. 303–309.

[9] K. MEHLHORN, *Bracket-languages are recognizable in logarithmic space*, Fachbereich Angewandte Mathematik und Informatik, Universitat des Saarlandes, Saarbrucken, W. Germany, 1975.

[10] E. MORIYA, *Associate languages and derivational complexity of formal grammars and languages*, Information and Control, 22 (1973), pp. 139–162.

[11] M. S. PATERSON, *Tape-bounds for time-bounded Turning machines*, J. Comput. System Sci., 6 (1972), pp. 116–124.

[12] M. PENTTONEN, *On derivation language corresponding to context-free grammars*, Acta Informatica, 3 (1974), pp. 285–291.

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

[14] A. SALOMAA, *Formal Languages*, Academic Press, London-New York, 1973.

[15] R. E. STEARNS, J. HARTMANIS AND P. M. LEWIS II, *Hierarchies of memory limited computations*, Proc. 6th Ann. Symp. on Switching Circuit Theory and Logical Design, IEEE, New York, 1965, pp. 179–190.

[16] I. H. SUDBOROUGH, *On tape-bounded complexity classes and multi-head finite automata*, Conf. Record 14th Ann. Symp. on Switching and Automata Theory, IEEE, Iowa, 1973, pp. 138–144.

SIAM J. COMPUT.
Vol. 6, No. 3, September 1977

THE COMPUTATIONAL COMPLEXITY OF PROVABILITY IN SYSTEMS OF MODAL PROPOSITIONAL LOGIC*

RICHARD E. LADNER†

Abstract. The computational complexity of the provability problem in systems of modal propositional logic is investigated. Every problem computable in polynomial space is log space reducible to the provability problem in any modal system between K and $S4$. In particular, the provability problem in K, T, and $S4$ are log space complete in polynomial space. The nonprovability problem in S5 is log space complete in nondeterministic polynomial time.

Key words. modal logic, computational complexity

Introduction. We investigate the computational complexity of deciding whether or not a modal propositional formula is provable in certain systems of modal propositional logic, including K, T, $S4$, and $S5$. In terms to be defined later we show (using a suggestion of S. K. Thomason) that if S is a modal system between K and $S4$, then every problem computable in polynomial space is log space reducible to the provability problem in S. We then show that there are polynomial space bounded algorithms for deciding if a formula is provable in any one of K, T, and $S4$. This implies that the provability problem for each of systems K, T, and $S4$ is log space complete in polynomial space. We also obtain upper and lower bounds on the space complexity of the provability problem in each of the systems K, T, and $S4$.

We show that the nonprovability problem for $S5$ is log space complete in nondeterministic polynomial time. Hence the provability problem in $S5$ and the provability problem in the classical propositional calculus have the same complexity modulo polynomial time.

All our proofs depend heavily on the semantic models for modal systems developed by Kripke [6].

As evidence that modal logic has some applications in Computer Science, we point to the work of V. R. Pratt and R. Moore [8], who have developed a system of modal logic as a basis for proving correctness and termination of programs. We briefly explain their application. Assume we have some underlying programming language and some underlying assertion language. For each program p define a new syntactic object $[p]$ which is understood to be a modal operator. We can now form new assertions of the form $[p]A$ where A is an arbitrary assertion. The intuitive meaning of $[p]A$ is that "if p terminates, then A holds." The fact that p always terminates can be expressed by the assertion $\langle p\rangle$true (where $\langle p\rangle A \overset{\text{def}}{\equiv} \sim[p]\sim A$). The Hoare formula $A\,\{p\}\,B$ is equivalent to the formula $A \supset [p]B$. An advantage of this modal system over the Hoare system is that more complicated assertions about programs are possible. For instance one can express the fact that "if a program p terminates with A true, then subsequently the

* Received by the editors August 21, 1975, and in revised form July 21, 1976.

† Department of Computer Science, University of Washington, Seattle, Washington 98195. This research was supported by the National Science Foundation under Grant GJ-43264.

program q can terminate with B being true" by the assertion "$[p](A \supset \langle q \rangle B)$".

If Σ is a finite alphabet, then define Σ^* to be the set of all finite words from letters in Σ and λ to be the empty word and $\Sigma^+ = \Sigma^* - \{\lambda\}$. If $x, y \in \Sigma^*$, then $|x|$ denotes the length of x, xy denotes x concatenated to y, and x^n denotes x concatenated to itself n times ($x^0 = \lambda$, $x^k = x \cdot x^{k-1}$ for $k \geqq 1$). Let $N = \{0, 1, 2, \cdots\}$. If $n \geqq 1$, then $\log n$ is defined to be $\lceil \log_2 n \rceil$ and $\log 0 = 0$.

1. Modal logic. We define formulas so that they are words in a finite alphabet. A *variable* is a member of $\mathrm{VAR} = \text{¢}\{0, 1\}^*\$$. A *Boolean formula* is either a variable or has the form $(A \wedge B)$ or $\sim A$ where A and B are Boolean formulas. The set of Boolean formulas, denoted by BF, is a subset of Δ_{BF}^* where $\Delta_{\mathrm{BF}} = \{\text{¢}, \$, 0, 1, \wedge, \sim, (,)\}$. A *modal formula* is either a variable or has the form $(A \wedge B)$, $\sim A$, or $\Box A$ where A and B are modal formulas. The set of modal formulas, denoted by MF, is a subset of Δ_{MF}^* where $\Delta_{\mathrm{MF}} = \Delta_{\mathrm{BF}} \cup \{\Box\}$. A formula of the form $\Box A$ is read "necessarily A". Another modal operator, $\Diamond$, is defined by $\Diamond A \equiv \sim \Box \sim A$ and $\Diamond$ can be read as "possibly". Technically $\Diamond$ and the standard logical operators $\wedge$, $\supset$, $\equiv$ do not appear in modal formulas, but for convenience they do appear in modal text. We also may drop parentheses from formulas to improve readability.

We will systematically use $\wedge, \vee, \sim$ as both logical symbols and as the Boolean operations on $\{T, F\}$ they represent.

Let PC (for propositional calculus) be some complete set of axioms for the valid Boolean formulas where the rules of inference are substitution and modus ponens. A *modal system* is a set of modal formulas. If S is a modal system, then define the provability relation, $\vdash_S$, inductively as follows.

(i) $\vdash_S A$ if $A \in \mathrm{PC} \cup S$,

(ii) $\vdash_S A'$ if $\vdash_S A$ and A' is the result of substituting uniformly in A a modal formula for a propositional variable (Rule of Substitution),

(iii) $\vdash_S B$ if $\vdash_S A$ and $\vdash_S A \supset B$ (Modus Ponens),

(iv) $\vdash_S \Box A$ if $\vdash_S A$ (Rule of Necessity),

(v) $\vdash_S$ is the smallest relation satisfying (i)–(iv).

If $\vdash_S A$, then we say that A is *provable in S* and we define S-PROVABLE $= \{A \in \mathrm{MF} : \vdash_S A\}$.

There are at least four important modal systems, K, T, $S4$, and $S5$ which are defined by

$$K = \{\Box(X \supset Y) \supset (\Box X \supset \Box Y)\},$$

$$T = K \cup \{\Box X \supset X\},$$

$$S4 = T \cup \{\Box X \supset \Box\Box X\},$$

$$S5 = S4 \cup \{\Diamond X \supset \Box \Diamond X\},$$

(X and Y are specific members of VAR.)

The reader unfamiliar with modal logic can appeal to Hughes and Cresswell [4].

Very useful semantic models for many modal systems were discovered by Kripke [6]. In particular, there are such semantics for the four systems K, T, $S4$, and $S5$. In the remainder of this section the facts we state are either due to Kripke [6] or are attributed to him.

A *model structure* is a triple (W, R, V) where W is a set, R is a binary relation on W, and V is a mapping from $\mathrm{VAR} \times W$ into $\{T, F\}$. The set W is a set of "possible worlds", R determines which worlds are "accessible" from other worlds, and V determines what is true in each of the worlds. Given a model structure (W, R, V) the mapping V can be extended to $\mathrm{MF} \times W$ inductively as follows:

$$V(A \wedge B, w) = T \quad \text{iff} \quad V(A, w) = T \text{ and } V(B, w) = T,$$

$$V(\sim A, w) = T \quad \text{iff} \quad V(A, w) = F,$$

$$V(\Box A, w) = T \quad \text{iff} \quad \text{for all } w' \in W, \text{ if } wRw', \text{ then } V(A, w') = T.$$

Define (W, R, V) to be a *K-model* if it is a model structure and to be a (i) *T-model*, (*ii*) *$S4$-model*, (iii) *$S5$-model* if R is respectively (i) reflexive, (ii) reflexive and transitive, (iii) reflexive, transitive and symmetric.

Let $S \in \{K, T, S4, S5\}$. Define a modal formula A to be *S-satisfiable* if there is an S-model (W, R, V) and a world $w \in W$ such that $V(A, w) = T$. Let S-SATISFIABLE $= \{A \in \mathrm{MF} : A$ is S-satisfiable$\}$. Define A to be *S-valid* if $\sim A$ is not S-satisfiable. Let S-VALID $= \{A \in \mathrm{MF} : A$ is S-valid$\}$. The crucial fact we use later is:

FACT 1.1 (Kripke). *For all* $S \in \{K, T, S4, S5\}$ S-VALID $=$ S-PROVABLE.

The *modal degree* of a formula is defined inductively: the modal degee of a variable is 0; degree of $\sim A$ = degree of A; degree of $A \wedge B$ = max{degree of A, degree of B}; and degree of $\Box A = 1 +$ degree of A.

2. Computational complexity. We adopt the *Turing machine* model of computation to measure time and space complexity. The reader may refer to Hopcroft and Ullman [3, Chap. 10] for background.

To be specific, our Turing machines will have three tapes: a two-way read-only input tape, one-way write-only output tape, and a two-way read-write work tape. Associated with such a machine are finite alphabets: Σ (input alphabet), Δ (output alphabet), and Γ (work tape alphabet); also a finite set of states Q, a start state q_0 and a transition function

$$\delta : Q \times \Sigma \times \Gamma \to 2^{Q \times \Gamma \times (\Delta \cup \{\lambda\}) \times \{R, L\}^2}.$$

Given a state, a symbol being read on the input tape, a symbol being read on the work tape, the machine does one of a finite number of "moves" each of which consists of going to a new state, writing a symbol on the work tape, outputting either a symbol or λ and moving the input tape and work tape heads. As defined, our Turing machines are *nondeterministic*. A Turing machine is *deterministic* if the cardinality of $\delta(q, \sigma, \tau) \leqq 1$ for each triple $(q, \sigma, \tau) \in Q \times \Sigma \times \Gamma$.

Given an input $x \in \Sigma^*$,a *computation of T on input x* is a finite sequence of configurations of the Turing machine which begins in the starting configuration (the machine is in state q_0, input tape contains x with the input head on the first letter of x, and the other tapes empty), each other configuration follows from the previous one via the transition rule, and ends in a configuration from which no configuration can follow. A Turing machine *T runs in time $t : N \to N$* if for each n and each $x \in \Sigma^*$ such that $|x| = n$ every computation of T on input x has length $\leqq t(n)$. A Turing machine *T runs in space $s : N \to N$* if for each n and each $x \in \Sigma^*$ of length n at most $s(n)$ distinct tape cells on the *work* tape are scanned in each computation of T on input x.

A *set* $L \subseteq \Sigma^*$ *is computable in nondeterministic time* (space) r if there is a Turing machine T that runs in time (space) r such that for all $x \in \Sigma^*$, $x \in L$ iff there is a computation of T on input x such that T outputs some symbol during that computation. A *set L is computable in time* (space) r if in the above definition the Turing machine is deterministic. A *function* $f: \Sigma^* \to \Delta^*$ *is computable in time* (space) r if there is a deterministic Turing machine T that runs in time (space) r such that for all $x \in \Sigma^*$, when T halts on input x the machine has outputted the string $f(x)$.

We define NP-TIME (NP-SPACE) to be the class of sets L such that there is a polynomial p such that L is computable in nondeterministic time (space) p. Similarly P-TIME (P-SPACE) is the class of sets L such that there is a polynomial p such that L is computable in time (space) p. A result of Savitch [9] implies P-SPACE $=$ NP-SPACE. There is the obvious containment relationship P-TIME $\subseteq$ NP-TIME $\subseteq$ P-SPACE. It is open whether or not either containment is proper.

If $s: N \to N$, then define (N)SPACE($s(n)$) = the class of sets computable in (nondeterministic) space s. We don't define the analogous time complexity classes for the same reason that we don't bother with multiple work tapes; the methods we use cannot be used to distinguish polynomial time complexity up to the degree of the polynomial.

Given sets $L \in \Sigma^*$ and $M \in \Delta^*$ we say that *L is log space reducible to M* ($L \leqq_{\log} M$) if there is a function $f: \Sigma^* \to \Delta^*$ such that f is computable in space log and for all $x \in \Sigma^*$, $x \in L$ iff $f(x) \in M$. We sometimes say $L \leqq_{\log} M$ *via f*. The relation $\leqq_{\log}$ is reflexive and transitive (cf. Stockmeyer and Meyer [12] or Jones [5]).

Let $\mathcal{S}$ be a class of sets. *A* set *L is log space complete in* $\mathcal{S}$ if $L \in \mathcal{S}$ and for all $M \in \mathcal{S}$, $M \leqq_{\log} L$. Cook [2] implicitly showed the existence of log space complete sets in NP-TIME while Stockmeyer and Meyer [12] showed the existence of log space complete sets in P-SPACE.

There is a well known relationship between complete problems and open problems concerning P-TIME, NP-TIME, and P-SPACE.

FACT 2.1. *If L is log space complete in* NP-TIME,*then* $L \in P$-TIME *if and only if* P-TIME $=$ NP-TIME.

FACT 2.2. *If L is log space complete in* P-SPACE, *then*

(i) $L \in P$-TIME *if and only if* P-TIME $=$ P-SPACE,

(ii) $L \in NP$-TIME *if and only if* NP-TIME $=$ P-SPACE.

If $l: N \to N$ and $f: \Sigma^* \to \Delta^*$ then f is *length* $l(n)$ *bounded* if for all $x \in \Sigma^*$, $|f(x)| \leqq l(|x|)$. The following fact due to Stockmeyer and Meyer [12] and Jones [5] is helpful later in establishing lower bounds.

FACT 2.3 (Stockmeyer and Meyer, and Jones). *If* $A \leqq_{\log} B$ *via f where f is length* $l(n)$ *bounded, then A is in* (N)SPACE($s(l(n)) + \log n$) *should B be in* (N)SPACE($s(n)$).

Let $\Delta_{\text{QBF}} = \Delta_{\text{BF}} \cup \{\forall, \exists\}$. A *quantified Boolean formula* (QBF) is a member of Δ^*_{QBF} of the form $Q_1X_1Q_2X_2 \cdots Q_nX_nA(X_1, \cdots, X_n)$ where $Q_i \in \{\forall, \exists\}$, $X_i \in$ VAR for $1 \leqq i \leqq n$ and $A(X_1, \cdots, X_n) \in$ BF whose variables are contained in

$X_1, \cdots, X_n$. The propositional variables range over $\{T, F\}$ so that if $A \in \mathrm{QBF}$, then the value of A is either T or F.

Define

$$\mathbf{B}_\omega = \{A \in \mathrm{QBF} : A \equiv T\},$$

$$\mathbf{B}_1 = \{A \in \mathrm{QBF} \cap (\exists \mathrm{VAR})^* \mathrm{BF} : A \equiv T\}.$$

The set $\mathbf{B}_\omega$ is the set of all valid quantified Boolean formulas, while $\mathbf{B}_1$ is essentially the set of all satisfiable Boolean formulas.

Stockmeyer and Meyer [12] have shown

FACT 2.4 (Stockmeyer and Meyer). *$\mathbf{B}_\omega$ is log space complete in P-SPACE.*

A more precise delineation of $\mathbf{B}_\omega$ is given in Stockmeyer [11].

FACT 2.5 (Stockmeyer). *Let d be an integer $\geqq 1$. If $A \in \mathrm{NSPACE}(n^d)$, then there is a function f and a constant $a > 0$ such that $A \leqq_{\log} \mathbf{B}_\omega$ via f and f is length $an^{2d} \log n$ bounded.*

When we investigate the complexity of $S5$ we will need a result of Cook [2].

FACT 2.6 (Cook). *$\mathbf{B}_1$ is log space complete in NP-TIME.*

Because of the transitivity of $\leqq_{\log}$ we can show that every problem computable in polynomial space is log space reducible to, say, L if we can show that $\mathbf{B}_\omega$ is log space reducible to L. In what follows we use $\mathbf{B}_\omega$ as a cornerstone in analyzing the space complexity of modal systems between K and $S4$.

One useful fact that we use later is:

FACT 2.7. *If $\mathbf{B}_\omega \leqq_{\log} A$ via a length $l(n)$ bounded function, then $\mathbf{B}_\omega \leqq_{\log} \bar{A}$ via a length $l(n+5)$ bounded function.*

Proof Let f be such that $\mathbf{B}_\omega \leqq_{\log} A$ via f and f is length $l(n)$ bounded. There is a g such that $\mathbf{B}_\omega \leqq_{\log} \bar{\mathbf{B}}_\omega$ via g and g is length $n+5$ bounded. Let $x \in \Delta^*_{\mathrm{QBF}}$ and let $n = |x|$. It can be determined in space $\log n$ whether or not $x \in \mathrm{QBF}$. If $x \notin \mathrm{QBF}$, then define $g(x) = (\exists ¢\$)¢\$$. If $x \in \mathrm{QBF}$ then define $g(x) = \bar{Q}_1 X_1 \cdots \bar{Q}_m X_m \sim A$ where $x = Q_1 X_1 \cdots Q_m X_m A$, $A \in \mathrm{BF}$, $\bar{\forall} = \exists$ and $\bar{\exists} = \forall$. Clearly, $x \in \mathbf{B}_\omega$ if and only if $g(x) \notin \mathbf{B}_\omega$. Now, $\mathbf{B}_\omega$ is log space reducible via $f \cdot g$ which is length $l(n+5)$ bounded. Q.E.D.

3. Log space reduction of $\mathbf{B}_\omega$ to modal systems between K and $S4$. We say that a modal system S *is between* S_1 *and* S_2 if S_1-PROVABLE $\subseteq S$-PROVABLE $\subseteq S_2$-PROVABLE. In this section we prove the following.

THEOREM 3.1. *If S is between K and $S4$, then $\mathbf{B}_\omega$ is log space reducible to S-PROVABLE.*

Proof. The crux of the proof is to show that given any quantified Boolean formula A, a modal formula B can be constructed (using only logarithmic space) with the properties: (i) $A \in \mathbf{B}_\omega$ implies $B \in S4$-SATISFIABLE and (ii) $B \in K$-SATISFIABLE implies $A \in \mathbf{B}_\omega$.

In light of Fact 2.7, the following claim yields the theorem.

CLAIM. $A \in \mathbf{B}_\omega$ if and only if $\sim B \notin S$-PROVABLE.

If $A \in \mathbf{B}_\omega$, then by (i) $B \in S4$-SATISFIABLE and hence $\sim\sim B \in S4$-SATISFIABLE. By the definition of $S4$-VALID, $\sim B \notin S4$-VALID. By Fact 1.1, $\sim B \notin S4$-PROVABLE. Since S-PROVABLE $\subseteq S4$-PROVABLE, then $\sim B \notin S$-PROVABLE. On the other hand if $\sim B \notin S$-PROVABLE, then because K-PROVABLE $\subseteq S$-PROVABLE, then $\sim B \notin K$-PROVABLE. Again using Fact 1.1, $B \in K$-SATISFIABLE, which in turn implies by (ii) that $A \in \mathbf{B}_\omega$.

Let $A = Q_1X_1 \cdots Q_mX_mA'(X_1, \cdots, X_m) \in \mathrm{QBF}$ where $Q_i \in \{\forall, \exists\}$ and $X_i \in \mathrm{VAR}$ for $1 \leqq i \leqq m$, and $A'(X_1, \cdots, X_m) \in \mathrm{BF}$. Let $Y_0, \cdots, Y_m$, and $Z_1, \cdots, Z_{1+\log m}$ be new variables an for $0 \leqq i \leqq m$and $1 \leqq j \leqq 1 + \log m$ define $\beta_{ij} = \sim$ if the jth bit of i written as a binary number of length $1 + \log m$ is 1 and $\beta_{ij} = \lambda$ otherwise.

Define B to be the conjunction of the following formulas:

(1) $\Box^{(m)}(Y_i \equiv \beta_{i1}Z_i \wedge \beta_{i2}Z_2 \wedge \cdots \wedge \beta_{i(1+\log m)}Z_{1+\log m})$ for $0 \leqq i \leqq m$,

(2) Y_0,

(3) $\Box^{(m)}(Y_i \supset \Diamond Y_{i+1})$ for $0 \leqq i < m$,

(4) $\Box^{(m)}(Y_i \supset ((X_i \supset \Box^{(m)}X_i) \wedge (\sim X_i \supset \Box^{(m)} \sim X_i)))$ for $0 < i \leqq m$,

(5) $\Box^{(m)}(Y_i \supset (\Diamond(Y_{i+1} \wedge X_{i+1}) \wedge \Diamond(Y_{i+1} \wedge \sim X_{i+1})))$ if $Q_{i+1} = \forall$ and $0 \leqq i < m$,

(6) $\Box^{(m)}(Y_m \supset A')$,

where $\Box^{(m)}D \equiv D \wedge \Box D \wedge \Box^2 D \wedge \cdots \wedge \Box^m D$.

The intuitive meaning of $\Box^{(m)}D$ is that in any model structure (W, R, V), $V(\Box^{(m)}D, w) = T$ if and only if D is true in any world reachable from w in i steps where $0 \leqq i \leqq m$.

The idea behind the formula B is to "simulate" the quantifiers of A. The variables Y_i are used to set up levels corresponding to the levels of quantification in A. The formula Y_i is true in each world on level i. If the ith quantifier of A is universal, then (5) guarantees a splitting for each of the two possibilities for X_i. At the final level, m, A' must be true. We begin by showing (i) mentioned earlier.

$A \in \mathbf{B}_\omega$ implies $B \in S4$-SATISFIABLE. Suppose $A \in \mathbf{B}_\omega$; then B is satisfied in the $S4$-model, (W_A, R_A, V_A). The set of worlds is a finite subset of $\{0, 1\}^*$ defined inductively by

(a) $\lambda \in W_A$,

(b) if $w \in W_A$ and $|w| = i < m$, then

(b1) $w0$ and $w1 \in W_A$ if and only if $Q_{i+1} = \forall$,

(b2) $w0 \in W_A$ if and only if $Q_{i+1} = \exists$,

(c) W_A is the smallest set satisfying (a) and (b).

The members of W_A form a tree with respect to extension. The tree is binary branching at level i if $Q_{i+1} = \forall$ and is unary branching if $Q_{i+1} = \exists$. The accessibility relation R_A is defined by

$$x R_A y \quad \text{iff} \quad x \text{ is a prefix of } y.$$

Clearly R_A is reflexive and transitive. Finally we define V_A inductively on the length of w in such a way that

(a′) if $|w| = i$, then $V_A(Y_i, w) = T$,

(b′) if $|w0| = |w1| = i$ and $Q_i = \forall$, then $V_A(X_i, w0) \neq V_A(X_i, w1)$,

(c′) if $|w| = i > j$, then $V_A(X_j, w) = V_A(X_j, w')$ where w' is the prefix of w of length $i - 1$,

(d′) if $|w| = i$, then $Q_{i+1}X_{i+1} \cdots Q_mX_mA'(V_A(X_1, w), \cdots, V_A(X_i, w), X_{i+1}, \cdots, X_m) = T$.

Assume (a′)–(d′) hold for all numbers $<i$. Let $|w|=i$. Set $V_A(Z_j, w)=T$ if $\beta_{ij}=\lambda$, $V_A(Z_j, w)=F$ if $\beta_{ij}=\sim$, $V_A(Y_j, w)=F$ if $j\neq i$, and $V_A(Y_i, w)=T$. If $1\leqq j<i$, then set $V_A(X_j, w)=V_A(X_j, w')$ where w' is the prefix of w of length $i-1$. If $j>i$, then set $V_A(X_j, w)=T$.

If $i=0$, then (a′)–(c′) hold by definition and (d′) holds because $A\in\mathbf{B}_\omega$. Assume then that $i>0$; then all that remains is the value of $V_A(X_i, w)$.

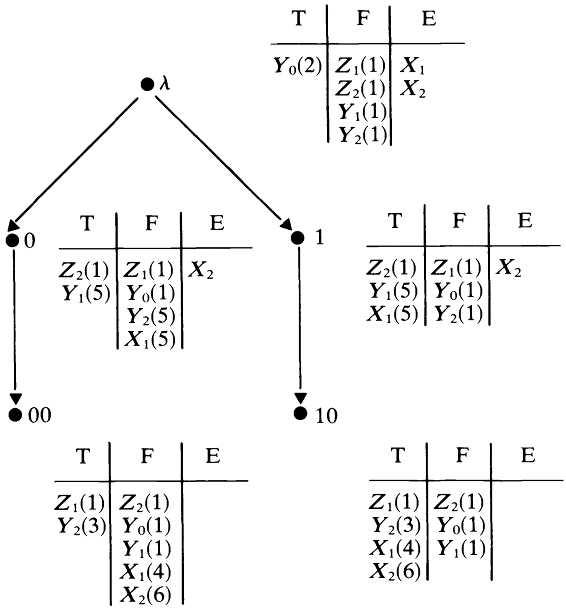
If $Q_i=\forall$, then set $V_A(X_i, w)=T$ if the last letter of w is 1 and set $V_A(X_i, w)=F$ otherwise.

If $Q_i=\exists$, then set $V_A(X_i, w)=V$ where $Q_{i+1}X_{i+1}\cdots Q_mX_mA'(V_A(X_1, w), \cdots, V_A(X_{i-1}, w), V, X_{i+1}, \cdots, X_m)=T$. Such a $V\in\{T, F\}$ exists because of the induction hypothesis.

It is straightforward to check that the induction hypothesis holds at i.

To establish that $B\in S4$-SATISFIABLE, we show that $V_A(B, \lambda)=T$. Clauses (1) through (5) in the definition of B hold by the construction of V_A. Clause (6) holds because by (a′) if $|w|<m$, then $V_A(Y_m, w)=F$ and by (d′) if $|w|=m$, then $V_A(A', w)=T$.

As an example of the preceding proof consider the formula $\forall X_1\exists X_2(X_1\equiv X_2)$. This formula is true and its modal companion B is satisfied in the $S4$-model graphically displayed in Fig. 1.



T	F	E
$Y_0(2)$	$Z_1(1)$	X_1
	$Z_2(1)$	X_2
	$Y_1(1)$	
	$Y_2(1)$	

T	F	E
$Z_2(1)$	$Z_1(1)$	X_2
$Y_1(5)$	$Y_0(1)$	
	$Y_2(5)$	
	$X_1(5)$	

T	F	E
$Z_2(1)$	$Z_1(1)$	X_2
$Y_1(5)$	$Y_0(1)$	
$X_1(5)$	$Y_2(1)$	

T	F	E
$Z_1(1)$	$Z_2(1)$	
$Y_2(3)$	$Y_0(1)$	
	$Y_1(1)$	
	$X_1(4)$	
	$X_2(6)$	

T	F	E
$Z_1(1)$	$Z_2(1)$	
$Y_2(3)$	$Y_0(1)$	
$X_1(4)$	$Y_1(1)$	
$X_2(6)$		

T—*variables that must be true.*
F—*variables that must be false.*
E—*variables that can have either value.*
The number in parentheses indicates the clause of B that forces the value of the variable.
The arrows represent the Hasse diagram of the accessibility relation.

FIG 1. *$S4$-model satisfying B associated with* $\forall X_1\exists X_2(X_1\equiv X_2)$

$B \in K$-SATISFIABLE *implies* $A \in \mathbf{B}_\omega$. Suppose that B is K-satisfiable in a model structure (W, R, V). We define a mapping, σ, of W_A into W inductively as follows.

(a″) Choose $\sigma(\lambda)$ such that $V(B, \sigma(\lambda)) = T$,

(b″) if $|w| = i > 0$ then choose $\sigma(w) \in W$ such that $\sigma(w')\, R\, \sigma(w)$ where w' is the prefix of w of length $i-1$ and

1. $V(\Box^{(m-i)}(Y_j \equiv \beta_{j1}Z_1 \wedge \cdots \wedge \beta_{j(1+\log m)}Z_{1+\log m}), \sigma(w)) = T$ for $0 \leqq j \leqq m$ (by clause (1) of B),
2. $V(Y_i, \sigma(w)) = T$ (by clause (3) of B),
3. $V(\Box^{(m-i)}(Y_j \supset \Diamond Y_{j+1}), \sigma(w)) = T$ for $0 \leqq j < m$ (by clause (3) of B),
4. $V(\Box^{(m-i)}(Y_j \supset ((X_j \supset \Box^{(m)}X_j) \wedge (\sim X_j \supset \Box^{(m)} \sim X_j))), \sigma(w)) = T$ for $0 < j \leqq m$ (by clause (4) of B),
5. $V(\Box^{(m-i)}(Y_j \supset (\Diamond(Y_{j+1} \wedge X_{j+1}) \wedge \Diamond(Y_{j+1} \wedge \sim X_{j+1}))), \sigma(w)) = T$ if $Q_{j+1} = \forall$ and $0 \leqq j < m$ (by clause (5) of B),
6. $V(\Box^{(m-i)}(Y_m \supset A'), \sigma(w)) = T$ (by clause (6) of B),
7. either $V(\Box^{(m-i)}X_j, \sigma(w)) = T$ or $V(\Box^{(m-i)} \sim X_j, \sigma(w)) = T$ for $j \leqq i$ (by the induction hypothesis for $j < i$ and by 2 and 4 above for $j = i$),
8. $V(X_j, \sigma(w)) = V(X_j, \sigma(w'))$ if $j < i$ and w' is the prefix of w of length $i-1$ (by 7 above),
9. $V(X_i, \sigma(w)) = T$ if $Q_i = \forall$ and w ends in 1 (by 2 and 5 above),
10. $V(X_i, \sigma(w)) = F$ if $Q_i = \forall$ and w ends in 0 (by 2 and 5 above).

We leave it to the reader to convince himself that such a mapping exists because B is K-satisfiable.

We may show by induction on $m - i$ that if $|w| = i$, then

$$Q_{i+1}X_{i+1} \cdots Q_mX_mA'(V(X_1, \sigma(w)), \cdots, V(X_i, \sigma(w)), X_{i+1}, \cdots, X_m) = T.$$

If $|w| = m$ then $V(Y_m, \sigma(w)) = T$ and $V(Y_m \supset A', \sigma(w)) = T$. Thus $V(A', \sigma(w)) = T$, which implies the equality for $m - i = 0$.

Let $|w| = i - 1$. It is straightforward to show that 8, 9, 10 above and the induction hypothesis imply the equality for w.

It remains to be shown that B can be constructed in logarithmic space given A. Technically speaking, we should be considering a mapping from Δ^*_{QBF} to Δ^*_{MF}, but it takes only space $\log n$ to check that a member of Δ^*_{QBF} is a member of QBF, so that we can essentially ignore non-well formed formulas. The ability to count the number of quantifiers in A is really all that is necessary in order to construct B. This amounts to a $\log n$ space bound. We leave the details to the reader. Q.E.D.

We originally just showed that $\mathbf{B}_\omega$ was log space reducible to each of T and $S4$. Subsequently S. K. Thomason showed us how to extend the proof to obtain the result for all systems between K and $S4$.

4. Space lower bounds for provability in *K*, *T*, and *S*4. We begin by trying to find the most efficient log space reductions of $\mathbf{B}_\omega$ to each of K, T, and $S4$.

LEMMA 4.1. *For each $S \in \{K, T, S4\}$ there is a function f_S such that $\mathbf{B}_\omega \leqq_{\log} S$-PROVABLE via f_S where f_S is length $l(n)$ bounded and*

(i) *$S = K$ implies $l(n) = O(n^3/\log^2 n)$,*

(ii) *$S = T$ implies $l(n) = O(n^2/\log^2 n)$,*

(iii) *$S = S4$ implies $l(n) = O(n \log n)$.*

Proof. Let $A = Q_1X_1 \cdots Q_mX_mA'$ where $A' \in \mathrm{BF}$ and let $n = |A|$. Without loss of generality we can assume that $X_i = \not{c}\#i\$$ where $\#i$ is the ith binary string in the ordering $\lambda, 0, 1, 00, 01, 10, 11, 000, \cdots$. It is important to notice that $|X_i| \leqq 2 + \log i$. If the Z_i's and Y_i's are chosen as follows, $Y_i = \not{c}\#(m+i)\$$ for $1 \leqq i \leqq m$, and $Z_i = \not{c}\#(2m+i)\$$ where $1 \leqq i \leqq 1 + \log m$ then $|Y_i| \leqq 1 + \log m$ and $|Z_i| \leqq 2 + \log m$. Note that $m = O(n/\log n)$.

Technically speaking, in Theorem 3.1 we reduced $\mathbf{B}_\omega$ to the complement of S-PROVABLE. By Fact 2.7 there is no loss (except for constant factors) in using the length bound of the reduction of $\mathbf{B}_\omega$ to the complement of S-PROVABLE as the length bound of the reduction of $\mathbf{B}_\omega$ to S-PROVABLE itself.

Case (i). $S = K$. To begin with we more efficiently encode $\Box^{(m)}D$ as $D \wedge \Box(D \wedge \Box(D \wedge \cdots (D \wedge \Box D)) \cdots)$ so that $|\Box^{(m)}D| = O(m|D|)$. Another improvement is to factor $\Box^{(m)}$ out using the rule $\Box^{(m)}(C \wedge D) \equiv \Box^{(m)}C \wedge \Box^{(m)}D$. Notice also that $|Y_i \equiv \beta_{i1}Z_1 \wedge \cdots \wedge \beta_{i(1+\log m)}Z_{1+\log m}| = O(\log^2 m)$. From this we can see that (4)· and (6) dominate the length of B with lengths $O(m^3 \log m)$ and $O(mn)$ respectively. We have that $|B|$ is $O(n^3/\log^2 n)$,

Case (ii). $S = T$. We may replace $\Box^{(m)}$ with just $\Box^m$. Again (4) and (6) dominate with lengths $O(m^2)$ and $O(n)$ respectively. Hence $|B|$ is $O(n^2/\log n)$.

Case (iii). We replace $\Box^{(m)}$ with simply $\Box$. In this case (1) and (6) dominate with lengths $O(m \log m)$ and $O(n)$ respectively. Hence $|B|$ is $O(n \log n)$. Q.E.D.

In the spirit of Stockmeyer [11] we use the lemma to show lower bounds on the space complexity of provability in K, T, and $S4$.

THEOREM 4.2. *If* $S \in \{K, T, S4\}$ *and* S-PROVABLE $\in$ NSPACE$(s(n))$, *then there is a constant* $c > 0$ *such that*

(i) $S = K$ *implies* $s(n) > c(n/\log n)^{1/6}$ *for infinitely many* n,

(ii) $S = T$ *implies* $s(n) > cn^{1/4}$ *for infinitely many* n,

(iii) $S = S4$ *implies* $s(n) > c(n/\log^2 n)^{1/2}$ *for infinitely many* n.

Proof. We begin with a proof of (i) which parallels almost exactly a proof of Stockmeyer [11, Cor. 6.6]. Suppose to the contrary that K-PROVABLE $\in$ NSPACE$(s(n))$ where for all $c > 0$, $s(n) \leqq c(n/\log^4 n)^{1/6}$ for all but finitely many n. We may assume that s is a nondecreasing function.

By the hierarchy theorem of Seiferas, Fischer and Meyer [10], we can conclude that there is a set $A \in$ NSPACE(n) such that for all s' if $\lim_n (s'(n+1)/n) = 0$, then $A \notin$ NSPACE$(s'(n))$. By Fact 2.5 $A \leqq_{\log} \mathbf{B}_\omega$ via a length $O(n^2 \log n)$ bounded function. By Lemma 4.1, $\mathbf{B}_\omega \leqq_{\log} K$-PROVABLE via a length $O(n^3/\log^2 n)$ bounded function. Hence $A \leqq_{\log} K$-PROVABLE via a length $O(n^6 \log n)$ bounded function. By Fact 2.3, $A \in$ NSPACE$(s(an^6 \log n) + \log n)$ for some constant $a > 0$. For all $c > 0$ $s(an^6 \log n) \leqq cn$ for all but finitely many n, contradicting the fact that $A \notin$ NSPACE$(s'(n))$ if $\lim_n (s'(n+1)/n) = 0$.

The proofs of (ii) and (iii) are analogous if we use the facts that if $A \in$ NSPACE(n), then $A \leqq_{\log} T$-PROVABLE via a length $O(n^4)$ bounded function and $A \leqq_{\log} S4$-PROVABLE via a length $O(n^2 \log^2 n)$ bounded function. Q.E.D.

5. Space upper bounds for provability in *K*, *T*, and *S*4. In this section we show that for $S \in \{K, T, S4\}$, S-PROVABLE $\in$ P-SPACE. In essence we actually show that S-SATISFIABLE $\in P$-SPACE. This may be surprising to some since there are modal formulas A_n of length $O(n \log(n) \log\log(n))$ with the property that $A_n \in S4$-SATISFIABLE and if (W, R, V) is an $S4$-model of A_n, then the cardinality of W is $\geqq 2^n$. What allows us to compute S-SATISFIABLE in polynomial space is the fact that if A is S-SATISFIABLE, then it is in a tree-like model structure, with each branch of only polynomial length. Hence the structure can be constructed one branch at a time.

THEOREM 5.1. *For* $S \in \{K, T, S4\}$, S-PROVABLE $\in P$-SPACE.

Proof. The algorithms that we will give are simply reformulations of the corresponding algorithms of Kripke [6] in such a way to optimize the space used. We do not necessarily give the most efficient algorithms, because we wish to present algorithms that are both understandable and run in polynomial space.

We begin with a procedure K-WORLD which has parameters $(\mathcal{T}, \mathcal{F}, \tilde{\mathcal{T}}, \tilde{\mathcal{F}})$, where each parameter is a finite set of modal formulas; the value of K-WORLD $(\mathcal{T}, \mathcal{F}, \tilde{\mathcal{T}}, \tilde{\mathcal{F}})$ is **true** if there is a K-model (W, R, V) and a $w \in W$ such that

$$V\Bigg(\bigwedge_{A\in\mathcal{T}} A \wedge \bigwedge_{A\in\mathcal{F}} \sim A \wedge \bigwedge_{A\in\tilde{\mathcal{T}}} \Box A \wedge \bigwedge_{A\in\tilde{\mathcal{F}}} \sim\Box A, w\Bigg) = T,$$

otherwise its value is **false.** More intuitively, K-WORLD$(\mathcal{T}, \mathcal{F}, \tilde{\mathcal{T}}, \tilde{\mathcal{F}})$ is **true** if there is a world w in which all the formulas of $\mathcal{T}$ are true, all the formulas of $\mathcal{F}$ are false, in each world accessible from w each member of $\tilde{\mathcal{T}}$ is true, and for each member, B, of $\tilde{\mathcal{F}}$ there is a world accessible from w where B is false.

```
procedure $K$-WORLD$(\mathcal{T}, \mathcal{F}, \tilde{\mathcal{T}}, \tilde{\mathcal{F}})$:
begin
  if $\mathcal{T} \cup \mathcal{F} \not\subseteq$ VAR then
    begin
1.    choose $A \notin \mathcal{T} \cup \mathcal{F} -$ VAR;
2.    if $A = \sim B$ and $A \in \mathcal{T}$ then return $K$-WORLD$(\mathcal{T} - \{A\}, \mathcal{F} \cup \{B\}, \tilde{\mathcal{T}}, \tilde{\mathcal{F}})$;
3.    if $A = \sim B$ and $A \in \mathcal{F}$ then return $K$-WORLD$(\mathcal{T} \cup \{B\}, \mathcal{F} - \{A\}, \tilde{\mathcal{T}}, \tilde{\mathcal{F}})$;
4.    if $A = B \wedge C$ and $A \in \mathcal{T}$ then return $K$-WORLD$((\mathcal{T} \cup \{B, C\}) - \{A\}, \mathcal{F},$
        $\tilde{\mathcal{T}}, \tilde{\mathcal{F}})$;
5.    if $A = B \wedge C$ and $A \in \mathcal{F}$ then return $K$-WORLD$(\mathcal{T}, (\mathcal{F} \cup \{B\}) - \{A\}, \tilde{\mathcal{T}},$
        $\tilde{\mathcal{F}}) \vee K$-WORLD$(\mathcal{T}, (\mathcal{F} \cup \{C\}) - \{A\}, \tilde{\mathcal{T}}, \tilde{\mathcal{F}})$;
6.    if $A = \Box B$ and $A \in \mathcal{T}$ then return $K$-WORLD$(\mathcal{T} - \{A\}, \mathcal{F}, \tilde{\mathcal{T}} \cup \{B\}, \tilde{\mathcal{F}})$;
7.    if $A = \Box B$ and $A \in \mathcal{F}$ then return $K$-WORLD$(T, F - \{A\}, \tilde{\mathcal{T}}, \tilde{\mathcal{F}} \cup \{B\})$
    end;
  if $\mathcal{T} \cup \mathcal{F} \subseteq$ VAR then
    begin
8.    if $\mathcal{T} \cap \mathcal{F} \neq \emptyset$ then return false;
9.    if $\mathcal{T} \cap \mathcal{F} = \emptyset$ then return $\bigwedge_{B \in \tilde{\mathcal{F}}} K$-WORLD$(\tilde{\mathcal{T}}, \{B\}, \emptyset, \emptyset)$
    end
end
```

(*Note.* The conjunction over the empty set is defined to be **true**.)

On line 1 we say "choose $A \in \mathcal{T} \cup \mathcal{F} -$VAR". We do not intend this as a nondeterministic step; it is just that it does not matter in what specific order the lists $\mathcal{T}$ and $\mathcal{F}$ are maintained.

The proof that K-WORLD works is essentially the same as that for Kripke's corresponding algorithm [6]. We can now give an algorithm for testing whether or not a modal formula $A \in K$-PROVABLE.

Test for $A \in K$-PROVABLE.

```
begin
  read $A$;
  $v \leftarrow \sim K$-WORLD($\{\sim A\}, \varnothing, \varnothing, \varnothing$);
end
```

The value of v determines if A is K-provable. This of course exploits the fact that $A \in K$-PROVABLE if and only if $\sim A \notin K$-SATISFIABLE.

We now examine the space complexity of this algorithm. The recursive nature of the algorithm is implemented on a Turing machine by simulating a stack. At each level of recursion the members of $\mathscr{T}$, $\mathscr{F}$, $\tilde{\mathscr{T}}$, $\tilde{\mathscr{F}}$ are just sets of subformulas of $\sim A$ so that their values an be indicated by using "pointers" to $\sim A$.

To implement the pointers, copy the original formula onto the stack and place a mark on the major connective of each subformula pointed to. There are four types of marks, one for each of the four subsets. The storage at each level of recursion is $O(n)$. We will also show that the numbers of levels of recursion is $O(n)$ so that the total space used is $O(n^2)$.

If $\mathscr{S}$ is a finite set of formulas then define $|\mathscr{S}| = \sum_{A \in \mathscr{S}} |A|$. We show by induction on $n = |\mathscr{T}| + |\mathscr{F}| + |\tilde{\mathscr{T}}| + |\tilde{\mathscr{F}}|$ that K-WORLD$(\mathscr{T}, \mathscr{F}, \tilde{\mathscr{T}}, \tilde{\mathscr{F}})$ has at most $2n+1$ levels of recursion. Assume the result for all numbers $<n$. Let the first recursive call of K-WORLD$(\mathscr{T}, \mathscr{F}, \tilde{\mathscr{T}}, \tilde{\mathscr{F}})$ be to $(\mathscr{T}', \mathscr{F}', \tilde{\mathscr{T}}', \tilde{\mathscr{F}}')$. If $\mathscr{T} \cup \mathscr{F} \neq \varnothing$, then by a case-by-case analysis $|\mathscr{T}'| + |\mathscr{F}'| + |\tilde{\mathscr{T}}'| + |\tilde{\mathscr{F}}'| < n$. If $\mathscr{T} \cup \mathscr{F} = \varnothing$, then we must be at line 9 of the program, so that $\mathscr{F}' \neq \varnothing$, which reduces us to the case $\mathscr{T}' \cup \mathscr{F}' \neq \varnothing$. Hence every two levels of recursion reduces $|\mathscr{T}| + |\mathscr{F}| + |\tilde{\mathscr{T}}| + |\tilde{\mathscr{F}}|$ by at least 1. Thus K-WORLD$(\{\sim A\}, \varnothing, \varnothing, \varnothing)$ has recursion depth $\leqq 2|A| + 1$.

We now argue that T-PROVABLE $\in$ SPACE(n^3). Only slight modifications of the procedure K-WORLD are necessary to produce the analogous procedure T-WORLD.

(T-1) Replace all K's with T's.

(T-2) Replace line 6 with
"**if** $A = \Box B$ and $A \in \mathscr{T}$ **then return** T-WORLD$((T \cup \{B\}) - \{A\}, \mathscr{F}, \tilde{\mathscr{T}} \cup \{B\}, \tilde{\mathscr{F}})$."

If $\mathscr{S}$ is a set of modal formulas then define $\deg(\mathscr{S}) = \max\{$modal degree of $C : C \in \mathscr{S}\}$. In this case the storage at each level of recursion remains $O(n)$ but the recursion depth is $O(n^2)$. This can be seen by noticing that there can be at most $O(n)$ successive recursive calls all with $\mathscr{T} \cup \mathscr{F} \not\subseteq$ VAR; and if $\tilde{\mathscr{T}}'$, $\tilde{\mathscr{F}}'$ and $\tilde{\mathscr{T}}''$, $\tilde{\mathscr{F}}''$ are the values of $\tilde{\mathscr{T}}$ and $\tilde{\mathscr{F}}$ on successive calls with $\mathscr{T} \cup \mathscr{F} \subseteq$ VAR, then $\deg(\tilde{\mathscr{T}}'' \cup \tilde{\mathscr{F}}'') < \deg(\tilde{\mathscr{T}}' \cup \tilde{\mathscr{F}}')$. Since the degree of any set of subformulas of $\sim A$ is $\leqq n$, then there can be at most $O(n^2)$ levels of recursion. The total space is $O(n^3)$.

More elaborate changes to T-WORLD are necessary to obtain an analogous procedure $S4$-WORLD. We need the ability to check if the current world is exactly the same as a prior world. To do this we introduce a new parameter $\mathscr{L}$ which is a sequence $\{(\mathscr{T}_1, B_1), (\mathscr{T}_2, B_2), \cdots, (\mathscr{T}_k, B_k)\}$ where $\mathscr{T}_1 \subseteq \mathscr{T}_2 \subseteq \cdots \subseteq \mathscr{T}_k$ are sets of modal formulas and $B_1, \cdots, B_k$ are modal formulas. The value of

$S4$-WORLD$(\mathcal{T}, \mathcal{F}, \tilde{\mathcal{T}}, \tilde{\mathcal{F}}, \mathcal{L})$ is **true** if there is an $S4$-model (W, R, V) and a sequence of words $w_1, \cdots, w_k, w$ in W with the properties: (a) w_{i+1} is accessible from w_i and w is accessible from w_k; (b) $V(\bigwedge_{A \in \mathcal{T}_i} A \wedge \sim B_i, w_i) = T$ for each i; and (c) $V(\bigwedge_{A \in \mathcal{T}} A \wedge \bigwedge_{A \in \mathcal{F}} \sim A \wedge \bigwedge_{A \in \tilde{\mathcal{T}}} \Box A \wedge \bigwedge_{A \in \tilde{\mathcal{F}}} \sim \Box A, w) = T$.

For clarity we give the entire algorithm for $S4$-WORLD. The major changes are in lines 6 and 9.

```
procedure S4-WORLD($\mathcal{T}, \mathcal{F}, \tilde{\mathcal{T}}, \tilde{\mathcal{F}}, \mathcal{L}$);
begin
  if $\mathcal{T} \cup \mathcal{F} \not\subseteq$ VAR then
    begin
1.    choose $A \in \mathcal{T} \cup \mathcal{F} -$ VAR;
2.    if $A = \sim B$ and $A \in \mathcal{T}$ then return S4-WORLD($\mathcal{T} - \{A\}, \mathcal{F} \cup \{B\}, \tilde{\mathcal{T}}, \tilde{\mathcal{F}}, \mathcal{L}$);
3.    if $A = \sim B$ and $A \in \mathcal{F}$ then return S4-WORLD($\mathcal{T} \cup \{B\}, \mathcal{F} - \{A\}, \tilde{\mathcal{T}}, \tilde{\mathcal{F}}, \mathcal{L}$);
4.    if $A = B \wedge C$ and $A \in \mathcal{T}$ then return S4-WORLD($(\mathcal{T} \cup \{B, C\}) - \{A\}, \mathcal{F}, \tilde{\mathcal{T}}, \tilde{\mathcal{F}}, \mathcal{L}$);
5.    if $A = B \wedge C$ and $A \in \mathcal{F}$ then return S4-WORLD($\mathcal{T}, (\mathcal{F} \cup \{B\}) - \{A\}, \tilde{\mathcal{T}}, \tilde{\mathcal{F}}, \mathcal{L}$) $\vee$ S4-WORLD($\mathcal{T}, (\mathcal{F} \cup \{C\}) - \{A\}, \tilde{\mathcal{T}}, \tilde{\mathcal{F}}, \mathcal{L}$);
6.    if $A = \Box B$ and $A \in \mathcal{T}$ then return S4-WORLD($(\mathcal{T} \cup \{B\}) - \{A\}, \mathcal{F}, \tilde{\mathcal{T}} \cup \{B\}, \tilde{\mathcal{F}}, \mathcal{L}$);
7.    if $A = \Box B$ and $A \in \mathcal{F}$ then return S4-WORLD($\mathcal{T}, \mathcal{F} - \{A\}, \tilde{\mathcal{T}}, \tilde{\mathcal{F}} \cup \{B\}, \mathcal{L}$)
    end;
  if $\mathcal{T} \cup \mathcal{F} \subseteq$ VAR then
    begin
8.    if $\mathcal{T} \cap \mathcal{F} \neq \varnothing$ then return false;
9.    if $\mathcal{T} \cap \mathcal{F} = \varnothing$ and $\tilde{\mathcal{F}} \neq \varnothing$ then return $\bigwedge_{B \in \tilde{\mathcal{F}}, (\tilde{\mathcal{T}}, B) \notin \mathcal{L}}$ S4-WORLD($\tilde{\mathcal{T}}, \{B\}, \tilde{\mathcal{T}}, \varnothing, \mathcal{L} \cdot (\tilde{\mathcal{T}}, B)$);
      return true
    end
end
```

(Note that $\mathcal{L} \cdot (\tilde{\mathcal{T}}, B)$ is the sequence $\mathcal{L}$ extended with $(\tilde{\mathcal{T}}, B)$ and the conjunction over the empty set defined to be **true**.)

Test for $A \in S4$-PROVABLE.

```
begin
  read $A$
  $v \leftarrow \sim$S4-WORLD($\{\sim A\}, \varnothing, \varnothing, \varnothing, \varnothing$);
end
```

The value of v determines if A is $S4$-provable. We leave the verification of the algorithm to the reader.

Let $n = |A|$. In order to improve the space complexity of the algorithm we should let $\mathcal{L}$ be a global stack. If $\mathcal{L} = \{(\mathcal{T}_1, B_1), (\mathcal{T}_2, B_2), \cdots, (\mathcal{T}_k, B_k)\}$ then $\mathcal{T}_1 \subseteq \mathcal{T}_2 \subseteq \cdots \subseteq \mathcal{T}_k \subseteq$ subformulas of A. Since no repetitions can occur in the sequence $\mathcal{L}$, then $k \leqq n^2$. Hence $O(n^3)$ storage suffices for $\mathcal{L}$. What remains is an analysis of the number of levels of recursion in $S4$-WORLD. Since $\mathcal{L}$ is now a global stack, then $O(n)$ is all that is needed at each level of recursion. As before there can be at most $O(n)$ successive recursive calls all with $\mathcal{T} \cup \mathcal{F} \not\subseteq \mathrm{VAR}$. Further, because the cardinality of $\mathcal{L}$ is bounded by $O(n^2)$ there can be at most $O(n^2)$ depth of recursive calls with $\mathcal{T} \cup \mathcal{F} \subseteq \mathrm{VAR}$. Thus the recursion depth is $O(n^3)$. Total space is bounded by $O(n^4)$. Q.E.D.

To summarize the specific space bounds we give this corollary to the proof of Theorem 5.1.

COROLLARY 5.2. *K-PROVABLE $\in$ SPACE(n^2), T-PROVABLE $\in$ SPACE(n^3), and $S4$-PROVABLE $\in$ SPACE(n^4).*

We do not claim that these bounds are best possible, but they do guarantee that these problems are computable in polynomial space.

COROLLARY 5.3. *For $S \in \{K, T, S4\}$, S-PROVABLE is log space complete in P-SPACE.*

6. The complexity of provability in $S5$. The provability problem in $S5$ seems to be easier than that for the systems we have considered so far. For example, in T, K and $S4$ we can construct satisfiable formulas which are only satisfiable in exponential size model structures. This phenomenon does not happen for $S5$-satisfiability. Hence we can only show that $S5$-SATISFIABLE is log space complete in NP-TIME.

LEMMA 6.1. *If $A \in S5$-SATISFIABLE has m modal connectives, then A is $S5$-satisfiable in an $S5$-model with $\leqq m+1$ worlds.*

PROOF. Let A be satisfied in an $S5$-model (W, R, V). We may assume that $u\,R\,v$ for all $u, v \in W$. We construct a mapping σ from all instances of subformulas of A into W in such a way that A is $S5$-satisfied in $(\mathrm{Range}(\sigma), R|\mathrm{Range}(\sigma), V|\mathrm{Range}(\sigma))$ and the cardinality of $\mathrm{Range}(\sigma) \leqq m+1$.

The function σ is defined inductively on the instances of subformulas of A.

(i) Choose $\sigma(A) \in w$ such that $V(A, \sigma(A)) = T$,

(ii) $\sigma(C) = \sigma(B)$ if $B = {\sim}C$,

(iii) $\sigma(C) = \sigma(D) = \sigma(B)$ if $B = C \wedge D$,

(iv) $\sigma(C) = \sigma(B)$ if $B = \Box C$ and $V(B, \sigma(B)) = T$,

(v) if $B = \Box C$ and $V(B, \sigma(B)) = F$, then choose $\sigma(C) \in W$ in such a way that $V(C, \sigma(C)) = F$.

Clearly the cardinality of $\mathrm{Range}(\sigma) \leqq m+1$. Let $W' = \mathrm{Range}(\sigma)$ and let R' and V' be respectively R and V restricted to W'. We may show inductively that for each instance of a subformula B of A, $V(B, \sigma(B)) = V'(B, \sigma(B))$. Q.E.D.

THEOREM 6.2. *$S5$-SATISFIABLE is log space complete in NP-TIME.*

Proof. Trivially $\mathbf{B_1}$ is log space reducible to $S5$-SATISFIABLE, $\exists X_1 \cdots \exists X_m A \in \boldsymbol{B_1}$ if and only if $A \in S5$-SATISFIABLE.

It remains to show that $S5$-SATISFIABLE $\in NP$-TIME. Let $A \in$ MF and let $|A| = n$. By Lemma 6.1 $A \in S5$-SATISFIABLE if and only if there is an $S5$-model (W, R, V) with the cardinality of $W \leqq n+1$ and a $w \in W$ such that $V(A, w) = T$. Such a model can be "guessed" nondeterministically and checked in polynomial time. Q.E.D.

7. Conclusion. It would be interesting to determine cut off points between $S4$ and $S5$ where the complexity of satisfiability changes from complete in P-SPACE to complete in NP-TIME. We conjecture that $S4.3$-SATISFIABLE is log space complete in NP-TIME.

Another interesting area is the complexity of provability or validity in intuitionistic propositional logic (IC). J. Cherniavsky [1] claimed that the nonvalid formulas in IC can be determined in NP-TIME. He has since informed us of mistakes in his proof. We conjecture that provability in IC is log space complete in P-SPACE. There is a very simple reduction of IC to $S4$ given by McKinsey and Tarski [7]. Define τ inductively:

(i) $\tau(A)=A$ if A is a variable,
(ii) $\tau(A \wedge B)=\tau(A) \wedge \tau(B)$,
(iii) $\tau(A \supset B)=\Box(\tau(A) \supset \tau(B))$,
(iv) $\tau(\sim A)=\Box \sim \tau(A)$.

Now, A is IC-provable if and only if $\tau(A)$ is $S4$-provable. Thus IC-PROVABLE $\in P$-SPACE. All that remains is to show that $\mathbf{B}_\omega$ or some other complete set is log space reducible to IC-PROVABLE.

Acknowledgments. We appreciate the suggestions of S. K. Thomason in obtaining the results of § 3. Also we are indebted to J. Cherniavsky in providing helpful ideas that we used in our algorithms for K, T, and $S4$.

Note added in proof. M. J. Fischer has suggested a new construction which improves the bounds of § 4. For example $S=K$ implies $l(n)=O(n^2/\log n)$ in Lemma 4.1.

REFERENCES

[1] J. Cherniavsky, *The complexity of some non-classical logics*, 14th Ann. IEEE Symp. on Switching and Automata Theory (1973), pp. 209–213.

[2] S. A. Cook, *The complexity of theorem proving procedures*, Proc. 3rd Ann. ACM Symp. on Theory of Computing (1971), pp. 151–158.

[3] J. E. Hopcroft and J. D. Ullman, *Formal Languages and their Relation to Automata*, Addison-Wesley, Reading, MA, 1969.

[4] G. E. Hughes and M. J. Cresswell, *An Introduction to Modal Logic*, Methuen, London, 1968.

[5] N. D. Jones, *Space-bounded reducibility among combinatorial problems*, J. Comput. System Sci., 11 (1975), pp. 68–85.

[6] S. A. Kripke, *Semantical analysis of modal logic, I. Normal modal propositional calculi*, Z. Math. Logik Grundlagen Math., 9 (1963), pp. 67–96.

[7] J. C. C. McKinsey and A. Tarski, *Some theorems about the sentential calculi of Lewis and Heyting*, J. Symbolic Logic, 13 (1948), pp. 1–15.

[8] V. R. Pratt, *Semantical considerations on Floyd–Hoare logic*, 17th Ann. IEEE Symposium on Foundations of Computer Science (1976), pp. 109–121.

[9] W. J. Savitch, *Relationship between nondeterministic and deterministic tape complexities*, J. Comput. System Sci., 4 (1970), pp. 177–192.

[10] J. I. Seiferas, M. J. Fischer and A. R. Meyer, *Refinements of the nondeterministic time and space hierarchies*, 14th Ann. IEEE Symp. on Switching and Automata Theory (1973), pp. 130–137.

[11] L. J. Stockmeyer, *The polynomial-time hierarchy*, IBM Tech. Rep. Yorktown Heights, NY, 1975.

[12] L. J. Stockmeyer and A. R. Meyer, *Word problems requiring exponential time: Preliminary report*, Proc. 5th Ann. ACM Symp. on Theory of Computing (1973), pp. 1–9.

SIAM J. COMPUT.
Vol. 6, No. 3, September 1977

A LINEAR TIME ALGORITHM FOR A $2\times n$ TRANSPORTATION PROBLEM*

D. L. ADOLPHSON AND G. N. THOMAS†

Abstract. This paper considers a special case of the standard transportation problem obtained by restricting the number of origins to two. Necessary and sufficient conditions are established which lead to a direct construction of the optimal solution. Using an extension of a recent selection algorithm, an algorithm is developed to solve this special case of the transportation problem in $O(n)$ time in the worst case. The algorithm applies to both capacitated as well as uncapacitated problems.

Key words. transportation problem, selection algorithm, computational complexity

1. Introduction. One of the most important problem structures in operations research is the standard transportation problem which can be described mathematically as follows: find a vector $x=(x_{ij})$ which minimizes

$$z=\sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij} \tag{1.1}$$

while satisfying

$$\sum_{j=1}^{n} x_{ij}=a_i, \qquad i=1,2,\cdots,m; \tag{1.2}$$

$$\sum_{i=1}^{m} x_{ij}=b_j, \qquad j=1,2,\cdots,n; \tag{1.3}$$

and

$$x_{ij}\geqq 0 \qquad i=1,2,\cdots,m; \quad j=1,2,\cdots,n; \tag{1.4}$$

where $c=(c_{ij})$, $a=(a_i)$ and $b=(b_j)$ are known constant vectors. Furthermore, it is assumed that $\sum_{i=1}^{m} a_i=\sum_{j=1}^{n} b_j$ so that a feasible solution exists.

The usual interpretation of the model is that we are given a transportation network with m origins and n destinations in which a_i represents the available supply at origin i and b_j represents the required demand at destination j. The constants c_{ij} represent a unit transportation cost from origin i to destination j. The objective is to find a minimum cost transportation schedule subject to the supply limitations at the origins and satifying the demand requirements at the destinations. Although this is the most common interpretation of the model, there are other problem situations such as production and inventory problems and scheduling problems which have the same mathematical structure.

An important variation of the transportation is obtained by assuming upper bounds on the x_{ij} values. Constraint (1.4) would then be replaced by

$$0\leqq x_{ij}\leqq u_{ij}, \qquad i=1,2,\cdots,m; \quad j=1,2,\cdots,n. \tag{1.4'}$$

* Received by the editors November 18, 1975, and in final revised form October 29, 1976.

† Department of Finance, Business Economics and Quantitative Methods, University of Washington, Seattle, Washington 98195.

Because of the broad applicability of this model, there has been a great deal of effort towards development of efficient algorithms for solving this problem. The most common approaches to solving this problem are the primal dual transportation algorithm and the transportation simplex algorithm. Both of these approaches are iterative methods in which the computation time is at least a higher order polynomial function of the number of nodes in the model. The primal dual approach has been analyzed in detail by Edmonds and Karp [3]; there is no theoretically known bound on the transportation simplex method although empirical studies have shown that it is roughly comparable to the primal dual method in terms of computation time.

In this paper, we restrict our attention to the special case in which $m = 2$ (or equivalently, $n = 2$). The main theorem is stated and proved in § 2 and provides a direct construction of an optimal solution. An implementation of the algorithm, given in § 3, is shown to require at most $O(n)$ steps. Section 4 provides a summary and conclusions.

2. The main theorem. The transportation problem with two rows is much simpler than the general problem. The reason for this can be seen by considering the numerical example in Fig. 1.
It can easily be verified that the solution given is an optimal solution for this problem. For this example the columns have been arranged so that the values of $d_j = c_{1j} - c_{2j}$ are nondecreasing. Given this ordering of the columns it will be shown by the following theorem that an optimal solution can be found by filling the top row from left to right and then filling in the values on the bottom row.

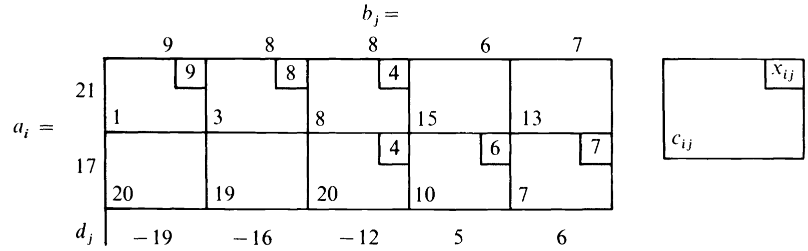
THEOREM 1. *Let* $x = (x_{ij})$ *be a vector satisfying* (1.2), (1.3) *and* (1.4) *for a* $2 \times n$ *transportation problem. Define index sets* $J = \{j | x_{1j} > 0\}$ *and* $K = \{k | x_{2k} > 0\}$. *If* $d_j = c_{1j} - c_{2j}$, *the solution* x *is optimal if and only if* $d_j \leqq d_k$ *for all* $j \in J$ *and all* $k \in K$.

Proof. The transportation problem can be viewed as a special case of the following minimum cost network flow problem:

$$\text{(2.1)} \qquad \text{Minimize} \sum_{(i,j)} c_{ij} x_{ij},$$

$$\text{(2.2)} \qquad \text{subject to} \sum_i (x_{ij} - x_{ji}) = \begin{cases} -v, & j = \text{source}, \\ 0, & j \neq \text{source, sink}, \\ v, & j = \text{sink}, \end{cases}$$

$$\text{(2.3)} \qquad 0 \leqq x_{ij} \leqq b_{ij}.$$



$b_j =$

	9	8	8	6	7
$a_i =$ 21	9 1	8 3	4 8	15	13
17	20	19	4 20	6 10	7 7
d_j	−19	−16	−12	5	6

FIG. 1

It is a well known result of minimum cost networks (see [4] for example) that a vector $x=(x_{ij})$ satisfying (2.2) and (2.3) is optimal if and only if the network contains no negative cycles based on the costs c^*_{ij} defined by

$$c^*_{ij}=\begin{cases} c_{ij}, & \text{if } 0\leqq x_{ij}<b_{ij},\\ \infty, & \text{if } x_{ij}=b_{ij},\\ -c_{ij}, & \text{if } 0<x_{ji}\leqq b_{ji}.\end{cases}$$

A "cycle" is used here to donate a sequence of edges $(i_0,j_0),(i_1,j_1),\cdots,(i_p,j_p)$ where $j_k=i_{k+1}$, $k=0,\cdots,p-1$ and $j_p=i_0$. We can assume without loss of generality that each i_k is distinct (i.e., the cycle is a simple cycle) since the existence of a negative cycle implies the existence of a negative simple cycle.

For a $2\times n$ transportation problem any negative simple cycle must have the form shown in Fig. 2.

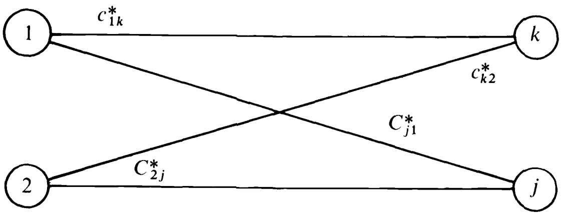


FIG. 2

Note that $C^*_{k2}<\infty$ implies $X_{2k}>0$ which implies $k\in K$. Similarly $C^*_{j1}<\infty$ implies $j\in J$. The total cost of the cycle is $(C^*_{2j}+C^*_{k2})+(C^*_{2j}+C^*_{j1})=(C_{1k}-C_{2k})+(C_{2j}-C_{1j})=d_k-d_j$.

From this we see that a negative cycle exists if and only if $d_k-d_j<0$ for some $k\in K$ and $j\in J$. In other words a given solution is optimal if and only if no negative cycles exist which occurs if and only if $d_j\geqq d_k$ for all $j\in J$ and $k\in K$. Q.E.D.

3. The algorithm. One way to construct a solution to a $2\times n$ transportation problem is to sort the columns according to d_j values and then fill in the first row starting with the smallest d_j value. However, we shall see that it is not necessary to sort the d_j values. In order to see this we consider the following numerical example (see Fig. 3) which is similar to the example of the previous section except that here all b_j values are 1 and the a_i values have been changed accordingly.

		$b_j=$				
		1	1	1	1	1
$a_i=$	3	$\boxed{1}$ 1	$\boxed{1}$ 3	$\boxed{1}$ 8	15	13
	2	20	19	20	$\boxed{1}$ 10	$\boxed{1}$ 7
	d_j	−19	−16	−12	5	6

FIG. 3

We see from this example that the following general procedure applies for the special case when $b_j = 1$. We will assume for the present that the n d_j values are distinct.

1. Given $d_1, d_2, \cdots, d_n$ find the a_1st smallest element (i.e., apply a selection algorithm).
2. Letting d_k be the value selected above set $x_{ij} \leftarrow 1$ if $d_j \leqq d_k$; otherwise set $x_{2j} \leftarrow 1$.

Since selection can be done in $O(n)$ time (see [1] and [4]) and since step 2 involves a single pass through the data, the procedure is clearly $O(n)$ for this special case. For the more general case with $b_j \neq 1$ we need to be able to solve the following "weighted selection problem".

Given a set S of reals and a function W from S into reals and given a nonnegative real number $w \leqq \sum_{x \in S} W(x)$, the weighted selection problem is to find an element $z \in S$ such that

$$L(z) < w \leqq U(z)$$

where

$$L(z) = \sum \{W(x) | x \in S \text{ and } x < z\},$$

$$U(z) = \sum \{W(x) | x \in S \text{ and } x \leqq z\}.$$

The following recursive procedure WSELECT, which is an extension of the selection algorithm given in [5], may be used to solve the weighted selection problem.

ALGORITHM WSELECT(S, W, w, z).

Step 1 (find median). Use a median finding algorithm to select the $\lceil |S|/2 \rceil$-ranked element of S (break ties arbitrarily). Let y be this element.

Step 2 (discard or halt). If $w > U(y)$ then use WSELECT recursively with

$$S := \{x \in S | x > y\}$$

and

$$w := w - U(y).$$

If $w \leqq L(y)$ then use WSELECT recursively with $S := \{x \in S | x < y\}$ and w unchanged. If $L(y) < w \leqq U(y)$ then $z := y$, stop.

The following numerical example illustrates the workings of WSELECT.

$$S = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10),$$

$$W(S) = (15, 7, 9, 22, 3, 12, 18, 14, 9, 16),$$

$$w = 105.$$

Step 1. $y = 5$, $L(y) = 53$, $U(y) = 56$.
Step 2. $w > U(y)$, $S := \{6, 7, 8, 9, 10\}$
$w := 105 - 56 = 49$.
Step 1. $y = 8$, $L(y) = 30$, $U(y) = 4$.
Step 2. $w > U(y)$, $S := \{9, 10\}$
$w := 49 - 44 = 5$.

Step 1. $y=9$, $L(y)=0$, $U(y)=9$.

Step 2. $L(y)<w<U(y)$, so $z=9$, stop.

Now that we have a procedure for solving the weighted selection problem we can state the algorithm for solving a $2\times n$ transportation problem.

ALGORITHM T. Given (c_{ij}), (a_i), (b_j) for $i=1,2$; $j=1,2,\cdots,n$; find an optimal solution to the $2\times n$ transportation problem.

Step 1 (compute differences). $d_j:=c_{ij}-c_{2j}, j=1,2,\cdots,n$.

Step 2 (find breakpoint). Use WSELECT with $S=\{d_1,d_2,\cdots,d_n\}$,

$$w(d_j)=bj,$$

$$w=a_1,$$

$$z=\text{the value of } d \text{ returned},$$

Step 3 (construct solution). For $j=1,2,\cdots,n$:

```
if $d_j<z$, then $x_{1j}:=b_j$, $x_{2j}:=0$;
if $d_j>z$, then $x_{1j}:=0$, $x_{2j}:=b_j$;
if $d_j=z$, then
    $x_{1j}:=\min(b_j, a_1-L(z))$,
    $x_{2j}:=(b_j-x_{1j})$,
    $L(z):=L(z)-x_{1j}$.
```

The procedure above can also be used to solve the capacitated transportation problem defined in § 1 when $m=2$. The key here is to realize that an upper bound on one row induces a lower bound on the other row. Given upper bounds u_{ij} on the x_{ij} values we can define lower bounds l_{ij} as

$$\begin{aligned} l_{1j}&=\max(0, b_j-u_{2j}),\\ l_{2j}&=\max(0, b_j-u_{1j}), \end{aligned} \qquad j=1,2,\cdots,n.$$

Then we can apply Algorithm T using

$$b'_j=b_j-l_{1j}-l_{2j}, \qquad j=1,2,\cdots,n,$$

$$a'_i=a_i-\sum_{j=1}^{n} l_{ij}, \qquad i=1,2,$$

to find

$$x'_{ij}=x_{ij}-l_{ij}, \qquad i=1,2; \quad j=1,2,\cdots,n.$$

The solution to the original capacitated transportation problem is $(x_{ij})=(x'_{ij})+(l_{ij})$.

4. Summary and conclusions. We have proved that the $2\times n$ transportation problem can be solved in linear time. We can consider some possible extensions of these results. First of all, these ideas may possibley be extended to an efficient algorithm for $3\times n$ or even $4\times n$ transportation problems. Second, the ideas may prove useful in a partitioning scheme for a $m \times n$ transportation problem by reducing the solution of these $m \times n$ problems to a series of $2 \times n$ transportation problems.

A third possible extension of these ideas involves the problem of scheduling n jobs on m machines so as to minimize mean finishing time. It has been shown by Bruno et al. [2] that this problem can be formulated as a transportation problem with $n \cdot m$ rows and n columns. The ideas contained in this paper may be particularly useful in providing an efficient solution for this problem with $m = 2$.

REFERENCES

[1] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest and R. E. Tarjan, *Time bounds for selection*, Computer and Systems Sci., 7 (1973), pp. 448–461.

[2] J. Bruno, E. G. Coffman, Jr. and R. Sethi, *Scheduling independent tasks to reduce mean finishing time*, Comm. ACM, 17 (1974), pp. 382–387.

[3] J. Edmonds and R. M. Karp, *Theoretical improvements in algorithm efficiency for network flow problems*, J. Assoc. Comput. Mach., 19 (1972), pp. 248–264.

[4] T. C. Hu, *Integer Programming and Network Flows*, Addison-Wesley, Reading, MA, 1969.

[5] A. Schonhage, M. Paterson and N. Pippenger, *Finding the median*, Theory of Computation Rep. 6, Univ. of Warwick, Coventry, England.

SIAM J. COMPUT.
Vol. 6, No. 3, September 1977

LINEAR-TIME COMPUTATION BY NONDETERMINISTIC MULTIDIMENSIONAL ITERATIVE ARRAYS*

JOEL I. SEIFERAS†

Abstract. It is shown by simulation that every language accepted within time n^d by a nondeterministic one-dimensional Turing machine is accepted in linear time by a nondeterministic d-dimensional iterative array. Conversely, every language accepted in linear time by such an iterative array is accepted within time n^{d+1} by a nondeterministic one-dimensional Turing machine. It follows that the class of languages accepted in linear time by nondeterministic multidimensional iterative arrays is precisely Karp's class NP, that nondeterministic $(d+2)$-dimensional iterative arrays are more powerful than nondeterministic d-dimensional iterative arrays, and that nondeterministic *two*-dimensional iterative arrays are more powerful than the entire class of nondeterministic *multi*dimensional Turing machines. Related deterministic results are surveyed and summarized for comparison.

Key words. iterative array, cellular automaton, nondeterminism, multidimensional, Turing machine, acceptor, on-line, off-line, simulation, time complexity

1. Introduction. Let $NIA(d)$ be the family of languages accepted *in linear time* by nondeterministic d-dimensional iterative arrays. (Deterministic multidimensional iterative arrays have been studied by Cole [2] and Kosaraju [7].) It has been observed [10] that every language accepted by a one-dimensional single-head Turing machine simultaneously within time n^2 and space n belongs to $NIA(2)$. Our main result (Theorem 1) generalizes this observation to $NTIME(n^d) \subseteq NIA(d)$, where $NTIME(T(n))$ is the family of languages accepted within time $T(n)$ by nondeterministic one-dimensional multihead Turing machines.

Conversely, we show that $NIA(d) \subseteq NTIME(n^{d+1})$ (Theorem 2). The two facts together show $\bigcup_d NIA(d) = \bigcup_d NTIME(n^d)$, which is the same as Karp's class NP [6]. We also use both facts in a proof that $NIA(d) \subsetneqq NIA(d+2)$.

Let $NTM(d)$ be the family of languages accepted in linear time by nondeterministic d-dimensional multihead Turing machines. A real-time simulation of Turing machines by iterative arrays gives $NTM(d) \subseteq NIA(d)$ [7], [12], but a less direct simulation using Theorem 1 gives the stronger result $\bigcup_d NTM(d) \subsetneqq NIA(2)$. Thus, in the nondeterministic case, *two*-dimensional iterative arrays outperform all *multi*dimensional multihead Turing machines.

In § 8 we examine related deterministic questions and summarize.

2. Notation.

Z is the set of integers.

Z^d is the set of d-tuples of integers.

N is the set of nonnegative integers.

* Received by the editors September 25, 1975, and in revised form January 24, 1977.

† Computer Science Department, Pennsylvania State University, University Park, Pennsylvania 16802. Most of this work was conducted at Massachusetts Institute of Technology Project MAC, supported by the National Science Foundation under Grant GJ-34671.

N^d is the set of d-tuples of nonnegative integers.
$|i|$ is the absolute value of the integer i.
$\pi_i(\mathbf{v})$ is the ith component of the tuple $\mathbf{v}$.
$\mathbf{0} \in Z^d$ is the d-tuple with $\pi_i(\mathbf{0}) = 0$ for $1 \leqq i \leqq d$.
$\mathbf{1} \in Z^d$ is the d-tuple with $\pi_i(\mathbf{1}) = 1$ for $1 \leqq i \leqq d$.
$\mathbf{e_j} \in Z^d$ is the d-tuple with $\pi_i(\mathbf{e_j}) = \begin{cases} 1 & \text{for } i = j, \\ 0 & \text{for } i \neq j. \end{cases}$
Σ^* is the set of finite strings of characters from Σ.
$|x|$ is the length of $x \in \Sigma^*$.
dom(f) is the domain of the function f.

3. String acceptance by Turing machines and iterative arrays. The basic *d-dimensional h-head Turing machine* consists of a finite-state control and a single d-dimensional worktape (infinite both ways in each dimension) on which the control reads, writes, and shifts with its h (initially coincident) worktape heads (see [12] for a formal definition). A Turing machine is *deterministic* or *nondeterministic*, respectively, if its finite-state control is. The *input-output terminal* of a Turing machine is its finite-state control.

The basic *d-dimensional iterative array* [2], [7] is a synchronized d-dimensional array (infinite both ways in each dimension) of identical finite-state machines, each of which communicates only with its $2d$ nearest neighbors in the array (see [12] for a formal definition). An iterative array is *deterministic* or *nondeterministic*, respectively, if each finite-state machine is. The *input-output terminal* of an iterative array is the finite-state machine at its origin. An iterative array is *quiescent* if every finite-state computing element is in the same designated *quiescent state*. For convenience, each identical finite-state machine of an iterative array may be thought of as a finite sequence of finite-state registers. If R is a name for one of these registers, then we will write $R(\mathbf{v})$ for the contents of register R of the computing element at location $\mathbf{v}$ in the array.

A Turing machine with initially blank tape or an initially quiescent iterative array M acts as an *acceptor* by receiving an input string sequentially at its input-output terminal. If for input string x the input-output terminal of M eventually enters some designated *accepting state*, then *M accepts x*. The *language accepted by M* is

$$L(M) = \{x \mid M \text{ accepts string } x\}.$$

The acceptor *M accepts within time T*: $N \to N$ if it accepts each string $x \in L(M)$ by step number $T(|x|)$ in some computation on that input.

If an acceptor M receives its input subject to the restriction that it must halt one step after detecting the end of the input string, then we say that M is an *on-line* acceptor. Without the restriction, M is an *off-line* acceptor (see [5]). Observe that the on-line restriction is no real restriction for nondeterministic acceptors; without spending extra time, they can anticipate the end of the input string by simply *guessing*. It is known, on the other hand, that the on-line restriction is a real one for both deterministic Turing machines and deterministic iterative arrays.[1]

DEFINITION. For $T: N \to N$,

$NTIME(T) = \{L \mid$ some nondeterministic one-dimensional multihead Turing machine accepts L within time $T\}$,

$NTM(d) = \{L \mid$ some nondeterministic d-dimensional multihead Turing machine accepts L in linear time (i.e., within time cn for some c)$\}$,

$NIA(d) = \{L \mid$ some nondeterministic d-dimensional iterative array accepts L in linear time$\}$.

Similarly define $DTIME(T)$, $DTM(d)$, $DIA(d)$, respectively, in terms of deterministic off-line acceptance and $QTIME(T)$, $QTM(d)$, $QIA(d)$, respectively, in terms of deterministic on-line acceptance.

It is well known that the classes $XIA(d)$ ($X \in \{N, D, Q\}$) are not affected if we permit the next state of each computing element of a d-dimensional iterative array to depend on the current states of more nearest neighbors than just $2d$ [2]. Nor are the classes affected if we permit the next state of each computing element to depend on the signs of its location's coordinates and the current states of the computing elements scanned by some finite number of independently shiftable (but initially coincident) array heads [7], [12]. The iterative array algorithms we describe in this paper, therefore, will be for iterative arrays which may employ these convenient features.

4. Arrays with multidimensional inputs. In this section we broaden our concept of languages accepted by d-dimensional iterative arrays to include sets of "d-boxes," which we define below to be a d-dimensional generalization of one-dimensional character strings.

DEFINITION. A *d-box* of *type* $(n_1, \cdots, n_d)$ over the finite alphabet Σ is a function

$$p:\{1, \cdots, n_1\} \times \cdots \times \{1, \cdots, n_d\} \to \Sigma.$$

The *diameter* $|p|$ of such a d-box is $n_1 + \cdots + n_d$. (A *string* of *length* n can be defined to be a 1-box of diameter n.) We shall often regard d-box p as a $(d+1)$-box by taking $n_{d+1} = 1$. (Thus a string of length n is also a 2-box of type $(n, 1)$, a 3-box of type $(n, 1, 1)$, etc.)

We are interested in d-boxes for their capacity to represent long strings (abbreviated Turing machine instantaneous descriptions (§ 5) in particular) in small diameters. With the next definition we fix such a representation.

DEFINITION. For each d-box p of type $(n_1, \cdots, n_d)$, using C_1 for ordinary 1-dimensional string concatenation, define $s(p)$ to be the following string of length $n_1 \cdots n_d$:

$$\underset{i_d=1}{\overset{n_d}{C_1}} \cdots \underset{i_1=1}{\overset{n_1}{C_1}} p(i_1, \cdots, i_d).$$

[1] In terms of the notation defined in this section, Hennie's set W^{d+1} [5] belongs to $DTM(1) \subseteq DTM(d) \subseteq DIA(d)$ for every d. Yet Hennie's argument shows that W^{d+1} belongs to neither $QTM(d)$ nor $QIA(d)$. (Aanderaa has shown that even the entire transduction actually performed by an on-line acceptor for W^{d+1} can be performed faster off-line than on-line in d dimensions [9].)

DEFINITION. If p, q are d-boxes of type $(n_1, \cdots, n_{d-1}, n_d)$, $(n_1, \cdots, n_{d-1}, n'_d)$, respectively, then the *d-concatenation* pC_dq of p and q is the d-box of type $(n_1, \cdots, n_{d-1}, n_d + n'_d)$ with

$$s(pC_dq) = s(p)C_1s(q) = s(p)s(q).$$

Remark. We can have both pC_dq and $pC_{d+1}q$ defined, but they are different; e.g.,

$$abC_1cd = abcd, \quad \text{but} \quad abC_2cd = \begin{matrix} cd \\ ab \end{matrix}.$$

A d-dimensional iterative array M is given the d-box p as a preloaded input in some designated input register, say "input," as follows:

$$\text{input}(\mathbf{v}) = \begin{cases} p(\mathbf{v}) & \text{for } \mathbf{v} \in \text{dom}(p), \\ \# & \text{for } \mathbf{v} \notin \text{dom}(p). \end{cases}$$

(We assume that # does not occur in the image of p.) Denote by $L'(M)$ the set of d-boxes accepted by the d-dimensional iterative array M. If M accepts each $p \in L'(M)$ within $T(|p|)$ steps, then *M accepts within time T*: $N \to N$. Let

$NIA'(d) = \{L'(M) | M$ is a d-dimensional iterative array which accepts in linear time$\}$,

$DIA'(d) = \{L'(M) | M$ is a deterministic d-dimensional iterative array which accepts in linear time$\}$.

LEMMA 1. *For each regular set L,*

$$\{p \,|\, p \textit{ is a } d\textit{-box (of any type) with } s(p) \in L\} \in DIA'(d).$$

Proof. Look at some deterministic one-way finite-state acceptor for L, and consider the alphabet consisting of transition matrices (i.e., functions from the state set into the state set) for that acceptor. Each string x has associated with it a unique transition matrix σ_x from this alphabet, and σ_{xy} is determined by σ_x and σ_y (it is their composition). The deterministic d-dimensional iterative array we construct will start operation in the configuration (input, accum, command), where

$$\text{input}(\mathbf{v}) = \begin{cases} p(\mathbf{v}) & \text{for } \mathbf{v} \in \text{dom}(p), \\ \# & \text{for } \mathbf{v} \notin \text{dom}(p), \end{cases}$$

$$\text{accum}(\mathbf{v}) = \text{command}(\mathbf{v}) = \# \quad \text{for all } \mathbf{v} \in Z^d.$$

By accumulating (i.e., composing) transition matrices along the successive dimensions (i.e., according to the definition of s), the array will successively compute the transition matrices of the strings of the forms

$$\underset{i_1}{C_1}\, p(i_1, \cdots, i_d),$$

$$\underset{i_2}{C_1}\, \underset{i_1}{C_1}\, p(i_1, \cdots, i_d),$$

$$\vdots$$

$$\underset{i_d}{C_1} \cdots \underset{i_1}{C_1}\, p(i_1, \cdots, i_d) = s(p).$$

The accumulation will take only linear time in each of the d dimensions.

The transition rules are as follows, with no change implied where no rule applies:

command($\mathbf{0}$) = #:
    command($\mathbf{0}$) ← "load accumulators."
command($\mathbf{0}$) = "load accumulators":
    accum($\mathbf{v}$) ← $\sigma_{\text{input}(\mathbf{v})}$ if input($\mathbf{v}$) ≠ #
    command($\mathbf{0}$) ← "accumulate in dimension 1"
command($\mathbf{0}$) = "accumulate in dimension i":
    If (accum($\mathbf{v}$), accum($\mathbf{v}+\mathbf{e_i}$), accum($\mathbf{v}+2\mathbf{e_i}$)) = (σ_x, σ_y, #), then set
        (accum($\mathbf{v}$), accum($\mathbf{v}+\mathbf{e_i}$), accum($\mathbf{v}+2\mathbf{e_i}$)) ← (σ_{xy}, #, #).
    If accum($\mathbf{1}+\mathbf{e_i}$) = # and $i<d$, then set
        command($\mathbf{0}$) ← "accumulate in dimension $i+1$."
    If accum($\mathbf{1}+\mathbf{e_i}$) = # and $i=d$, then set

$$\text{command}(\mathbf{0}) \leftarrow \begin{cases} \text{"accept"} & \text{if accum}(\mathbf{1}) \text{ maps the start state to an accepting state,} \\ \text{"reject"} & \text{otherwise.}\square \end{cases}$$

LEMMA 2. *Let Σ be a Cartesian product alphabet with fields "key," "sex," and "data." Let L be the set of d-boxes $p=p_1C_d\cdots C_dp_n$ such that each p_i is a $(d-1)$-box over Σ and the following holds for each i with* sex(p_i) *identically* 1:
The maximum $j<i$ with

$$\text{key}(p_j)=\text{key}(p_i), \qquad \text{sex}(p_j)\neq\text{sex}(p_i)$$

exists and satisfies

$$\text{data}(p_j)=\text{data}(p_i).$$

Then $L\in DIA'(d)$.

Proof. Informally, for each p_i whose sex is totally 1, the acceptor will have to search sequentially through $p_{i-1}, p_{i-2}, \cdots, p_1$ until it finds the nearest "mate" p_j for p_i. (A "mate" must differ at least slightly in sex but agree completely in key.) In addition, the acceptor will have to verify that p_i and p_j agree completely in data.

The deterministic d-dimensional iterative array we construct will start operation with all registers containing #, except input($\mathbf{v}$) = $p(\mathbf{v})$ for $\mathbf{v}\in\text{dom}(p)$. For each i, the array will shift copies of $p_i, p_{i-1}, \cdots, p_1$ past p_i, watching for p_j as required. So that the successive comparisons can be performed without delay, the copies will be "skewed" in such a way that, for $1\leqq k\leqq d-1$, each p_j goes by $p_i(\mathbf{v})$ one step after it goes by $p_i(\mathbf{v}+\mathbf{e_k})$ (see Figs. 1 and 2 below). This will allow the instantaneous accumulation at $p_i(\mathbf{v})$ of the results of comparing $p_i(\mathbf{v}')$ with $p_j(\mathbf{v}')$ for all $\mathbf{v}'\in\text{dom}(p_i)=\text{dom}(p_j)$ with $\pi_k(\mathbf{v}')\geqq\pi_k(\mathbf{v})$ for all $k\leqq d-1$.

The algorithm consists of the four steps below. Documentary remarks follow the entire algorithm.

1. Set

$$\text{status}(\mathbf{v}) \leftarrow \begin{cases} 0 & \text{if sex (input}(\mathbf{v}))=1, \\ 1 & \text{if sex (input}(\mathbf{v}))\notin\{\#, 1\}, \end{cases}$$

```
and then repeat the following $|p|$ times:
    $\text{status}(\mathbf{v}) \leftarrow \begin{cases} 0 & \text{if status}(\mathbf{v}) = 0 \text{ and status}(\mathbf{v}+\mathbf{e_i}) \neq 1 \text{ for all } i < d, \\ 1 & \text{if status}(\mathbf{v}) = 1 \text{ or status}(\mathbf{v}+\mathbf{e_i}) = 1 \text{ for some } i < d. \end{cases}$
2. Set copy($\mathbf{v}$) $\leftarrow$ input($\mathbf{v}$).
   Station a head at some $\mathbf{v_0} \in Z^d$ with
       input($\mathbf{v_0}$) $\neq \#$, input($\mathbf{v_0}+\mathbf{e_i}$) $= \#$ for all $i < d$,
   and then repeat the following until mobility($\mathbf{v_0}$) $= 1$:
   If either $\pi_i(\mathbf{v}) = 1$ for all $i < d$
       or mobility($\mathbf{v}-\mathbf{e_i}$) $= 1$ for some $i < d$,
   then set mobility($\mathbf{v}$) $\leftarrow 1$.
   If mobility($\mathbf{v}$) $= 1$, then set copy($\mathbf{v}$) $\leftarrow$ copy($\mathbf{v}+\mathbf{e_d}$).
3. Repeat the following $2|p|$ times:
   If input($\mathbf{v}$) $\neq \#$, copy($\mathbf{v}$) $\neq \#$, then set
       xkey($\mathbf{v}$) $\leftarrow 1$ if
           either key (input($\mathbf{v}$)) $\neq$ key (copy($\mathbf{v}$))
           or     xkey ($\mathbf{v}+\mathbf{e_i}$) $= 1$ for some $i < d$;
       xsex($\mathbf{v}$) $\leftarrow 1$ if
           either sex (input($\mathbf{v}$)) $\neq$ sex(copy($\mathbf{v}$))
           or     xsex ($\mathbf{v}+\mathbf{e_i}$) $= 1$ for some $i < d$;
       xdata($\mathbf{v}$) $\leftarrow 1$ if
           either data (input($\mathbf{v}$)) $\neq$ data (copy($\mathbf{v}$))
           or     xdata ($\mathbf{v}+\mathbf{e_i}$) $= 1$ for some $i < d$.
   Set copy($\mathbf{v}$) $\leftarrow$ copy($\mathbf{v}-\mathbf{e_d}$).
   If $\pi_i(\mathbf{v}) = 1$ for all $i < d$ and status($\mathbf{v}$) $= 0$, xkey($\mathbf{v}$) $\neq 1$, xsex($\mathbf{v}$) $= 1$, then set
       $\text{status}(\mathbf{v}) \leftarrow \begin{cases} 1 & \text{if xdata}(\mathbf{v}) \neq 1, \\ -1 & \text{if xdata}(\mathbf{v}) = 1. \end{cases}$
4. Accept if status($\mathbf{v}$) $= 1$ for every $\mathbf{v}$ with
       $\pi_i(\mathbf{v}) = 1$ for all $i < d$, input($\mathbf{v}$) $\neq \#$.
```

From the following documentary remarks, it is straightforward to prove that such an iterative array does recognize L in linear time.

1) No transition ever modifies input($\mathbf{v}$).

2) Step 1 modifies only the "status" registers. After that step, the following holds for each $\mathbf{v} \in \text{dom}(p) \subseteq N^d$ of the form $(1, \cdots, 1, i)$:

$$\text{status}(\mathbf{v}) = \begin{cases} 0 & \text{if sex}(p_i) \text{ is identically } 1, \\ 1 & \text{otherwise.} \end{cases}$$

In the rest of the algorithm, status($\mathbf{v}$) will be significant only for $\mathbf{v}$ of the form $(1, \cdots, 1, i)$, $1 \leqq i \leqq n$. Status 0, 1, or -1, respectively, will indicate that the search for a satisfactory mate p_j for p_i is continuing, successfully concluded, or unsuccessfully concluded. (A mate is required only if sex(p_i) is identically 1.)

3) Step 2 modifies only the "copy" and "mobility" registers. The step ends after $|p_1|-d+1$ repetitions with the "copy" registers shifted (or "skewed") so that

$$\text{copy}(v_1, \cdots, v_{d-1}, i-|p_1|+v_1+\cdots+v_{d-1}) = p_i(v_1, \cdots, v_{d-1})$$

for $(v_1, \cdots, v_{d-1}, i) \in \text{dom}(p)$. The "mobility" registers are not used after step 2.

4) In step 3, the "skewed" copies of the p_i's are rigidly shifted past the original p_i's (see Figs. 1 and 2). This allows each p_i to be compared with the copy of each p_j for $j \leqq i$. *Mis*matches in the "key," "sex," and "data" fields, respectively, are accumulated as 1 in the "xkey," "xsex," and "xdata" registers associated with the "input" registers. The sequence of these accumulated comparisons for each p_i is accumulated in status$(1, \cdots, 1, i)$, which finally indicates whether the search for an appropriate p_j succeeds. Step 4 merely polls the registers status$(1, \cdots, 1, 1), \cdots$, status$(1, \cdots, 1, n)$ to see whether the search succeeded for every p_i.

5) Steps 1, 3, respectively, call for $1+|p|, 2|p|$ transitions. Step 2 requires only $1+|p_1|+(|p_1|-d+1)$ transitions. Only the positions $(1, \cdots, 1, 1), \cdots, (1, \cdots, 1, n)$ must be checked in step 4, so $n \leqq |p|$ transitions suffice. □

5. Abbreviated instantaneous descriptions.

Definition. An *instantaneous description* (*id*) of a one-dimensional h-head Turing machine M with finite control state set $Q (0 \notin Q)$ and tape alphabet Σ is a string x over the $(h+1)$-track alphabet

$$(Q \cup \{0\}) \times \cdots \times (Q \cup \{0\}) \times \Sigma$$

such that

$$(\exists\, q \in Q)(\pi_i(x) \in 0^* q 0^* \quad \text{for } 1 \leqq i \leqq h),$$

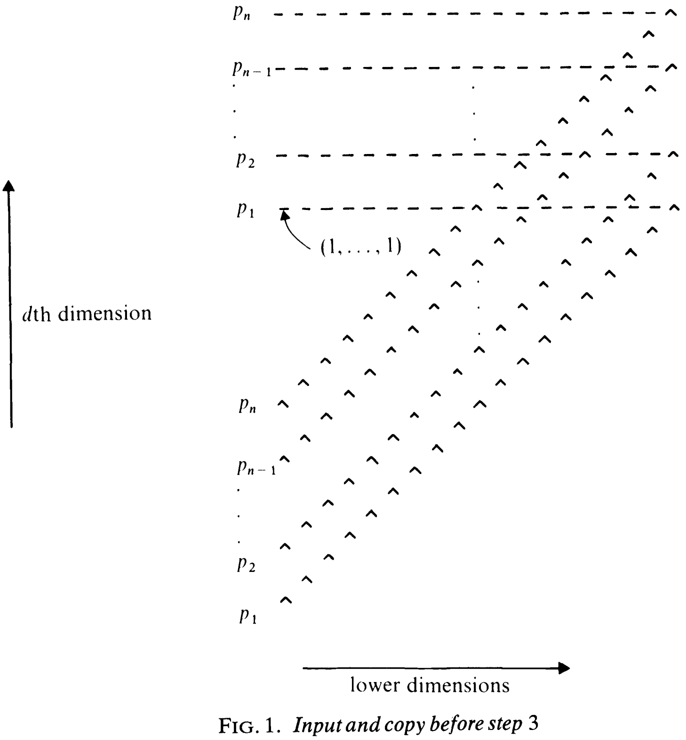


Fig. 1. *Input and copy before step* 3

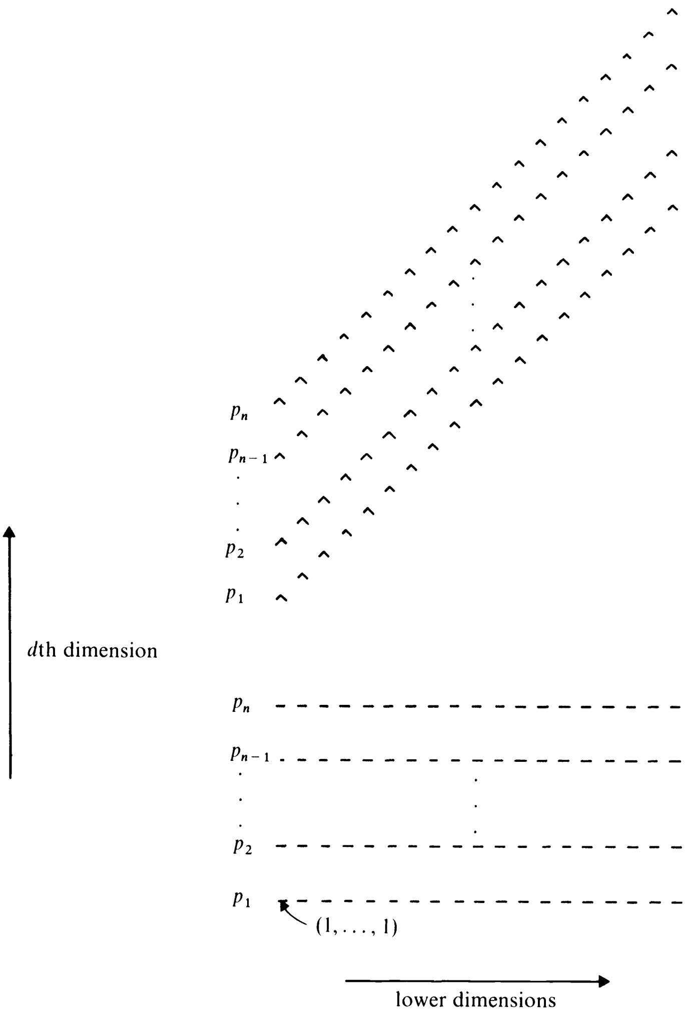


FIG. 2. *Input and copy after step* 3

where $0^*q0^* = \{0^iq0^j | i, j \geqq 0\}$ and we treat a string of length n over the $(h+1)$-track alphabet as an $(h+1)$-tuple of strings of length n over the respective component alphabets. Informally, $\pi_{h+1}(x)$ is the nonblank portion of the tape of M, q is the control state of M, and the position of q in $\pi_i(x)$ is the position of head i of M for $1 \leqq i \leqq h$. An *abbreviated instantaneous description* (*aid*) of M is a string x over the $(h+2)$-track alphabet

$$(Q \cup \{0\}) \times \cdots \times (Q \cup \{0\}) \times \Sigma \times \{0, [,]\}$$

such that

$$(\exists\, q \in Q)(\pi_i(x) \in 0^*q0^* \text{ for } 1 \leqq i \leqq h), \qquad \pi_{h+2}(x) \in ([0^*])^*.$$

A boundary between bracketed expressions on the extra track indicates the possible excision at that point of some unscanned tape squares from a full id. For x, y aids of M with $|x| = |y|$ and $\pi_{h+2}(x) = \pi_{h+2}(y)$, we say $x \mid\!\frac{k}{M}\, y$ if M can move in k steps from aid x to aid y without shifting a head to a (possibly) missing tape square. When M is fixed by context, we write $\mid\!\frac{k}{}$ for $\mid\!\frac{k}{M}$. If $\pi_i(x) = \pi_i(y)$ for $1 \leqq i < h+2$, then we say $x \approx y$. If $\pi_{h+2}(y)$ can be obtained from $\pi_{h+2}(x)$ by replacing some 0's by brackets, then we say $x \leqq y$. (We do not define $x \leqq y$ to imply $x \approx y$ or vice versa.)

Now we formulate a convenient set of conditions necessary and sufficient for $p \mid\!\frac{n^{d+1}}{M}\, q$, where M is a fixed one-dimensional h-head Turing machine. Since each head can shift at most n^{d+1} times in n^{d+1} steps, we may as well further abbreviate the aids to length at most $h \cdot (2n^{d+1})$, omitting some of the tape positions which the heads could not possibly reach on the way from aid p to aid q. Then we can easily pad p and q out to length exactly $2hn^{d+1}$, again without affecting derivability. The conditions we give are in terms of n computations of length only n^d, so we can parse the long aids into $2hn$ aids of length n^d and involve exactly $2h$ of them in each such short computation. (Then we will be able to use the conditions recursively in § 6.)

LEMMA 3. *Let*

$$p = p_1 \cdots p_{2hn}, \qquad q = q_1 \cdots q_{2hn}$$

be aids of length $2hn^{d+1}$ *with* $\pi_{h+2}(p) = \pi_{h+2}(q)$, $|p_j| = |q_j| = n^d$ *for* $1 \leqq j \leqq 2hn$. *Then* $p \mid\!\frac{n^{d+1}}{}\, q$ *if and only if there are* $2n$ *aids*

$$x_i = x_{i,1} \cdots x_{i,2h}, \qquad 1 \leqq i \leqq n,$$

$$y_i = y_{i,1} \cdots y_{i,2h}, \qquad 1 \leqq i \leqq n,$$

with $|x_{i,j}| = |y_{i,j}| = n^d$, $x_i \mid\!\frac{n^d}{}\, y_i$ *for* $1 \leqq i \leqq n$, $1 \leqq j \leqq 2h$ *and also an "address" function*

$$\text{addr}:\{1, \cdots, n\} \times \{1, \cdots, 2h\} \to \{1, \cdots, 2hn\}$$

such that the seven conditions below hold for all i, j. (Informally, $\text{addr}(i, j)$ is the "page number" within the longer aids ($2hn$ pages long) of the jth pages from the shorter aids x_i and y_i (only $2h$ pages long) taking part in the ith of the n subcomputations.)

1) $\text{addr}(i, j+1) > \text{addr}(i, j)$.

2) $\text{addr}(i, j+1) = 1 + \text{addr}(i, j)$
   if] *is not a suffix of* $\pi_{h+2}(x_{i,j})$.

3) $p_j \leqq x_{i_1,j_1}$ *whenever* $j = \text{addr}(i_1, j_1)$.

4) $p_j \approx q_j$ *whenever*
   $1 \leqq i' \leqq n \Rightarrow \text{addr}(i', j') \neq j$.

5) $p_j \approx x_{i_1,j_1}$ *whenever*
   $1 \leqq i' < i_1 \Rightarrow \text{addr}(i', j') \neq j$,
   $\text{addr}(i_1, j_1) = j$.

6) $y_{i_1,j_1} \approx x_{i_2,j_2}$ *whenever* $i_1 < i_2$ *and*
   $i_1 < i' < i_2 \Rightarrow \text{addr}(i', j') \neq \text{addr}(i_1, j_1)$,
   $\text{addr}(i_2, j_2) = \text{addr}(i_1, j_1)$.

7) $y_{i_1,j_1} \approx q_j$ *whenever*
$i_1 < i' \leqq n \Rightarrow \mathrm{addr}(i', j') \neq \mathrm{addr}(i_1, j_1)$,
$j = \mathrm{addr}(i_1, j_1)$.

Proof. ($\Rightarrow$) If $p \vdash^{nd+1} q$, then there are aids $r_0, r_1, \cdots, r_n$ such that

$$p = r_0 \vdash^{nd} r_1 \vdash^{nd} \cdots \vdash^{nd} r_n = q.$$

If we parse r_{i-1}, r_i into blocks of length n^d, then a computation $r_{i-1} \vdash^{nd} r_i$ can involve at most $2h$ pairs of blocks (two pairs for each head). We can select these pairs (in order) to be

$$x_{i,1}, y_{i,1}, \cdots, x_{i,2h}, y_{i,2h}, \quad \text{respectively,}$$

adding new brackets on track $h+2$ to indicate new excisions. For $\mathrm{addr}(i, j)$, we can take the position of the jth selected pair of blocks.

($\Leftarrow$) Assume we have the x_i's, y_i's, addr as described. For each $i (0 \leqq i \leqq n)$, we can construct an aid $r_i = r_{i,1} \cdots r_{i,2hn}$ with

$$r_{0,j} \approx p_j \quad \text{for } 1 \leqq j \leqq 2hn,$$

$$r_{i,j} \approx \begin{cases} r_{i-1,j} & \text{if } j \notin \{\mathrm{addr}(i, j_1) | 1 \leqq j_1 \leqq 2h\}, \\ y_{i,j_1} & \text{if } j = \mathrm{addr}(i, j_1) \end{cases}$$

$$\text{for } 0 < i \leqq n, \quad 1 \leqq j \leqq 2hn,$$

$$\pi_{h+2}(r_i) = \pi_{h+2}(p) \quad \text{for } 0 \leqq i \leqq n.$$

Then $p = r_0 \vdash^{nd} r_1 \vdash^{nd} \cdots \vdash^{nd} r_n = q$. □

6. Simulation of one-dimensional Turing machines.

LEMMA 4. *Let M be a fixed nondeterministic one-dimensional h-head Turing machine. For each d, let*

$$L_d = \bigcup_n \{(p, q) | (p \text{ and } q \text{ are } d\text{-boxes of type } (n, \cdots, n, 2hn) \text{ with } s(p) \vdash^{nd}_{M} s(q)\},$$

where (p, q) is the d-box defined by

$$(p, q)(\mathbf{v}) = (p(\mathbf{v}), q(\mathbf{v}))$$

for every $\mathbf{v} \in \mathrm{dom}((p, q)) = \mathrm{dom}(p) = \mathrm{dom}(q)$. *Then $L_d \in NIA'(d)$.*

Proof. Let Γ be the $(h+2)$-track alphabet

$$(Q \cup \{0\}) \times \cdots \times (Q \cup \{0\}) \times \Sigma \times \{0, [,]\},$$

where $Q (0 \notin Q)$ is the finite control state set of M and Σ is the tape alphabet of M.

The proof is by induction on d. By [12] $NTM'(1) \subseteq NIA'(1)$, and obviously $L_1 \in NTM'(1)$. In the induction step, we use the fact $L_d \in NIA'(d)$ to show $L_{d+1} \in NIA'(d+1)$. At a high level, the $(d+1)$-dimensional algorithm closely follows Lemma 3, so its correctness should be apparent.

1. Check that the input $(d+1)$-box (p, q) is of type $(n, \cdots, n, 2hn)$ for some n, with $s(p)$, $s(q)$ aids of M, $\pi_{h+2}(s(p)) = \pi_{h+2}(s(q))$. Let p_j, q_j $(1 \leqq j \leqq 2hn)$ be the d-boxes of type $(n, \cdots, n)$ with

$$p = p_1 C_{d+1} \cdots C_{d+1} p_{2hn},$$

$$q = q_1 C_{d+1} \cdots C_{d+1} q_{2hn}.$$

(See Fig. 3.)

2. Nondeterministically guess (x, y) to be the $(d+1)$-concatenation of n d-boxes (x_i, y_i) over Γ $(1 \leqq i \leqq n)$, each of type $(n, \cdots, n, 2hn)$. Let $x_{i,j}, y_{i,j}$ $(1 \leqq i \leqq n, 1 \leqq j \leqq 2h)$ be the d-boxes of type $(n, \cdots, n)$ with

$$x_i = x_{i,1} C_d \cdots C_d x_{i,2h},$$
$$y_i = y_{i,1} C_d \cdots C_d y_{i,2h}.$$

(See Fig. 4.)

3. Use the induction hypothesis to check that $(x_i, y_i) \in L_d$ for $1 \leqq i \leqq n$.

4. Nondeterministically guess an address for each of the d-boxes p_j and $x_{i,j}$. Assign the address addr(j) of p_j to q_j as well, and assign the address addr(i, j) of $x_{i,j}$ to $y_{i,j}$ as well.

5. Check that addr$(j) = j$.

6. Check that

$$\operatorname{addr}(i, j+1) > \operatorname{addr}(i, j),$$
$$\operatorname{addr}(i, j+1) = 1 + \mathit{addr}(i, j)$$
$$\text{whenever }] \text{ is not a suffix of } \pi_{h+2}(s(x_{i,j})).$$

7. Nondeterministically change some brackets to 0's in $\pi_{h+2}(x)$, and set $\pi_{h+2}(y) \leftarrow \pi_{h+2}(x)$ after these changes. (This is a slight departure from 3) of Lemma 3. The intention is to remove precisely the brackets (if any) that occur in $x_{i,j}$ but not in $p_{\text{addr}(i,j)}$. By strengthening the $\approx$'s to equalities in 4)–7) of Lemma 3, we will check to make sure that the following holds for all i, j after this modification:

$$\pi_{h+2}(x_{i,j}) = \pi_{h+2}(y_{i,j}) = \pi_{h+2}(p_{\text{addr}(i,j)}).)$$

8. Check that $s(p_j) = s(q_j)$ whenever

$$1 \leqq i' \leqq n \Rightarrow \operatorname{addr}(i', j') \neq \operatorname{addr}(j).$$

9. Check that $s(p_j) = s(x_{i_1, j_1})$ whenever

$$1 \leqq i' < i_1 \Rightarrow \operatorname{addr}(i', j') \neq \operatorname{addr}(j),$$
$$\operatorname{addr}(i_1, j_1) = \operatorname{addr}(j).$$

10. Check that $s(y_{i_1, j_1}) = s(x_{i_2, j_2})$ whenever $i_1 < i_2$ and

$$i_1 < i' < i_2 \Rightarrow \operatorname{addr}(i', j') \neq \operatorname{addr}(i_1, j_1),$$
$$\operatorname{addr}(i_2, j_2) = \operatorname{addr}(i_1, j_1).$$

11. Check that $s(y_{i_1, j_1}) = s(q_j)$ whenever

$$i_1 < i' \leqq n \Rightarrow \operatorname{addr}(i', j') \neq \operatorname{addr}(i_1, j_1),$$
$$\operatorname{addr}(j) = \operatorname{addr}(i_1, j_1).$$

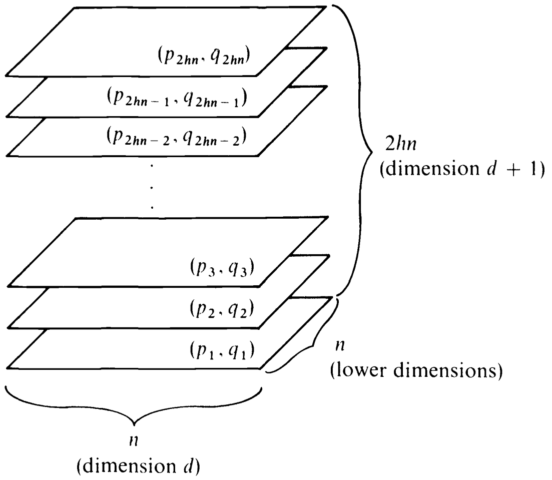


FIG. 3. *The given pair* (p, q)

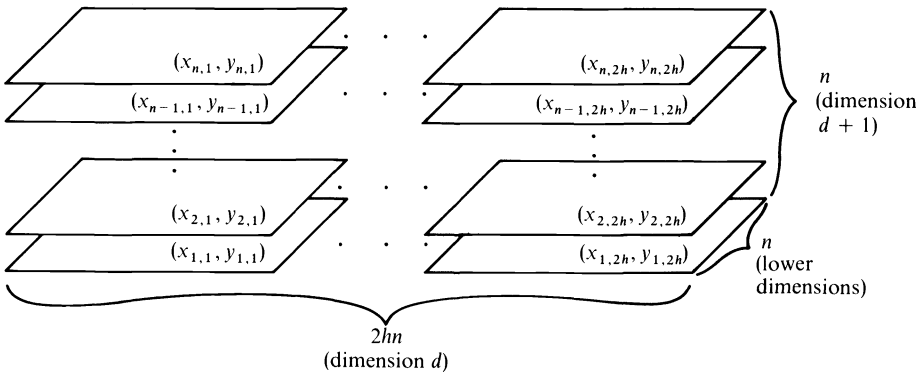


FIG. 4. *The guessed pairs* (x_i, y_i)

Finally, let us elaborate on how the steps above may be carried out in linear time.

1. Even a $(d+1)$-dimensional multihead Turing machine can check in linear time that its input $(d+1)$-box is of type $(n, \cdots, n, 2hn)$ for some n, so a $(d+1)$-dimensional iterative array can, too [12]. By Lemma 1, the check that $(s(p), s(q))$ belongs to the regular language

$$\{(u, v)|u, v \text{ are aids of } M \text{ with } \pi_{h+2}(u) = \pi_{h+2}(v)\}$$

also requires only linear time.

2. First, the array sets up a constant $(d+1)$-box of type $(n, \cdots, n, 2hn, n)$ in linear time by starting with a trivial constant $(d+1)$-box of type $(1, \cdots, 1)$, converting that to one of type $(n, 1, \cdots, 1)$ by shifting $n-1$ times in dimension 1,

converting that to one of type $(n, n, 1, \cdots, 1)$ by shifting $n-1$ times in dimension 2, and so on. In a single final transition, the array independently guesses a character from Γ at each position in the constant $(d+1)$-box.

3. In this step the $(d+1)$-dimensional array behaves essentially like a one-dimensional array of d-dimensional arrays, each of which recognizes L_d in linear time, recording the outcome at its "origin"—actually a position of the form $(0, \cdots, 0, i) \in Z^{d+1}$. After enough time has passed, the array need only check whether the outcomes recorded at the positions $(0, \cdots, 0, i)$ for $1 \leqq i \leqq n$ are all "accept."

4. The guessed objects are actually d-boxes of type $(n, \cdots, n)$ over $\{0, 1\}$, superimposed on each p_j and $x_{i,j}$. The addresses are the numbers whose binary representations are the images of those respective d-boxes under the map s. The address of each p_j or $x_{i,j}$ is assigned also to the corresponding q_j or $y_{i,j}$ by simply creating copies of the guessed d-boxes.

5–6. To delimit the strings $s(p_j)$ (and their addresses) within $s(p)$, specially mark the last character of each $s(p_j)$ (and of its address); i.e., specially mark the positions $(n, \cdots, n, i) \in Z^{d+1}$ for $1 \leqq i \leqq 2hn$. Similarly delimit the strings $s(q_j)$, $s(x_{i,j})$, $s(y_{i,j})$ (and their addresses). Use some other special mark to delimit the strings $s(x_i)$, $s(y_i)$ as well.

For step 5, shift a copy of each p_j's address d-box up in dimension $d+1$ to p_{j+1}'s address d-box. Appeal to Lemma 1 for linear-time verification that $\text{addr}(1) = 1$ and that $\text{addr}(j+1) = 1 + \text{addr}(j)$ for $1 \leqq j < 2hn$. For step 6, shift a copy of each $x_{i,j}$ and its address d-box up n times in dimension d to $x_{i,j+1}$ and its address d-box, and again appeal to Lemma 1.

8–11. By routine shifting, the array can construct in linear time a $(d+1)$-box $r = r_1 C_{d+1} \cdots C_{d+1} r_{8hn}$ of type $(n, \cdots, n, 8hn)$ with

$$(\text{data}(r_k), s(\text{key}(r_k)), \text{sex}(r_k)) \leftarrow \begin{cases} (p_k, \text{addr}(k), m) & \text{if } 1 \leqq k \leqq 2hn, \\ (x_{i,j}, \text{addr}(i,j), f) & \text{if } 2hn+1 \leqq k \leqq 6hn \textit{ and } k \\ & \quad = 2hn + (i-1) \cdot 4h + (2j-1), \\ (y_{i,j}, \text{addr}(i,j), m) & \text{if } 2hn+1 \leqq k \leqq 6hn \text{ and } k \\ & \quad = 2hn + (i-1) \cdot 4h + 2j, \\ (q_{k-6hn}, \text{addr}(k-6hn), f) & \text{if } 6hn+1 \leqq k \leqq 8hn, \end{cases}$$

where $m, f: \{1, \cdots, n\}^d \to \{0, 1\}$ are constantly 0, 1, respectively. It is easy to see that steps 8–11 should merely check whether r belongs to the language of Lemma 2. □

THEOREM 1. *NTIME*$(n^d) \subseteq$ *NIA*(d).

Proof. Let M be a fixed nondeterministic one-dimensional h-head Turing machine which accepts within time n^d. Without loss of generality, make the following assumptions about M:

1. The worktape of M is infinite to the right only.

2. The input to M is found initially written at the beginning of an otherwise blank worktape. (Assume B is the blank symbol of M.)

3. Initially M is in control state q_0 with all h heads scanning the leftmost worktape square.

4. M accepts by entering control state q_f with all h heads scanning the leftmost square of a blank worktape.

5. Rather than "halt" when it enters control state q_f, M technically continues to go through transitions which do not change anything.

Let L_d be as in Lemma 4. A nondeterministic d-dimensional iterative array can accept $L(M)$ in linear time as follows:

1. If the input string is $x, |x|=n$, then set up a d-box (p, q) of type $(n, \cdots, n, 2hn)$ with $s(p)$, $s(q)$ aids of M such that

$$\pi_i(s(p))=q_0 0^{2hn^{d}-1} \quad \text{for } 1 \leqq i \leqq h,$$

$$\pi_{h+1}(s(p))=xB^{2hn^{d}-n},$$

$$\pi_{h+2}(s(p))=[0^{2hn^{d}-2}],$$

$$\pi_i(s(q))=q_f 0^{2hn^{d}-1} \quad \text{for } 1 \leqq i \leqq h,$$

$$\pi_{h+1}(s(q))=B^{2hn^{d}},$$

$$\pi_{h+2}(s(q))=[0^{2hn^{d}-2}].$$

2. Check that $(p, q) \in L_d$ (Lemma 4). □

COROLLARY 1.1. *For* any *time bound T, all of NTIME(T) is contained in the class of languages accepted within time proportional to $n+T(n)^{1/d}$ by nondeterministic d-dimensional iterative arrays.*

Proof. If the one-dimensional Turing machine M accepts L within time T and p is a new "padding" character, then $L' = \{xp^k | x \in L, |xp^k|^d \geqq \text{minimum time for } M \text{ to accept } x\} \in NTIME(n^d) \subseteq NIA(d)$. If the d-dimensional iterative array M' accepts L' within time cn, then a d-dimensional iterative array can accept L within time proportional to $n+T(n)^{1/d}$ by operating as follows on input x: For $i = 0, 1, 2, \cdots$, simulate M' on input xp^{2^i} for $c \cdot |xp^{2^i}|$ steps, accepting if M' ever does. □

7. Simulation by one-dimensional Turing machines.

THEOREM 2. *$XIA(d) \subseteq XTIME(n^{d+1})$ for each prefix $X \in \{N, D, Q\}$.*

Proof. We give the simulation for $X \in \{D, Q\}$; the simulation for the prefix $X = N$ is similar. Without loss of generality, let the acceptor M be a deterministic d-dimensional iterative array (without extra features such as direct central control [12]) in which only the computing elements at positions belonging to N^d (i.e., with nonnegative coordinates) ever leave the quiescent state (see [2]). We give a step-by-step simulation of M by a deterministic one-dimensional $(2d+1)$-head Turing machine which spends about t^d steps to simulate the tth step in each computation by M.

For any fixed input to M and each nonnegative integer t, let $p(t)$ be the d-box of type $(t+1, \cdots, t+1)$ with $p(t)(\mathbf{v})$ equal for each $\mathbf{v} \in \text{dom}(p(t))$ to the state after t steps of computation by M of the finite-state computing element at position $\mathbf{v}-\mathbf{1}$. (All other finite-state computing elements must still be quiescent at time t.) The simulating Turing machine will write $s(p(0))$ on its worktape and then use a linear-time algorithm to construct $s(p(t+1))$ from $s(p(t))$ repeatedly. Thus $s(p(t))$ will be constructed within time proportional to $t \cdot |s(p(t))| = t \cdot (t+1)^d$, as desired.

For each prefix of $s(p(t))$, the simulator will also record the largest $d' \leqq d$ for which the length of the prefix is divisible by $(t+1)^{d'}$. This information can be recorded interspersed between the characters of $s(p(t))$. Formally, then, the maintained string will actually be a member of $\Sigma(K\Sigma)^*$, where K is the state set of each finite-state computing element of M and $\Sigma = \{0, 1, \cdots, d\}$. For example, the first character of this string is always d since the length 0 of the null prefix is certainly divisible by $(t+1)^d$.

We divide the derivation of $s(p(t+1))$ from $s(p(t))$ into two subtasks, the first of which is to actually simulate a transition by M. For each $\mathbf{v} \in Z^d$ with $0 \leqq \pi_i(\mathbf{v}) \leqq t$ for all i, let loc($\mathbf{v}$, t) be the position in the string $s(p(t))$ of the state at time t of the computing element at position $\mathbf{v}$ of M. To simulate a transition by M, we initially position head H_0 at position loc($\mathbf{0}$, t) = 1 and head H_{+i} (for $1 \leqq i \leqq d$) at position loc($\mathbf{0}$, t) + $(t+1)^{i-1}$ in $s(p(t))$. (The latter is the position just beyond the first $s(p(t))$ boundary to the right that is labeled by $i-1 \in \Sigma$.) We position head H_{-i} (for $1 \leqq i \leqq d$) equally distant from H_0 but in the other direction. (Initially, then, H_{-i} is positioned somewhere to the left of $s(p(t))$.) Moving all $2d+1$ heads synchronously across $s(p(t))$ until H_0 reaches loc($(t, \cdots, t)$, t) = $|s(p(t))|$ and keeping track of when a boundary labeled $\geqq i$ intervenes between H_0 and H_{+i} or between H_{-i} and H_0 (for $1 \leqq i \leqq d$), the simulator can produce an updated version $s(p'(t))$ of $s(p(t))$ in linear time. To see this, just make the following observations:

1) If no boundary labeled $\geqq i$ intervenes between positions loc($\mathbf{v}$, t) and loc($\mathbf{v}$, t) + $(t+1)^{i-1}$ in $s(p(t))$, then loc($\mathbf{v}+\mathbf{e}_i$, t) = loc($\mathbf{v}$, t) + $(t+1)^{i-1}$; otherwise, the computing element at position $\mathbf{v}+\mathbf{e}_i$ in the array is still in the quiescent state at time t.

2) If no boundary labeled $\geqq i$ intervenes between positions loc($\mathbf{v}$, t) − $(t+1)^{i-1}$ and loc($\mathbf{v}$, t) in $s(p(t))$, then loc($\mathbf{v}-\mathbf{e}_i$, t) = loc($\mathbf{v}$, t) − $(t+1)^{i-1}$; otherwise, the computing element at position $\mathbf{v}-\mathbf{e}_i$ in the array is in the quiescent state because the position has a negative ith coordinate.

The second subtask, of course, is to derive $s(p(t+1))$, along with the appropriate boundary labels, from $s(p'(t))$ and its boundary labels. Observe that $p(t+1)$ is the type $(t+2, \cdots, t+2)$ d-box that agrees with the type $(t+1, \cdots, t+1)$ d-box $p'(t)$ on its domain and is the quiescent state q_0 elsewhere. One linear-time algorithm for the subtask involves copying $s(p'(t))$ and its boundary labels onto a new track, making appropriate amendments in the process. The old string, a member of $(\Sigma - \{0\})(K\Sigma)^*$, parses unambiguously into pieces from $(\Sigma - \{0\})(K\{0\})^*$. In the algorithm, each piece is copied whole, the only amendments being made between pieces. The amendments are determined in each case by the initial label in the following piece and the current cumulative contents of the new track. If the initial label of the next piece is d', then d' consecutive amendments are made. The d''-th amendment matches everything accumulated on the new track so far, starting with the last label $d''-1$ and including even the most recent amendments, except that q_0 is substituted for every character from K. One head on the old track and two heads on the new track suffice. It is easy to see that the algorithm requires time that is only linear in the length of the string on the new track, and it is easy to prove by induction that the amendments are correct. □

COROLLARY 2.1. *For* any *time bound T, $NTIME(T^{d+1})$, $DTIME(T^{d+1})$, $QTIME(T^{d+1})$ include all languages accepted within time T by d-dimensional iterative arrays which are nondeterministic, deterministic off-line, deterministic on-line, respectively.*

COROLLARY 2.2. $\bigcup_d NIA(d) = \bigcup_d NTIME(n^d) = \mathrm{NP}$.

COROLLARY 2.3. $NIA(d) \subsetneqq NIA(d+2)$.

Proof. Cook's nondeterministic time hierarchy theorem [3] gives $NTIME(n^{d+1}) \subsetneqq NTIME(n^{d+2})$, so

$$\begin{aligned} NIA(d) &\subseteq NTIME(n^{d+1}) \quad \text{(by Theorem 2)} \\ &\subsetneqq NTIME(n^{d+2}) \\ &\subseteq NIA(d+2) \quad \text{(by Theorem 1).} \quad \square \end{aligned}$$

Remarks. (i) It follows that Theorem 1 is optimal to within one dimension; for, if we had $NTIME(n^{d+2}) \subseteq NIA(d)$, the proof of Corollary 2.3 would yield $NIA(d) \subsetneqq NIA(d)$. Our conjecture is that Theorem 1 actually is optimal.

(ii) Similarly, Theorem 2 is optimal to within one dimension.

(iii) The refined nondeterministic time hierarchy theorem of [13] gives tighter results such as $NTIME(n^{d+1}/\log n) \subsetneqq NTIME(n^{d+1})$, so strengthening Theorem 2 just a bit to $XIA(d) \subseteq XTIME(n^{d+1}/\log n)$ would tighten Corollary 2.3 to $NIA(d) \subsetneqq NIA(d+1)$. Kosaraju has shown that every context-free language belongs to $DIA(2)$ [8], so it would also give a less-than-cubic time upper bound on context-free language recognition by deterministic one-dimensional Turing machines. Our conjecture, however, is that Theorem 2 (but probably not Corollary 2.3) already is optimal.

COROLLARY 2.4. $\bigcup_d NTM(d) \subsetneqq NIA(2)$.

Proof. For each d and each $X \in \{N, D, Q\}$, $NTM(d) \subseteq NTIME(n^{2-1/d})$ [11]. Using this fact for $X = N$, we have

$$\begin{aligned} \bigcup_d NTM(d) &\subseteq \bigcup_d NTIME(n^{2-1/d}) \\ &\subseteq NTIME(n^2/\log n) \\ &\subsetneqq NTIME(n^2) \quad \text{(by [13])} \\ &\subseteq NIA(2) \quad \text{(by Theorem 1).} \quad \square \end{aligned}$$

8. Related deterministic questions and summary. A motivation for our work has been to learn the effects, if any, of parallelism (the unbounded activity of iterative arrays) and dimension on computing time. Allowing nondeterminism as well has allowed us to draw some interesting conclusions, especially Corollary 2.4, which says that unbounded activity in two dimensions is more powerful than bounded activity in any number of dimensions. Because it is not known for sure whether nondeterminism ever saves time (except for on-line models (see below)), however, our results are of interest in connection with their more important deterministic analogues, even if the proof of Theorem 1 does not generalize. In this section we bring together the known deterministic analogues of the results of the preceding sections. (See Table 1 for a summary.)

TABLE 1
Summary

Assertion	Prefix	Best known results	Conjectures
$XTIME(n^d) \subseteq XIA(d')$	Q		$\neg\exists\, d'$
	D		$\neg\exists\, d'$
	N	$d'=d$	$\neg\exists\, d'<d$
$XIA(d) \subseteq XTIME(T(n))$	Q	$T(n)=n^{d+1}$	Optimal
	D	$T(n)=n^{d+1}$	Optimal
	N	$T(n)=n^{d+1}$	Optimal
$XTM(d) \subseteq XIA(d')$	Q	$d'=d$	$\neg\exists\, d'<d$
	D	$d'=d$	$\neg\exists\, d'<d$
	N	$d'=2$	$\neg\exists\, d'<2$
$XIA(d')-XTM(d) \neq \varnothing$	Q	$d'=1$	----------------
	D		$d'=1$
	N	$d'=2$	$d'=1$
$XIA(d) \subseteq XTM(d')$	Q	$\neg\exists\, d'$	----------------
	D		$\neg\exists\, d'$
	N	$\neg\exists\, d'$ unless $d=1$	$\neg\exists\, d'$
$XTM(d')-XIA(d) \neq \varnothing$	Q	$d'=d+1$	----------------
	D		$d'=d+1$
	N	$\neg\exists\, d'$ unless $d=1$	$NTM(2)-NIA(1) \neq \varnothing$
$XTM(d) \subsetneqq XTM(d')$	Q	$d'=d+1$	----------------
	D		$d'=d+1$
	N		$d'=d+1$
$XIA(d) \subsetneqq XIA(d')$	Q	$d'=d+1$	----------------
	D		$d'=d+1$
	N	$d'=d+2$	$d'=d+1$

Although we have shown $NTIME(n^d) \subseteq NIA(d)$, our proof sheds no real light on the related deterministic questions. We do not know whether even $DTIME(n^d) \subseteq \bigcup_d DIA(d)$ or $QTIME(n^d) \subseteq \bigcup_d QIA(d)$ holds. Conversely, our proof that $XIA(d) \subseteq XTIME(n^{d+1})$ goes through for any $X \in \{N, D, Q\}$.

Similarly, we have shown that $NIA(d) \subsetneqq NIA(d+2)$, but we have no idea whether even $DIA(d) \subsetneqq \bigcup_d DIA(d)$ is true. In the on-line case, however, we have even stronger results, witnessed by examples of Hennie [5]. Cook and Aanderaa [4] have observed that those examples and Hennie's information-theoretic argument apply to iterative arrays as well as to Turing machines, giving $QTM(d+1) - QIA(d) \neq \varnothing$.[2] By [12], $QTM(d) \subseteq QIA(d)$, so we get both $QTM(d) \subsetneqq$

[2] The witness language $W^{d+1} \in QTM(d+1) - QIA(d)$, in fact, is accepted *in real time* by an on-line deterministic $(d+1)$-dimensional *single-head* Turing machine.

$QTM(d+1)$ (cf., Hennie [5]) and $QIA(d) \subsetneqq QIA(d+1)$ (cf., Cole [2]) as corollaries. (Both $DTM(d) \subsetneqq DTM(d+1)$ and $NTM(d) \subsetneqq NTM(d+1)$ are unsettled conjectures.)

Our result $\bigcup_d NTM(d) \subsetneqq NIA(2)$ easily yields $NTM(d) \subsetneqq NIA(d)$ for every $d \geqq 2$. For off-line deterministic machines, as usual, all such questions are open. For on-line deterministic machines, however, a strong result is again known: $QIA(1) - \bigcup_d QTM(d) \neq \varnothing$. In particular, Atrubin [1] has designed a deterministic on-line one-dimensional iterative array that multiplies in linear time (in fact in real time), while Cook and Aanderaa [4] have shown that no on-line deterministic multidimensional multihead Turing machine can do so. (On-line multiplication of binary numbers, of course, is easily converted to an on-line language acceptance problem). Thus $QTM(d) \subsetneqq QIA(d)$ for every d. Of course we cannot have $\bigcup_d QTM(d) \subseteq QIA(d)$ for any d, for that would contradict $QTM(d+1) - QIA(d) \neq \varnothing$.

REFERENCES

[1] A. J. Atrubin, *A one-dimensional real-time iterative multiplier*, IEEE Trans. Electronic Computers, EC-14 (1965), pp. 394–399.

[2] S. N. Cole, *Real-time computation by n-dimensional iterative arrays of finite-state machines*, Ibid., EC-18 (1969), pp. 349–365.

[3] S. A. Cook, *A hierarchy for nondeterministic time complexity*, J. Comput. System Sci., 7 (1973), pp. 343–353.

[4] S. A. Cook and S. O. Aanderaa, *On the minimum computation time of functions*, Trans. Amer. Math. Soc., 142 (1969), pp. 291–314.

[5] F. C. Hennie, *On-line Turing machine computations*, IEEE Trans. Electronic Computers, EC-15 (1966), pp. 35–44.

[6] R. M. Karp, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–103.

[7] S. R. Kosaraju, *Computations on iterative automata*, Doctoral Thesis, University of Pennsylvania, Philadelphia, August 1969.

[8] ———, *Speed of recognition of context-free languages by array automata*, this Journal, 4 (1975), pp. 331–340.

[9] A. R. Meyer, Personal communication, November 1973.

[10] A. R. Meyer and M. S. Paterson, Personal communication, September 1973.

[11] N. J. Pippenger, in preparation.

[12] J. I. Seiferas, *Iterative arrays with direct central control*, Acta Informatica, to appear.

[13] J. I. Seiferas, M. J. Fischer and A. R. Meyer, *Separating nondeterministic time complexity classes*, J. Assoc. Comput. Mach., to appear.

SIAM J. COMPUT.
Vol. 6, No. 3, September 1977

A NEW ALGORITHM FOR GENERATING ALL THE MAXIMAL INDEPENDENT SETS[*]

SHUJI TSUKIYAMA,[†] MIKIO IDE,[‡] HIROMU ARIYOSHI,[¶] AND ISAO SHIRAKAWA[†]

Abstract. The problem of generating all the maximal independent sets (or maximal cliques) of a given graph is fundamental in graph theory and is also one of the most important in terms of the application of graph theory. In this paper, we present a new efficient algorithm for generating all the maximal independent sets, for which processing time and memory space are bounded by $O(nm\mu)$ and $O(n+m)$, respectively, where n, m, and μ are the numbers of vertices, edges, and maximal independent sets of a graph.

Key words. algorithm, backtracking, graph, maximal clique, maximal independent set

1. Introduction. The efficient search for all the maximal independent sets (or maximal cliques) is fundamental in the theory of graphs and its applications [4], [6], and is also interesting in terms of complexity of computation.

The problem of listing all the maximal independent sets of a given graph is equivalent to that of listing all the maximal cliques of a graph, since each maximal independent set of a graph G corresponds one-to-one to each maximal clique of the complementary graph of G. Thus, the former can be reduced to the latter, and vice versa.

A number of authors have proposed a variety of approaches to this problem [1], [2], [3], [5], [7], [9], [10], among which those proposed by Bierstone noted in [2], Akkoyunlu [1], and Bron–Kerbosch [5] are distinctive in the sense that they can be applied to a graph of comparatively large size. Specifically, the algorithm in [2] finds all the maximal cliques of a graph in processing time proportional to ν^2 in the worst case, where ν is the number of the maximal cliques, and the algorithm in [5] empirically can be applied more efficiently than that of [2], but without any theoretical estimation of complexity.

The present paper considers a new efficient algorithm for generating all the maximal independent sets of a given graph G in processing time bounded by $O(nm\mu)$, where n, m, and μ are the numbers of vertices, edges, and all the maximal independent sets of G, respectively. This method is based on a modified application of the vertex sequence method of [2], combined with a backtracking method different from that employed in [1] or [5].

A *graph* is denoted by $G=[V, E]$, where V is a set of *vertices*, and E is a set of *edges*, each represented by an unordered pair $\langle v, w\rangle$ of its end vertices v and w.

[*] Received by the editors November 18, 1975. This paper was supported in part by the Grant in Aid for Scientific Research of the Ministry of Education, Science and Culture of Japan under Grant Cooperative Research (A) 035012 (1975) and 135017 (1976).

[†] Department of Electronic Engineering, Faculty of Engineering, Osaka University, Suita, Osaka, 565 Japan.

[‡] Takasago Laboratory, Mitsubishi Heavy Industries Ltd., Takasago, Hyogo, 676 Japan.

[¶] Department of Electrical Engineering, Faculty of Engineering, Ehime University, Matsuyama, Ehime, 790 Japan.

Unless otherwise stated, a graph of our interest henceforth does not contain any self-loop or any multiple edge. For any edge $\langle v, w\rangle$ of graph G, $\langle v, w\rangle$ is said to be *incident* to v and w, and v and w are said to be *adjacent* each other. Given a graph $G=[V,E]$, let $\Gamma_G(v)$ for any $v\in V$ be a set of vertices adjacent to v in G, that is,

$$\Gamma_G(v)\triangleq\{w|\langle v, w\rangle\in E\}. \tag{1}$$

Given a set $S\subset V$ of vertices of G, if any pair of distinct vertices in S are not adjacent each other, then S is called an *independent set* of G, henceforth abbreviated simply to IS, and any such S which is not contained in any other IS is called a *maximal independent set* of G, abbreviated to MIS.

2. Main theorem. Given a set $W\subset V$ of vertices of G, let

$$E(W)\triangleq\{\langle v, w\rangle\in E|v, w\in W\}, \tag{2}$$

and designate a subgraph $G(W)$ of G as

$$G(W)\triangleq[V, E(W)]. \tag{3}$$

For any W_{i-1}, $W_i\subset V$ such that $W_i=W_{i-1}\cup\{x\}$ with $x\in V-W_{i-1}$,[1] let $\mathcal{M}_{i-1}$ and $\mathcal{M}_i$ be sets of the MIS's of $G(W_{i-1})$ and $G(W_i)$, respectively, and consider a relation between $\mathcal{M}_{i-1}$ and $\mathcal{M}_i$ in the following.

For simplicity, let $\Gamma_i(v)\triangleq\Gamma_{G(W_i)}(v)$ and $A\triangleq\Gamma_i(x)$ for $x\in W_i-W_{i-1}$; then for

$$\mathcal{M}_{i-1}(x,\bar{A})\triangleq\{M'\in\mathcal{M}_{i-1}|M'\cap A=\emptyset\}, \tag{4}$$

$$\mathcal{M}_{i-1}(x,A)\triangleq\{M'\in\mathcal{M}_{i-1}|M'\cap A\neq\emptyset\}, \tag{5}$$

$\mathcal{M}_{i-1}$ can be partitioned in the form

$$\mathcal{M}_{i-1}=\mathcal{M}_{i-1}(x,\bar{A})+\mathcal{M}_{i-1}(x,A), \tag{6}$$

where "+" denotes the union of two disjoint sets. It should be noted that x is contained in any $M'\in\mathcal{M}_{i-1}$, since x is an isolated vertex in $G(W_{i-1})$ and also $G(W_{i-1})$ has all the vertices of the original graph G.

LEMMA 1. *Let*

$$\mathcal{M}_i(\bar{x},A)\triangleq\{M=M'-\{x\}|M'\in\mathcal{M}_{i-1}(x,A)\}; \tag{7}$$

then we have

$$\mathcal{M}_i\supset\mathcal{M}_{i-1}(x,\bar{A})+\mathcal{M}_i(\bar{x},A). \tag{8}$$

Proof. Since $\mathcal{M}_i\supset\mathcal{M}_{i-1}(x,\bar{A})$ and $\mathcal{M}_{i-1}(x,\bar{A})\cap\mathcal{M}_i(\bar{x},A)=\emptyset$ can be readily verified, we shall show $\mathcal{M}_i(\bar{x},A)\subset\mathcal{M}_i$. Suppose that any $M\in\mathcal{M}_i(\bar{x},A)$ is not an element of $\mathcal{M}_i$; then there exists a nonempty set X such that $M\cap X=\emptyset$ and $M+X\in\mathcal{M}_i$, since M is an IS of $G(W_i)$. This implies that $M+X$ is an IS of $G(W_{i-1})$. Moreover, we have $x\notin X$, since $M\cap A\neq\emptyset$ and $M+X\in\mathcal{M}_i$. Therefore, $M+X+\{x\}$ is an IS of $G(W_{i-1})$. This contradicts $M\in\mathcal{M}_i(\bar{x},A)$ or $M+\{x\}\in\mathcal{M}_{i-1}$. Hence the lemma.

Next, we shall show that any element of $\mathcal{M}_i-\mathcal{M}_{i-1}(x,\bar{A})-\mathcal{M}_i(\bar{x},A)$ is generated from an element of $\mathcal{M}_{i-1}(x,A)$.

[1] For sets X and Y, let $X-Y\triangleq\{a|a\in X$ and $a\notin Y\}$.

LEMMA 2. *Each $M \in \mathcal{M}_i - \mathcal{M}_{i-1}(x, \bar{A}) - \mathcal{M}_i(\bar{x}, A)$ contains x, and for any such M there exists $M' \in \mathcal{M}_{i-1}(x, A)$ such that*

(i) *$M \subset M'$ and $M' - M \subset A$ $(=\Gamma_i(x))$, and*

(ii) *for any $y \in M' \cap A$ either $\Gamma_{i-1}(y) = \emptyset$ or $\Gamma_{i-1}(z) \cap (M' - A) \neq \emptyset$ for all $z \in \Gamma_{i-1}(y) - A$.*

Proof. 1°. We first prove that $M \in \mathcal{M}_i - \mathcal{M}_{i-1}(x, \bar{A}) - \mathcal{M}_i(\bar{x}, A)$ contains x. Suppose $x \notin M$, then $M \cap A \neq \emptyset$, since M is an MIS of $G(W_i)$. Moreover, $\Gamma_{i-1}(z) \cap M \neq \emptyset$ for any $z \notin M + \{x\}$, since $M \in \mathcal{M}_i$ or $\Gamma_i(z) \cap M \neq \emptyset$ for any $z \notin M$. Thus, $M + \{x\}$ is an MIS of $G(W_{i-1})$, and hence $M + \{x\} \in \mathcal{M}_{i-1}(x, A)$ and $M \in \mathcal{M}_i(\bar{x}, A)$. This contradicts the assumption for M.

2°. We now show the existence of $M' \in \mathcal{M}_{i-1}(x, A)$ satisfying (i). As can be readily seen, any $M \in \mathcal{M}_i - \mathcal{M}_{i-1}(x, \bar{A}) - \mathcal{M}_i(\bar{x}, A)$ is an IS of $G(W_{i-1})$, and hence we can find an MIS M' of $G(W_{i-1})$ which contains M. If $M' = M$, then since $x \in M$ and $M \cap A = \emptyset$, $M' = M$ is an element of $\mathcal{M}_{i-1}(x, \bar{A})$, which contradicts the assumption for M. Thus, M' contains M properly. Suppose that a vertex $v \notin A$ belongs to $M' - M$; then $\Gamma_{i-1}(v) \cap M' = \emptyset$, and hence $\Gamma_i(v) \cap M = \emptyset$, since $M' \supset M$ and $\Gamma_{i-1}(v) = \Gamma_i(v)$ for $v \notin A$. This contradicts $M \in \mathcal{M}_i$. Therefore, this M' belongs to $\mathcal{M}_{i-1}(x, A)$ and satisfies (i).

3°. In the following we show that any such M' as found in 2° also satisfies the condition (ii). Since M' is an MIS of $G(W_{i-1})$, for any $y \in M' - M = M' \cap A$ either

$$\Gamma_{i-1}(y) = \emptyset, \tag{9}$$

or

$$z \notin M \subset M' \quad \text{for all } z \in \Gamma_{i-1}(y). \tag{10}$$

Consider only the latter case, and as can be readily verified, $\Gamma_i(z) \cap M = \emptyset$ implies that $M + \{z\}$ is an IS of $G(W_i)$. This contradicts $M \in \mathcal{M}_i$. Thus, we have $\Gamma_i(z) \cap M = \Gamma_i(z) \cap (M' - A) \neq \emptyset$, and hence $\Gamma_{i-1}(z) \cap (M' - A) \neq \emptyset$ for $z \in \Gamma_{i-1}(y) - A$, since $\Gamma_{i-1}(z) = \Gamma_i(z)$ for $z \notin A$. This completes the proof.

Given $M' \in \mathcal{M}_{i-1}(x, A)$, if M' satisfies the condition (ii) in Lemma 2, then let M' be said to satisfy *condition* A.

Based on Lemma 2, we can construct a derivation policy for all the MIS's of $G(W_i)$, which are contained in $\mathcal{M}_i - \mathcal{M}_{i-1}(x, \bar{A}) - \mathcal{M}_i(\bar{x}, A)$, as follows.

Let $\mathscr{C}_A$ be a set of those MIS's satisfying condition A which are contained in $\mathcal{M}_{i-1}(x, A)$. We now introduce an equivalence relation "$\equiv$" in $\mathscr{C}_A$ such that $M_1 \equiv M_2$ for $M_1, M_2 \in \mathscr{C}_A$ if and only if $M_1 - A = M_2 - A$. Then we partition $\mathscr{C}_A$ into equivalence classes, and let $\tilde{\mathscr{C}}_A$ denote a set of the representatives of the equivalence classes. Define

$$\tilde{\mathscr{C}}_B \triangleq \{M = M' - A \mid M' \in \tilde{\mathscr{C}}_A\}, \tag{11}$$

then we can see from Lemma 2 that $\mathcal{M}_i - \mathcal{M}_{i-1}(x, \bar{A}) - \mathcal{M}_i(\bar{x}, A) \subset \tilde{\mathscr{C}}_B$.

Furthermore, we can prove the following lemma.

LEMMA 3. *There holds the following equation*:

$$\tilde{\mathscr{C}}_B = \mathcal{M}_i - \mathcal{M}_{i-1}(x, \bar{A}) - \mathcal{M}_i(\bar{x}, A). \tag{12}$$

Proof. It is sufficient to show that $\tilde{\mathscr{C}}_B \subset \mathcal{M}_i - \mathcal{M}_{i-1}(x, \bar{A}) - \mathcal{M}_i(\bar{x}, A)$. By definition, $\tilde{\mathscr{C}}_B \cap \mathcal{M}_{i-1}(x, \bar{A}) = \emptyset$ and $\tilde{\mathscr{C}}_B \cap \mathcal{M}_i(\bar{x}, A) = \emptyset$. Thus, we have only to show that $\tilde{\mathscr{C}}_B \subset \mathscr{M}_i$. Suppose that $M \in \tilde{\mathscr{C}}_B$ is not an MIS of $G(W_i)$, then there exists a nonempty set X such that $M+X$ is an MIS of $G(W_i)$. On the other hand, we can see from the definition of $\tilde{\mathscr{C}}_B$ that there exists an MIS $M' \in \mathscr{M}_{i-1}(x, A)$ of $G(W_{i-1})$ such that $M'-A=M$. Now, we shall show that for such a set X and any $y \in M' \cap A$ there hold

$$y \notin X, \tag{13}$$

and

$$\Gamma_{i-1}(y) \cap X = \varnothing. \tag{14}$$

Since $M+X \in \mathscr{M}_i$ and $x \in M$, (13) can be readily verified. Consider any $z \in \Gamma_{i-1}(y)$. If $z \in \Gamma_{i-1}(y) \cap A$, then since $x \in M$ and $M+X \in \mathscr{M}_i$, we obviously have $z \notin X$. Otherwise, we have $\Gamma_i(z) \cap M = \Gamma_{i-1}(z) \cap M \neq \varnothing$, since M' satisfies condition A, and hence $z \notin X$, since $M+X$ is an MIS of $G(W_i)$. Thus, we see that (14) also holds. However, (13) and (14) imply that $M+X+(M' \cap A)$ is an IS of $G(W_{i-1})$, contrary to the fact that $M'=M+(M' \cap A)$ is an MIS of $G(W_{i-1})$. Hence the lemma.

We then consider how to derive $\tilde{\mathscr{C}}_B$ from $\mathscr{M}_{i-1}(x, A)$. Let all the vertices in $A=\Gamma_i(x)$ be arranged in an arbitrary order with subscripts 1 through $p \triangleq |A|$ such that

$$A \triangleq \{y_1, y_2, y_3, \cdots, y_p\}, \tag{15}$$

and let

$$\{y_j\}_{j=k}^{h} \triangleq \{y_k, y_{k+1}, \cdots, y_{h-1}, y_h\}. \tag{16}$$

For any MIS $M' \in \mathscr{M}_{i-1}(x, A)$ of $G(W_{i-1})$, let M' be said to satisfy *condition* B, if for any vertex $y_k \in M' \cap A (1 \leqq k \leqq p)$ either $\Gamma_{i-1}(y_k)=\varnothing$ or $\Gamma_{i-1}(z) \cap (M'-\{y_j\}_{j=1}^{k}) \neq \varnothing$ for all $z \in \Gamma_{i-1}(y_k)-\{y_j\}_{j=1}^{k}$. In other words, what M' satisfies condition B implies that for any $v \in \bigcup_{y \in M' \cap A} \Gamma_{i-1}(y)$ $(\subset V-M')$,

(i) if $v \notin A$, then v is adjacent to some vertex of $M'-A$ in $G(W_{i-1})$, or

(ii) if $v \in A$ and we let $y_j \triangleq v \in A$, then in $G(W_{i-1}) v$ is adjacent to some vertex of $M'-A$ or some $y_k \in M' \cap A$ with $k>j$.

Note that if M' satisfies condition A, then any $v \in \bigcup_{y \in M' \cap A} \Gamma_{i-1}(y)$ satisfies (i), but not always (ii). In this sense, for M' condition B is more tight than condition A.

We now prove the following lemmas.

LEMMA 4. *Let*

$$\mathscr{M}_i'(x, \bar{A}) \triangleq \{M=M'-A \mid M' \in \mathscr{M}_{i-1}(x, A), \text{ and } M' \text{ satisfies condition } B\}; \tag{17}$$

then

$$\mathscr{M}_i'(x, \bar{A}) \supset \tilde{\mathscr{C}}_B. \tag{18}$$

Proof. It is sufficient to show that one and only one element of each equivalence class of $\mathscr{C}_A$ satisfies condition B.

1°. We shall first show that any two distinct M_1' and M_2' contained in an equivalence class of $\mathscr{C}_A$ do not simultaneously satisfy condition B. Let y_{h_1} be the vertex in $M_1'-M_2'$ $(\subset A=\Gamma_i(x))$ with the largest subscript h_1, and let y_{h_2} be the one in $M'_2 - M'_1$ ($\subset A$) with the largest subscript h_2. Unless otherwise stated, henceforth let M'_1 be said to be in a higher level than M'_2, if $h_1 > h_2$, and without loss of generality assume that M'_1 is in a higher level than M'_2. Then, since $y_{h_1} \in M'_1 - M'_2$ and $M'_2 \in \mathcal{M}_{i-1}(x, A)$, we have $\Gamma_{i-1}(y_{h_1}) \cap M'_2 \neq \varnothing$, and moreover noting that $M'_1 - A = M'_2 - A$ and $h_1 > h_2$, we can see that $\Gamma_{i-1}(y_{h_1}) \cap M'_2 \subset \{y_j\}_{j=1}^{h_1} \subset A$. Let y_t be the vertex with the largest subscript in $\Gamma_{i-1}(y_{h_1}) \cap M'_2$; then for this $y_t (\in M'_2 \cap A)$ there exists $y_{h_1} (\in \Gamma_{i-1}(y_t) - \{y_j\}_{j=1}^{t})$ such that $\Gamma_{i-1}(y_{h_1}) \cap (M'_2 - \{y_j\}_{j=1}^{t}) = \varnothing$, and hence M'_2 does not satisfy condition B.

2°. Given an equivalence class of $\mathscr{C}_A$, let M'_m be in the highest level of all the MIS's in the class, and we shall show that M'_m satisfies condition B. Suppose that M'_m does not satisfy condition B. Then, for some $y_k \in M'_m \cap A$ there exists a vertex $z \in \Gamma_{i-1}(y_k) - \{y_r\}_{r=1}^{k}$ such that

$$\Gamma_{i-1}(z) \cap (M'_m - \{y_r\}_{r=1}^{k}) = \varnothing. \tag{19}$$

If we assume that $z \in \Gamma_{i-1}(y_k) - A$, then since M'_m satisfies condition A, we have

$$\Gamma_{i-1}(z) \cap (M'_m - \{y_r\}_{r=1}^{p}) \neq \varnothing. \tag{20}$$

Hence, there hold $z \in \{y_r\}_{r=k+1}^{p}$ and

$$\Gamma_{i-1}(z) \cap M'_m \subset \{y_r\}_{r=1}^{k}. \tag{21}$$

Now, let $y_j \triangleq z (j > k)$, and consider a set X defined by

$$X \triangleq M'_m + \{y_j\} - (\Gamma_{i-1}(y_j) \cap M'_m). \tag{22}$$

Then, X is an IS of $G(W_{i-1})$, and therefore we can construct an MIS $M''_m \triangleq X + Y$ of $G(W_{i-1})$ by adding to X a suitably chosen set Y if necessary (see Fig. 1). For this M''_m, we shall show in the following that (i) $M'_m - A = M''_m - A$, and (ii) M''_m satisfies condition A:

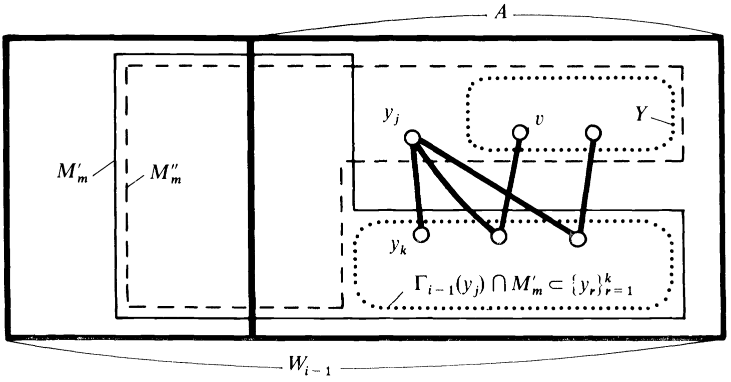


FIG. 1. *Illustrating example for proof*

(i) For any $v \in Y$, we have $\Gamma_{i-1}(v) \cap M'_m \neq \varnothing$ and $\Gamma_{i-1}(v) \cap X = \varnothing$. Therefore, from the definition of X there holds $\Gamma_{i-1}(v) \cap M'_m \subset \Gamma_{i-1}(y_j) \cap M'_m$, and hence by (21)

$$\Gamma_{i-1}(v) \cap M'_m \subset \{y_r\}_{r=1}^{k} \subset A. \tag{23}$$

Let y_t be a vertex in $M'_m \cap A$ which is adjacent to v in $G(W_{i-1})$, and if we assume that $v \in \Gamma_{i-1}(y_t) - A$, then since M'_m satisfies condition A, for this v there holds $\Gamma_{i-1}(v) \cap (M'_m - A) \neq \emptyset$, which contradicts (23). Thus, $v \in \Gamma_{i-1}(y_t) \cap A$, and hence

$$M'_m - A = M''_m - A. \tag{24}$$

(ii). For any $y \in (M''_m \cap A) - \{y_j\} - Y$, we have obviously either $\Gamma_{i-1}(y) = \emptyset$ or $\Gamma_{i-1}(z) \cap (M''_m - A) \neq \emptyset$ for all $z \in \Gamma_{i-1}(y) - A$, since y is contained in M'_m which satisfies condition A. Consider any $v \in Y + \{y_j\}$ $(\subset M''_m \cap A)$, and suppose that this v does not satisfy the condition (ii) of Lemma 2. Then there exists a vertex $w \in \Gamma_{i-1}(v) - A$ such that $\Gamma_{i-1}(w) \cap (M''_m - A) = \emptyset$, and hence by (24), $\Gamma_{i-1}(w) \cap (M'_m - A) = \emptyset$. Moreover, since M'_m satisfies condition A, w is not adjacent to any vertex of $M'_m \cap A$ in $G(W_{i-1})$. Hence , we have

$$\Gamma_{i-1}(w) \cap M'_m = \emptyset. \tag{25}$$

On the other hand, since $w \notin A$, $w \notin M''_m$, and $M'_m - A = M''_m - A$, we can see that

$$w \notin M'_m. \tag{26}$$

Therefore, from (25) and (26), $M'_m + \{w\}$ is an IS of $G(W_{i-1})$, which contradicts $M'_m \in \mathcal{M}_{i-1}$. This completes the proof of (ii):

Thus, from (i) and (ii) M'_m and M''_m belong to the same equivalence class of $\mathscr{C}_A$. Noting that M''_m contains $y_j = z$ $(j > k)$, and that from (21) and (22) there holds $M'_m - M''_m \subset \Gamma_{i-1}(y_i) \cap M'_m \subset \{y_r\}_{r=1}^{k}$, we see that M''_m is in a higher level than M'_m. This contradicts the assumption for M'_m. Hence the lemma.

LEMMA 5. *There also holds* $\mathcal{M}'_i(x, \bar{A}) \subset \tilde{\mathscr{C}}_B$, *and consequently we have*

$$\mathcal{M}'_i(x, \bar{A}) = \tilde{\mathscr{C}}_B. \tag{27}$$

Proof. It is sufficient to show that any $M' \in \mathcal{M}_{i-1}(x, A)$ satisfying condition B also satisfies condition A. Suppose that any such M' does not satisfy condition A, then there exists a vertex $z \in \Gamma_{i-1}(y_j) - A$ for some $y_j \in M' \cap A$ such that $\Gamma_{i-1}(z) \cap (M' - A) = \emptyset$. Among such vertices y_j, let y_k be the one with the largest subscript k, and let $z \in \Gamma_{i-1}(y_k) - A$ be a vertex such that

$$\Gamma_{i-1}(z) \cap (M' - A) = \emptyset. \tag{28}$$

Then, from the definition of y_k there holds

$$\Gamma_{i-1}(z) \cap (M' \cap \{y_r\}_{r=k+1}^{p}) = \emptyset. \tag{29}$$

From (28) and (29) we obtain $\Gamma_{i-1}(z) \cap (M' - \{y_r\}_{r=1}^{k}) = \emptyset$, which implies that M' does not satisfy condition B, contrary to the assumption. Hence the lemma.

Lemmas 4 and 5 indicate that $\tilde{\mathscr{C}}_B$ can be derived directly from $\mathcal{M}_{i-1}(x, A)$ without duplication, and moreover by Lemmas 3 and 5 the following theorem follows.

THEOREM 1. *There holds the equation*

$$\mathcal{M}_i = \mathcal{M}_{i-1}(x, \bar{A}) + \mathcal{M}'_i(x, \bar{A}) + \mathcal{M}_i(\bar{x}, A). \tag{30}$$

With the use of this theorem, we can show an outline to generate all the MIS's of $G(W_i)$ from those of $G(W_{i-1})$ as in Fig. 2.

```
begin
  empty the set $\mathcal{M}_i$;
  for each $M' \in \mathcal{M}_{i-1}$ do
      if $M' \cap \Gamma_i(x) = \emptyset$ then put $M' \in \mathcal{M}_{i-1}(x, \bar{A})$ into $\mathcal{M}_i$
      else begin (in this case $M' \in \mathcal{M}_{i-1}(x, A)$)
              put $M' - \{x\} \in \mathcal{M}_i(\bar{x}, A)$ into $\mathcal{M}_i$;
              if $M'$ satisfies condition $B$ then
                 put $M' - \Gamma_i(x) \in \mathcal{M}_i'(x, \bar{A})$ into $\mathcal{M}_i$;
           end;
end;
```

FIG. 2. *Outline to generate the MIS's of $G(W_i)$ from those of $G(W_{i-1})$*

3. Algorithm. Based on the consideration so far stated, in what follows we shall describe the details of a scheme to generate all the MIS's of a given graph $G = [V, E]$.

Considering that in each stage of generating the MIS's of $G(W_i)$ from those of $G(W_{i-1})$ the processing time depends primarily on the procedure of determining whether each $M' \in \mathcal{M}_{i-1}(x, A)$ satisfies condition B or not, we first discuss how to achieve this determination efficiently.

Introduce a mapping $f_i : \mathcal{M}_i \times V \to I$, where I is the set of nonnegative integers, which is defined such that for any MIS M and vertex v of $G(W_i) = [V, E(W_i)]$,

$$f_i(M, v) \triangleq |\Gamma_i(v) \cap M|. \tag{31}$$

Then we can see that $f_i(M, v) = 0$ if and only if $v \in M$, and furthermore from Theorem 1 that the following relations are derived.

I. For each $M' \in \mathcal{M}_{i-1}(x, \bar{A})$,

$$f_i(M', y_j) = f_{i-1}(M', y_j) + 1 \quad \text{for all } y_j \in A = \Gamma_i(x), \tag{32}$$

$$f_i(M', v) = f_{i-1}(M', v) \quad \text{for all } v \in V - A. \tag{33}$$

II. For each $M' \in \mathcal{M}_{i-1}(x, A)$,

$$f_i(M' - \{x\}, x) = |M' \cap A|, \tag{34}$$

$$f_i(M' - \{x\}, v) = f_{i-1}(M', v) \quad \text{for all } v \in V - \{x\}. \tag{35}$$

III. For each $M' \in \mathcal{M}_{i-1}(x, A)$ satisfying condition B,

$$f_i(M' - A, y_j) = f_{i-1}(M', y_j) + 1 - |M' \cap A \cap \Gamma_{i-1}(y_j)| \quad \text{for all } y_j \in A, \tag{36}$$

$$f_i(M' - A, v) = f_{i-1}(M', v) - |M' \cap A \cap \Gamma_{i-1}(v)| \quad \text{for all } v \in V - A. \tag{37}$$

We can also observe that $M' \in \mathcal{M}_{i-1}(x, A)$ satisfies condition B if and only if for any $y_k \in M' \cap A$ there holds either $\Gamma_{i-1}(y_k) = \emptyset$, or $f_{i-1}(M', z) - |M' \cap \{y_j\}_{j=1}^k \cap \Gamma_{i-1}(z)| \neq 0$ for all $z \in \Gamma_{i-1}(y_k) - \{y_j\}_{j=1}^k$. Noting that given k $(1 \leqq k \leqq p = |A|)$, $f_{i-1}(M', z) - |M' \cap \{y_j\}_{j=1}^k \cap \Gamma_{i-1}(z)| + 1 > 0$ for any $z \in \Gamma_{i-1}(y_k) \cap \{y_j\}_{j=1}^k$, we can show an outline how to determine whether M' satisfies condition B, as follows.

1°. Provide an integer variable $\mathrm{IS}(v)$ for each $v \in V$, and initially set $\mathrm{IS}(v) \leftarrow f_{i-1}(M', v)$ for all v.

2°. In the order of subscript numbers of $y_j \in A = \{y_j\}_{j=1}^{p}$, conduct the following process for each y_j: Set $\mathrm{IS}(z) \leftarrow IS(z) - 1$ for each $z \in \Gamma_{i-1}(y_j)$ only if $y_j \in M'$, and then put $\mathrm{IS}(y_j) \leftarrow \mathrm{IS}(y_j) + 1$ for y_j.

3°. In the process of each iteration for y_j in 2°, if there exists a vertex z adjacent to any $y_j \in M'$ such that $\mathrm{IS}(z) - 1 = 0$, then M' is determined not to satisfy condition B, or else M' is proved to satisfy condition B.

Here, we should note that if M' satisfies condition B, then at the termination of 2°, $\mathrm{IS}(v)$ for each v has been set to $f_i(M' - A, v)$.

For simplicity, henceforth let V of a given graph $G = [V, E]$ be represented by integers 1 through $n \triangleq |V|$, and let the incidence structure of G be specified by a set of the adjacency lists $\mathrm{Adj}(v) \triangleq \Gamma_G(v)$ for all $v \in V$. For each $i = 1, 2, \cdots, n$, put $W_i \triangleq \{1, 2, \cdots, i\}$, and denote by $\mathcal{M}_i \triangleq \{M_1^i, M_2^i, \cdots, M_{\mu_i}^i\}$ a set of the MIS's of subgraph $G(W_i)$.

To describe the proposing algorithm compactly, we introduce a digraph $\mathscr{G}$ associated with G to be defined below. Let each vertex M_j^i of $\mathscr{G}$ corresponds one-to-one to each element M_j^i of $\mathcal{M}_i$ for $i = 1, 2, \cdots, n$, and let an edge be incident from vertex $M_k^{i-1} \in \mathcal{M}_{i-1}$ to vertex $M_h^i \in \mathcal{M}_i$, if and only if $M_h^i \in \mathcal{M}_i$ is generated from $M_k^{i-1} \in \mathcal{M}_{i-1}$ in the sense stated in the procedure shown in Fig. 2.

As readily seen, $\mathscr{G}$ is an *arborescence* [3] with a root vertex $M_1^1 = V \in \mathcal{M}_i$, as illustrated in Fig. 3, and there exist either one or two edges incident from each M_j^i $(i \neq n)$, and moreover if there exist two edges from any M_j^{i-1}, then $M_j^{i-1} \in \mathcal{M}_{i-1}$ can be seen to satisfy condition B.

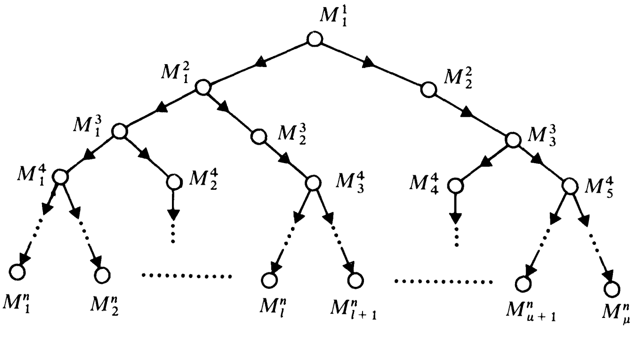


FIG. 3. *Digraph* $\mathscr{G}$

We now show in Fig. 4 our algorithm called **procedure** MIS to generate all the MIS's in a given graph G. This procedure starts at the root vertex M_1^1 of $\mathscr{G}$ associated with G, and proceeds to each vertex of $\mathscr{G}$ by backtracking. In this process, **procedure** *BACKTRACK* is called for each vertex.

For any $M_j^{i-1} \in \mathcal{M}_{i-1}$, consider the stage when $BACKTRACK(i-1)$ is called for vertex M_j^{i-1}, then the following process is to be conducted: Statement C1 examines whether $M_j^{i-1} \cap \Gamma_i(i) = \varnothing$ or not with the use of counter "c", for at this time each $\mathrm{IS}(v)$ is equal to $f_{i-1}(M_j^{i-1}, v)$ (this can be verified by induction). If $c = 0$, which implies that $M_j^{i-1} \cap \Gamma_i(i) = \varnothing$, statement B1 calls $BACKTRACK(i)$ for vertex M_k^i such that $M_j^{i-1} = M_k^i \in \mathcal{M}_i$. If $c \neq 0$, statement B2 calls $BACKTRACK(i)$ for vertex M_k^i such that $M_k^i = M_j^{i-1} - \{i\} \in \mathcal{M}_i$, and then statement C2

```
procedure MIS;
    comment Procedure MIS is a routine for generating all the MIS's of a graph $G$ represented by
      adjacency lists $Adj(\cdot)$. Integer $n$ is a global variable denoting the number of vertices;
begin integer list array $Bucket(n)$; integer array $IS(n)$;
  procedure $BACKTRACK$(integer value $i$);
    begin integer $c, x$;   logical $f$;
      if $i<n$ then
        begin
          $x := i+1$;
          $c := 0$;
C1:       for $y \in Adj(x)$ such that $y \leqq i$ do
            if $IS(y)=0$ then $c := c+1$;
          if $c=0$ then
            begin
L1:           for $y \in Adj(x)$ such that $y \leqq i$ do $IS(y) := IS(y)+1$;
B1:           $BACKTRACK(x)$;
L2:           for $y \in Adj(x)$ such that $y \leqq i$ do $IS(y) := IS(y)-1$;
            end
          else
            begin
              $IS(x) := c$;
B2:           $BACKTRACK(x)$;
              $IS(x) := 0$;
              $f :=$ true;
C2:           for $y \in Adj(x)$ such that $1 \leqq y \leqq i$, in increasing order do
                begin
                  if $IS(y)=0$ then
                    begin
                      put $y$ in $Bucket(x)$;
C3:                   for $z \in Adj(y)$ such that $z \leqq i$ do
                        begin
                          $IS(z) := IS(z)-1$;
                          if $IS(z)=0$ then $f :=$ false;
                        end;
                    end;
                  $IS(y) := IS(y)+1$;
                end;
B3:           if $f=$ true then $BACKTRACK(x)$;
L3:           for $y \in Adj(x)$ such that $y \leqq i$ do $IS(y) := IS(y)-1$;
L4:           for $y \in Bucket(x)$ do
                begin
L5:               for $z \in Adj(y)$ such that $z \leqq i$ do $IS(z) := IS(z)+1$;
                  delete $y$ from $Bucket(x)$;
                end;
            end;
        end
      else output new MIS designated by $IS(\cdot)$;
    end $BACKTRACK$;
  for $j := 1$ until $n$ do
      begin
        $IS(j) := 0$;
        $Bucket(j) := \varnothing$;
      end;
  $BACKTRACK(1)$;
end MIS;
```

FIG. 4. *An algorithm for generating all the MIS's*

examines whether M_j^{i-1} satisfies condition B or not. Only if f = true, which implies that M_j^{i-1} satisfies condition B, statement B3 calls *BACKTRACK*(i) for vertex M_k^i such that $M_k^i = M_j^{i-1} - \Gamma_i(i) \in \mathcal{M}_i$.

Here, we can see that every time *BACKTRACK*(i) is called for M_k^i in statements B1, B2, and B3, each IS(v) has just been replaced by $f_i(M_k^i, v)$, and that after the termination of *BACKTRACK* (i), each IS(v) is subsequently to be restored to the original value $f_{i-1}(M_j^{i-1}, v)$. However, we should remark that the restoring process for IS(v) in the statements L3 and L4 is different from those in other statements. In fact, a list *Bucket*$(\cdot)$ is newly introduced with a function such that all the vertices in $\Gamma_i(i)$, for which statement C3 is executed, are to be put into *Bucket*(i), whether M_j^{i-1} satisfies condition B or not.

Associated with this algorithm, the following theorem can be verified by Theorem 1 and the above discussion.

THEOREM 2. *The application of procedure MIS yields all the MIS's of a given graph without duplication.*

We can also prove the following theorem, which reveals the efficiency of the algorithm.

THEOREM 3. *Given a graph $G = [V, E]$ with $n \triangleq |V|$, $m \triangleq |E|$, and μ MIS's, procedure MIS requires processing time and memory space bounded by $O(nm\mu)$ and $O(n+m)$, respectively.*

Proof. Noting that memory space of $O(n+m)$ is necessary for list arrays *Bucket*$(\cdot)$ and *Adj*$(\cdot)$, respectively, the second half of the theorem may be easily verified. Thus, we shall estimate the processing time to be required in the algorithm. First, we can see that in each *BACKTRACK*(i), the iteration number of any of the statements L1, L2, L3, L4, C1, and C2 is not greater than the degree $|\Gamma_G(x)|$ of vertex $x = i+1$. On the other hand, we can also observe that any of the total numbers of iterations of the statements L5 and C3 in each *BACKTRACK*(i) does not exceed $\sum_{y \in \Gamma_G(x)} |\Gamma_G(y)| \leqq 2m$ $(x = i+1)$. Thus, the processing time $T(i)$ spent by one call of *BACKTRACK*(i) exclusive of recursive calls to *BACKTRACK*, is bounded by $T(i) < k_1 m + k_2 |\Gamma_G(i+1)| + k_3$, where k_1, k_2, and k_3 are constants. Consequently, the processing time required to find an MIS is bounded by $\sum_{i=1}^{n-1} T(i) < k_1 nm + 2k_2 m + k_3 n$, or $O(nm)$, and hence the total processing time is bounded by $O(nm\mu)$. This completes the proof.

4. Implemented results. To observe how efficiently this algorithm is virtually executed, it is programmed in FORTRAN and run on NEAC 2200/700.

We first consider a class of graphs of $n = m = 3k$ consisting of k vertex-disjoint triangles (or 3-cliques), which have 3^k MIS's and are proved to contain the largest number of MIS's per vertex [8]. Associated with this class, to save a large amount of printing, we modify the algorithm so as to count only the number of MIS's, not to list the members of each MIS. Table 1 shows the implemented result for $k = 4, 5, \cdots, 13$, which demonstrates that the processing time is bounded by $O(\mu)$. This time bound can also be theoretically verified as follows: As shown in the proof of Theorem 3, one call of *BACKTRACK*(i) for vertex M_j^i of $\mathcal{G}$ requires processing time of $O(\sum_{y \in \Gamma_G(x)} |\Gamma_G(y)| + |\Gamma_G(x)|)$, exclusive of recursive calls to *BACKTRACK*. Considering that in such a graph the degree $|\Gamma_G(v)|$ of each vertex v is 2, this processing time is of order $O(1)$, since the modified algorithm does not list each MIS.[2] Note that the total processing time is proportional to the

[2] In the original algorithm, this processing time is of order $O(n)$.

number ζ of vertices in $\mathscr{G}$; then we can see that it is bounded by $O(\zeta+n)$, where additional $O(n)$ is required in the initialization of IS($\cdot$) and *Bucket*($\cdot$). On the other hand, since each such graph of $n=3k$ contains 3^k MIS's, the number ζ of vertices of $\mathscr{G}$ is less than 3^{k+1} and is bounded by $O(\mu)$.

TABLE 1

Implemented results for graphs of $|V|=|E|=3k$ consisting of k vertex-disjoint 3-cliques

k	4	5	6	7	8
Number of MIS's	81	243	729	2187	6561
CPU time per MIS (μsec)	555.6	551.4	558.3	547.3	554.8

k	9	10	11	12	13
Number of MIS's	19683	59049	177147	531441	1594323
CPU time per MIS (μsec)	585.4	586.8	580.3	546.3	546.3

We now consider randomly generated graphs ranging between $n=10$ and 50. For each value of n, we generated three classes of graphs with $\xi(\triangleq 2m/[n(n-1)])=0.25$, 0.50, and 0.75. The CPU time per MIS averaged over each such class is shown in Figs. 5 and 6, plotted with respect to n and m, respectively. This result demonstrates that the CPU time per MIS is almost $O(m)$.

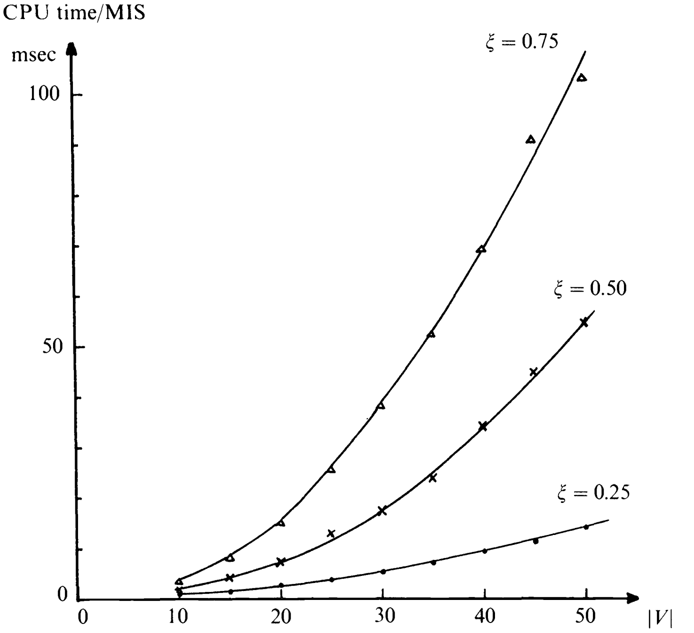


FIG. 5. *CPU time per MIS for random graphs plotted with the size of $n \triangleq |V|$*

5. Conclusion. A new algorithm for generating all the MIS's of a given graph is presented. The improvement of efficiency accomplished in this algorithm is mainly due to the investigation of a close relation between $\mathcal{M}_{i-1}$ and $\mathcal{M}_i$ which is summarized as follows:

(i) Each $M_j^{i-1} \in \mathcal{M}_{i-1}$ generates either one or two $M_k^i \in \mathcal{M}_i$ without duplication.

(ii) Any $M_k^i \in \mathcal{M}_i$ to be generated from $M_j^{i-1} \in \mathcal{M}_{i-1}$ depends only on M_j^{i-1} and the topology of $G(W_i)$, independently of any other MIS of $G(W_{i-1})$ or $G(W_i)$.

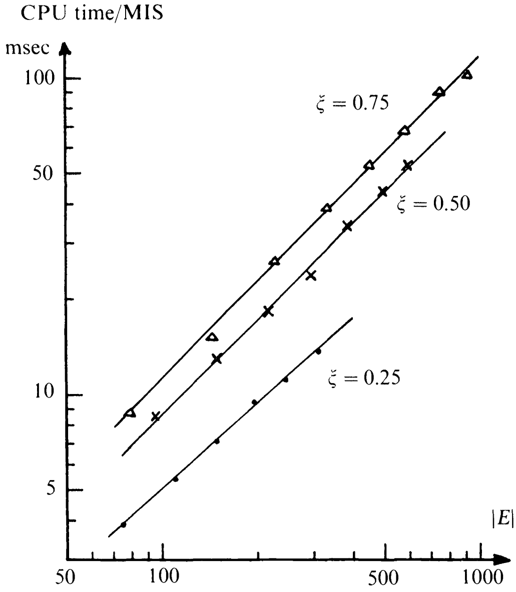


FIG. 6. *CPU time per MIS for random graphs plotted with the size of m* $\triangleq |E|$

Acknowledgments. The authors express their appreciation to Professor H. Ozaki, Osaka University, for his encouragement, and Professor R. E. Tarjan, University of California at Berkeley, presently Stanford University, for his instructive suggestion that the memory space of this algorithm can be reduced to $O(n+m)$. They also wish to thank the reviewers for their helpful suggestions and constructive criticisms.

REFERENCES

[1] E. A. AKKOYUNLU, *The enumeration of maximal cliques of large graphs*, this Journal, 2 (1973), pp. 1–6.

[2] J. G. AUGUSTON AND J. MINKER, *An analysis of some graph theoretical cluster techniques*, J. Assoc. Comput. Mach., 17 (1970), pp. 571–588; Correction, G. D. MULLIGAN AND D. G. CORNEIL, *Corrections to Bierstone's algorithm for generating cliques*, Ibid., 19 (1972), pp. 244–247.

[3] C. BERGE, *The Theory of Graphs and Its Applications*, John Wiley, New York, 1962.

[4] M. A. BREUER, *Design Automation of Digital Systems*, vol. 1, Theory and Techniques, Prentice-Hall, Englewood Cliffs, N.J., 1972.

[5] C. Bron and J. Kerbosch, *Finding all cliques of an undirected graph—Algorithm 457*, Comm. ACM, 16 (1973), pp. 575–577.

[6] N. Deo, *Graph Theory with Applications to Engineering and Computer Science*, Prentice-Hall, Englewood Cliffs, N.J, 1974.

[7] P. M. Marcus, *Derivation of maximal compatibles using Boolean algebra*, IBM J. Res. Develop., 8 (1964), pp. 537–538.

[8] J. W. Moon and L. Moser, *On cliques in graphs*, Israel J. Math., 3 (1965), pp. 23–28.

[9] R. E. Osteen, *Clique detection algorithms based on line addition and line removal*, SIAM J. Appl. Math., 26 (1974), pp. 126–135.

[10] M. C. Paull and S. H. Unger, *Minimizing the number of states in incompletely specified sequential switching functions*, IRE Trans. Electronic Computers, EC-8 (1959), pp. 356–367.

SIAM J. COMPUT.
Vol. 6, No. 3, September 1977

WORST CASE ANALYSIS OF TWO SCHEDULING ALGORITHMS*

SHUI LAM† AND RAVI SETHI‡

Abstract. Coffman and Graham give an algorithm to schedule unit execution time task systems nonpreemptively. On two processors, their algorithm is optimal. We show that in general, if ω is the length of a schedule produced by their algorithm and ω_0 the length of an optimal schedule, then $\omega/\omega_0 \leqq 2 - 2/m$, where m is the number of processors. The preemptive equivalent of the above algorithm has been considered by Muntz and Coffman. Again we show that the ratio of the lengths of theirs and an optimal schedule is bounded by $2 - 2/m$. In both the nonpreemptive and the preemptive cases there exist task systems for which the ratio $2 - 2/m$ can be approached arbitrarily closely. On a small number of machines, $2 - 2/m$ is not too far from 1. In particular, as noted above, on two machines the ratio *is* 1.

Key words. preemptive, nonpreemptive scheduling, list scheduling, algorithm analysis, processor sharing, level algorithms, critical path algorithms, minimal length schedules

1. Overview. Given a set of tasks to be executed on a multiprocessor system, the time taken to finish all the tasks (the *schedule length*) provides a measure of processor utilization. Schedule length has therefore been a popular cost criterion in the design of scheduling algorithms. Most such scheduling algorithms are "critical path" or "level by level" algorithms. Whenever a processor becomes available, the algorithms preferentially execute the tasks at the highest level.

In this paper we will consider two kinds of schedules: *nonpreemptive* schedules, in which a task once started is executed to completion, and *preemptive* schedules, where it is possible to suspend and at a later stage resume execution of a task at the point of suspension. Preemption costs are assumed to be negligible.

Scheduling problems tend to fall into the class of combinatorial problems that are called NP-complete [7], [12], [21]. NP-complete problems are notorious in that all known solutions for them are enumerative, and it is strongly suspected that no essentially faster (nonenumerative, polynomial time bounded) algorithms will ever be found. Heuristics for determining approximate solutions are therefore desirable.

A well-known rule for scheduling independent tasks nonpreemptively on m identical processors is the LPT (largest processing time first) rule. Whenever a processor becomes available it is assigned the longest unexecuted task. Schedules constructed by the LPT rule are not always the shortest possible. Karp [12] shows that even on two processors, determining shortest schedules for independent tasks is an NP-complete problem. LPT schedules are, however, easy to construct, and from Graham's analysis, [8], are within 33 percent of the optimal schedule length. More precisely, the ratio of the lengths of the LPT and optimal schedules is bounded by $(4m - 1)/3m$, where m is the number of processors. The LPT rule is a critical path algorithm since the task with the longest remaining execution time determines the critical path at any point.

* Received by the editors January 6, 1975, and in revised form October 18, 1976. This work was supported in part by the National Science Foundation under Grant GJ-282890 at Pennsylvania State University.

† Computer Science Department, Pennsylvania State University, University Park, Pennsylvania. Now at School of Computer Science, McGill University, Montreal, Quebec, Canada H3C 3J7.

‡ Computer Science Department, Pennsylvania State University, University Park, Pennsylvania. Now at Bell Laboratories, Murray Hill, New Jersey 07974.

In general, there are constraints on the order in which tasks in a given system can be executed. Precedence constraints between tasks are conveniently represented by directed acyclic graphs (dags) as in Figs. 1 and 8. Tasks are represented by nodes in Fig. 1. An edge from T to T' means that T precedes T' and must be completed before T' can be started. All tasks in Fig. 1 have the same execution time. Such systems will be referred to as unit execution time, or UET systems.

UET systems are interesting because in certain special cases optimal schedules can be easily constructed. Let the "level" of a task T be the length of a longest path from T to a terminal. Hu [11] shows that executing tasks "level by level" leads to an optimal (shortest) schedule when the precedence relation defines a tree. A similar strategy applied to trees with tasks of arbitrary execution time results in a bound of $1 + (m - 1)\tau/x$ on the ratio of constructed and optimal schedule lengths [14]. Here m is the number of processors, τ the execution time of the longest task, and x the sum of the execution times of all tasks in the system.

When the precedence relation defines an arbitrary directed acyclic graph (dag), level by level execution is no longer optimal, even for UET systems on two processors. Chen and Liu [2], [3] show worst case bounds of $\frac{4}{3}$ for two processors and $2 - 1/(m - 1)$ for $m > 2$ processors. Meanwhile, for general m, determining minimal length schedules for UET systems is an NP-complete problem [21], [22].

Coffman and Graham [5] give an algorithm that also executes tasks level by level, but when there is more than one task at the highest level, it makes a judicious choice of which task to execute first. As a result, for UET systems, their algorithm is optimal on two processors. In § 2 we analyze the performance of their algorithm on $m > 2$ processors and show that its worst case bound is $2 - 2/m$. The bound is best possible in that for every m there exist task systems for which the bound is approached arbitrarily closely. It should be noted that optimal solutions for UET dags on two processors have been obtained by Fujii et al. [6] and Muraoka [18], using quite different approaches.

All of the above algorithms, except the ones by Fujii et al. and Muraoka, belong to the more general class of *list scheduling* algorithms [8], [9]: first a list of the given tasks is constructed. Whenever a processor becomes available, the list is scanned and the first unexecuted task that is ready to execute is assigned to the processor. Graham [8], [9] compares an arbitrary list schedule with an optimal one and obtains the worst case ratio of $2 - 1/m$. Kaufman [13] has given examples showing that the $2 - 1/m$ bound is achievable even for UET systems. As shown in Table 1, the difference between $2 - 2/m$ and $2 - 1/m$ is significant when m is small. In particular, $2 - 2/m$ implies an optimal schedule on 2 processors.

TABLE 1
A comparison of two bounds of ω/ω_0

m	$2 - 2/m$	$2 - 1/m$
2	1.00	1.50
3	1.33	1.67
4	1.50	1.75
5	1.60	1.80
10	1.80	1.90

When preemption is allowed, a task of execution time τ can be viewed as a chain of τ one unit tasks. As in the nonpreemptive case, optimal schedules can easily be constructed when the precedence relation defines a tree, or if there are two processors available. A level by level algorithm due to Muntz and Coffman [16], [17] constructs optimal schedules for both the above cases. In § 3 we analyze the Muntz–Coffman algorithm and show that $2 - 2/m$ is again a best bound when such schedules are compared with optimal schedules.

2. The Coffman–Graham algorithm. We are given a task system $(\mathscr{T}, <)$, where $\mathscr{T} = \{T_1, \cdots, T_n\}$ is a set of n tasks to be executed on m processors, and $<$ is a partial order on $\mathscr{T}$ that specifies the precedence relation among tasks. For tasks T and T' in $\mathscr{T}$, if $T < T'$, then T is called a *predecessor* of T' and T' a *successor* of T, and the execution of T must be completed before T' can begin. If there exists no task T'' such that $T < T'' < T'$ then T will be called an *immediate predecessor* of T' and T' an *immediate successor* of T. T is a *terminal* (*initial*) task if T has no successors (predecessors). The graph that represents a task system $(\mathscr{T}, <)$ has $\mathscr{T}$ as the node set. There is a directed edge from T to T' in the graph if and only if T is an immediate predecessor of T'.

Let $(\mathscr{T}, <)$ be a UET system. A *chain* of length k is a list of k tasks $(T_{i_1}, T_{i_2}, \cdots, T_{i_k})$ such that for all j, $1 \leqq j < k$, $T_{i_j} < T_{i_{j+1}}$. The *level* of a task T is the length of a longest chain from T to a terminal task.

The idea of the Coffman–Graham algorithm is to assign a distinct label to each task and then construct a schedule using the list of tasks in order of decreasing label. Tasks at higher levels will have higher labels, while tasks at the same level are assigned labels according to the lexicographical order of the labels of their immediate successors.

In this section we will use the following notation. For all tasks $T \in \mathscr{T}$, let $S(T)$ denote the set of immediate successors of T. Let $\alpha(T)$ denote an integer label assigned to task T. $N(T)$ is used to denote the decreasing sequence of integers formed by ordering the set $\{\alpha(T') | T' \in S(T)\}$.

As noted in [20], for the Coffman–Graham algorithm to be optimal, the graph specifying the UET system $(\mathscr{T}, <)$ must not contain any transitive edges. The definition of immediate successor and graph excludes the possibility of having transitive edges in the graph representing a task system. Note, however, that for an n-node graph it takes $O(n^{2.81})$ time to delete all the transitive edges in the graph [1].

COFFMAN–GRAHAM (CG-) ALGORITHM. Given a UET System $(\mathscr{T}, <)$, this algorithm determines a nonpreemptive schedule for $(\mathscr{T}, <)$ on m identical processors.

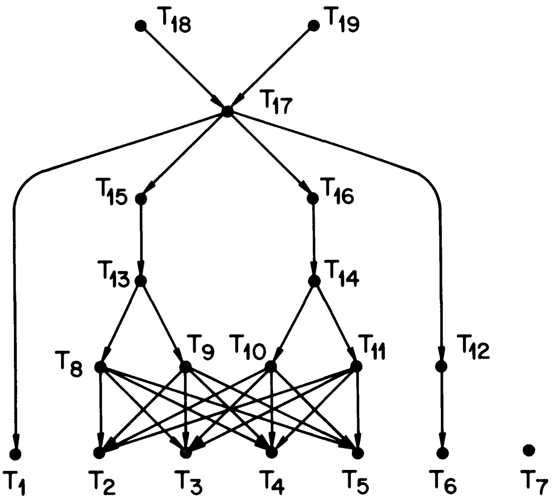
1. Choose an arbitrary task T_0 from $\mathscr{T}$ with $S(T_0) = \varnothing$, and define $\alpha(T_0)$ to be 1.

2. Suppose, for some $i \leqq n$, that labels $1, 2, \cdots, i - 1$ have been assigned. Let R be the set of tasks with no unlabeled successors, i.e., for all $T \in R$, $N(T)$ is defined. Let T^* be a task in R such that $N(T^*)$ is lexicographically smaller than $N(T)$ for all T in R. (Break ties at will.) Define $\alpha(T^*)$ to be i.

3. When all tasks have been labeled, construct a list of tasks $L = (U_n, U_{n-1}, \cdots, U_1)$ such that $\alpha(U_i) = i$ for all i, $1 \leqq i \leqq n$. Use list L to determine a list schedule for $(\mathscr{T}, <)$.

Schedules constructed by this algorithm will be referred to as CG-schedules.

Lexicographic order is dictionary order, so that (5, 4, 3) is smaller than (6, 2) and (5, 4, 3, 2). Fig. 1 gives an example of a task system and its CG-schedule on 3 processors.



	1	2	3	4	5	6	7	8	
P_1	T_{19}	T_{17}	T_{16}	T_{14}	T_{11}	T_8	T_5	T_2	▨
P_2	T_{18}	▨	T_{15}	T_{13}	T_{10}	T_1	T_4	▨	▨
P_3	T_7	▨	T_{12}	T_6	T_9	▨	T_3	▨	▨

FIG. 1. *A task system and its CG-schedule on 3-processors. One possible labeling of the above task system yields labels given by the index for each task, i.e.,* $\alpha(T_j) = j$. $L = (T_{19}, T_{18}, \cdots, T_1)$.

Since all tasks have unit execution times, and are scheduled nonpreemptively, it will be convenient to treat time as being discrete rather than continuous. All schedules start at time 0. For all integers i, *time unit* i is the time interval $(i - 1, i)$. $\lambda(T)$ will denote the time unit during which task T is executed. Note that all processors become available simultaneously at the beginning of every time unit. If we have m processors, then *by convention*, let processor P_k be assigned a task before P_{k+1} is, for all k, $1 \leqq k < m$. Then we can state the following simple lemma from [5].

LEMMA 2.1. *In a CG-schedule, if task T is executed by processor P_1, then all tasks executed along with or after T have a lower label than T, i.e., $\lambda(T) \leqq \lambda(T')$ implies $\alpha(T) > \alpha(T')$.*

Proof. List L is in order of decreasing labels. By convention, processor P_1 is assigned before any of the other processors, thereby getting the unexecuted task with the highest label. □

In order to prove that CG-schedules are optimal on 2 processors, Coffman and Graham [5] divide the schedules into "segments" (called sets X_i in [5]). These segments have the important property that all tasks in one segment must be completed before any task in the next segment can begin. Hence an optimal schedule corresponding to a given CG-schedule will be no shorter than one that arranges tasks in each segment optimally. On 2 processors, the CG-algorithm does arrange tasks in each segment optimally.

We extend the notion of segments to m processors, $m \geqq 2$. By showing that the ratio of the lengths of a segment W in a CG-schedule and the optimal length for W is no larger than $2 - 2/m$, we will show that the ratio of the lengths of CG-schedules and optimal schedules is no larger than $2 - 2/m$. Finally, we will give examples of task systems for which the ratio can be approached arbitrarily closely.

Given a particular CG-schedule S, segments for S will be defined in terms of "blocks". Informally, a block is a portion of the schedule in which no task is executed "ahead of its turn". An idle period is considered as an empty task with label 0.

Given a CG-schedule, blocks will be defined from right to left. We start with task T_2 in Fig. 2 and look for time units during which tasks with lower labels are executed. In time unit 6, T_1 and the empty task have lower labels than T_2, leaving just one task T_8 with a higher label than T_2. T_8 must precede T_2, T_3, T_4 and T_5, for otherwise one of these tasks, and not T_1 would have been executed during time unit 6. We therefore divide the schedule and place T_2, T_3, T_4 and T_5 into block X_0 and repeat the process starting with T_8.

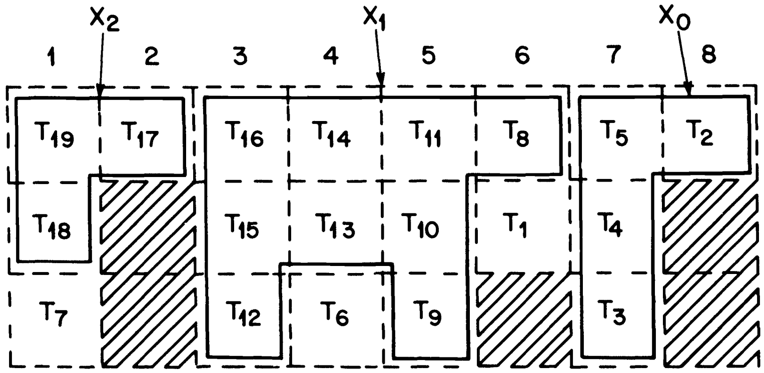


FIG. 2. *An illustration of the definitions of blocks and segments, using the CG-schedule given in Fig.* 1.

T_6 in time unit 4, and the two empty tasks in time unit 2 have lower labels than T_8. We split the schedule between time units 2 and 3 since time unit 2 has just one task, T_{17}, with a higher label than T_8. T_6 is excluded from block X_1 since it has a lower label than T_8. A more precise definition of blocks follows.

DEFINITION. Given a CG-schedule S, we define blocks $X_q, \cdots, X_0$, for some $q \geqq 0$ as follows:

1. Define U_0 to be the last task executed by processor P_1 during the last time unit of S.
2. For $i \geqq 1$, U_i is defined to be the task executed by P_1 in the most recent (maximal) time unit λ, such that for all other tasks T' executed in λ, $\alpha(T') < \alpha(U_{i-1})$.

Block X_{i-1} is the set of all tasks T satisfying $\lambda(U_i) < \lambda(T) \leqq \lambda(U_{i-1})$ and $\alpha(T) \geqq \alpha(U_{i-1})$. Suppose the above steps define U_i for $0 \leqq i \leqq q$, and U_{q+1} does not exist. Then X_q is the set of tasks T with $\lambda(T) \leqq \lambda(U_q)$ and $\alpha(T) \geqq \alpha(U_q)$.

It is often convenient to talk about the time units that tasks from X_i are executed in. For all T in X_i we say $\lambda(T)$ is *in* X_i. Extending our notation, we use $\lambda(X_i)$ to refer to min $\{\lambda(T) | T \in X_i\}$.

A task in S that does not belong to any block will be called an *extra* task.

Note from Fig. 2 that task T_{12} in X_1 does not precede T_5 in block X_0. Since we want segments to be such that all tasks in one segment precede all tasks in the next, we will form segments by merging blocks.

In proving that $2 - 2/m$ is the appropriate bound for segments, we will have to account for at least two tasks during all time units except the last. By definition, block X_{i+1} has just one task U_{i+1} during $\lambda(U_{i+1})$. Thus, when merging X_{i+1} and X_i into the same segment we will have to find an extra task B (like T_6 in Fig. 2) that can be included into the segment. We will first define segments, and then show that the extra tasks mentioned in step 2 can always be found.

DEFINITION. Given a CG-schedule S divided into blocks $X_q, \cdots, X_0$, for $q \geqq 0$, form *segments* $W_0, \cdots, W_r$, for some $r \geqq 0$, as follows (note that unlike blocks segments are defined from left to right):

```
1. $W_0 := X_q$; $i := 0$; $j := q - 1$;
2. while $j \geqq 0$ do
       if for all tasks $T$ in $W_i$ and $T'$ in $X_j$, $T < T'$
           then we have completed segment $W_i$, so
                start a new segment by: $i = i + 1$; $W_i := X_j$;
                $j := j - 1$.
           else let $T$ in $W_i$ be such that for some $T'$ in $X_j$,
                $T < T'$ is false.
                Find task $B \notin W_i$ such that
                $\lambda(T) < \lambda(B) < \lambda(X_j)$, $\alpha(B) > \alpha(U_j)$ and $T < B$.
                $W_i := W_i \cup X_j \cup \{B\}$; $j := j - 1$.
3. If integer $r$ is the final value of $i$ then segments $W_0, \cdots, W_r$ have been
   defined. The length of a segment is given by the number of time units in
   the block contained in the segment.
```

We first show that the above definition is consistent.

LEMMA 2.2. *During some execution of step* 2, *let there be a task* T *in* W_i *such that for some* T' *in* X_j $T < T'$ *is false. Then there exists* $B \notin W_i$ *such that* $\lambda(T) < \lambda(B) < \lambda(X_j)$, $\alpha(B) > \alpha(U_j)$ *and* $T < B$.

Proof. Choose T so that it has no successors in W_i, and T' so that it has no predecessors in X_j. Let I be the set of tasks in X_j that have no predecessors in X_j. Clearly $T' \in I$.

Consider U_{j+1} the last task in X_{j+1}. From the definition of segments $U_{j+1} \in W_i$, and from the definition of blocks U_{j+1} precedes all tasks in X_j. (Recall that all other tasks executed during $\lambda(U_{j+1})$ have lower labels than U_j, and hence lower labels than all tasks in X_j. Therefore U_{j+1} precedes all tasks in X_j.) Since there are no transitive edges in the graph for a task system, the labels of all tasks in I are considered when assigning a label to U_{j+1}.

From the definition of segments all tasks in W_i have higher labels than U_{j+1}. Thus task T has a higher label than U_{j+1}. Since T has no successors in W_i, for

$\alpha(T) > \alpha(U_{j+1})$ to be true, T must either precede all tasks in I or must precede an extra task B as claimed. Since T does not precede T' in·I, B must exist. The arguments used here are essentially those used in [5] to show optimality on two processors. □

Having shown that segments are well defined, we now show that all tasks in one segment precede all tasks in the next.

LEMMA 2.3. *Let W be a segment immediately to the left of segment W'. Then for all tasks T in W and T' in W', T must be completed before T' can start, i.e., $T < T'$.*

Proof. Let X_i be the leftmost block in W'. Then from the definition of segments, all tasks T in W precede all tasks T' in X_i. If X_i is the only block in W', then the proof is complete.

So assume that W' contains blocks $X_i, X_{i-1}, \cdots, X_{i-k}$, for $k \geqq 1$, as well as some extra tasks. From the definition of blocks it follows that for all j, U_j precedes all tasks in X_{j-1}. By transitivity for all T in W, T precedes all tasks in $X_i \cup \cdots \cup X_{i-k}$.

The first extra task B added to W' is preceded by some task in X_i. Any subsequent extra task added to W' is either preceded by a task in some block in W', or by an extra task already in W'. In either case, by transitivity, the extra task is preceded by an element of X_i.

The lemma follows. □

Lemma 2.3 permits us to restrict attention to individual segments in comparing the lengths of optimal and CG-schedules. In the case of 2 processors, a segment of length ω contains $2\omega - 1$ tasks and requires ω time units to finish in any schedule. Therefore the CG-schedule is optimal. For $m > 2$, Fig. 3 shows the CG and optimal schedules for a segment. From this example, we can say that on 3 processors, a CG-schedule may be as much as 33% longer than an optimal schedule. Other examples for $m > 3$ that we will consider in this paper have a quite different structure.

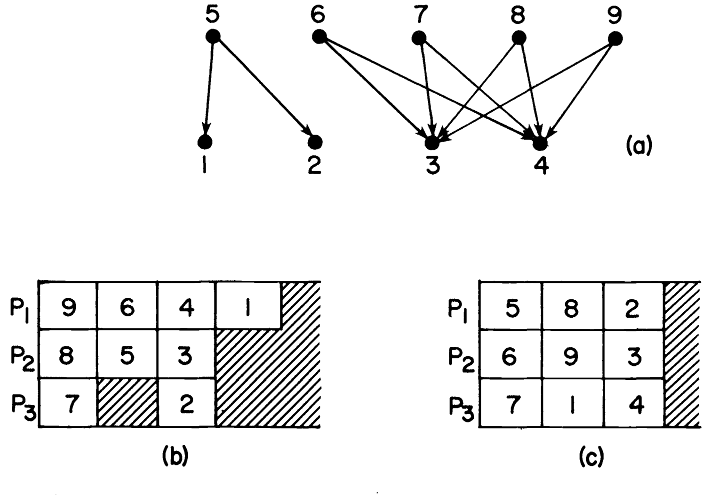


FIG. 3. *A task system* (a), *its CG-schedule* (b) *and optimal schedule* (c) *on* 3 *processors. The integer at each node is the task label.*

For the next two lemmas, we reserve ω and ω_0 to refer to the lengths of a CG-schedule and an optimal schedule, respectively, for some segment. In proving $\omega/\omega_0 \leqq 2 - 2/m$ we will need to examine the structure of a block.

A time unit during which all processors are executing tasks in a block X will be referred to as a *full column* of X. Time units in X that are not full columns will be referred to as *partial columns*. We will determine lower bounds on ω_0 in terms of the number of partial columns in the blocks contained in a segment.

LEMMA 2.4. *Let $X_i, \cdots, X_{i-k}$, $k \geqq 0$ be the blocks in segment W, and let there be p partial columns in these blocks. If X_i starts with a full column, then an optimal schedule for the tasks in W must have length $\omega_0 \geqq p + 1$.*

Proof. Let the partial columns of the segment be as shown in Fig. 4. Let V_1 be the task executed by P_1 during $\lambda(X_i)$. Number the partial columns $1, 2, \cdots, p$, from left to right. For $1 \leqq j < p$, let V_{j+1} be the task executed by P_1 in the time unit following partial column j. Let T be the highest labeled task in the first partial column. We first observe the following two facts:

(i) Since T is in a partial column, any extra tasks in this column must have a label lower than $\alpha(V_2)$. Hence there must be a task T^* in $\lambda(T)$ such that $T^* < V_2$. If T^* is T then we have $T < V_2$. If $T^* \neq T$, then since $\alpha(T) > \alpha(T^*)$, and V_2 is the highest labeled task that T can precede, we must also have $T < V_2$.

(ii) Similarly, V_j, $2 \leqq j < p$, and every task T_j such that $\alpha(T_j) > \alpha(V_j)$ precedes either V_{j+1} or some task R_{j+1} such that $\alpha(R_{j+1}) > \alpha(V_{j+1})$.

Combining (i) and (ii) we conclude that T precedes a chain of at least $p - 1$ tasks, and so does every task with label higher than $\alpha(T)$.

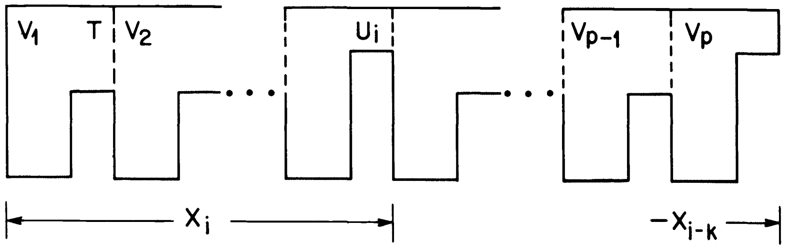


FIG. 4. *Proof of Lemma* 2.4

Let $A_1, \cdots, A_m$ be the tasks executed during the very first time unit of X_i. If there is a task A_j such that $\alpha(A_j) < \alpha(T)$, then there must be a task A_l, $l \neq j$, such that $A_l < T$, for otherwise T would have replaced A_j in the schedule. With $A_l < T$ and the fact that T precedes a chain of $p - 1$ tasks, it follows that the optimal schedule must have length at least $p + 1$.

If for all j, $1 \leqq j \leqq m$, $\alpha(A_j) > \alpha(T)$, then we have at least $m + 1$ tasks that each precede a chain of at least $p - 1$ tasks. Again the optimal schedule length must be at least $p + 1$. □

LEMMA 2.5. *Let ω be the length of segment W and ω_0 the length of an optimal schedule for the tasks in the segment. Then $\omega/\omega_0 \leqq 2 - 2/m$ for $m \geqq 3$.*

Proof. Let $X_i, \cdots, X_{i-k}$, be the blocks contained in segment W and let there be p partial columns and f full columns in these blocks. Clearly $\omega = f + p$.

From the definition of segments, k extra tasks must have been merged into W. For each of the $k + 1$ blocks there is one partial column with one task. Since every other partial column has at least two tasks, the total number of tasks x in W is at least $mf + 2(p - k - 1) + (k + 1) + k = mf + 2p - 1$.

We consider two cases.

Case 1. The leftmost block X_i starts with a full column. Then from Lemma 2.4, $\omega_0 \geqq p + 1$. Since the shortest possible schedule has no idle time, $m\omega_0 \geqq x \geqq mf + 2p - 1$.

Since $\omega = f + p$, we have

$$m\omega = m(f + p) = mf + 2p - 1 + (m - 2)(p + 1) - m + 3.$$

On substitution we get

$$m\omega \leqq m\omega_0 + (m - 2)\omega_0 - (m - 3).$$

Since $m \geqq 3$, it follows that $\omega/\omega_0 \leqq 2 - 2/m$.

Case 2. The leftmost block X_i starts with a partial column. It follows from the proof of Lemma 2.3 that tasks in X_i precede all other tasks in the segment. Any extra tasks in the first time unit of X_i must have a lower label than all other tasks in X_i. Thus the tasks in the partial column of X_i precede all other tasks in X_i, and by transitivity, precede all other tasks in the segment. Thus the optimal schedule for the segment must have at least one idle period. Hence $m\omega_0 \geqq x + 1 \geqq mf + 2p$.

From the proof of Lemma 2.4 it follows that there must be a chain of at least p tasks in W. Therefore $\omega_0 \geqq p$. Since $\omega = f + p$, we have

$$m\omega = m(f + p) = mf + 2p + (m - 2)p.$$

Substituting, we get

$$m\omega \leqq m\omega_0 + (m - 2)\omega_0.$$

Hence, $\omega/\omega_0 \leqq 2 - 2/m$. □

THEOREM 2.1. *Let S be a CG-schedule for a given UET system on m processors. Let ω be the length of S and ω_0 be the length of an optimal schedule for the same task system. Then for $m \geqq 2$,*

$$\omega/\omega_0 \leqq 2 - 2/m.$$

Furthermore, the bound is best possible for m equal to 3 or any even number.[1]

Proof. For $m = 2$, the bound follows from the optimality of CG-schedules [4], [5], [20]. Therefore suppose $m \geqq 3$. Let there be $r + 1$ segments, $W_0, \cdots, W_{r-1}, W_r$, in the CG-schedule S. Lemma 2.3 shows that an optimal schedule can be no shorter than one that arranges each individual segment optimally. Therefore, letting $\omega_0(k)$ be the optimal schedule length of segment W_k, $0 \leqq k \leqq r$, we have

$$\omega_0 \geqq \sum_{k=0}^{r} \omega_0(k).$$

[1] In § 3 it will be shown that $2 - 2/m$ is a best bound for odd values of m also.

From Lemma 2.5 we have $\omega(k)/\omega_0(k) \leqq 2 - 2/m$, where $\omega(k)$ is the length of segment W_k in the CG-schedule. Thus

$$\omega = \sum_{k=0}^{r} \omega(k) \leqq \sum_{k=0}^{r} (2 - 2/m)\omega_0(k) \leqq (2 - 2/m)\omega_0,$$

and the bound is obtained.

That the bound is in fact best possible for the specified values of m can be shown by examples. For $m = 3$, see Fig. 3. For even values of m, we show a task system for 6 processors that realizes the bound asymptotically. The construction of task systems for any other even number of processors can be done following the same pattern.

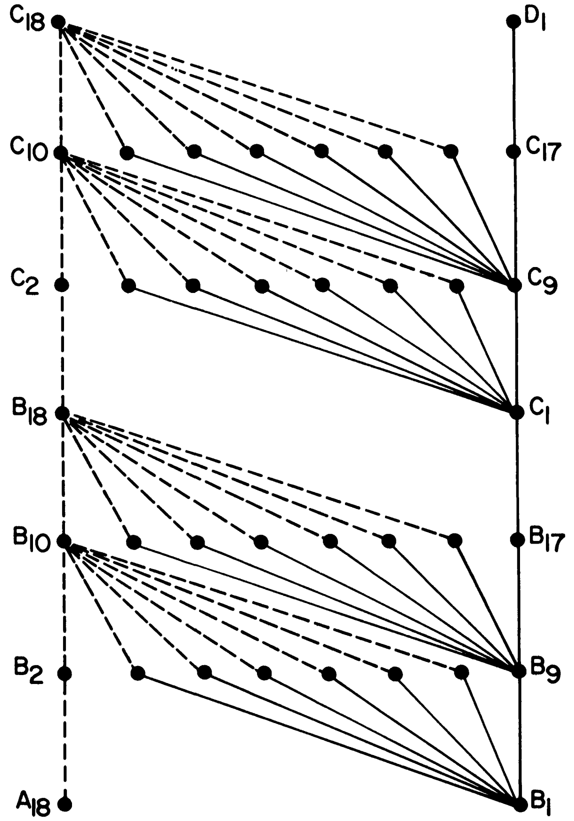
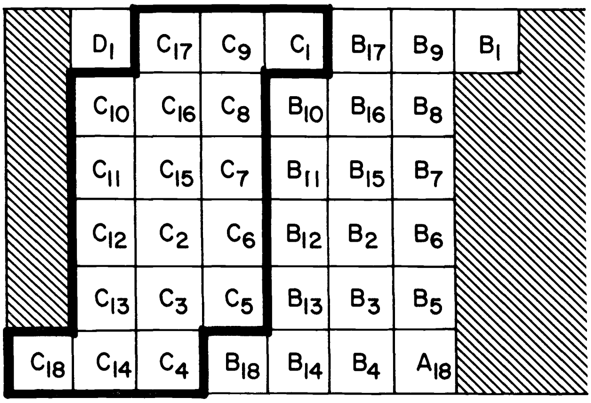


FIG. 5. *By repeating the pattern above, we can construct task systems for which the ratio of the lengths of a CG-schedule and an optimal schedule on 6 processors approaches* $2 - 2/6 = 5/3$. *For even m, a similar construction leads to task systems for which the ratio approaches* $2 - 2/m$. *Some of the edges have been drawn dotted for clarity. All edges are directed downwards.*

Consider the task system in Fig. 5. Tasks $B_1, \cdots, B_{18}$ form a pattern that is repeated by tasks $C_1, \cdots, C_{18}$. If the pattern is repeated k times by adding tasks $D_1, \cdots, D_{18}, E_1, \cdots, E_{18}$ and so on, we will get a task system with $18k + 2$

tasks. If the task system is executed as in Fig. 6, then the length of the schedule is $3k + 2$.



▨	D_1	C_{17}	C_9	C_1	B_{17}	B_9	B_1	▨
▨	C_{10}	C_{16}	C_8	B_{10}	B_{16}	B_8	▨	▨
▨	C_{11}	C_{15}	C_7	B_{11}	B_{15}	B_7	▨	▨
▨	C_{12}	C_2	C_6	B_{12}	B_2	B_6	▨	▨
▨	C_{13}	C_3	C_5	B_{13}	B_3	B_5	▨	▨
C_{18}	C_{14}	C_4	B_{18}	B_{14}	B_4	A_{18}	▨	▨

FIG. 6. *A schedule for the task system in Fig. 5. Bold lines enclose a repeating pattern.*

It is easy to verify that one possible labeling for the task system in Fig. 5 would be to assign labels $1, 2, \cdots$ in the order $A_{18}, B_1, B_2, \cdots, B_{18}, C_1, \cdots, C_{18}, D_1, \cdots$. With $18k + 2$ tasks, the CG-schedule would then be of length $5k + 1$, since it would take 1 unit for each level with 2 tasks and 2 units for each level with $m + 2 = 8$ tasks. The ratio of ω/ω_0 approaches $\frac{5}{3} = 2 - 2/m$ as k increases.

With m processors where m is even, instead of the pattern $B_1, \cdots, B_{18}$, we need $m^2/2$ tasks; 1 task at the lowest level, $m + 2$ tasks at each of the next $m/2 - 1$ levels, and the remaining task at the highest level. It is left to the reader to fill in the details. A formal specification for general m is cumbersome. □

Examination of Fig. 5, which provides the worst case example, shows that the worst case occurs if ties are broken badly in step 2 of the CG-algorithm. The interesting question is whether a "good" rule for breaking ties will improve the worst case performance of the CG-algorithm. It is left to the reader to verify that adding 4 tasks at the lowest levels and some extra edges as in Fig. 7, the CG-algorithm can be forced to label tasks as required in the proof of Theorem 2.1. The 4 tasks added lengthen both the optimal and CG-schedules by a constant, which is insignificant in the limit. The extra edges are clearly not transitive edges.

3. The Muntz-Coffman algorithm. The preemptive counterpart of the algorithm in the last section is given by Muntz and Coffman [16], [17]. A treatment of the algorithm may also be found in [4] and [20].

DEFINITION. Let $(\mathscr{T}, <)$ be a task system containing tasks $T_1, T_2, \cdots, T_n$ having execution times $\tau_1, \tau_2, \cdots, \tau_n$. $\mathscr{C} = \{T_{i_1}, T_{i_2}, \cdots, T_{i_k}\}$ is a *chain* from task T_{i_1} to task T_{i_k} if for all j, $1 \leqq j < k$, $T_{i_j} < T_{i_{j+1}}$. The *length* of $\mathscr{C}$ is given by $\sum_{j=1}^{k} \tau_{i_j}$. The *level* of a task T in $\mathscr{T}$ is the maximum over the lengths of all chains from T to a terminal task.

The idea behind the algorithm in [16], [17] is to preferentially execute tasks at higher levels. Tasks at the same level get the same level of service. In order to

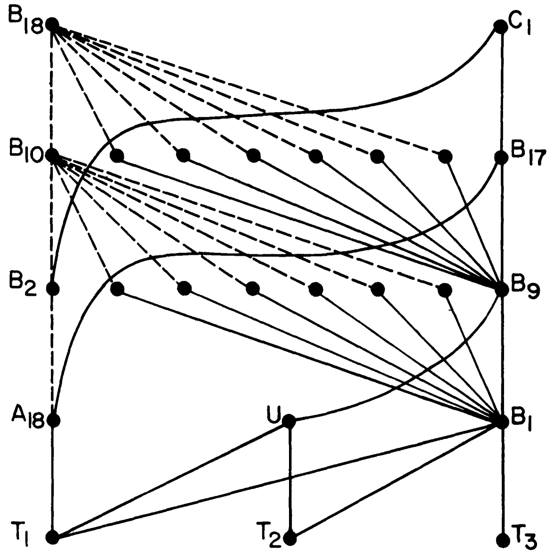


FIG. 7. *Adding edges to force the CG algorithm to assign labels in the order* $B_1, B_2, \cdots, B_{18}$

illustrate diagrammatically how tasks get the same level of service, the notion of *processor sharing* is introduced. If there are more processors than tasks, then each task gets a processor to itself. Otherwise, the tasks will have to share the available resources. Thus each task is assigned β processors, where $0 < \beta \leqq 1$. Instead of talking of the execution time τ of a task T, we sometimes say that T has a *service requirement* of τ processor-time units. An example of a task system scheduled using processor sharing is given in Fig. 8. Task T assigned β processors will execute for τ/β units, where $\tau/\beta \geqq \tau$, since $0 < \beta \leqq 1$.

Since levels of tasks change as execution proceeds, we need the following definition.

DEFINITION. Let S be a schedule for a task system $(\mathcal{T}, <)$. $L_t(T)$, the *level at time t* of task T with respect to schedule S, is the level of task T in the unexecuted portion of the task system.

MUNTZ–COFFMAN (MC-) ALGORITHM. Given a task system $(\mathcal{T}, <)$, this algorithm constructs a processor shared schedule which can easily be converted into a preemptive schedule of the same length. Let s be the time when assignment of processors is made. Initially $s = 0$.

Among the tasks that are ready to execute, assign one processor each to the tasks at the highest level. If there is a tie among b tasks (because they are at the same level) for the last $a (a < b)$ processors, then assign a/b of a processor to each of these b tasks. Continue such an assignment until a time t at which one of the following events occurs.

Event 1. A task is completed at t.

Event 2. There are two tasks T and T' such that $L_s(T) > L_s(T')$ but $L_t(T) = L_t(T')$. That is, the level of T' has caught up with that of T at time t.

In either case set $s = t$ and reassign the processors to the unexecuted portion of the task system.

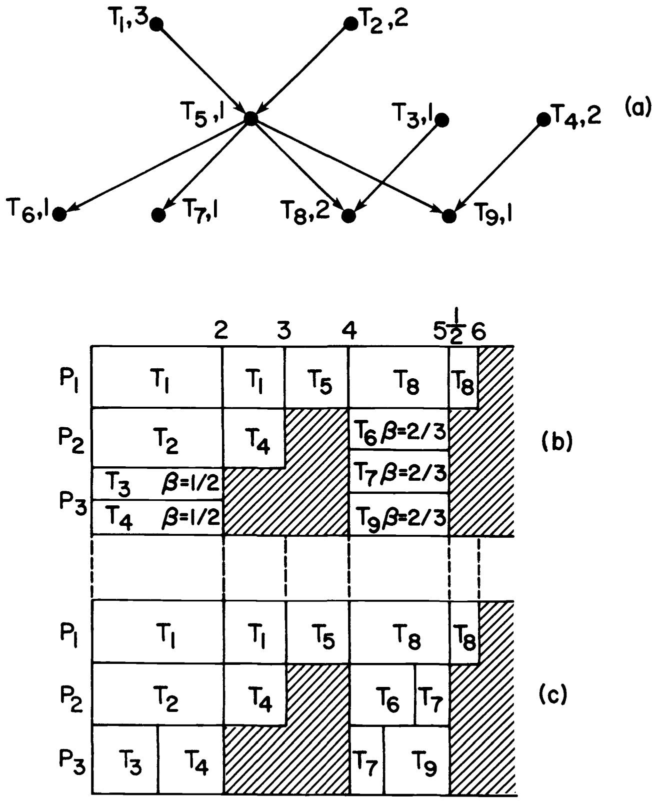


FIG. 8. (a) *A task system. Execution times of tasks appear next to the task names.* (b) *A shared schedule for the task system, and* (c) *the preemptive schedule constructed from the shared schedule.*

The resultant schedule is called an MC-schedule for $(\mathcal{T}, <)$.

Let $0 = t_1 < t_2 < \cdots < t_k$ be the sequence of times at which Event 1 or 2 occurs in the above construction. The following procedure will transform an MC-schedule into a preemptive schedule of the same length. Consider interval (t_i, t_{i+1}) and let $T_{i_1}, \cdots, T_{i_s}$ be the tasks executed in this interval at rates $\beta_{i_1}, \cdots, \beta_{i_s}$, respectively. Now using the quantity $\beta_{i_j}(t_{i+1} - t_i)$ as the execution time for T_{i_j}, we can apply McNaughton's preemptive scheduling algorithm [4], [15], [19], [20] for tasks $T_{i_1}, \cdots, T_{i_s}$ in this time interval and construct a valid preemptive schedule for these tasks. The length of the new schedule is exactly $(t_{i+1} - t_i)$. Do this for all intervals.

Figure 8 gives an example illustrating the above algorithm and the transformation. Intuitively, one can visualize the MC-algorithm as pruning a partial order from top down. Only "leaves" are ready to be pruned. The rate of pruning at any time depends on the number of "leaves" at the top and the machine capacity. If there is machine capacity left, then "leaves" at lower levels will be considered.

In MC-schedules, a task may be split into several pieces, each executed in a different interval of time. Given an MC-schedule S for an n-task system $(\mathcal{T}, <)$, let $U_{u_1}, \cdots, U_{i_{n_i}}$, for some $n_i \geqq 1$, be all the pieces of task T_i in S, with piece U_{i_j} started before $U_{i_{j+1}}$ in S, for all j, $1 \leqq j < n_i$. Also let $\tau_{i_1}, \cdots, \tau_{i_{n_i}}$ be the execution requirements of the pieces $U_{i_1}, \cdots, U_{i_{n_j}}$, respectively. Then obviously we have $\sum_{j=1}^{n_i} \tau_{i_j} = \tau_i$. If we define the precedence relation $<'$ among the pieces with $U_{i_j} <' U_{i_{j+1}}$ for all $1 \leqq j < n_i$, $1 \leqq i \leqq n$, and $U_{i_{n_i}} <' U_{k_1}$ if and only if $T_i < T_k$ for $1 \leqq i, k \leqq n$, it is easy to see that each of these pieces is just like a task and S is an MC-schedule for the task system $(\mathscr{T}', <')$, where $\mathscr{T}' = \{U_{i_j}, 1 \leqq j \leqq n_i, 1 \leqq i \leqq n\}$ and $<'$ is defined as above. Therefore from now on we shall refer to each of these pieces as a task as if S is constructed for the task system formed by the pieces, and use the terms "piece" and "task" interchangeably.

It has been shown in [16], [17], [20] that the MC-algorithm constructs optimal schedules if $(\mathscr{T}, <)$ is tree structured, or if there are only two processors. Now we will show that if ω and ω_0 are the lengths of an MC-schedule and an optimal schedule, respectively, then $\omega/\omega_0 \leqq 2 - 2/m$, and that $2 - 2/m$ is an asymptotic best bound. Since all tasks at the same level are executed at the same rate, it will be easier than the last section to show that $2 - 2/m$ is an upper bound. The hard part is coming up with a task system that achieves the bound. We first show that the bound is true for a special type of MC-schedule and then extend the result to arbitrary MC-schedules.

LEMMA 3.1. *Let S be an MC-schedule of length ω for a task system $(\mathscr{T}, <)$ on $m \geqq 2$ processors. If at all times in the schedule at least two processors are busy, then $\omega/\omega_0 \leqq 2 - 2/m$, where ω_0 is the length of an optimal preemptive schedule for $(\mathscr{T}, >)$.*

Proof. Since an MC-schedule has no unnecessary idle time, there exists a chain of tasks in S such that whenever a processor is idle in $(0, \omega)$ one of the other processors is executing a task in the chain. If L is the total length of time in S when at least one processor is idle, then there must be a chain of length at least L in $(\mathscr{T}, <)$. Hence $\omega_0 \geqq L$. Moreover, if x is the sum of the execution times of all tasks in $(\mathscr{T}, <)$, then $\omega_0 \geqq x/m$.

Let I be the total idle time in S. Since no more than $m - 2$ processors are idle at any time, $I \leqq (m - 2)L$. Therefore $I \leqq (m - 2)\omega_0$.

Since $m\omega = x + I$, on substitution for x and I, we get $m\omega \leqq m\omega_0 + (m - 2)\omega_0$, which yields $\omega/\omega_0 \leqq 2 - 2/m$. □

MC-schedules do not introduce unnecessary idle time, but they may not always have two busy processors. We shall show that any MC-schedule can be divided into segments such that (a) within each segment, except for the last task, there are always at least two busy processors, and (b) tasks in one segment must be completed before those in the next segment can begin.

From the MC-algorithm, if there are not enough tasks at the highest level to occupy all the processors, tasks at the next level will be considered. Therefore some tasks are executed "ahead" of their levels. There are some "bottleneck tasks" T in the sense that all tasks at a higher level than T must be executed before T. In dividing an MC-schedule, we are essentially looking for these bottleneck tasks. The tasks that are executed "ahead" of their levels will be deleted from the MC-schedule, resulting in a reduced schedule. The task system corresponding to the reduced schedule is called the reduced task system. In the reduced schedule, all tasks at a higher level than a bottleneck task T must be completed before T

can start. Similarly all tasks at a lower level than T must be executed after T is completed.

DEFINITION. Let S be an MC-schedule of length ω, and let

$$0 = t_1 < t_2 < \cdots < t_k$$

be the sequence of times at which either Event 1 or 2 occurs. Clearly $\omega = t_k$. Define *segments* $W_r, \cdots, W_1, W_0$ as follows:

(i) If the last time interval (t_{k-1}, t_k) is executing only one task, then call this task U_0; otherwise $U_0 = \varnothing$. U_0 is *in* segment W_0. Set $j = 0$ and $t = t_{k-1}$ (or $t = t_k$ if $U_0 = \varnothing$).

(ii) If $t = 0$, then the entire schedule is divided, so stop. Otherwise let t be t_i for some $i > 1$. For h from i down to 2 do:

(a) Compute $L_{t_h}(T)$ for each task T executed in the interval (t_{h-1}, t_h) with respect to the reduced task system at this point.

(b) Remove all tasks T in interval (t_{h-1}, t_h) that have $L_{t_h}(T) < L_t(U_j)$.

We will show that this step removes all tasks that are executed before U_j but do not precede U_j. Figure 9 illustrates the above step.

(iii) Find the first interval (t_l, t_{l+1}) before t in which only one processor is assigned a task. Define this task as U_{j+1} and add the portion of the schedule between U_{j+1} and U_j to W_j. U_{j+1} is in W_{j+1}.

(iv) Set $j = j + 1$ and $t = t_l$ and go to (ii).

THEOREM 3.1. *Let S be an MC-schedule of length ω for a task system $(\mathscr{T}, <)$ on $m \geqq 2$ processors; then $\omega/\omega_0 \leqq 2 - 2/m$, where ω_0 is the length of an optimal schedule. Moreover, for all values of m there exists a task system that approaches the bound arbitrarily closely.*

Proof. As in the definition above, let $W_r, W_{r-1}, \cdots, W_0$ be the segments of S. It is easy to see that the total length of the segments is ω. If in the definition of segment W_j a task T executed in the interval (t_{h-1}, t_h) is deleted, then from step (ii) of the definition, $L_{t_h}(T) < L_t(U_j) = L_{t_h}(U_j)$. Then from the MC-algorithm there exists V, a predecessor of U_j, executing concurrently with T such that $L_{t_{h-1}}(V) > L_{t_{h-1}}(T)$. It follows that V is assigned a full processor in the interval (t_{h-1}, t_h). Therefore deleting tasks like T does not change the length of S. The reader can verify that $W_r, W_{r-1}, \cdots, W_0$ taken together form an MC-schedule of length ω for the reduced task system.

Consider a task V in segment W_j for some j. By definition, segment W_{j+1} ends with task U_{j+1}, and while U_{j+1} is being executed, all other processors are idle in the reduced schedule. Since the reduced schedule is an MC-schedule, it must be true that U_{j+1} precedes V, for otherwise V would have been scheduled concurrently with U_{j+1}.

To show that $T < U_{j+1}$ for all $T \neq U_{j+1}$ in W_{j+1}, we assume the contrary. That is, there exists a task T in W_{j+1} such that $T < U_{j+1}$ is false. Choose T so that T is the last such task to finish in W_{j+1}. This choice ensures that T has no successors in W_{j+1}. That is, T can only have immediate successors in W_j or some later segment, which implies that $L_{t_h}(T) < L_t(U_{j+1})$, where (t_{h-1}, t_h) is the interval that T is in and t is the time when U_{j+1} starts execution. However, $L_{t_h}(T) < L_t(U_{j+1})$ contradicts the fact that T is not deleted. Consequently we have $T < U_{j+1} < V$ for all $T \neq U_{j+1}$ in W_{j+1} and all V in W_j.

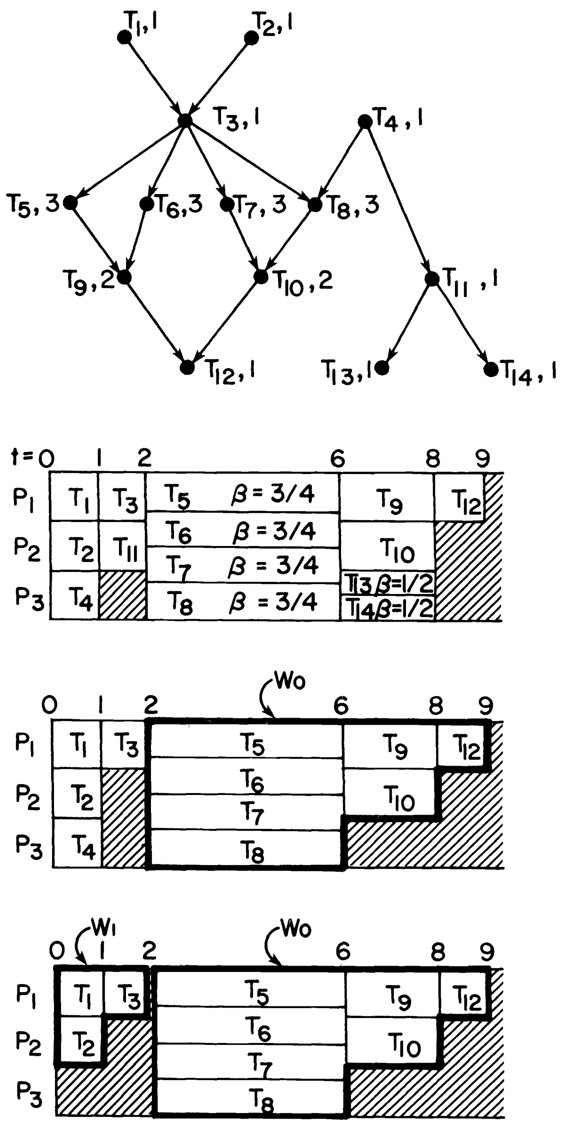


FIG. 9. *The top schedule is the original MC-schedule for the task system. The middle schedule is the reduced schedule after removal of tasks when compared with $L_8(T_{12})$. Finally, there is the reduced schedule after removal of tasks when compared with $L_1(T_3)$.*

It follows that as in Theorem 2.1, we need only consider one segment W_j at a time. Let U_j be the last task executing in W_j. As noted above, $T < U_j$ for all other tasks T in W_j. Clearly the optimal strategy can do no better than executing all other tasks optimally and then executing U_j. Thus consider all tasks in W_j except U_j. By peeling off U_j, we are left with a portion of the schedule in which at all times at least two processors are busy. From Lemma 3.1, for this part of the segment $\omega'/\omega'_0 \leqq 2 - 2/m$, where ω' is the length of this part of the segment and ω'_0 the optimal length for it. Clearly $\omega'/\omega'_0 \geqq (\omega' + c)/(\omega'_0 + c)$ when $\omega'/\omega'_0 \geqq 1$ and $c \geqq 0$. Thus taking task U_j into account cannot change the bound.

As in Theorem 2.1 if the bound for each individual segment is $2 - 2/m$, then $2 - 2/m$ is an upper bound for the entire schedule.

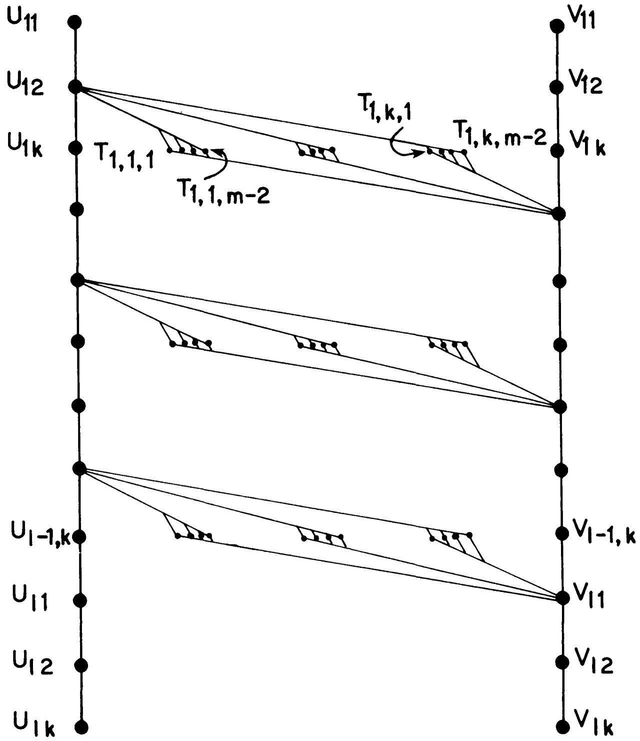


FIG. 10. *The task system for the proof of Theorem* 3.1. *All tasks have unit execution time. The figure has been drawn for* $k = 3$, $l = 4$ *and* $m = 6$. *As* $k \to \infty$ *and* $l \to \infty$, $\omega/\omega_0 \to 2 - 2/m$, *where* ω *and* ω_0 *are the lengths of the MC- and optimal schedules for the task system.*

To see that $2 - 2/m$ is (asymptotically) achievable, consider the UET system in Fig. 10. The task system has l repetitions of a pattern that consists of k levels of tasks. The figure has been drawn with $k = 3$, $l = 4$ and $m = 6$. We can have a schedule that executes $U_{11}, U_{12}, \cdots, U_{1k}$ in the first k time units. In the next time unit, $V_{11}, T_{1,1,1}, \cdots, T_{1,1,m-2}$ and U_{21} are executed. After that, V_{12}, $T_{1,2,1}, \cdots, T_{1,2,m-2}$ and U_{22} are executed, and so on, until $V_{l-1,k}$, $T_{l-1,k,1}, \cdots$, $T_{l-1,k,m-2}$ and U_{lk} are executed. Then k units are required to finish the schedule. The optimal schedule of length ω_0 is therefore no longer than

$$2k + (l - 1)k = k(l + 1).$$

The MC-schedule, on the other hand, is of length

$$\omega = (l - 1)(k - 1 + (k(m - 2) + 2)/m) + k.$$

The ratio ω/ω_0 is then given by

$$\frac{\omega}{\omega_0} \geqq \frac{(l-1)(k-1+(k(m-2)+2)/m)+k}{k(l+1)} = c.$$

As k tends to infinity, we get

$$\lim_{k\to\infty} c = \frac{(l-1)(1+(m-2)/m)+1}{(l+1)}.$$

Taking limits again, we find

$$\lim_{l\to\infty}\lim_{k\to\infty} c = 1 + \frac{m-2}{m} = 2 - \frac{2}{m}. \qquad \square$$

In Fig. 10, as k increases, the contribution of U_{1k} and V_{1k} to the schedule length becomes insignificant. Then as l increases, the k extra time units in the beginning and in the end of the schedule become insignificant. The example in Fig. 10 can be modified to yield an example showing that $2 - 2/m$ is an asymptotic best bound for all values of m for the CG-algorithm of § 2.

4. Discussion. As reviewed in [21], a large number of scheduling problems are NP-complete. It is therefore desirable to devise heuristics for these problems. We have taken two algorithms, one preemptive and the other nonpreemptive, which yield optimal schedules on two processors. We have shown that on m processors, $m \geqq 2$, the two algorithms produce schedules that are always within a factor of $2 - 2/m$ of the optimal. We then gave an example of task systems for which the ratio approached $2 - 2/m$ in the limit.

Acknowledgments. The authors acknowledge valuable suggestions from the referees and helpful discussions with Ned Horvath and Chandra Kintala. The example with the $\frac{4}{3}$ bound in Fig. 3 was also communicated to the authors by J. L. Baer. Deepak K. Goyal pointed out an error in an earlier version of this paper. The definition of segments in § 2 is adapted from the approach used in [23].

REFERENCES

[1] A. V. Aho, M. R. Garey and J. D. Ullman, *The transitive reduction of a directed graph*, this Journal, 1 (1972), pp. 131–137.

[2] N. F. Chen and C. L. Liu, personal communication, December, 1974.

[3] ———, *On a class of scheduling algorithms for multiprocessor computing systems*, Parallel Processing, Lecture Notes in Computer Science, vol. 24, Springer-Verlag, New York, 1975, pp. 1–16.

[4] E. G. Coffman, Jr. and P. J. Denning, *Operating Systems Theory*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[5] E. G. Coffman, Jr. and R. L. Graham, *Optimal scheduling for two processor systems*, Acta Informatica, 1 (1972), pp. 200–213.

[6] M. Fujii, T. Kasami and K. Ninomiya, *Optimal sequencing of two equivalent processors*, SIAM J. Appl. Math., 17 (1969), pp. 784–789; *Erratum*, 20 (1971), p. 141.

[7] M. R. Garey and D. S. Johnson, *Complexity results for multiprocessor scheduling under resource constraints*, this Journal, 4 (1975), pp. 397–411.

[8] R. L. Graham, *Bounds on multiprocessing timing anomalies*, SIAM J. Appl. Math., 17 (1969), pp. 416–429.

[9] ———, *Bounds for certain multiprocessing anomalies*, Bell System Tech. J., 45 (1966), pp. 1563–1581.

[10] ———, *Bounds on multiprocessing anomalies and related packing algorithms*, AFIPS Conf. Proc., 40 (1972), pp. 205–217.

[11] T. C. Hu, *Parallel sequencing and assembly line problems*, Operations Res., 9 (1961), pp. 841–848.

[12] R. M. Karp, *Reducibility among combinatorial problems*, Complexity of Computer Computation, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.

[13] M. T. Kaufman, *Anomalies in scheduling unit-time tasks*, Stanford Electronics Lab. Tech. Rep. 34, Stanford Univ., Stanford, Calif., 1972.

[14] ———, *An almost-optimal algorithm for the assembly-line scheduling problem*, IEEE Trans. Computers, C-23 (1974), pp. 1169–1174.

[15] R. McNaughton, *Scheduling with deadlines and loss functions*, Management Sci., 6 (1959), pp. 1–12.

[16] R. R. Muntz and E. G. Coffman, Jr., *Preemptive scheduling of real time tasks on multiprocessor systems*, J. Assoc. Comput. Mach., 17 (1970), pp. 324–338.

[17] ———, *Optimal preemptive scheduling on two-processor systems*, IEEE Trans. Computers, C-18 (1969), pp. 1014–1020.

[18] Y. Muraoka, *Parallelism, exposure and exploitation in programs*, Ph.D. thesis, Univ. of Illinois, Urbana, 1971.

[19] M. H. Rothkopf, *Scheduling independent tasks on parallel processors*, Management Sci., 12 (1966), pp. 437–447.

[20] R. Sethi, *Algorithms for minimal length schedules*, Computer and Job-shop Scheduling Theory, E. G. Coffman, Jr., ed., John Wiley, New York, 1976, pp. 51–99.

[21] J. D. Ullman, *Complexity of sequencing problems*, Computer and Job-shop Scheduling Theory, E. G. Coffman, Jr., ed., John Wiley, New York, 1976, pp. 139–164.

[22] ———, *NP-complete scheduling problems*, J. Comput. Systems Sci., 10 (1975), pp. 384–393.

[23] D. K. Goyal, Scheduling equal execution time tasks under unit resource restriction, unpublished manuscript, 1976.

SIAM J. COMPUT.
Vol. 6, No. 3, September 1977

FINDING A MAXIMUM INDEPENDENT SET*

ROBERT ENDRE TARJAN† AND ANTHONY E. TROJANOWSKI‡

Abstract. We present an algorithm which finds a maximum independent set in an n-vertex graph in $O(2^{n/3})$ time. The algorithm can thus handle graphs roughly three times as large as could be analyzed using a naive algorithm.

Key words. algorithm, clique, computational complexity, graph, maximum independent set, NP-complete problem

1. Introduction. A *graph* $G=(V, E)$ is an ordered pair consisting of a finite set V of *vertices* and a set of unordered pairs (v, w) of distinct vertices, called *edges*. Two vertices v, w are *adjacent* if $(v, w) \in E$. A set S of vertices is *independent* (or *internally stable*) if $(v, w) \notin E$ for all $v, w \in S$. A set S of vertices is a *clique* if $(v, w) \in E$ for all pairs of distinct vertices $v, w \in S$. The complement of a graph $G=(V, E)$ is the graph $\bar{G}=(V, \bar{E})$ where $\bar{E}=\{(v, w) | v, w \in V, v \neq w$, and $(v, w) \notin E\}$. Clearly $S \subseteq V$ is an independent set of G if and only if S is a clique of $\bar{G}$. (Note: some authors require that a clique be a *maximal* set of pairwise adjacent vertices; we do not.)

A *path* from v_1 to v_k in a graph $G=(V, E)$ is a sequence of vertices $v_1, v_2, \cdots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leqq i < k$. A set of vertices S is *connected* if, for all $v, w \in S$, there is a path from v to w containing only vertices in S. The vertices of a graph G can be partitioned into maximal connected subsets, called the *connected components* of G. If $G=(V, E)$ is a graph and S is a set of vertices, the graph $G(S)=(S, E(S))$, where $E(S)=\{(v, w) \in E | v, w \in S\}$, is called the *subgraph* of G *induced by* the vertex set S.

We consider the problem of finding a maximum-size independent set in a given graph $G=(V, E)$; or, equivalently, finding a maximum-size clique in a given graph. This problem has been studied extensively, but no polynomial-time algorithm is known. In fact, the maximum independent set problem is NP-complete [4], [7], and thus is unlikely to have a polynomial-time algorithm. Our goal is to provide an algorithm, which, though not polynomial, is significantly faster in the worst case than the obvious enumeration algorithm or any other algorithm known to us.

Let $n=|V|$. The number of subsets of V is 2^n. By listing each possible subset of V and testing it for independence, one can find a maximum clique in $O(p(n)2^n)$ time, where $p(n)$ is some polynomial. Other algorithms have been proposed [2], [9], [10], but for none except the one in [10] has a worst-case time bound better than $O(2^n)$ been proved.

We extend the algorithm of [10] to provide an $O(2^{n/3})$-time algorithm. The algorithm is recursive and depends upon a somewhat complicated case analysis.

* Received by the editors April 26, 1976.

† Computer Science Department, Stanford University, Stanford, California 94305. This research was supported in part by National Science Foundation Grant MCS 75-22870 and by the Office of Naval Research.

‡ Computer Science Department, University of Illinois, Urbana, Illinois 61801. This research was supported in part by National Science Foundation Grant DCR 74-02774.

Though the algorithm is tedious to state in detail, it would be straightforward to program, and we suspect that it would perform well in practice. Nevertheless, its main interest seems to be theoretical; its existence shows that at least one *NP*-complete problem can be solved in a time bound significantly better than that of the obvious enumeration algorithm. For a similar algorithm to solve another *NP*-complete problem,, see [5].

The maximum number of independent sets maximal with respect to the subset relation in a graph of n vertices is $3^{n/3}$. One could find a maximum-size independent set by enumerating all maximal independent sets (using an algorithm such as in [1], [3], [6], [8]) and choosing the largest. However, the algorithm to be proposed is substantially better than even this method, in the worst case.

The algorithm uses a recursive backtracking scheme. Its starting point is the following observation. Let $v \in V$. Let $A(v)$ be the set of vertices adjacent to v. Then any maximum independent set either contains v or does not contain v. Thus any maximum independent set of G is either $\{v\}$ combined with a maximum independent set in $G(V-\{v\}-A(v))$, or it is a maximum independent set in $G(V-\{v\})$.

We extend this idea. For any $S \subseteq V$, let $A(S) = \bigcup_{v \in S} A(v)$. If $S \subseteq V$, then any maximum independent set I in G consists of an independent set $I \cap S$ in $G(S)$ and a maximum independent set $I-S$ in $G(V-S-A(I))$. Our algorithm selects a subset $S \subseteq V$, finds each independent set J in $G(S)$, and, for each such J, recursively finds a maximum independent set in $G(V-S-A(J))$.

We improve this method further by introducing the concept of *dominance*. If $S \subseteq V$ and I, J are independent in $G(S)$, we say *I dominates J* if, for any $J' \subseteq V-S$ such that $J \cup J'$ is independent, there is a set $I' \subseteq V-S$ such that $I \cup I'$ is independent and $|I \cup I'| \geqq |J \cup J'|$. For any such dominated set J, we need not solve a subproblem, since we get an independent set at least as large by solving a subproblem for I.

Dominance is important because in certain cases it can be confirmed quickly. We give two examples which are used extensively in the algorithm. Let $v \in V$. Let $S = \{v\} \cup A(v)$. If $w \in A(v)$, then $\{v\}$ dominates $\{w\}$ in S, since if $I \subseteq V-S$ and $I \cup \{w\}$ is independent, then $I \cup \{v\}$ is independent. Similarly, $\{v\}$ dominates $\emptyset$ in S.

Let $S \subseteq V$. Let I and $J = I \cup \{v\}$ be independent in $G(S)$. Suppose $(V-S) \cap (A(v)-A(I)) = \{w_1, w_2\}$. In $S \cup \{w_1, w_2\}$, J dominates both $I \cup \{w_1\}$ and $I \cup \{w_2\}$. We distinguish three possibilities.

(i) $(w_1, w_2) \in E$ or $I \cap A(\{w_1, w_2\}) \neq \emptyset$. Then J dominates I in S: if $I' \subseteq V-S$ and $I' \cup I$ is independent, then $|I' \cap \{w_1, w_2\}| \leqq 1$. Thus $J' = I' - \{w_1, w_2\}$ satisfies $|J' \cup J| \geqq |I' \cup I|$ and $J' \cup J$ is independent.

(ii) $(w_1, w_2) \notin E$, $I \cap A(\{w_1, w_2\}) = \emptyset$, and $|(V-S-A(J)) \cap A(\{w_1, w_2\})| \leqq 1$. Then I dominates J in S (and $I \cup \{w_1, w_2\}$ dominates J in $S \cup \{w_1, w_2\}$): If $J' \subseteq V-S$ and $J' \cup J$ is independent, then $I' = (J' \cup \{w_1, w_2\}) - A(\{w_1, w_2\})$ satisfies $|I' \cup I| \geqq |J' \cup J|$ and $I' \cup I$ is independent.

(iii) $(w_1, w_2) \notin E$, $I \cap A(\{w_1, w_2\}) = \emptyset$, and $|(V-S-A(J)) \cap A(\{w_1, w_2\})| \geqq 2$. In this case we need further information to determine whether I dominates J or vice-versa.

In summary, the algorithm selects a set $S \subseteq V$, determines a set of dominating independent sets in S using the two observations on the previous page, and recursively solves one subproblem for each dominating set.

2. The algorithm. A detailed specification of the algorithm appears below. A call *maxset*(S) will return an integer which is the size of a maximum independent set in $G(S)$; the graph $G=(V, E)$ is assumed to be a global variable. The statement of the algorithm consists of a sequence of cases and subcases. The *first* case which applies is used to define the value of *maxset*(S). Thus, inside a given case, the hypotheses of all previous cases can be assumed to be false. We use this convention to avoid a confusing nesting of **if-then-else** statements. Throughout the procedure $d(v)$ denotes the degree of v in $G(S)$. It is easy to modify the algorithm so that it returns a maximum independent set as well as the size of such a set.

```
procedure maxset(S);
begin
   0: $S$ is not connected in $G(S)$.
      Let $S_1, S_2, \cdots, S_k$ be the connected components of $G(S)$. Note that
      every maximum independent set consists of a union of maximum
      independent sets, one from each connected component. Let maxset =
      $\sum_{i=1}^{k}$ maxset$(S_i)$.
not 0: $S$ is connected.
      Let $v$ be a vertex of minimum degree in $G(S)$. One of the following six
      cases applies.
   1: $d(v)=1$.
      Let $A(v) \cap S=\{w\}$.
      Let maxset = $1+$maxset$(S-\{v, w\})$.
   2: $d(v)=2$.
      2.1: $d(w)=2$ for all $w \in V$.
           Note that the vertices of $S$ form a cycle in $G(S)$.
           Let maxset = $\lfloor |S|/2 \rfloor$.
      2.2: There exist $u$, $w_1$ such that $d(u)=2$, $d(w_1) \geqq 3$, and $(u, w_1) \in E$.
           Let $A(u) \cap S=\{w_1, w_2\}$.
           2.2.1: $(w_1, w_2) \in E$.
                  Let maxset = $1+$maxset$(S-\{u, w_1, w_2\})$.
           2.2.2: $(w_1, w_2) \notin E$.
                  Let maxset = max $\{1+$maxset$(S-\{u, w_1, w_2\})$,
                                     $2+$maxset$(S-A(w_1)-A(w_2))\}$.
   3: $d(v)=3$.
      Let $A(v) \cap S=\{w_1, w_2, w_3\}$.
      3.1: $(w_1, w_2), (w_1, w_3), (w_2, w_3) \in E$.
           Let maxset = $1+$maxset$(S-\{v, w_1, w_2, w_3\})$.
      3.2: $(w_1, w_2), (w_1, w_3) \in E$ (or any symmetric case).
           Let maxset = max $\{1+$maxset$(S-\{v, w_1, w_2, w_3\})$,
                             $2+$maxset$(S-A(w_2)-A(w_3))\}$.
      3.3: $(w_1, w_2) \in E$ (or any symmetric case).
           For $i=1, 2, 3$, let $\bar{A}_i=S-\{w_1, w_2, w_3\}-A(w_i)$.
```

Note that $|\bar{A}_1|, |\bar{A}_2| \leqq |S|-5$, $|\bar{A}_3| \leqq |S|-6$.

3.3.1: $|\bar{A}_1 \cap \bar{A}_3| \leqq |\bar{A}_2 \cap \bar{A}_3| = |S|-6$ (or the symmetric case).

Note that $\bar{A}_2 \cap \bar{A}_3 = \bar{A}_3$. Thus $\{w_2, w_3\}$ dominates $\{w_1, w_3\}$.

Let *maxset* = max $\{1+$*maxset*$(S-\{v, w_1, w_2, w_3\})$,
$2+$*maxset*$(\bar{A}_3)\}$.

3.3.2: $|\bar{A}_1 \cap \bar{A}_3|, |\bar{A}_2 \cap \bar{A}_3| \leqq |S|-7$.

Let *maxset* = max $\{1+$*maxset*$(S-\{v, w_1, w_2, w_3\})$,
$2+$*maxset*$(\bar{A}_1 \cap \bar{A}_3)$,
$2+$*maxset*$(\bar{A}_2 \cap \bar{A}_3)\}$.

3.4: $(w_i, w_j) \notin E$ for $i, j \in \{1, 2, 3\}$.

For $i = 1, 2, 3$, let $\bar{A}_i = S - \{w_1, w_2, w_3\} - A(w_i)$.

Note that $|\bar{A}_i| \leqq |S|-6$ for $i = 1, 2, 3$.

3.4.1: $|\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3| \geqq |S|-7$.

Set $\{w_1, w_2, w_3\}$ dominates $\{w_i, w_j\}$ for $i, j \in \{1, 2, 3\}$.

Let *maxset* = max $\{1+$*maxset*$(S-\{v, w_1, w_2, w_3\})$,
$3+$*maxset*$(\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3)\}$.

3.4.2: $|\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3| = |S|-8$ or $|S|-9$.

If, for some i, j, $|\bar{A}_i \cap \bar{A}_j| \leqq |\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3|+1$, then $\{v_i, w_j\}$ is dominated by $\{w_1, w_2, w_3\}$.

For distinct i, j, k,

$$|\bar{A}_i| \geqq |\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3| + |\bar{A}_i \cap \bar{A}_j - (\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3)| + |\bar{A}_i \cap \bar{A}_k - (\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3)|,$$

$|\bar{A}_i| \leqq |S|-6$, and $|\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3| \geqq |S|-9$.

Thus $|\bar{A}_i \cap \bar{A}_j| \geqq |\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3|+2$ for only one possible pair $i \neq j$. Let 1, 2 be the pair (if any).

3.4.2.1: $|\bar{A}_i \cap \bar{A}_j| \leqq |\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3|+1$ for all $i \neq j$.

Let *maxset* = max $\{1+$*maxset*$(S-\{v, w_1, w_2, w_3\})$,
$3+$*maxset*$(\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3)\}$.

3.4.2.2: $|\bar{A}_1 \cap \bar{A}_2| \geqq |\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3|+2$ (or any symmetric case).

Let *maxset* = max $\{1+$*maxset*$(S-\{v, w_1, w_2, w_3\})$,
$2+$*maxset*$(\bar{A}_1 \cap \bar{A}_2)$,
$3+$*maxset*$(\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3)\}$.

3.4.3; $|\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3| \leqq |S|-10$.

3.4.3.1: $|\bar{A}_i \cap \bar{A}_j| \leqq |\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3|+1$ for all $i \neq j$.

Same as 3.4.2.1.

3.4.3.2: $|\bar{A}_1 \cap \bar{A}_2| \geqq |\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3|+2$ (or any symmetric case).

Same as 3.4.2.2.

3.4.3.3: $|\bar{A}_1 \cap \bar{A}_2|, |\bar{A}_1 \cap \bar{A}_3| \geqq |\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3|+2$ (or any symmetric case).

Let *maxset* = max $\{1+$*maxset*$(S-\{v, w_1, w_2, w_3\})$,
$2+$*maxset*$(\bar{A}_1 \cap \bar{A}_2)$,
$2+$*maxset*$(\bar{A}_1 \cap \bar{A}_3)$,
$3+$*maxset*$(\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3)\}$.

3.4.3.4: $|\bar{A}_1 \cap \bar{A}_2|, |\bar{A}_1 \cap \bar{A}_3|, |\bar{A}_2 \cap \bar{A}_3| \geqq |\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3|+2$.

For $i = 1, 2, 3$, let $u_{i1}, u_{i2} \in (\bar{A}_j \cap \bar{A}_k) - \bar{A}_i$ $(j, k \neq i)$.

3.4.3.4.1: $|\bar{A}_j \cap \bar{A}_k| = |\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3| + 2$ and $(u_{i1}, u_{i2}) \in E$ for some distinct i, j, k.
Then $\{w_1, w_2, w_3\}$ dominates $\{w_j, w_k\}$.
Same as 3.4.3.3.

3.4.3.4.2: $|\bar{A}_j \cap \bar{A}_k| = |\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3| + 2$ and $(u_{i1}, u_{i2}) \notin E$ for all distinct i, j, k. Let

$$\begin{aligned} maxset = \max\{&1 + maxset(S - \{v, w_1, w_2, w_3\}),\\ &4 + maxset(\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3 - A(u_{11}) - A(u_{12})),\\ &4 + maxset(\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3 - A(u_{21}) - A(u_{22})),\\ &4 + maxset(\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3 - A(u_{31}) - A(u_{32})),\\ &3 + maxset(\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3)\}. \end{aligned}$$

3.4.3.4.3: $|\bar{A}_1 \cap \bar{A}_2|, |\bar{A}_1 \cap \bar{A}_3| = |\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3| + 2$ (or any symmetric case).
Let

$$\begin{aligned} maxset = \max\{&1 + maxset(S - \{v, w_1, w_2, w_3\}),\\ &4 + maxset(\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3 - A(u_{31}) - A(u_{32})),\\ &4 + maxset(\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3 - A(u_{21}) - A(u_{22})),\\ &2 + maxset(\bar{A}_2 \cap \bar{A}_3),\\ &3 + maxset(\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3)\}. \end{aligned}$$

3.4.3.4.4: $|\bar{A}_1 \cap \bar{A}_2| = |\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3| + 2$ (or any symmetric case).
Let

$$\begin{aligned} maxset = \max\{&1 + maxset(S - \{v, w_1, w_2, w_3\}),\\ &4 + maxset(\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3 - A(u_{31}) - A(u_{32})),\\ &2 + maxset(\bar{A}_1 \cap \bar{A}_3),\\ &2 + maxset(\bar{A}_2 \cap \bar{A}_3),\\ &3 + maxset(\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3)\}. \end{aligned}$$

3.4.3.4.5: $|\bar{A}_i \cap \bar{A}_j| \geqq |\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3| + 3$ for $i \neq j$.
Let

$$\begin{aligned} maxset = \max\{&1 + maxset(S - \{v, w_1, w_2, w_3\}),\\ &2 + maxset(\bar{A}_1 \cap \bar{A}_2),\\ &2 + maxset(\bar{A}_1 \cap \bar{A}_3),\\ &2 + maxset(\bar{A}_2 \cap \bar{A}_3),\\ &3 + maxset(\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3)\}. \end{aligned}$$

4: $d(v) = 4$.

4.1: $d(w) = 4$ for all vertices w.

4.1.1: There are vertices v, w such that $(v, w) \notin E$ and $|A(v) \cap A(w) \cap S| \geqq 2$.

4.1.1.1: $|A(v) \cap A(w) \cap S| \geqq 3$.
Then $\{v, w\}$ dominates both $\{v\}$ and $\{w\}$ in $\{v, w\}$.
Let

$$\begin{aligned} maxset = \max\{&2 + maxset(S - \{v, w\} - A(v) - A(w)),\\ &maxset(S - \{v, w\})\}. \end{aligned}$$

4.1.1.2: $|A(v) \cap A(w) \cap S| = 2$.
Let $x, y \in (A(v) - A(w)) \cap S$, q,
$q, r \in (A(w) - A(v)) \cap S$.
Let $\bar{A}(z) = S - \{z\} - A(z)$ for $z \in S$.

4.1.1.2.1: $(x, y), (q, r) \in E$.
Then $\{v, w\}$ dominates both $\{v\}$ and $\{w\}$ in $\{v, w\}$.
Let
$$maxset = \max\{2 + maxset(\bar{A}(v) \cap \bar{A}(w)), maxset(S - \{v, w\})\}.$$

4.1.1.2.2: $(x, y) \in E, (q, r) \notin E$ (or symmetric case).
Let
$$maxset = \max\{2 + maxset(\bar{A}(v) \cap \bar{A}(w)),$$
$$3 + maxset(\bar{A}(v) \cap \bar{A}(w) \cap \bar{A}(q) \cap \bar{A}(r)),$$
$$maxset(S - \{v, w\})\}.$$

4.1.1.2.3: $(x, y), (q, r) \notin E$,
$|\bar{A}(v) \cap \bar{A}(w) \cap \bar{A}(q) \cap \bar{A}(r)| \geqq |S| - 9$
(or symmetric case).
Let
$$maxset = \max\{3 + maxset(\bar{A}(v) \cap \bar{A}(w) \cap \bar{A}(x) \cap \bar{A}(y)),$$
$$3 + maxset(\bar{A}(v) \cap \bar{A}(w) \cap \bar{A}(q) \cap \bar{A}(r)),$$
$$maxset(S - \{v, w\})\}.$$

4.1.1.2.4: $(x, y), (q, r) \notin E$,
$|\bar{A}(v) \cap \bar{A}(w) \cap \bar{A}(q) \cap \bar{A}(r)|$,
$|\bar{A}(v) \cap \bar{A}(w) \cap \bar{A}(x) \cap \bar{A}(y)| \leqq |S| - 10$.
Let
$$maxset = \max\{2 + maxset(\bar{A}(v) \cap \bar{A}(w)),$$
$$3 + maxset(\bar{A}(v) \cap \bar{A}(w) \cap \bar{A}(x) \cap \bar{A}(y)),$$
$$3 + maxset(\bar{A}(v) \cap \bar{A}(w) \cap \bar{A}(q) \cap \bar{A}(r)),$$
$$maxset(S - \{v, w\})\}.$$

4.1.2: If $(v, w) \notin E$, then $|A(v) \cap A(w) \cap S| \leqq 1$.
Let $A(v) \cap S = \{w_1, w_2, w_3, w_4\}$. For $i = 1, 2, 3, 4$, let $\bar{A}_i = S - A(v) - A(w_i)$.Then, for $i \neq j$, $\bar{A}_i \cap \bar{A}_j = \varnothing$. Also, if $(w_i, w_j), (w_i, w_k) \in E$, then $(w_j, w_k) \in E$.*

4.1.2.1: $(w_1, w_i) \in E$ for $i = 2, 3, 4$ (or any symmetric case).
It follows from * above that the problem graph is a complete graph of five vertices. Let $maxset = 1$.

4.1.2.2: $(w_1, w_2), (w_1, w_3), (w_2, w_3) \in E$,
$(w_1, w_4), (w_2, w_4), (w_3, w_4) \notin E$ (or any symmetric case).
Let $maxset = \max\{1 + maxset(S - \{v\} - A(v)),$
$$2 + maxset(\bar{A}_1 \cap \bar{A}_4),$$
$$2 + maxset(\bar{A}_2 \cap \bar{A}_4),$$
$$2 + maxset(\bar{A}_3 \cap \bar{A}_4)\}.$$

4.1.2.3: $(w_1, w_2), (w_3, w_4) \in E$,
$(w_1, w_3), (w_1, w_4), (w_2, w_3), (w_2, w_4) \notin E$ (or any symmetric case).
Let $maxset = \max\{1 + maxset(S - \{v\} - A(v)),$
$$2 + maxset(\bar{A}_1 \cap \bar{A}_3),$$
$$2 + maxset(\bar{A}_2 \cap \bar{A}_3),$$
$$2 + maxset(\bar{A}_1 \cap \bar{A}_4),$$
$$2 + maxset(\bar{A}_2 \cap \bar{A}_4)\}.$$

```
            4.1.2.4: $(w_1, w_2) \in E$,
                     $(w_1, w_3), (w_2, w_3), (w_1, w_4), (w_2, w_4), (w_3, w_4) \notin E$
                     (or any symmetric case).
                     Let $maxset = \max\{1 + maxset(S - \{v\} - A(v))$,
                                    $2 + maxset(\bar{A}_1 \cap \bar{A}_3)$,
                                    $2 + maxset(\bar{A}_2 \cap \bar{A}_3)$,
                                    $2 + maxset(\bar{A}_1 \cap \bar{A}_4)$,
                                    $2 + maxset(\bar{A}_2 \cap \bar{A}_4)$,
                                    $2 + maxset(\bar{A}_3 \cap \bar{A}_4)$,
                                    $3 + maxset(\bar{A}_1 \cap \bar{A}_3 \cap \bar{A}_4)$,
                                    $3 + maxset(\bar{A}_2 \cap \bar{A}_3 \cap \bar{A}_4)\}$.
            4.1.2.5: $(w_i, w_j) \notin E$ for $i \neq j$.
                     Let $maxset = \max\{1 + maxset(S - \{v\} - A(v))$,
                                    $2 + maxset(\bar{A}_1 \cap \bar{A}_2)$,
                                    $2 + maxset(\bar{A}_1 \cap \bar{A}_3)$,
                                    $2 + maxset(\bar{A}_1 \cap \bar{A}_4)$,
                                    $2 + maxset(\bar{A}_2 \cap \bar{A}_3)$,
                                    $2 + maxset(\bar{A}_2 \cap \bar{A}_4)$,
                                    $2 + maxset(\bar{A}_3 \cap \bar{A}_4)$,
                                    $3 + maxset(\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3)$,
                                    $3 + maxset(\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_4)$,
                                    $3 + maxset(\bar{A}_1 \cap \bar{A}_3 \cap \bar{A}_4)$,
                                    $3 + maxset(\bar{A}_2 \cap \bar{A}_3 \cap \bar{A}_4)$,
                                    $4 + maxset(\bar{A}_1 \cap \bar{A}_2 \cap \bar{A}_3 \cap \bar{A}_4)\}$.
      4.2: $d(w) \geqq 5$ for some vertex $w$.
           Let $v, w$ be such that $d(v) = 4$, $d(w) \geqq 5$, $(v, w) \in E$.
           Let $maxset = \max\{1 + maxset(S - \{w\} - A(w))$,
                               $maxset(S - \{w\})\}$.
           Note that $S - \{w\}$ contains a vertex of degree three and all vertices are
           of degree three or greater.
   5: $d(w) = 5$ for all vertices $w$.
      5.1: $|S| = 6$.
           Let $maxset = 1$.
      5.2: $|S| > 6$.
           Let $maxset = \max\{1 + maxset(S - \{v\} - A(v))$,
                               $maxset(S - \{v\})\}$.
           Note that $S - \{v\}$ contains a vertex of degree four, a vertex of degree
           five, and all vertices are of degree four or greater.
   6: Some vertex $w$ has $d(w) \geqq 6$.
      Let $maxset = \max\{1 + maxset(S - \{w\} - A(w))$,
                          $maxset(S - \{w\})\}$.
end maxset.
```

3. Resource bounds. Let $T(n)$ be an upper bound on the worst-case running time of *maxset*(S) when $|S| = n$. Let $T_i(n)$ be an upper bound on the worst-case running time of *maxset*(S) when $|S| = n$ and case i occurs at the outermost level of recursion. Let $p(n)$ be a polynomial which bounds the running time of the outermost level of recursion, exclusive of recursive calls. We have the following inequalities. (Starred inequalities are cases which are not obviously better than other cases.)

$$T_0(n) \leqq \max\left\{ \sum_{i=1}^{k} T(n_i) \,\middle|\, \sum_{i=1}^{k} n_i = n,\ 1 \leqq n_i \leqq n \right\} + p(n).$$

$$T_1(n) \leqq T(n-2) + p(n).$$

$$T_{2.1}(n) \leqq p(n).$$

$$T_{2.2}(n) \leqq T(n-3) + p(n).$$

$$T_{2.3}(n) \leqq T(n-3) + T(n-5) + p(n).^*$$

$$T_{3.1}(n) \leqq T(n-4) + p(n).$$

$$T_{3.2}(n) \leqq T(n-4) + T(n-5) + p(n).$$

$$T_{3.3.1}(n) \leqq T(n-4) + T(n-6) + p(n).$$

$$T_{3.3.2}(n) \leqq T(n-4) + 2T(n-7) + p(n).^*$$

$$T_{3.4.1}(n) \leqq T(n-4) + T(n-7) + p(n).$$

$$T_{3.4.2.1}(n) \leqq T(n-4) + T(n-8) + p(n).$$

$$T_{3.4.2.2}(n) \leqq T(n-4) + T(n-6) + T(n-8) + p(n).^*$$

$$T_{3.4.3.1}(n) \leqq T(n-4) + T(n-10) + p(n).$$

$$T_{3.4.3.2}(n) \leqq T(n-4) + T(n-8) + T(n-10) + p(n).$$

$$T_{3.4.3.3}(n) \leqq T(n-4) + 2T(n-8) + T(n-10) + p(n).^*$$

$$T_{3.4.3.4.1}(n) \leqq T(n-4) + 2T(n-8) + T(n-10) + p(n).$$

$$T_{3.4.3.4.2}(n) \leqq T(n-4) + 4T(n-10).^*$$

$$T_{3.4.3.4.3}(n) \leqq T(n-4) + T(n-8) + 3T(n-11).^*$$

$$T_{3.4.3.4.4}(n) \leqq T(n-4) + 2T(n-9) + 2T(n-12).^*$$

$$T_{3.4.3.4.5}(n) \leqq T(n-4) + 3T(n-10) + T(n-13).$$

$$T_{4.1.1.1}(n) \leqq T(n-2) + T(n-6) + p(n).^*$$

$$T_{4.1.1.2.1}(n) \leqq T(n-2) + T(n-8) + p(n).$$

$$T_{4.1.1.2.2}(n) \leqq T(n-2) + 2T(n-8) + p(n).^*$$

$$T_{4.1.1.2.3}(n) \leqq T(n-2) + 2T(n-8) + p(n).$$

$$T_{4.1.1.2.4}(n) \leqq T(n-2) + T(n-8) + 2T(n-10) + p(n).^*$$

$$T_{4.1.2.1}(n) \leqq p(n).$$

$$T_{4.1.2.2}(n) \leqq T(n-5) + 3T(n-9) + p(n).$$

$$T_{4.1.2.3}(n) \leqq T(n-5) + 4T(n-9) + p(n).^*$$

$$T_{4.1.2.4}(n) \leqq T(n-5) + 4T(n-10) + T(n-11) + 2T(n-13) + p(n).$$

$$T_{4.1.2.5}(n) \leqq T(n-5)+6T(n-11)+4T(n-14)+T(n-17)+p(n).^*$$

$$\begin{aligned} T_{4.2}(n) &\leqq T_3(n-1)+T(n-6)+p(n) \\ &\leqq \max\{T(n-5)+T(n-6),\ T(n-5)+2T(n-8),\ T(n-5)+T(n-7) \\ &\qquad +T(n-9), \\ &\qquad T(n-5)+2T(n-9)+T(n-11),\ T(n-5)+4T(n-11), \\ &\qquad T(n-5)+T(n-9)+3T(n-12),\ T(n-5)+2T(n-10) \\ &\qquad +2T(n-13)\} \\ &\quad +T(n-6)+p(n). \end{aligned}$$

$$T_{5.1}(n) \leqq p(n).$$

$$\begin{aligned} T_{5.2}(n) &\leqq T_{4.2}(n-1)+T(n-6)+p(n) \\ &\leqq \max\{T(n-6)+T(n-7),\ T(n-6)+2T(n-9),\ T(n-6)+T(n-8) \\ &\qquad +T(n-10), \\ &\qquad T(n-6)+2T(n-10)+T(n-12),\ T(n-6)+4T(n-12), \\ &\qquad T(n-6)+T(n-10)+3T(n-13),\ T(n-6)+2T(n-11) \\ &\qquad +2T(n-14)\} \\ &\quad +T(n-6)+p(n) \end{aligned}$$

$$T_6(n) \leqq T(n-1)+T(n-7)+p(n).$$

$$T(n) \leqq \max_i T_i(n).$$

From each of the recursive bounds

$$T_i(n) \leqq \sum_{i=1}^{k} a_i T(n-b_i)+p(n)$$

we get a polynomial equation

$$x^{b_k} = \sum_{i=1}^{k} a_i x^{b_k-b_i}.$$

If y is the maximum of the positive solutions to all these equations, $cy^{n+\varepsilon}$ is a bound on the running time of the algorithm. It happens that the value of y is slightly less than $\sqrt[3]{2}$. By means of a tedious calculation using Table 1, one can prove by induction that $T(n) \leqq c2^{n/3}$ without solving lots of polynomials. The constant c depends upon $p(n)$. The worst cases of the recursion are 4.1.1.2.4 and 4.1.2.5.

The storage required by the algorithm is certainly polynomial, since the depth of recursion is only $O(n)$. With careful programming, the storage required can be made linear in the size of the graph.

4. Conclusions. We have presented a recursive algorithm which finds a maximum independent set in a graph of n vertices in $O(2^{n/3})$ time. The algorithm is an extension and improvement of one described in [10]. Though the case

TABLE 1
Fractional exponentials for inductive proof of time bound

n	$2^{n/3}$	n	$2^{n/3}$
1	1.2599^+	9	8.0000
2	1.5876^+	10	10.079^+
3	2.0000	11	12.699^+
4	2.5198^+	12	16.000
5	3.1747^+	13	20.158^+
6	4.0000	14	25.398^+
7	5.0397^+	15	32.000
8	6.3496^+	16	40.317^+

analysis used is lengthy, the algorithm could be programmed easily, and we believe the algorithm would perform well in practice. It may also be possible to simplify the algorithm somewhat by combining cases; we leave such a simplification as an open problem.

Nevertheless, the main interest of the result is theoretical; it shows that even for *NP*-complete problems it is sometimes possible to develop algorithms which are substantially better in the worst case than the obvious enumeration algorithms. Whether the algorithm presented here can be improved substantially, and whether similar algorithms can be developed for other *NP*-complete problems, are open questions. Recently, Chvatal (private communication) has shown that certain kinds of recursive algorithms for finding a maximum independent set must use $2^{\varepsilon n}$ time in the worst case, for some small positive ε. His lower bound applies to the algorithm presented here.

REFERENCES

[1] J. G. AUGUSTIN AND J. MINKER, *An analysis of some graph theoretical cluster techniques*, J. Assoc. Comput. Mach., 17 (1970), pp. 571–588.

[2] E. BALAS AND A. SAMUELSON, *Finding a minimum node cover in an arbitrary graph*, Management Sciences Research Rep. 325, Graduate School of Business Administration, Carnegie-Mellon Univ., Pittsburgh, PA, 1973.

[3] C. BRON AND J. KERBOSCH, *Algorithm 457: Finding all cliques of an undirected graph*, Comm. ACM, 16 (1973), pp. 575–577.

[4] S. COOK, *The complexity of theorem-proving procedures*, Proc. Third ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1970, pp. 151–158.

[5] E. HOROWITZ AND S. SAHNI, *Computing partitions with applications to the knapsack problem*, Tech. Rep. 72-134, Computer Sci. Dept., Cornell Univ., Ithaca, NY, 1972.

[6] H. C. JOHNSTON, *Cliques of a graph: Variations on the Bron–Kerbosch algorithm*, Internat. J. Comput. and Information Sci., 5 (1976), pp. 209–238.

[7] R. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.

[8] G. D. MULLIGAN AND D. G. CORNEIL, *Corrections to Bierstone's algorithm for generating cliques*, J. Assoc. Comput. Mach., 19 (1972), pp. 244–247.

[9] G. L. NEMHAUSER AND L. E. TROTTER, JR., *Vertex packings: Structural properties and algorithms*, Math. Programming, 8 (1975), pp. 232–248.

[10] R. TARJAN, *Finding a maximum clique*, Tech. Rep. 72-123, Computer Sci. Dept., Cornell Univ., Ithaca, NY, 1972.

SIAM J. COMPUT.
Vol. 6, No. 3, September 1977

SUCCINCTNESS OF DESCRIPTIONS OF UNAMBIGUOUS CONTEXT-FREE LANGUAGES*

ERIK MEINECHE SCHMIDT† AND THOMAS G. SZYMANSKI‡

Abstract. There is no recursive function bounding the succinctness gained using ambiguous grammars rather than unambiguous ones in the description of unambiguous context-free languages.

Key words. inherent ambiguity, size of grammars, encoding of TM computations, halting problem

1. Introduction. In this paper we examine the relationship between the size of ambiguous and unambiguous context-free grammars generating the same language. We show that for certain languages the presence of ambiguity in a grammar allows it to be much smaller than any unambiguous one. Specifically we show that for any recursive function there is a language such that the gap between the size of the two types of grammars is not bounded by the function.

The same result is known in two other cases. Meyer and Fischer [3] proved it for descriptions of regular (in fact cofinite) languages using finite automata and ambiguous context-free grammars, and Valiant [6] recently proved it for deterministic pushdown automata (dpda) and unambiguous grammars. Both results are based on the idea of encoding large Turing machine (TM) computations in small context-free grammars, as described by Hartmanis [2]. Meyer and Fischer's result follows easily from this, whereas Valiant has to prove a "repetition lemma" for dpda's in order to get his theorem.

Here we also use the encoding of TM computations but rather than the dpda property we use Ogden's Lemma [4] for context-free languages in an argument which is similar to the proof of the existence of (inherently) ambiguous languages.

2. Preliminaries. We assume that the reader is familiar with the usual concepts associated with context-free grammars as described, for example, in the book by Aho and Ullman [1]. We will denote the *length* of a string x by $|x|$, and the *reversal* of x by x^R. The empty string will be represented by λ. The symbols * and $^+$ will be used in their usual sense of "zero or more" and "one or more" respectively.

In order to talk about economy of descriptions we first have to define what we mean by the size of Turing machines and context-free grammars. We use the same definitions as in [6].

DEFINITION.

(a) Let M be a TM with q states and t tape symbols. The size of M (*size*(M)) is equal to $q \cdot t$.

* Received by the editors April 30, 1976, and in revised form October 6, 1976.

† Department of Computer Science, Cornell University, Ithaca, New York 14853. The work of this author was supported in part by Aarhus University, Aarhus, Denmark, and by "Thanks to Scandinavia, Inc.", New York.

‡ Department of Electrical Engineering and Computer Science, Princeton University, Princeton, New Jersey 08540. The work of this author was supported in part by the National Science Foundation under Grant DCR 74-21939.

(b) Let G be a context-free grammar. Its size ($size(G)$) equals the total number of occurrences of terminal and nonterminal symbols in the productions.

Let $\mathcal{U}$ denote the class of deterministic one-tape Turing machines which always halt (eventually) when started on a blank input tape. For technical reasons we assume that these machines cannot write the blank symbol. Thus at the end of such a machine's computations, every tape cell which had been scanned by the machine during the course of its computation will be left nonblank. We also assume that machines in $\mathcal{U}$ always halt after performing an *odd* number of steps.

Let M be an arbitrary machine from $\mathcal{U}$. Let Q and Γ represent respectively the state set and set of tape symbols of M. The blank symbol is *not* considered to be a member of Γ. A *configuration* of M is any string in $\Gamma^*Q\Gamma^*$. A configuration represents, in the natural fashion, the nonblank portion of M's tape, M's state and head position at some moment in time. A *computation* of M on input x is a sequence of configurations $z_1, z_2, \ldots, z_n$ such that 1) each z_i is a configuration of M, 2) z_1 is the starting configuration of M on x (i.e. $z_1 = q_0x$ where $q_0 \in Q$ is the start state of M), 3) each z_i follows from z_{i-1} by application of the transition function of M (denoted $z_i = NEXT_M(z_{i-1})$) and 4) z_n is a *halting configuration* of M, that is, a string in $\Gamma^*F\Gamma^*$ where F is the set of final states of M.

3. Languages associated with $\mathcal{U}$. In this section we will describe a way to embed Turing machine computations in context-free languages.

Let M be an arbitrary machine from $\mathcal{U}$. Let $\{\#, a, b, c\}$ be a set of new symbols disjoint from both Q and Γ, and let $\Delta = Q \cup \Gamma \cup \{\#\}$. We associate two languages, $\bar{L}_M$ and $\hat{L}_M$, with M as follows:

$\bar{L}_M = \{x_1 \# x_2^R \# x_3 \# x_4^R \# \cdots \# x_{2n}^R a^i b^i c^j \mid$

(a) $n \geqq 1, i \geqq 1, j \geqq 1$

(b) x_1 is the starting configuration of M when started on blank tape,

(c) $x_{2p} = NEXT_M(x_{2p-1})$ for $1 \leqq p \leqq n\}$,

$\hat{L}_M = \{y_1 \# y_2^R \# y_3 \# y_4^R \# \cdots \# y_{2n}^R a^{|y_{2n}|-1} b^j c^j \mid$

(a) $n \geqq 1, j \geqq 1$

(b) $y_{2p+1} = NEXT_M(y_{2p})$ for $1 \leqq p < n$

(c) y_{2n} is a halting configuration of $M\}$.

Observe that both $\bar{L}_M$ and $\hat{L}_M$ are subsets of $\Delta^+a^+b^+c^+$. $\bar{L}_M$ and $\hat{L}_M$ have been chosen to generate strings consisting of pairs of M-configurations. In $\bar{L}_M$, the odd-even pairs represent single steps of M, and in $\hat{L}_M$, it is the even-odd pairs which represent single steps.

It is easy to see that $\bar{L}_M$ and $\hat{L}_M$ have unambiguous context-free grammars, $\bar{G}_M$ and $\hat{G}_M$ respectively, whose sizes are no bigger than a constant times the size of M. If we take their union, we get a grammar G_M for the language

$$L_M = \bar{L}_M \cup \hat{L}_M$$

having the property that

$$size(G_M) = C \cdot size(M)$$

for some constant C independent of M. This grammar, however, is ambiguous because $\bar{L}_M \cap \hat{L}_2$ is nonempty. Indeed, this intersection contains exactly one string

z which corresponds to the computation $z_1, z_2, \cdots, z_{2n}$ of M started on blank tape in the following way:

$$z = z_1 \# z_2^R \# \cdots \# z_{2n}^R a^N b^N c^N$$

where N is the amount of tape used during the computation.

Since $\bar{L}_M \cap \hat{L}_M$ is a finite set, L_M also has an unambiguous context-free grammar. One way of producing such a grammar is to first construct unambiguous pda's for the languages $\bar{L}_M$ and $\hat{L}_M$. Modify the pda for $\hat{L}_M$ by adding appropriate states to its final control to enable it to reject z. Convert the pda's to grammars, take their union, and the result is an unambiguous grammar for L_M. Its size, however, is large due to the extra machinery necessary to avoid generating z in two different ways.

In the next section we are going to prove that *any* unambiguous grammar for L_M must be huge if z is long.

4. Property of unambiguous grammars for L_M. Here we prove that since the word z has inherited two different structures (one from $\bar{L}_M$ and another from $\hat{L}_M$) it will have two different derivations in any small grammar generating L_M. The proof is very similar to the proof that $\{a^ib^jc^k \mid i = j \vee j = k\}$ is an ambiguous language which can be found on p. 205 of Aho and Ullman [1]. It uses the following result from [4].

LEMMA (Ogden). *Let G be a context-free grammar with m symbols (terminals and nonterminals), let l be the length of the longest right-hand side of the productions and let $k = \max\{3, l^{2m+3}\}$. If $z \in L(G)$, $|z| \geqq k$ and if k or more positions in z are designated as being "distinguished" then z can be written as $uvwxy$ such that*

1) *w contains at least one of the distinguished positions,*

2) *either u and v both contain distinguished positions or x and y both contain distinguished positions,*

3) *vwx has at most k distinguished positions,*

4) *there is a nonterminal A such that*

$$S \overset{*}{\Rightarrow} uAy \overset{*}{\Rightarrow} uv^iAx^iy \overset{*}{\Rightarrow} uv^iwx^iy \quad \textit{for all } i \geqq 0.$$

Using Ogden's Lemma we can prove the key lemma of the paper.

LEMMA 1. *Let M be in $\mathcal{U}$ and let N be the amount of tape used in its computation on blank tape. Let G be any context-free grammar generating L_M and let k be the constant from Ogden's Lemma. If $N \geqq k! + k$ then G is ambiguous.*

Proof. Consider the language L_M and the corresponding string z as described in the previous section. Let us rewrite z as $\alpha a^N b^N c^N$ where $\alpha \in \Delta^+$ is that portion of z representing the computation of M. We will show that if $N \geqq k! + k$ then z has two different derivations in G (recall that $\{z\} = \bar{L}_M \cap \hat{L}_M$).

Assume that $N = k! + j$ where $j \geqq k$ and consider the word $z' = \alpha a^N b^j c^j$ which is a member of $\hat{L}_M$ and therefore of L_M. Since $j \geqq k$, we may distinguish all the b's in z' and write z' as

$$z' = uvwxy$$

where u, v, w, x and y satisfy the conditions of Ogden's lemma. We claim that

v consists entirely of b's, *x consists entirely of c*'s *and* $|v|=|x|$. The argument is as follows.

If x and y both have distinguished positions, then $x \in b^+$ because w has at least one distinguished position. This implies that v is λ or else v is a member of Δ^+, Δ^+a^+, $\Delta^+a^+b^+$, a^+, a^+b^+ or b^+.

If v does not contain any a's then the string uwy is of the form $\alpha' a^N b^{j-i} c^j$ for some $i>0$. Since this string has different numbers of a's, b's and c's it can't be a member of L_M and so v must contain at least one a. If v contains other symbols than a's, then the string uv^2wx^2y is not in $\Delta^+a^*b^*c^*$ and hence not in L_M. Thus, $v \in a^+$. Now $uwy = \alpha a^{N-|v|}b^{j-|x|}c^j$ and $uv^2wx^2y = \alpha a^{N+|v|}b^{j+|x|}c^j$ are both in L_M. This implies that $N-|v| = j-|x|$ and $N+|v| = j+|x|$ which contradicts the fact that $N \neq j$. Hence the possibilities for v are exhausted and we may conclude that x and y do not both have distinguished positions.

Accordingly, u and v must both have distinguished positions. Moreover, $v \in b^+$. The possible locations for x are $x=\lambda$, $x \in b^+$, $x \in b^+c^+$ or $x \in c^+$. The first two possibilities are eliminated by considering $uwy = \alpha a^N b^{j-|v|-|x|}c^j$ and the third possibility is eliminated by noting that uv^2wx^2y is a member of $\Delta^+a^+b^+c^+b^+c^+$. This means that $v \in b^+$ and $x \in c^+$. Moreover, if $|v| \neq |x|$, then the string $uwy = \alpha a^N b^{j-|v|}c^{j-|x|}$ is certainly not in L_M, and the proof of our claim is completed.

According to Ogden's Lemma there exists a nonterminal A such that z' can be derived as shown below:

$$S \overset{*}{\Rightarrow} uAy \overset{*}{\Rightarrow} uvAxy \overset{*}{\Rightarrow} uvwxy = z'.$$

Since v consists entirely of distinguished positions and vwx has at most k distinguished positions, $|v|<k$ and so $|v|$ divides $k!$ Let $l = k!/|v|$. By repeating the subderivation $A \overset{*}{\Rightarrow} vAx$ exactly l times, we can derive z in the following way:

$$S \overset{*}{\Rightarrow} uAy \overset{*}{\Rightarrow} uv^lAx^ly$$

$$\overset{*}{\Rightarrow} uv^lwx^ly = \alpha a^N b^{j+l|v|}c^{j+l|v|} = z.$$

(Recall that $N = k!+j = k!/|v| \cdot |v|+j = l|v|+j$.)

Now let us consider the string $z'' = \alpha a^j b^j c^N$ which is a member of $\bar{L}_M$ and hence L_M. By distinguishing all the b's in z'' and arguing in a fashion similar to the above, we can write z'' as $z'' = \bar{u}\bar{v}\bar{w}\bar{x}\bar{y}$ with $|\bar{v}| = |\bar{x}|$, $\bar{v} \in a^+$ and $\bar{x} \in b^+$. As before, there exists some nonterminal B and integer m for which

$$S \overset{*}{\Rightarrow} \bar{u}B\bar{y} \overset{*}{\Rightarrow} \bar{u}\bar{v}^mB\bar{x}^m\bar{y}$$

$$\overset{*}{\Rightarrow} \bar{u}\bar{v}^m\bar{w}\bar{x}^m\bar{y} = z.$$

To complete the proof we will show that the two derivations of z described above have different derivation trees. Assume the contrary. Then the derivation tree for z contains a node labeled A and a node labeled B. No A node can be a descendant of a B node in any derivation tree, for if it were, there would be a derivation of the form $B \overset{+}{\Rightarrow} t_2At_4$ in the grammar, and consequently L_M would contain words of the form

$$t_1a^{|\bar{v}|}t_2b^{|v|}t_3c^{|v|}t_4b^{|\bar{v}|}t_5.$$

Similarly, B is not a descendent of A. This means that A and B are incomparable in the derivation tree for z and hence there are terminal strings s_1, s_2, and s_3 such that $S \overset{+}{\Rightarrow} s_1Bs_2As_3$ in G. By inserting the subderivations $B \overset{+}{\Rightarrow} \bar{v}B\bar{x}$ and $A \overset{+}{\Rightarrow} vAx$ we can produce, for any integer i, the string $s_1\bar{v}^i\bar{w}\bar{x}^is_2v^iwx^is_3$. By choosing i sufficiently large, we can produce strings having many more b's than either a's or c's. Such a word is clearly not in L_M, contradicting our assumption that the two derivations of z correspond to the same tree. Accordingly, G is ambiguous. □

5. The size of an unambiguous grammar for L_M. We use Lemma 1 to show that the size of any unambiguous grammar for L_M must grow with the amount of tape used in M's computation.

LEMMA 2. *There exists a constant C with the following property: Let M be a machine in $\mathcal{U}$ which uses N tape cells in its computation. Let G be an unambiguous grammar generating L_M. Then*

$$size(G) \geqq C \cdot [\log \log N]^{1/2}.$$

Proof. Let m and l be, respectively, the number of symbols in G and the length of the longest right side of a production in G. Let $g = size(G)$. We know from Lemma 1 that

$$N < (l^{2m+3})! + l^{2m+3}.$$

Since m is at least 2 (otherwise G could only generate strings of length 0 and 1) we must have $2m+3 \leqq 4m$. In addition, $m \leqq g$ and $l \leqq g$. Hence

$$N < (g^{4g})! + g^{4g} \leqq 2(g^{4g})!$$

which implies (since $g \geqq 2$) that

$$N < (g^{4g})^{(g^{4g})}$$

Taking logarithms twice, we obtain

$$\begin{aligned} \log \log N &< 4g \log g + \log 4 + \log g + \log \log g \\ &\leqq 4g \log g + 4 \log g \\ &\leqq 6g \log g \\ &\leqq 6g^2 \end{aligned}$$

which is what we want. □

Now we can prove our main result.

THEOREM. *There is no recursive function F with the following property: For all ambiguous context-free grammars G_a generating an unambiguous language there*

exists an unambiguous grammar G_u generating the same language such that

$$size(G_u) \leqq F(size(G_a)).$$

Proof. Assume the contrary and consider some L_M where M is in $\mathcal{U}$. Since L_M has an ambiguous grammar G_a^M such that

$$size(G_a^M) \leqq \text{const.} \cdot size(M)$$

we get—using Lemma 2 and the function F (which we may assume to be increasing)—

$$C \cdot [\log \log N]^{1/2} \leqq size(G_u^M) \leqq F(size(G_a^M)) \leqq F(\text{const.} \cdot size(M))$$

where N is the amount of tape used in M's computation. But this means that there is a fixed recursive relation between N and $size(M)$ for all machines in $\mathcal{U}$ and this immediately enables us to decide whether an arbitrary TM halts when started on blank tape.

Thus we have a contradiction and we may conclude that the theorem is true. □

6. Conclusion. The result proved in this paper answers one of the questions left open by Valiant [6]. If we consider the relative succinctness gained using finite automata, deterministic pushdown automata, unambiguous context-free grammars and arbitrary context-free grammars we get the following table (see Table 1)

TABLE 1

Bounds on relative succinctness

Original descriptor \ New descriptor	fa	dpda	ucfg	cfg
fa	—	recursive	?	nonrec
dpda		—	nonrec	nonrec
ucfg			—	nonrec
cfg				—

representing bounds on the size savings achieved by substituting a more powerful descriptor for the original descriptor of a given language. The recursiveness of fa-dpda was proved by Stearns [5]. The nature of the relation between fa and ucfg is still open.

Acknowledgment. The authors wish to thank Leonard Berman and the referees for their remarks on earlier drafts of this paper.

REFERENCES

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

[2] J. HARTMANIS, *Context-free languages and Turing machine computations*, Proc. Sympos. Appl. Math., vol. 19, American Mathematical Society, Providence, RI, 1967 pp. 42–51.

[3] A. R. MEYER AND M. J. FISHER, *Economy of description by automata, grammars and formal systems,* Proc. 12th An. Sympos. on Switching and Automata Theory (1971), IEEE Computer Society, Silver Spring, MD, pp. 188–191.

[4] W. OGDEN, *A helpful result for proving inherent ambiguity*, Math. Systems Theory, 2 (1968), no. 3, pp. 191–194.

[5] R. E. STEARNS, *A regularity test for pushdown machines*, Information and Control 11 (1967), pp. 323–340.

[6] L. G. VALIANT, *A note on the succinctness of descriptions of deterministic languages*, Information and Control, 32 (1976), no. 2, pp. 139–145.

SIAM J. COMPUT.
Vol. 6, No. 3, September 1977

THE PARTIAL FRACTION EXPANSION PROBLEM AND ITS INVERSE*

FRANCIS Y. CHIN†

Abstract. The partial fraction expansion problem and its inverse are studied and it is shown that these two problems can be solved in $O(N \log^2 N)$ steps for those rational functions with N simple poles, $O(N \log N)$ steps for those with a single multiple pole of order N and $O(N \log N(\log n + 1))$ steps for the general multiple pole case, where N is the degree of the denominator polynomial and n is the number of distinct poles. We further show that the evaluation of a rational function and its derivatives at a given point can be done more efficiently than previously known. Previous known algorithms for the partial fraction problem and its inverse require $O(n^2)$ steps.

Key words. partial fraction decomposition, asymptotic arithmetic complexity, polynomial evaluation, rational functions

1. Introduction. In Laplace and z-transform analysis and many other applications, we encounter the problem of breaking up rational functions into a sum of simple components, which sum is called a *partial fraction expansion*. Consider the following rational function which has no common factors in the numerator and denominator polynomials,

$$F(x) = P(x)/Q(x) = \prod_{i=0}^{m-1} (x - z_i)^{d_i} \prod_{j=0}^{n-1} (x - p_j)^{c_j}$$

with zeros z_i of multiplicity d_i and poles p_j of multiplicity c_j. If p_j is a simple zero ($c_j = 1$) of the denominator polynomial $Q(x)$, then p_j is said to be a *simple pole* of the rational function. If $c_j = r > 1$, then p_j is called a *multiple pole of order r*.

The partial fraction expansion can be obtained by solving a system of linear equations, which takes $O(N^{2.81})$ steps by Strassen's method [ST69]. Here, N is the degree of the denominator polynomial, assumed to be greater than the numerator's degree. Another approach is to find the residues of the poles, which still takes $O(N^2)$ steps for the simple pole case, and more than $O(N^2)$ steps for the multiple pole case by Brugia [BR] and Linner's method [LI]. By applying those better algorithms given in [CS] or in Pottle's paper [PO], the general problem for the multiple pole case can be solved in $O(N^2)$ steps. In this paper, we are going to study the asymptotic complexity of this problem. We have shown that the partial expansion problem is reducible to the polynomial evaluation problem and can be done in $O(N \log^2 N)$ steps[1] for those rational functions with simple poles, $O(N \log N)$ steps[1] for the case of a single multiple pole, and $O(N \log N$ $(\log n + 1))$ steps for the general multiple pole case, where n is the number of distinct poles in $F(x)$. Furthermore, we show that the evaluation of the derivatives of a rational function at a certain point is reducible to the partial fraction

* Received by the editors May 21, 1976.

† Department of Computing Science, University of Alberta, Edmonton, Canada T6G 2H1. This research is part of the author's Ph.D. dissertation and was carried out at Princeton University. This work was supported by National Science Foundation under Grant GK-42028 and the U.S. Army Research Office-Durham under Contract DAHC04-69-C0012.

[1] Part of these results also appear in [CU].

expansion problem and thus can be done more efficiently than by previously known algorithms.

For the second section of this paper, we shall study the inverse of the partial fraction expansion problem and show that it will take $O(N \log^2 N)$ steps for the simple pole case, $O(N \log N)$ steps for the single multiple pole case and $O(N \log N(\log n+1))$ steps for the general multiple pole case, analogously to the results of the partial fraction expansion problem.

2. The partial fraction expansion problem (PF problem). In what follows we shall work over the field of complex numbers. The PF problem can be defined thusly:

PF problem: Given a rational function

$$F(x)=P(x)/Q(x)=\sum_{i=0}^{M} d_i x^i \Big/ \prod_{j=0}^{n-1}(x-p_j)^{c_j} \tag{1}$$

with $N=\sum_{j=0}^{n-1} c_j > M$, we want to find the $K_{j,i}$'s such that

$$F(x)=\sum_{j=0}^{n-1}\sum_{i=1}^{c_j} K_{j,i}/(x-p_j)^i. \qquad \square$$

Usually, $K_{j,i}$ is calculated by the following formula:

$$K_{j,i}=[1/(c_j-i)!][D^{(c_j-i)}(P(x)/Q_j(x))]|_{x=p_j}$$

where

$$Q_j(x)=Q(x)/(x-p_j)^{c_j}=\prod_{\substack{i=0\\ i\neq j}}^{n-1}(x-p_i)^{c_i},$$

and D is the differentiation operator. The $K_{j,i}$ can be related to the truncated Taylor expansion of $P(x)/Q_j(x)$ by

$$P(x)/Q_j(x)=K_{j,c_j}+K_{j,c_j-1}(x-p_j)+\cdots+K_{j,1}(x-p_j)^{c_j-1}+\cdots.$$

Now, we are going to bound from above the asymptotic complexity of the PF problem by means of algorithms for fast polynomial evaluation, multiplication and division. Furthermore, we shall make use of the $O(n \log^2 n)$ n-point evaluation algorithm [MB], [KU] to solve this problem for the simple pole case.

THEOREM 1. *The PF problem for the case of N simple poles can be done in* $O(N \log^2 N)$ *steps.*

Proof. Writing

$$F(x)=P(x)/Q(x)=\sum_{i=0}^{M} d_i x^i \Big/ \prod_{j=0}^{N-1}(x-p_j)$$
$$=\sum_{j=0}^{N-1} K_j/(x-p_j)$$

we know that

$$K_k=[P(x)/\prod_{\substack{j=0\\ j\neq k}}^{N-1}(x-p_j)]|_{x=p_k}=P(x)/Q'(x)|_{x=p_k}$$

for $k=0,1,\cdots,N-1$.

$Q(x)$ can be computed as $\sum_{j=0}^{N} a_j x^j$ in $O(N \log^2 N)$ steps and its derivative $Q'(x)$ can be found in $O(N)$ steps. Since $Q'(x)$ and $P(x)$ are two polynomials of degree at most $(N-1)$, there exists an $O(N \log^2 N)$ algorithm [MB], [KU] for the evaluation of $Q'(x)$ and $P(x)$ at p_k, $k = 0, 1, \cdots, N-1$. □

Now we shall show that the PF problem for the single multiple pole of order N is equivalent to evaluating an $(N-1)$st degree polynomial and all its derivatives.

DEFINITION. $A \leqq B$ means that problem A is reducible to problem B in linear time, and we say that problem A is *equivalent* to problem B iff $A \leqq B$ and $B \leqq A$. □

THEOREM 2. *The PF problem for the single multiple pole of order N is equivalent to the evaluation problem of an* $(N-1)$*-st degree polynomial and all its derivatives at any point.*

Proof. We have

$$F(x) = P(x)/(x-p_1)^N = \sum_{i=1}^{N} K_i/(x-p_1)^i$$

with $K_{N-j} = (1/j!)(D^j P(x))|_{x=p_1}$ where $j = 0, \cdots, N-1$.

So the PF problem is reducible to the evaluation problem for $P(x)$. Conversely, by letting $P(x)$ be any $(N-1)$st degree polynomial, the evaluation problem for $P(x)$ and all its derivatives at p_1 is reducible to the PF problem for a single multiple pole of order N at p_1. □

COROLLARY 1. *The PF problem for the single multiple pole of order N can be done in* $O(N \log N)$ *steps.*

Proof. An $(N-1)$st degree polynomial and all its derivatives can be evaluated at any point in $O(N \log N)$ steps [ASU], [VA]. So by the above theorem, this PF problem can be done in $O(N \log N)$ steps. □

Besides these two particular cases, the general multiple pole problem can also be solved asymptotically more efficiently than in $O(N^2)$ steps. Before showing this, we are going to prove that the first N terms in the truncated Taylor series of any rational polynomial function can be obtained in $O(N \log N)$ steps.

LEMMA 1. *Given* $P(x) = \sum_{i=0}^{m} a_i x^i$ *and* $Q(x) = \sum_{i=0}^{n} b_i x^i$, *where* $Q(x)$ *does not have a root at* $x = 0$, *the first N terms in the truncated Taylor expansion of* $P(x)/Q(x)$ *at* $x = 0$ *can be computed in* $O(N \log N)$ *steps.*

Proof. From [SI], the first N terms of $1/Q(x)$ can be found from the first N terms of $Q(x)$ in $O(N \log N)$ steps. With another $O(N \log N)$ steps, we can multiply the first N terms of $1/Q(x)$ by the first N terms of $P(x)$ and obtain the first N terms in the truncated Taylor expansion of $P(x)/Q(x)$ at $x = 0$. □

LEMMA 2. *Given* $P(x) = \sum_{i=0}^{m} a_i x^i$ *and* $Q(x) = \sum_{i=0}^{n} b_i x^i$, *the first N terms in the truncated Taylor expansion of* $P(x)/Q(x)$ *at any point which is not a root of* $Q(x)$ *can be found in* $O(N \log N)$ *steps, where* $m, n \leqq N$.

Proof. Taylor expansion of a polynomial at any point requires $O(N \log N)$ steps [ASU], the proof then proceeds as in Lemma 1. □

LEMMA 3. *All* $I_i(x) = (x-p_i)^{c_i}$; $i = 0, \cdots, n-1$, *where* $\sum_{i=0}^{n-1} c_i = N$ *can be computed in* $O(N)$ *steps.*

Proof. It is easy to see that each $I_i(x)$ takes only $O(c_i)$ steps by the recurrence formula $a_k = ((c_i - k + 1)p_i/k)a_{k-1}$ where $I_i(x) = \sum_{j=0}^{c_i} a_{c_i - j} x^j$. Thus it follows immediately that all the $I_i(x)$ can be computed in $O(N)$ steps. □

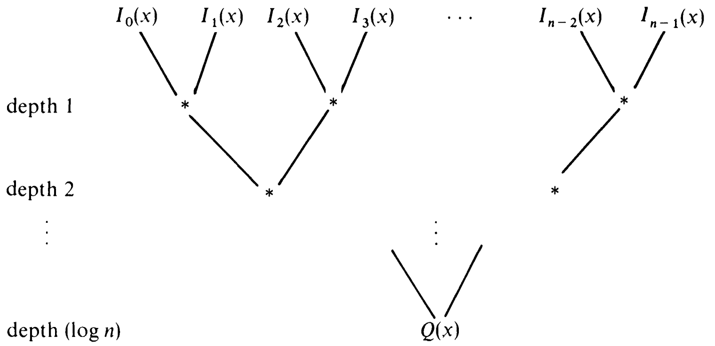


FIG. 1

LEMMA 4.

$$Q(x)=\prod_{i=0}^{n-1}(x-p_i)^{c_i}$$

can be obtained in $O((N\log N)(\log n))$ *steps where* $N=\sum_{i=0}^{n-1}c_i$.

Proof. See Fig. 1.

Since the sum of the degrees of all the terms at any depth is N, the computation of all the terms at depth i from the terms at depth $(i-1)$ takes $O(N\log N)$ steps. The result follows immediately from the fact that the depth of the tree is $\log n$. □

Now we are going to show that the general PF problem is reducible to the problem of evaluating a polynomial of degree N and its first (c_j-1) derivatives at points x_j where $j=0,\cdots,n-1$ and $N=\sum_{j=0}^{n-1}c_j$. This evaluation problem has been shown in [CH] that it can be done in $O(N\log N(\log n+1))$ steps. Thus, we can show that the general PF problem can be done in the same number of steps which matches with its two extreme cases, $O(N\log^2 N)$ and $O(N\log N)$ steps when $n=N$ and $n=1$ respectively.

THEOREM 3. *The general PF problem with n distinct poles requires no more than* $O(N\log N(\log n+1))$ *steps.*

Proof. Let

$$F(x)=P(x)/\prod_{j=0}^{n-1}(x-p_j)^{c_j}$$

$$=\sum_{j=0}^{n-1}\sum_{i=1}^{c_j}K_{j,i}/(x-p_j)^i$$

where $\sum_{j=0}^{n-1}c_j=N$. It is shown in Lemma 4 that the computation of $Q(x)=\prod_{j=0}^{n-1}(x-p_j)^{c_j}$ takes no more than $O(N\log N\log n)$ steps.

Let's first consider

$$G_k(x)=P(x)/\prod_{\substack{j=0\\j\neq k}}^{n-1}(x-p_j)^{c_j}=P(x)/Q_k(x)$$

$$=\sum_{i=0}^{c_k-1}K_{k,c_k-i}(x-p_k)^i+\cdots$$

where $K_{k,c_k-i}=(1/i!)(D^iG_k(x))|_{x=p_k}$.

Basically, we do the Taylor expansions at each of the poles, and it will be sufficient if the first (c_j-1) terms of the truncated Taylor expansions of $P(x)$ and $Q_j(x)$ at the poles x_j are known. As for $P(x)$, the evaluation of $P(x)$ and its first (c_j-1) derivatives at the poles x_j will take no more than $O(N \log N(\log n+1))$ steps by the results in [CH]. Although each $Q_j(x)$ is different for each pole p_j, we can write

$$Q(x)=\prod_{k=0}^{n-1}(x-p_k)^{c_k},$$

$$D^iQ(x)=\sum_{k=0}^{i}\binom{i}{k}[D^k(x-p_j)^{cj}][D^{i-k}(Q_j(x))],$$

so

$$D^i(Q(x))|_{x=p_j}=\binom{i}{c_j}c_j!D^{i-c_i}(Q_j(x))|_{x=p_j}$$

or

$$D^h(Q_j(x))|_{x=p_j}=[h!/(c_j+h)!][D^{c_j+h}(Q(x))]|_{x=p_j}.$$

Therefore, to evaluate all the $D^iQ_j(x)|_{x=p_j}$ for $i=0, 1, \cdots, c_j-1$ and all $j=0, 1, \cdots, n-1$, we evaluate $Q(x)$ and all its $(2c_j-1)$ derivatives at all the p_j's and this will take no more than $O(N \log N(\log n+1))$ steps [CH]. Hence all the $K_{j,i}$'s can be found in $O(N \log N(\log n+1))$ steps. □

3. The evaluation problem for rational functions. In Theorem 2, we have shown that the evaluation problem of an $(N-1)$st degree polynomial and all its derivatives at any point is equivalent to the PF problem for the single multiple pole of order N. We can now show some similar equivalences.

THEOREM 4. *The problem of finding the L coefficients of* $1/(x-p_0)^i$ *where* $i=1, 2, \cdots, L$, *in the PF expansion of* $P(x)/[(x-p_0)^LQ(x)]$ *is equivalent to the evaluation problem of* $P(x)/Q(x)$ *and its* $(L-1)$ *derivatives at* p_0, *as long as* p_0 *is not a root of either* $P(x)$ *or* $Q(x)$.

Proof. The result follows trivially from the equations below:

$$F(x)=P(x)/(x-p_0)^LQ(x)=\sum_{i=1}^{L}K_i/(x-p_0)^i+\cdots$$

and

$$K_{L-i}=(1/i!)[D^i(P(x)/Q(x))]|_{x=p_0}$$

where $i=0, 1, \cdots, L-1$. □

COROLLARY 2. *If* $F(x)=P(x)/Q(x)=\sum_{i=0}^{M}d_ix^i/\prod_{j=0}^{n-1}(x-p_j)^{c_j}$ *where* $M \leqq \sum_{j=0}^{n-1}c_j+L$, *then the evaluation of* $F(x)$ *and its first* $(L-1)$ *derivatives at* p_0 *which is not a root of* $P(x)$ *or* $Q(x)$ *can be done in* $O((N+L)\log(N+L)\log(n+1))$ *steps, where* $N=\sum_{j=0}^{n-1}c_j$. □

COROLLARY 3. *Similarly for*

(i) $F(x)=P(x)/\prod_{j=0}^{N-1}(x-p_j)$ *in* $O((N+L)\log^2(N+L))$ *steps,*

(ii) $F(x)=P(x)/(x-p_1)^N$ *in* $O((N+L)\log(N+L))$ *steps,*

(iii) $F(x)=P(x)/(x-p_1)$ *in* $O(L\log L)$ *steps.* □

4. The inverse of the partial fraction expansion problem (IPF problem). The IPF problem is defined as:

IPF problem. Given

$$F(x)=\sum_{j=0}^{n-1}\sum_{i=1}^{c_j}\frac{K_{j,i}}{(x-p_j)^i}, \tag{2}$$

find the d_i's, $i=1,\cdots,N-1$, such that

$$F(x)=\sum_{j=0}^{N-1}d_ix^i/\prod_{j=0}^{n-1}(x-p_j)^{c_j},$$

where $N=\sum_{j=0}^{n-1}c_j$. □

THEOREM 5 (simple pole case). *If all the* c_j*'s in* (2) *are* 1 (i.e. *all the poles are simple*) *the IPF problem can be done in* $O(N\log^2N)$ *steps.*

Proof.

$$F(x)=\sum_{j=0}^{N-1}\frac{K_{j,1}}{(x-p_j)}=\sum_{j=0}^{N-1}K_{j,1}\Bigg(\prod_{\substack{k=0\\k\neq j}}^{N-1}(x-p_k)\Bigg)\Bigg/\prod_{j=0}^{N-1}(x-p_j).$$

By the divide and conquer approach to compute the summation in the numerator

$$\sum_{j=0}^{N-1}K_{j,1}\Bigg(\prod_{\substack{k=0\\k\neq j}}^{N-1}(x-p_k)\Bigg)=\Bigg(\prod_{k=N/2}^{N-1}(x-p_k)\Bigg)\Bigg(\sum_{j=0}^{N/2-1}K_{j,1}\Bigg(\sum_{\substack{k=0\\k\neq j}}^{N/2-1}(x-p_k)\Bigg)\Bigg)$$
$$+\Bigg(\prod_{k=0}^{N/2-1}(x-p_k)\Bigg)\Bigg(\sum_{j=N/2}^{N-1}K_{j,1}\Bigg(\prod_{\substack{k=N/2\\k\neq j}}^{N-1}(x-p_k)\Bigg)\Bigg),$$

this IPF problem can be solved in $O(N\log^2N)$ steps. □

THEOREM 6 (single multiple pole case). *If* $n=1$ *in* (2), *i.e. F has only one multiple pole of order N, then this IPF problem can be solved in* $O(N\log N)$ *steps.*

Proof.

$$F(x)=\sum_{j=1}^{N}K_j/(x-p_1)^j$$
$$=\sum_{j=1}^{N}K_j(x-p_1)^{N-j}/(x-p_1)^N,$$

$$\begin{aligned}\sum_{j=1}^{N} K_j(x-p_1)^{N-j} &= \sum_{j=0}^{N-1} K_{N-j}(x-p_1)^j \\ &= \sum_{j=0}^{N-1} K_{N-j}\left[\sum_{i=0}^{j}\binom{j}{i}p_1^{j-i}x^i\right] \\ &= \sum_{i=0}^{N-1}\sum_{j=i}^{N-1} K_{N-j}\binom{j}{i}p_1^{j-i}x^i \\ &= \sum_{i=0}^{N-1}\left[\sum_{j=0}^{N-1} f(j)g(i-j)\right]x^i/i!\end{aligned}$$

where $f(k)=K_{N-k}k!$ for $0 \leqq k \leqq N-1$,

$$g(k)=\begin{cases} p_1^{-k}/(-k)!, & -N+1 \leqq k \leqq 0, \\ 0, & 1 \leqq k \leqq N-1.\end{cases}$$

It is easy to see that $f(k)$ and $g(k)$ can be computed in $O(N)$ steps, and since the inner summation in the above formula is a convolution which can be evaluated in $O(N \log N)$ steps, the coefficients of x^i can be found in $O(N \log N)$ steps. □

There exists an $O(N \log N(\log n+1))$ algorithm for the general IPF problem with multiple poles.

THEOREM 7. *The IPF problem can be solved in* $O(N \log N(\log n+1))$ *steps.*

Proof. It suffices for us to consider the case when $n>1$ and not all the c_j's are 1. Let's first consider the summation of all the terms with the same pole with $c_j>1$,

$$G_j(x)=\sum_{i=1}^{c_j} K_{j,i}/(x-p_j)^i = H_j(x)/(x-p_j)^{c_j}$$

where

$$H_j(x)=\sum_{i=1}^{c_j} K_{j,i}(x-p_j)^{c_j-i}$$

which can be calculated in $O(c_j \log c_j)$ steps as proved in Theorem 6. Let C_p be the constant for this computation; then all the $H_j(x)$ can be computed in $C_p \sum_{j=0}^{n-1} c_j \log c_j \leqq C_pN \log N = O(N \log N)$ steps.

Let $I_j(x)=(x-p_j)^{c_j}$. Lemma 3 shows that all the $I_j(x)$'s can be computed in $O(N)$ steps. Now we can write (2) as

$$\begin{aligned}F(x) &= \sum_{j=0}^{n-1}[H_j(x)/I_j(x)] \\ &= \left[\sum_{j=0}^{n-1} H_j(x)\prod_{\substack{k=0\\k\neq j}}^{n-1} I_k(x)\right]\Big/\prod_{j=0}^{n-1} I_j(x).\end{aligned}$$

Assume $n=2^r$. We have

$$\begin{aligned}\sum_{j=0}^{n-1} H_j(x)\left[\prod_{\substack{k=0\\k\neq j}}^{n-1} I_k(x)\right] &= J_{n/2,n-1} * \left[\sum_{j=0}^{n/2-1} H_j(x)\left(\prod_{\substack{k=0\\k\neq j}}^{n/2-1} I_k(x)\right)\right] \\ &\quad + J_{0,n/2-1} * \left[\sum_{j=n/2}^{n-1} H_j(x)\left(\prod_{\substack{k=n/2\\k\neq j}}^{n-1} I_k(x)\right)\right]\end{aligned}$$

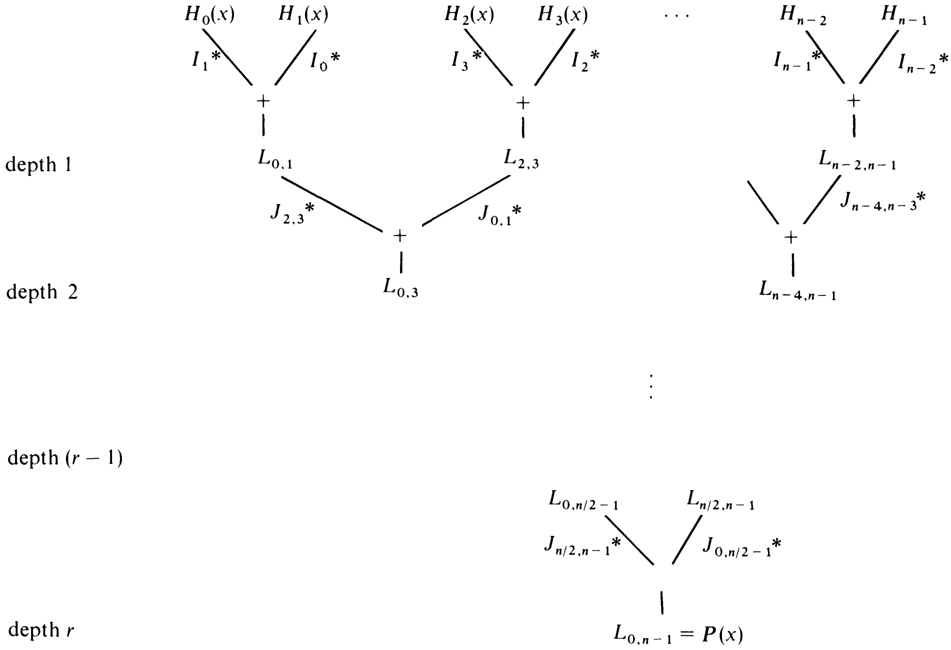


FIG. 2

where $J_{e,f} = \prod_{j=e}^{f} I_j(x)$. All the $J_{2^i(j-1)+1,2^ij}$ where $j = 1, \cdots, 2^{r-i}$; $i = 1, \cdots, r$, can be computed in $O(N \log N \log n)$ steps by saving the intermediate results when building up the tree in Lemma 4. Finally, we want to show that $\sum_{j=0}^{n-1} H_j(x) \prod_{k=0,k\neq j}^{n-1} I_k(x)$ can be computed in $O(N \log N \log n)$ steps.

Letting $L_{e,f} = \sum_{i=e}^{f} H_i(x) \prod_{j=e,j\neq i}^{f} I_j(x)$, we have the computation tree in Fig. 2.

So, it can be shown by the similar argument as given in Lemma 4 that the number of steps for computing all the terms at each depth is $O(N \log N)$. Since the height of this tree is $\log n$, $L_{0,n-1}$ can be in $O(N(\log N)(\log n + 1))$ steps. □

5. Conclusion. The partial fraction expansion problem is also investigated by E. Horowitz [HO]. Instead of assuming a linearly factored denominator, he considers the partial fraction expansion problem with the denominator polynomial in its square-free factorization form.[2] Algorithms of cost $>O(N^4)$ are presented, but there is no fundamental dispute with the results in this paper. In [HO], Horowitz deals with rational numbers where the complexity of each arithmetic step depends on the size of the numbers, while in this paper, the field of complex number is used and each complex arithmetic step is assumed to be unit cost. Anyway, the assumption that the denominator of the rational function is

[2] Let B be a polynomial of positive degree and $B = \prod_{i=0}^{k} B_i^i$ where each B_i is *square-free* (i.e., B_i has no divisor of multiplicity $\geqq 2$), and the B_i are pairwise relative prime for $1 \leqq i \leqq k$. Then $\prod_{i=0}^{k} B_i^i$ is called the *square-free factorization* of B. Algorithms for square-free factorization are presented in [YU].

given in an already factored form is one basic weakness of the paper; the general case is considered by Kung and Tong in [KT].

REFERENCES

[ASU] A. V. Aho, K. Steiglitz and J. D. Ullman, *Evaluation of polynomials at a fixed set of points*, this Journal, 4 (1975), pp. 533–539.

[BR] O. Brugia, *A noniterative method for the partial fraction expansion of rational functions with higher order poles*, SIAM Rev., 7 (1965), pp. 381–387.

[CH] F. Y. Chin, *A generalized asymptotic upper bound on polynomial evaluation and interpolation*, this Journal, 5 (1976), pp. 682–690.

[CS] F. Y. Chin and K. Steiglitz, *An $O(N^2)$ algorithm for partial fraction expansion*, IEEE Trans. Circuits and Systems, CAS-24, 1 (1977), pp. 42–45.

[CU] F. Y. Chin and J. D. Ullman, *Asymptotic complexity of partial fraction expansion*, Tech. Rep. 172, Dept. of Electrical Engineering, Princeton Univ., Princeton, NJ.

[HO] E. Horowitz, *Algorithms for partial fraction decomposition and rational function integration*, Proc. 2nd Symposium on Symbolic and Algebraic Manipulation, 1971, pp. 441–457.

[KU] H. T. Kung, *Fast evaluation and interpolation*, Computer Science Tech. Rep., Carnegie-Mellon University, Jan. 1973.

[KT] H. T. Kung and D. M. Tong, *Fast algorithms for partial fraction decomposition*, Proc. Symposium on New Directions and Recent Results in Algorithms and Complexity at Carnegie-Mellon Univ., J. Traub, ed., April, 1976.

[LI] L. J. P. Linner, *The computation of the K-th derivative of polynomials and rational functions in factor form and related matters*, IEEE trans. Circuits and Systems, (1974), pp. 233–236.

[MB] R. Moenck and A. Borodin, *Fast modular transforms via division*, Proc. of the IEEE 13th Annual Symposium on Switching and Automatia Theory, (1972), pp. 90–96.

[PO] C. Pottle, *On the partial fraction expansion of a rational function with multiple poles by digital computer*, IEEE trans. on Circuit Theory, (1964), pp. 161–162.

[SI] M. Sieveking, *An algorithm for division of power series*, Computing 10 (1972), pp. 153–156.

[ST69] V. Strassen, *Guassian elimination is not optimal*, Numer. Math., 13 (1969), pp. 354–356.

[ST72] ———, *Die Berechnungskomplexitat von Elementarsymmetrischen Funktionen und von Interpolationskoeffizioten*, Ibid., 20 (1972), pp. 238–251.

[VA] T. M. Vari, *Some complexity results for a class of Toeplitz matrices*, Tech. Rep., Dept. of Computer Science and Mathematics, York University, Toronto, 1974.

[YU] D. Y. Y. Yun, *On square-free decomposition algorithms*, Proc. of 1976 ACM Symposium on Symbolic and Algebraic Computation, pp. 26–35.

SIAM J. COMPUT.
Vol. 6, No. 3, September 1977

AN ANALYSIS OF SEVERAL HEURISTICS FOR THE TRAVELING SALESMAN PROBLEM*

DANIEL J. ROSENKRANTZ,† RICHARD E. STEARNS† AND PHILIP M. LEWIS II†

Abstract. Several polynomial time algorithms finding "good," but not necessarily optimal, tours for the traveling salesman problem are considered. We measure the closeness of a tour by the ratio of the obtained tour length to the minimal tour length. For the nearest neighbor method, we show the ratio is bounded above by a logarithmic function of the number of nodes. We also provide a logarithmic lower bound on the worst case. A class of approximation methods we call insertion methods are studied, and these are also shown to have a logarithmic upper bound. For two specific insertion methods, which we call nearest insertion and cheapest insertion, the ratio is shown to have a constant upper bound of 2, and examples are provided that come arbitrarily close to this upper bound. It is also shown that for any $n \geqq 8$, there are traveling salesman problems with n nodes having tours which cannot be improved by making $n/4$ edge changes, but for which the ratio is $2(1-1/n)$.

Key words. traveling salesman problem, approximation algorithm, k-optimal, minimal spanning tree, triangle inequality

1. Introduction.

The traveling salesman problem has long been of great interest. The problem has been formulated in several different ways. We use the following formulation:

A traveling salesman graph G is a complete weighted undirected graph specified by a pair (N, d) where N is a set of nodes, d is a *distance function* mapping pairs of nodes (or edges) into real numbers, and d satisfies

a) $d(i,j)=d(j,i)$ for all i and j in N,

b) $d(i,j)\geqq 0$ for all i and j in N,

c) $d(i,j)+d(j,k)\geqq d(i,k)$ for all i, j, k in N.

Condition c) is referred to as the *triangle inequality*. The number $d(i,j)$ is called the *length* or *weight* of (i,j).

A *tour* for a traveling salesman graph G is a circuit on the graph containing each node exactly once (i.e. a Hamiltonian circuit). The *length* of a tour is the sum of the lengths of the edges composing the circuit. An *optimal tour* or *solution* for G is a tour of minimal length. The *traveling salesman problem* is to take a traveling salesman graph and find an optimal tour.

The traveling salesman problem is sometimes formulated (Bellmore and Nemhauser [1]) as the problem of finding a minimal length circuit containing each node at least once for an undirected graph in which the distances are not constrained by the triangle inequality. However, a problem stated in this manner can always be reduced (Hardgrave and Nemhauser [6]) to the problem considered here by the technique of changing each $d(i,j)$ to the length of the shortest path between i and j. This conversion can be done in time proportional to the cube of the number of nodes (Floyd [4]). Each tour in the new problem corresponds to a

* Received by the editors July 19, 1976, and in revised form December 13, 1976.

† General Electric Research and Development Center, Schenectady, New York 12345. An extended abstract of this paper is in the Proceedings of the IEEE Fifteenth Annual Symposium on Switching and Automata Theory, 1974, under the title *Approximate algorithms for the traveling salesperson problem.*

circuit of the same length in the original problem, and the two problems have solutions of the same length. Therefore, our results, which are stated in terms of the new problem, also apply to the original problem.

Another formulation requires that a shortest tour be found for distances not constrained by the triangle inequality. A problem stated this way can always be reduced to the type of problem considered here by adding a suitably large constant k to each distance. The altered problem has the same optimal tour as the original, but the lengths of the optimal tours will differ by the amount $n \cdot k$ where n is the number of nodes. Our results do not apply to this formulation, since our results pertain to the tour lengths.

The best known methods of solving the traveling salesman problem take an amount of time exponential in the number of nodes. Furthermore, the problem is easily seen to be *NP*-hard. Karp [8] shows that determining whether an undirected graph has a Hamiltonian circuit is *NP*-complete. This problem can be reduced to a traveling salesman problem by forming the complete weighted graph whose edges are of length one if there is a corresponding edge in the original graph, and of length two otherwise. For an n node graph, the minimal tour of the new graph has length n if and only if the original graph is a Hamiltonian circuit.

In view of the computational difficulties in obtaining optimal tours, a number of algorithms have been published which run faster but do not necessarily produce an optimal tour. A number of these approximation algorithms have been experimentally observed to perform well, but there has not been a theoretical characterization of how the obtained tours compare with the optimal.

In this paper, we analyze some of these methods to bound the ratio of the obtained tour length to the optimal tour length. In some cases, these bounds grow as a function of the number of nodes and in other cases a constant bound is found for all traveling salesman problems. In contrast, if the distance function is unconstrained by the triangle inequality then for any constant $k \geqq 1$, the problem of finding a tour with a ratio bounded by k is *NP*-complete (Sahni and Gonzalez [16]).

Another approximation method was recently announced and analyzed in Christofides [2]. This method produces a better worst case approximation than the methods analyzed here, but requires more running time.

In the material which follows, we exclude the trivial case where the distance function is identically zero. This assumption together with the triangle inequality implies that every tour has a length greater than zero. We also adopt the convention that OPTIMAL represents the length of the optimal tour. Under the assumption of nontriviality,

$$\text{OPTIMAL} > 0 \tag{1.1}$$

2. Nearest neighbor algorithm.

2. Nearest neighbor algorithm. The first approximation algorithm we study is the nearest neighbor method (Bellmore and Nemhauser [1]), also called the next best method in Gavett [5]. In this algorithm, a path is constructed as follows:

1. Start with an arbitrary node.
2. Find the node not yet on the path which is closest to the node last added and add to the path the edge connecting these two nodes.

3. When all nodes have been added to the path, add an edge connecting the starting node and the last node added.

We assume that when there are ties in step 2, they can be broken arbitrarily.

We note that the nearest neighbor algorithm can be programmed to operate in time proportional to n^2 where n is the number of nodes. This time is linear in the input length if the input is a list of all distances.

Let NEARNEIBER be the length of the tour obtained by the nearest neighbor algorithm. Let lg denote the logarithm to the base 2, and $\lceil x \rceil$ denote the smallest integer greater than or equal to x.

THEOREM 1. *For a traveling salesman graph with n nodes*

$$\frac{\text{NEARNEIBER}}{\text{OPTIMAL}} \leqq \frac{1}{2}\lceil \lg (n) \rceil + \frac{1}{2}.$$

The proof of Theorem 1 is given after the proof of the following lemma.

LEMMA 1. *Suppose that for a n node graph* (N, d) *there is a mapping assigning each node p a number* l_p *such that the following two conditions hold*:

a) $d(p, q) \geqq \min (l_p, l_q)$ *for all nodes p and q.*

b) $l_p \leqq \frac{1}{2}$ OPTIMAL *for all nodes p.*

Then $\sum l_p \leqq \frac{1}{2}(\lceil \lg (n) \rceil + 1)$OPTIMAL.

Proof. We can assume without loss of generality that node set N is $\{i \mid 1 \leqq i \leqq n\}$ and that $l_i \geqq l_j$ whenever $i \leqq j$. The key to the proof is the following inequality:

$$\text{OPTIMAL} \geqq 2 \sum_{i=k+1}^{\min(2k,n)} l_i \tag{2.1}$$

for all k satisfying $1 \leqq k \leqq n$.

To prove (2.1), we let H be the complete subgraph defined on the set of nodes

$$\{i \mid 1 \leqq i \leqq \min (2k, n)\}.$$

We let T be the tour in H which visits the nodes of H in the same order as these nodes are visited in an optimal tour of the original graph. Let LENGTH be the length of T. By the triangle inequality, each edge (b, c) of T must have a length which is less than or equal to the length of the path from b to c used in the optimal tour. Since the edges of T sum to LENGTH and the corresponding paths in the original graph sum to OPTIMAL we conclude that

$$\text{OPTIMAL} \geqq \text{LENGTH}. \tag{2.2}$$

By condition a) of the Lemma, for each (i, j) in T, $d(i, j) \geqq \min (l_i, l_j)$. Therefore,

$$\text{LENGTH} \geqq \sum_{(i,j) \in T} \min (l_i, l_j) = \sum_{i \in H} \alpha_i l_i \tag{2.3}$$

where α_i is the number of edges (i, j) in T for which $i > j$ (and hence $l_i = \min (l_i, l_j)$).

We want to obtain a lower bound on the right hand side of (2.3). Observe that each α_i is at most 2 (because i is the endpoint of only two edges in tour T) and that the α_i sum to the number of edges in T. Because k is at least half of the number of edges in T, we certainly get a lower bound on the right hand side of (2.3) if we assume that the k largest l_i have $\alpha_i = 0$ and the remaining $\min(2k, n) - k$ of the l_i have $\alpha_i = 2$. By assumption, the k largest are $\{l_i | 1 \leqq i \leqq k\}$ so the estimated lower bound is

$$(2.4) \qquad \sum_{i \in H} \alpha_i l_i \geqq 2 \sum_{i=k+1}^{\min(2k,n)} l_i$$

and (2.2), (2.3), and (2.4) together establish (2.1).

We now sum inequalities 2.1 for all values of k equal to a power of two less than n, namely:

$$\sum_{j=0}^{\lceil \lg(n) \rceil - 1} \text{OPTIMAL} \geqq \sum_{j=0}^{\lceil \lg(n) \rceil - 1} 2 \cdot \sum_{i=2^j+1}^{\min(2^{j+1},n)} l_i,$$

which reduces to

$$(2.5) \qquad \lceil \lg(n) \rceil \cdot \text{OPTIMAL} \geqq 2 \cdot \sum_{i=2}^{n} l_i.$$

Now condition b) of the lemma implies

$$(2.6) \qquad \text{OPTIMAL} \geqq 2 \cdot l_1$$

and (2.5) and (2.6) combine to give the conclusion of the lemma.

Proof of Theorem 1. For each node p, let l_p be the length of the edge leaving node p and going to the node selected as the nearest neighbor to p. We want to show that the l_p satisfy the conditions of Lemma 1.

If node p was selected by the algorithm before node q, then q was a candidate for the closest unselected node to node p. This means that edge (p, q) is no shorter than the edge selected and hence

$$(2.7) \qquad d(p, q) \geqq l_p.$$

Conversely, if q was selected before p, then

$$(2.8) \qquad d(p, q) \geqq l_q.$$

Since one of the nodes was selected before the other, either (2.7) or (2.8) must hold and condition a) of Lemma 1 must be satisfied.

To prove condition b) it suffices to prove that for any edge (p, q)

$$(2.9) \qquad d(p, q) \leqq \tfrac{1}{2} \cdot \text{OPTIMAL}.$$

The optimal tour can be considered to consist of two disjoint parts, each of which is a path between nodes p and q. From the triangle inequality, the length of any path between p and q cannot be less than $d(p, q)$, establishing (2.9).

Because the l_p are the lengths of the pairs comprising tour T,

$$(2.10) \qquad \sum l_p = \text{NEARNEIBER}.$$

The conclusion of Lemma 1 together with (2.10) and (1.1) imply the inequality of Theorem 1.

THEOREM 2. *For each $m > 3$, there exists a traveling salesman graph with $n = 2^m - 1$ nodes such that*

$$\frac{\text{NEARNEIBER}}{\text{OPTIMAL}} > \frac{1}{3} \lg(n+1) + \frac{4}{9}.$$

Proof. For all $i \geqq 1$, we define an incomplete weighted graph F_i with three distinguished nodes. The distinguished nodes are called the *start node*, the *middle node*, and the *right node*. These graphs are defined recursively using Fig. 1 where the start node appears to the left, the middle node in the middle, and the right node on the right. Each F_i has a path P_i which goes from the start node to the middle node visiting each node of F_i on the way. The P_i are also defined recursively in Fig. 1.

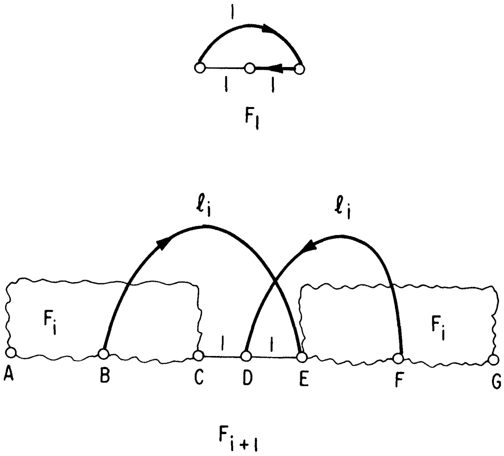


FIG. 1

Graph F_1 consists of precisely three nodes with each pair of nodes having an edge of weight 1. Path P_1 consists of two edges, the edge from the start node to the right node and the edge from the right node to the middle node.

To construct graph F_{i+1}, one takes two copies of F_i (which we call the *left copy* and *right copy*) and one additional node (which becomes the middle node of F_{i+1}). This additional node is called D in Fig. 1. The additional node D is connected to the right node of the left copy (node C) and the start node of the right copy (node E) by edges of length 1. The additional node D is also connected to the middle node of the right copy (node F) by an edge of length l_i (defined below). Finally, the middle node of the left copy (node B) is connected to the start node of the right copy (node E) by an edge of length l_i. The start node of F_{i+1} is the start node of the left copy (node A) and the right node is the right node of the right copy (node G). The path P_{i+1} consists of the two copies of path P_i plus the two edges (B, E) and (F, D) of length l_i. The length l_i is given by the formula

$$l_i = \tfrac{1}{6}(4 \cdot 2^i - (-1)^i + 3). \tag{2.11}$$

Let L_i be the length of path P_i. Length L_i is described by the difference equation

$$L_{i+1} = 2 \cdot L_i + 2 \cdot l_i$$

since P_{i+1} consists of two copies of P_i and two edges of length l_i. Given that $L_1 = 2$, the solution of this difference equation is

$$L_i = \tfrac{1}{9}(6 \cdot i \cdot 2^i + 8 \cdot 2^i + (-1)^i - 9). \tag{2.12}$$

For each F_i, we define a graph G_i obtained by connecting the start and right nodes of F_i by an edge of length 1, and connecting the middle node to the start node with an edge of length $l_i - 1$. The start node of F_i is then also referred to as the start node of G_i. Figure 2 is a picture of G_4. We define $\bar{G}_i$ to be the complete graph on the nodes of G_i where $d(a, b)$ is the length of the minimal path from a to b in G_i. Therefore, the distances in $\bar{G}_i$ satisfy the triangle inequality.

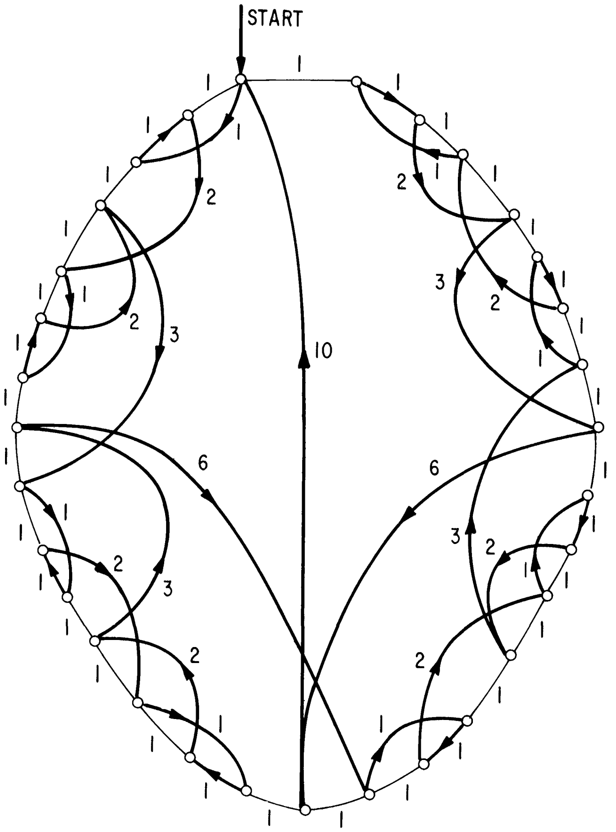


FIG. 2

Graph $\bar{G}_i$ has two important properties:

a) the edges of G_i have the same lengths in $\bar{G}_i$ as they have in G_i;

b) if the nearest neighbor method is started with the start node of G_i, the method can (with suitable resolution of ties) produce path P_i followed by the edge of length $l_i - 1$ returning from the middle node (which is the last node of path P_i) to the start node.

We return to prove properties a) and b) after completing the main thread of the proof.

Each $\bar{G}_i$ has an optimal tour whose length is equal to the number of nodes n in $\bar{G}_i$ (namely $2^{i+1} - 1$). This tour is found, starting with the start node, by visiting the nodes in left to right order and then returning from the right node back to the start node. Each of the edges in this tour has weight one.

The example satisfying the theorem is $\bar{G}_{m-1}$. Its ratio is exactly

$$\frac{\text{NEARNEIBER}}{\text{OPTIMAL}}=(L_i+l_i-1)/n \quad \text{where } i=\lg(n+1)-1.$$

This ratio is greater than the ratio indicated in the theorem.

All that remains is to prove properties a) and b). Referring back to Fig. 1, we first show that for each F_{i+1}

$$\overline{AB}=\overline{BC}=\overline{EF}=\overline{FG}=l_i-1, \tag{2.13}$$

$$\overline{AC}=\overline{EG}=l_{i+1}-2, \tag{2.14}$$

$$\overline{BE}=\overline{DF}=l_i, \tag{2.15}$$

$$\overline{AD}=\overline{DG}=l_{i+1}-1, \tag{2.16}$$

$$\overline{AG}=l_{i+2}-2. \tag{2.17}$$

The notation $\overline{XY}$ indicates the length of the shortest path between X and Y in F_{i+1}. The equations are routinely verified for $i=1$. We continue by induction. Assume that (2.13)–(2.17) are true for $i \leqq I-1$ (i.e. for F_I). Figure 3a shows the relevant nodes in F_{I+1} before the two copies of F_I are connected. Associated with each pair of nodes from the same copy of F_I, an edge is shown whose weight is the length of the shortest path in F_I connecting these nodes. These shortest path lengths in F_I are specified by the induction hypothesis. For instance, edge (A, B) in Fig. 3a connects the start and middle nodes of F_I and from (2.16), the shortest path length in F_I connecting these nodes is l_I-1. Figure 3b shows Fig. 3a with the addition of the four edges created in the construction of F_{I+1} from the two copies of F_I. Because each of the edge weights in Fig. 3a represents a shortest path in F_I, by applying formula (2.11) for l_I to all possible paths in F_{I+1}, we can conclude that

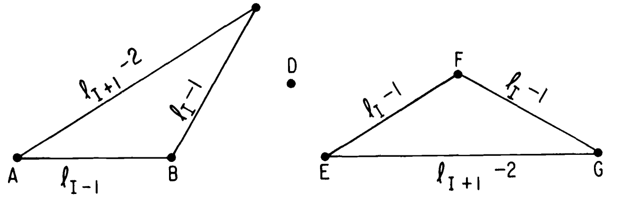


FIG. 3a

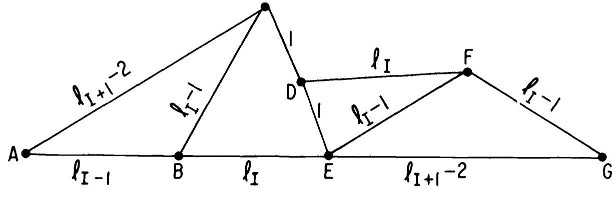


FIG. 3b

each of the edge weights in Fig. 3b is the length of the shortest path in F_{I+1} *connecting the end nodes of the edges. This establishes equations* (2.13)–(2.15) *for* F_{I+1}. Equations (2.16) and (2.17) are established by a similar consideration of all paths in Fig. 3b. The path of length $l_{I+2}-2$ from A to G is $ABEG$.

We note also that (2.13)–(2.16) also hold when F_{I+1} is converted to G_{I+1}. This is proven by connecting A and G in Fig. 3b by an edge of weight one and A and D by an edge at length $l_{I+1}-1$ and again checking the paths. Note also that the shortest path from A to D is the edge (A, D).

Now we return to property a). Equation (2.15) shows that, as each F_{i+1} is constructed, the newly added edges constitute the shortest paths between their endpoints. All distances among points in an F_i are maintained when that F_i is embedded in F_{i+1}, because the distances among the three exit points at F_i are maintained. (Compare (2.13) with (2.16) and (2.14) with (2.17).)

We have already noted that the final edge added in constructing a G_i is also a shortest path and the fact that the length one edges are also shortest paths requires no argument. Thus property a) is established.

Property b) is established by observing that the middle node of an F_i is reached only after all of the nodes of F_i have been visited, and the node at the end of the edge of length l_i is at least as close as any node reached by a path through the start or right nodes. These nodes are already distance l_{i-1} away from the middle node and are at least distance 1 from a node not yet selected. □

One way to improve a nearest neighbor result is to repeat the method for each possible starting node and then take the minimum solution among these. This idea is described in Gavett [5]. However, for the examples used to prove Theorem 2, the result of this method (with suitable resolution of ties) is also proportional to $\lg(n)$.

3. Insertion methods.

We now consider a class of methods we call insertion methods. The basic idea of these methods is to construct the approximation tour by a sequence of steps in which tours are constructed for progressively larger subsets of the nodes.

DEFINITION. Given a traveling salesman graph (N, d), a tour T on a subset S of N will be called a *subtour* of (N, d). We write $a \in T$ to mean $a \in S$. We treat a one node subset as a tour without edges.

DEFINITION. Given a traveling salesman graph (N, d), a subtour T, and a node k in N which is not in T, we define TOUR(T, k) to be a subtour obtained as follows:

if T passes through more than one point, then

a) find an edge (x, y) in T which minimizes

$$d(x, k)+d(k, y)-d(x, y), \tag{3.1}$$

b) delete edge (x, y) and add edges (x, k) and (k, y) to obtain TOUR(T, k);

if T passes through a single node i, then make TOUR(T, k) the two node tour consisting of edges (i, k) and (k, i).

In either case, we say that TOUR(T, k) is obtained by *inserting* k into T.

Formula (3.1) represents the difference in length between tour T and the tour obtained by replacing (x, y) by (x, k) and (k, y). Thus, when T has two or more nodes, TOUR(T, k) is the shortest tour that can be obtained from T and k by the alteration described in step b). When T has only one node, TOUR(T, k) is the only tour that can be made from k and the point in T.

DEFINITION. An approximation method is called an *insertion method* if it takes a traveling salesman graph (N, d) with n nodes and constructs a sequence of subtours $T_1, \cdots, T_n$ so that

1. T_1 consists of a single node a_0,
2. for each $i < n$, there is a node a_i not in T_i such that

$$T_{i+1} = \mathrm{TOUR}(T_i, a_i), \tag{3.2}$$

3. T_n is the approximation.

In later sections, we consider specific selection criteria for choosing the nodes a_i. Here we are concerned with results which hold regardless of the selection method.

DEFINITION. Given a subtour T and a node k not in T, we define COST(T, k) to be the length of TOUR(T, k) minus the length of T.

An important consequence of the triangle inequality is the following:

LEMMA 2. *If (N, d) is a traveling salesman graph, T is a subtour, k a node not in T, and j a node in T, then*

$$\mathrm{COST}(T, k) \leqq 2 \cdot d(k, j). \tag{3.3}$$

Proof. In the case that T has only one node, the result is obvious. When T consists of more than one node, j is an endpoint of some edge (i, j) in T. Because k is inserted to minimize (3.1),

$$\mathrm{COST}(T, k) \leqq d(i, k) + d(k, j) - d(i, j) \tag{3.4}$$

where the right-hand side is (3.1) with (i, j) substituted for (x, y). The triangle inequality says

$$d(i, k) - d(i, j) \leqq d(j, k). \tag{3.5}$$

Inequalities (3.4) and (3.5) together with $d(j, k) = d(k, j)$ give (3.3).

We let INSERT represent the length of a path constructed by an insertion algorithm.

THEOREM 3. *For a traveling salesman graph with n nodes,*

$$\frac{\mathrm{INSERT}}{\mathrm{OPTIMAL}} \leqq \lceil \lg(n) \rceil + 1. \tag{3.6}$$

Proof. Let (N, d) be the graph and let T_i for $1 \leqq i \leqq n$ and a_i for $0 \leqq i < n$ be the subtours and nodes referred to in the definition of an insertion method. An obvious consequence of the definition of cost is

$$\mathrm{INSERT} = \sum_{i=1}^{n-1} \mathrm{COST}(T_i, a_i) \tag{3.7}$$

For each node a_i in N-$\{a_0\}$, define

$$l_{a_i} = \tfrac{1}{2} \cdot \mathrm{COST}(T_i, a_i) \tag{3.8}$$

and define

(3.9) $$l_{a_0}=0.$$

We want to show that the l_p for p in N satisfy the hypothesis of Lemma 1. To verify condition a), consider two nodes a_i and a_j with $i>j$. By our naming conventions, $i>j$ means that a_j belongs to T_i and a_i was inserted in T_i. By Lemma 2,

(3.10) $$\mathrm{COST}(T_i, a_i) \leqq 2 \cdot d(a_i, a_j).$$

With (3.8) this implies

(3.11) $$l_{a_i} \leqq d(a_i, a_j),$$

which implies condition a).

Condition b) is trivial for l_{a_0}. For other l_{a_i}, (3.8) requires us to prove

(3.12) $$\mathrm{COST}(T_i, a_i) \leqq \mathrm{OPTIMAL}.$$

In the case of a_1, this cost is just $2d(a_0, a_1)$ and by the triangle inequality, OPTIMAL is at least as large as the distance between two points and back. For $i>1$, a_i is inserted between two distinct points x and y with cost

(3.13) $$d(x, a_i)+d(a_i, y)-d(x, y),$$

which is the length of the added edges minus the length of the deleted edge. There is a subpath of the optimal tour between x and a_i which does not contain y and a disjoint subpath between a_i and y not containing x. By the triangle inequality, these subpaths are no shorter than $d(x, a_i)$ and $d(a_i, y)$ respectively and hence (3.13) must be no greater than OPTIMAL and condition b) is established.

Lemma 1 together with (3.8), (3.9), (3.7), and (1.1) imply the theorem. □

We do not know if the logarithmic growth permitted by Theorem 3 can actually be achieved. In fact, we know of no examples such that INSERT/OPTIMAL>4 so there could even be a constant upper bound. In the next section we present some insertion methods for which we can establish a constant upper bound.

4. Nearest insertion and cheapest insertion. We now consider two insertion methods which produce a tour no longer than twice the optimal regardless of the number of nodes in the problem. We call these two methods the nearest insertion method and the cheapest insertion method.

Given a subtour T and a node p, we define the distance $d(T, p)$ between T and p as

(4.1) $$\min\{d(x, p) \text{ for } x \text{ in } T\}.$$

We say that a tour is constructed by *nearest insertion* if each a_i, $1 \leqq i < n$, in the definition of an insertion method satisfies

(4.2) $$d(T_i, a_i)=\min\{d(T_i, x) \text{ for } x \text{ in } N-T_i\}.$$

We say a tour is constructed by *cheapest insertion* if the a_i satisfy

(4.3) $$\mathrm{COST}(T_i, a_i)=\min\{\mathrm{COST}(T_i, x) \text{ for } x \text{ in } N-T_i\}.$$

The nearest insertion method is easily programmed to run in a time proportional to n^2. The only programming trick is to compute the value of $d(T_{i+1}, x)$ as the minimum of the two numbers $d(T_i, x)$ and $d(a_i, x)$. Thus the nearest insertion method runs in time proportional to the nearest neighbor method.

The cheapest insertion method is described in Nicholson [12]. The fastest algorithm we have devised for the cheapest insertion method runs proportional to $n^2 \cdot \log(n)$. Each time a node a_i is inserted in T_i, the new subtour T_{i+1} contains two new edges not in T_i. For each new edge (x, a_i) in T_{i+1}, the algorithm involves performing a sort of the $n-(i+1)$ values of

$$d(x, k)+d(k, a_i)-d(x, a_i)$$

obtained for all k in $N-T_{i+1}$.

THEOREM 4. *If a tour of length* INSERT *is obtained by nearest insertion or cheapest insertion, then*

$$\frac{\text{INSERT}}{\text{OPTIMAL}}<2. \tag{4.4}$$

We prove this theorem after proving the following lemma:

LEMMA 3. *Suppose that, for a traveling salesman graph* (N, d) *with n nodes, a tour of length* INSERT *is constructed by the insertion method of* § 3. *Suppose further that for i satisfying* $1 \leqq i < n$, *the tour* T_i *and node* a_i *selected by the insertion method satisfy*

$$\text{COST}(T_i, a_i) \leqq 2 \cdot d(p, q) \tag{4.5}$$

for all nodes p and q such that p is in T_i *and q is not in* T_i. *Then*

$$\text{INSERT} \leqq 2 \cdot \text{TREE} \tag{4.6}$$

where TREE *is the length of a minimal spanning tree for* (N, d).

Proof. Let M be a minimal spanning tree. The idea of the proof is to establish a correspondence between steps in the insertion procedure and edges of M. For the step of inserting node a_i into T_i, the corresponding edge of M will have one endpoint in T_i and the other endpoint in $N-T_i$. Thus (4.5) can be used to show that the cost of each step is no more than twice the corresponding edge.

First, since M is a tree, there is a unique path in M connecting each pair of nodes. For each node a_i with $i>0$, we say that node a_j is *compatible* with node a_i if $j<i$ and all the intermediate nodes in the unique path in M connecting a_i and a_j have indices greater than i. Thus a_j compatible with a_i implies that a_j is the first node in T_i encountered in the path from a_i to a_j. For each a_i with $i>0$, the *critical node* is the node with the largest index that is compatible with a_i. The *critical path* for a_i is the unique path in M between a_i and its critical node. The *critical edge* for a_i is the edge in the critical path, one of whose endpoints is the critical node. Observe that the critical edge for a_i has one endpoint (the critical node) in T_i, and the other endpoint in $N-T_i$.

We now show that no two nodes can have the same critical edge. Assume to the contrary that a_i and a_j (with $j>i$) have the same critical edge. Let the endpoints of this critical edge be a_k and a_l with $l>k$. For any critical edge, the node with the lower index is the critical node and the node with the higher index is on the critical path, so node a_k is the critical node for both a_i and a_j. Thus, the critical paths for a_i and a_j both pass through a_l before reaching a_k. Therefore, there is a path P in M connecting a_j and a_i, such that every edge in P belongs to either the critical path for a_j or the critical path for a_i (or both). Therefore every intermediate node on P has an index greater than i. Since the path P from a_j reaches a node of lower index (a_i), some node a_m along path P is compatible with a_j. Now $m \geqq i$ because a_m is on path P and $i > k$ because a_k is a compatible node for a_i. This implies $m > k$ and so a_m is a compatible node for a_j with a higher index than a_k. This contradicts the assumption that a_k is critical for a_j. Therefore no two nodes can have the same critical edge. Thus given a minimal spanning tree we can associate a unique edge in that tree with each node inserted by the insertion method.

Let e_i be the critical edge for node a_i. Since one endpoint of e_i is in T_i and the other endpoint is not, by (4.5).

$$\text{COST}(T, a_i) \leqq 2 \cdot d(e_i). \tag{4.7}$$

Summing (4.7) gives

$$\sum_{i=1}^{n-1} \text{COST}(T_i, a_i) \leqq 2 \cdot \sum_{i=1}^{n-1} d(e_i). \tag{4.8}$$

The left-hand side of (4.8) is INSERT by (3.7). Since M consists of $n-1$ edges, and each e_i is distinct, the right-hand side of (4.8) is $2 \cdot$ TREE. Thus (4.8) implies (4.6).

Proof of Theorem 4. We first show that, for both insertion methods, (4.5) holds. For the nearest insertion, there is for each i by (4.2) a node y_i in T_i such that

$$d(y_i, a_i) \leqq d(p, q) \tag{4.9}$$

for all p in T_i and q in $N - T_i$. Lemma 2 says that

$$\text{COST}(T_i, a_i) \leqq 2 \cdot d(y_i, a_i) \tag{4.10}$$

and (4.9) and (4.10) imply (4.5). For the cheapest insertion, the cost of inserting a_i is by (4.3) even less than the cost of inserting an a_i chosen to satisfy (4.2). Therefore (4.5) must also hold in this case and Lemma 3 applies to both cases.

The optimal tour can be made into a tree by deleting its longest edge and this longest edge has a length at least OPTIMAL$/n$ where n is the number of nodes in the problem. Since the minimal spanning tree is no longer than this tree,

$$\text{TREE} \leqq \left(1 - \frac{1}{n}\right) \cdot \text{OPTIMAL}. \tag{4.11}$$

Equations (4.11), (4.6), and (1.1) imply (4.4),

Corollary. *For a traveling salesman graph on n nodes, equation* (4.4) *in Theorem* 4 *may be replaced by*

$$\frac{\text{INSERT}}{\text{OPTIMAL}} \leqq 2 \cdot \left(1 - \frac{1}{n}\right). \tag{4.12}$$

For the nearest insertion method, a simpler correspondence than that in the proof of Lemma 3 can be established between the cost of the insertion steps and the edges of a minimal spanning tree. Since each a_i is selected in accordance with (4.2), there is an edge (x, a_i) such that x is in T_i and

$$(4.13) \qquad d(x, a_i) = \min\{d(p, q) \text{ for } p \text{ in } T_i \text{ and } q \text{ in } N - T_i\}.$$

Let e_i be this edge (x, a_i) and observe from Lemma 2 that

$$\mathrm{COST}(T_i, a_i) \leqq 2 \cdot e_i.$$

Moreover, the set of edges $\{e_i | 1 \leqq i < n\}$ constitute a minimal spanning tree since the method of selecting edges satisfying (4.13) is a method of constructing a minimal spanning tree (Kruskal [9], Prim [13]).

We now show that there exist traveling salesman graphs for which the bound (4.12) is actually achieved. The examples can be interpreted as cities placed uniformly on a circular road. The case for 8 nodes is shown in Fig. 4. The optimal path is simply to go around the circle. The insertion methods may construct a path such as that in the figure, a path which goes almost all the way around and then doubles back on itself. Thus, each edge of the circle (except one) is traveled twice instead of the one time actually required, and the ratio of the path obtained to OPTIMAL is roughly two.

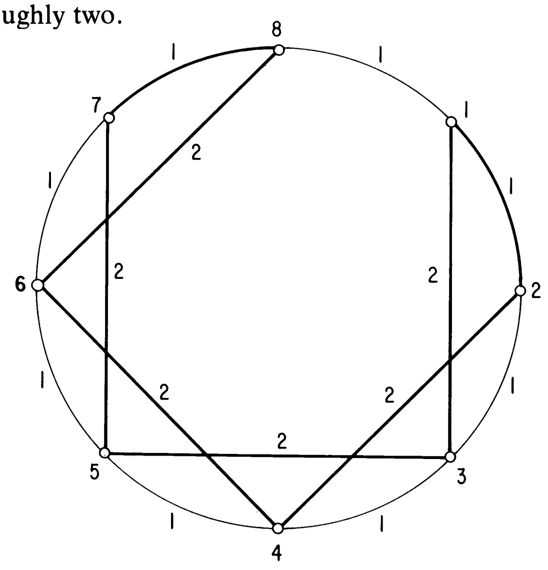


FIG. 4

THEOREM 5. *For $n \geqq 6$, there exists a traveling salesman graph on n nodes such that*

$$(4.14) \qquad \frac{\mathrm{INSERT}}{\mathrm{OPTIMAL}} = 2 \cdot \left(1 - \frac{1}{n}\right)$$

for either the nearest insertion or cheapest insertion methods.

Proof. We define graph (N_n, d_n) as follows:

$$N_n = \{i | 1 \leqq i \leqq n\},$$

$d_n(i, j)$ = smallest nonnegative integer m such that

$$i - j \equiv m \pmod{n} \text{ or } j - i \equiv m \pmod{n}.$$

We define T_1 to be the tour on set $\{1\}$, we define

$$T_2 = \{(1,2),(2,1)\}$$

and for $3 \leqq i \leqq n$ we define

$$T_i = \{(1,2),(i-1,i)\} \cup \{(j,j+2) | 1 \leqq j \leqq i-2\}.$$

Figure 4 shows T_8 for the case $n = 8$.

We define

$$a_i = i + 1 \quad \text{for } 0 \leqq i < n.$$

Obviously the T_i defined above are tours. We will show that the T_i together with the a_i satisfy (3.2), (4.2), and (4.3).

T_{i+1} is obtained from T_i by deleting edge $(i-1, i)$ and adding edges $(i-1, i+1)$ and $(i, i+1)$.

We compute that

$$\text{COST}(T_i, a_i) = 2$$

and that $(i-1, i)$ is the edge in T_i which minimizes (3.1); this proves (3.2). We also compute that

$$d(T_i, a_i) = 1$$

because $d(a_{i-1}, a_i) = (i+1) - i = 1$ and (4.2) is satisfied because 1 is the shortest distance between distinct nodes. It can also be calculated that no insertion can cost less than 2 and so (4.2) holds. We omit these calculations but note that they require the assumption $n \geqq 6$.

We note finally that the approximation T_n has two edges of length one and $n-2$ edges of length two for a total length of $2 \cdot (n-1)$. The optimal tour is obviously the tour of length n that starts with node 1 and visits the nodes in numerical order. Equation (4.14) is obtained by dividing these two lengths. □

For $i > 3$ in the above proof, the insertion of a_i into T_i to obtain T_{i+1} involves a tie between edges $(i-1, i)$ and $(i-2, i)$, both of which minimize (3.1). An example with no ties in (3.1) is obtained from the example by decreasing the length of all edges greater than 1 by a small number ε. The choice $(i-1, i)$ of the proof becomes the unique choice and the construction proceeds as in the proof. The resulting ratio is very close (depending on ε) to (4.14).

Theorem 5 shows that, in the worst case, nearest insertion can create paths which double back on themselves and are roughly twice as long as necessary. We examined a number of problems with nodes placed randomly on a plane, and observed that the nearest insertion method often produced paths in which portions doubled back on themselves.

5. Farthest insert. There is another insertion method which has some intuitive and empirical appeal, a method we call farthest insertion.

We say that a tour is constructed by *farthest insertion* if each a_i, $1 \leqq i < n$, in the definition of an insertion method satisfies

$$d(T_i, a_i) = \max\{d(T_i, x) \text{ for } x \text{ in } N - T_i\}. \tag{5.1}$$

Contrasting (5.1) with (4.2), we observe that farthest insertion has a max where nearest insertion has a min. The intuitive appeal is that the method establishes the general outline of the approximate tour at the outset and then fills in the details. The early establishment of a general outline is appealing because we expect better performance when the number of nodes is small. Inserting nearby points late in the approximation is appealing because the short edges used late in the procedure are less likely to be accidentally deleted by some still later insertion.

The empirical appeal is that, in a series of experiments, we found that farthest insertion usually produced a better tour than nearest insertion, cheapest insertion, and the nearest neighbor. For example, when tried on problems obtained by placing 50 nodes randomly on a unit square, nearest insertion was from 7 to 22% worse than farthest insertion, nearest neighbor was from 0 to 38% worse, and cheapest insertion ranged from 7% better to 12% worse. The usual ranking was thus farthest insertion first, cheapest insertion second, nearest insertion third, and nearest neighbor last.

The largest example we tried was 2000 points placed uniformly at random in the unit square. The score was farthest insertion 36.8, nearest insertion 41.4, nearest neighbor 39.9. A path of length 37.2 was obtained by randomly selecting the order in which points were chosen for insertion. The farthest insertion path was no more than 1.25 times the optimal since the minimal spanning tree had length 29.5.

The advantage of picking random or arbitrary points for insertion is that virtually no computation time is needed to select an arbitrary point. On the 2000 city problem, the nearest neighbor tour was constructed in 751 seconds, the arbitrary insertion in 820 seconds, and the nearest and farthest insertions in 1628 seconds each.

Theorems 2 and 4 tell us that, in the worst case, the nearest neighbor paths become progressively worse than the nearest insertion paths as the number of nodes increase. We found no evidence of such a trend in our experiments. For example, in the 2000 node example described above, nearest neighbor actually did better than nearest insertion.

Altogether, our experiments suggest that the performance of the methods is not strongly tied to their worst case behavior.

6. Some other approximation methods. There are a variety of other approximation methods for which the cost of each step in the construction of the tour corresponds to a unique edge in a minimal spanning tree and for which the reasoning of Lemma 3 and Theorem 4 can be used to demonstrate a worst case ratio bound of 2. In this section, we discuss two such methods.

The first method, which we call *nearest addition*, is similar to nearest insertion. The nearest addition method takes a traveling salesman graph (N, d) with n nodes and constructs a sequence of subtours $T_1, T_2, \cdots, T_n$ so that

1. T_1 consists of a single node a_0;
2. for each $i < n$ there are nodes a_i in $N - T_i$ and b_i in T_i such that

$$d(b_i, a_i) = \min\{d(y, x) \text{ for } y \text{ in } T_i \text{ and } x \text{ in } N - T_i\} \tag{6.1}$$

and T_{i+1} is constructed from T_i by deleting some edge (c, b_i) from T_i and adding the two edges (c, a_i) and (b_i, a_i);

3. T_n is the approximation.

At each step of the procedure, the closest node is selected and added to the subtour next to the node to which it is the closest.

The increase in length between T_i and T_{i+1} is

$$d(c, a_i)+d(b_i, a_i)-d(c, b_i). \tag{6.2}$$

From the triangle inequality

$$d(c, a_i)\leqq d(c, b_i)+d(b_i, a_i) \tag{6.3}$$

so that (6.2) is bounded by $2 \cdot d(b_i, a_i)$. The set of edges (b_i, a_i) selected in accordance with (6.1) is identical to the set of edges that would be selected for the nearest insertion method in accordance with (4.2), and constitutes a minimal spanning tree. Therefore results similar to Lemma 3 and Theorem 4 apply, and the ratio of the obtained tour length to the optimal tour length is bounded by 2.

Another approximation method is one we call *nearest merger*. First, given two disjoint subtours (i.e., subtours having no nodes in common) T_1 and T_2, their merger MERGE(T_1, T_2) is defined as follows:

a) If T_1 consists of a single node k, then

$$\text{MERGE}(T_1, T_2)=\text{TOUR}(T_2, k)$$

else if T_2 consists of a single node k_1, then

$$\text{MERGE}(T_1, T_2)=\text{TOUR}(T_1, k).$$

b) If T_1 and T_2 each contain at least two nodes, let a, b, c, d be nodes such that (a, b) is an edge in T_1, (c, d) is an edge in T_2 and

$$d(a, c)+d(b, d)-d(a, b)-d(c, d) \tag{6.4}$$

is minimized. Then MERGE(T_1, T_2) is the tour obtained from T_1 and T_2 by deleting (a, b) and (c, d) and adding (a, c) and (b, d).

The *nearest merger* method takes a problem (N, d) with n nodes and constructs a sequence $S_1, \cdots, S_n$ such that each S_i is a set of $n-i+1$ disjoint subtours covering all the nodes in N. The sequence is constructed as follows:

1. S_1 consists of n subtours, each containing a single node.
2. For each $i<n$, find an edge (a_i, c_i) such that

$$d(a_i, c_i)=\min\{d(x, y) \text{ for } x \text{ and } y \text{ in different subtours in } S_i\}. \tag{6.5}$$

Then S_{i+1} is obtained from S_i by merging the subtours containing a_i and c_i.

At each step in the procedure, the two closest subtours are merged.

Observe that in a merger corresponding to (6.4), from the triangle inequality

$$d(b, d)\leqq d(b, a)+d(a, c)+d(c, d)$$

so that (6.4) is bounded by $2 \cdot d(a, c)$. Also observe that the set of edges (a_i, c_i) chosen in accordance with (6.5) form a minimal spanning tree (Kruskal [9]). From these facts, results similar to Lemma 3 and Theorem 4 can be proved for nearest merger, and so the ratio of the obtained tour length to the optimal tour length is bounded by 2.

We also observe that Theorem 5 is also true for both nearest addition and nearest merger. For the examples in the proof of Theorem 5 both of these methods produce the same approximate tour as nearest insertion and cheapest insertion.

One possible way to improve nearest insertion, cheapest insertion, and nearest addition is to repeat each of these methods for each possible starting node and then take the minimum solution among these. However, for the examples in the proof of Theorem 5, these methods produce tours of the same length for all starting nodes. Therefore the approach of trying all starting nodes does not improve the worst case ratio.

The methods of this section and § 4 are all proven to have constant bounds because of comparisons with the minimal spanning tree. There are also known bounded methods which actually construct a tour by first constructing the minimal spanning tree. One widely known but unpublished method is to construct the minimal spanning tree, double its edges to obtain an Eulerian circuit containing each point at least once, and then make the circuit into a tour by removing extra occurrences of each node. This method also has an upper bound of 2.

The method of Christophides [2] also starts with the minimal spanning tree, but this is converted into an Eulerian circuit by solving the matching problem among the nodes of odd order. This method has an upper bound of $\frac{3}{2}$, an improvement on the bounds for the methods studied here. However, the running time of this method is n^3, which is slower than the n^2 methods studied here.

7. *k*-optimality. One approach to obtaining approximate solutions is to first find some tour and then perturb it somewhat to see if a better tour results. If a better tour does result, the original tour is discarded and perturbations on the new tour are tried. Methods of this kind are described in Croes [3], Lin [10], Reiter and Sherman [14], Roberts and Flores [15] and Nicholson [12]. The local optimum obtained by these perturbation methods can be further adjusted to obtain a global optimum (Croes [3]). Lin and Kernighan [11] generalize these techniques in a powerful way.

Define a *k-change* of a tour as the deletion of k edges and their replacement by k other edges so that another tour is obtained.

Define a tour as *k-optimal* (Lin [10]) if no k-change produces a better tour.

Lin [10] describes a method whereby several random initial tours are obtained, each is improved until a 3-optimal tour is obtained and the best of these 3-optimal tours is used.

In this section, we investigate how far a k-optimal tour can be from the optimal tour.

THEOREM 6. *For each $n \geqq 8$ there exists a traveling salesman graph having a tour which is k-optimal for all $k \leqq n/4$, and for which the length of that tour,* LOCALOPT, *satisfies*

$$\frac{\text{LOCALOPT}}{\text{OPTIMAL}} = 2 \cdot \left(1 - \frac{1}{n}\right). \tag{7.1}$$

Proof. The example is the graph (N_n, d_n) and tour T_n constructed in the proof of Theorem 5. In particular, the tour shown in Fig. 4 will be shown to be 2-optimal.

For each n, define the set of edges

$$E_n = \{(1, n)\} \cup \{(i, i+1) \text{ for } 1 \leqq i < n\}$$

E_n is the set of edges which have length one. Because of the way function d_n is defined, each pair of points (a, b) from N_n is connected by some path in E_n of length equal to $d_n(a, b)$. For each tour T, there is a circuit $\alpha(T)$ obtained by replacing each edge of T by a path of equal length from E_n. Circuit $\alpha(T)$ has the same length as T and visits each node at least once. Circuit $\alpha(T_n)$ visits node 1 and n once and every other node twice.

For each edge e in E_n and each tour T, we let COUNT(e, T) be the number of times edge e occurs in circuit $\alpha(T)$. For tour T_n we have

$$\text{COUNT}((i, i+1), T_n) = 2 \quad \text{for } 1 \leqq i < n, \tag{7.2}$$

$$\text{COUNT}((1, n), T_n) = 0. \tag{7.3}$$

Because the edges of E_n are of unit length, the length $L(T)$ of tour T is given by

$$L(T) = \sum_{e \text{ in } E_n} \text{COUNT}(e, T). \tag{7.4}$$

We say that a tour T is *even* if COUNT(e, T) is even for all e in E_n. We say that a tour T is *odd* if COUNT(e, T) is odd for all e in E_n. We next show that any tour must be either odd or even.

By construction, each node a is the endpoint of exactly two edges of E_n, namely $(a, a+1)$ and $(a, a-1)$ (mod n). Since each occurrence of a in $\alpha(T)$ is associated with two edges of $\alpha(T)$

$$\text{COUNT}((a, a+1), T) + \text{COUNT}((a, a-1), T) = 2 \cdot j_a \tag{7.5}$$

where j_a is the number of times node a occurs in circuit $\alpha(T)$. Therefore, COUNT$((a, a+1), T)$ and COUNT$((a, a-1), T)$ sum to an even number and are either both even or both odd. Since the edges in E_n form a connected graph, if T were neither odd nor even, some node would have one incident edge with an odd count and its other incident edge with an even count. This contradicts (7.5), so T is either odd or even.

For any tour T, there can be only one edge e in E_n such that COUNT$(e, T) = 0$ since otherwise the tour could not be connected. Therefore, T_n with its one edge of count 0 and other edges of count 2 (see (7.2) and (7.3)) is the shortest even tour. Consequently, any tour improving on T_n must be odd.

Now suppose that tour T_n is changed by a k-change to an odd tour. Since the largest edges of T_n are at length 2, the decrease resulting from deleting k edges is at most $2k$. Since at most $2k$ of the counts in $\alpha(T)$ are reduced, and since E_n has n edges, $n-2k$ edges of E_n do not get their counts decreased. When edges are added to complete the k-change, the counts for the edges not decreased must in fact be increased in order to change from an even number to an odd number. Therefore, the increases are at least $n-2k$. If the k-change is to improve the tour length, the decreases must be greater than the increases or

$$2k > n - 2k.$$

This inequality is only true when $k > n/4$ so T is indeed k-optimal for $k \leqq n/4$.

We already know from the proof of Theorem 5 that T_n and the optimal tour have ratio $2 \cdot (1-1/n)$ so the theorem is proved.

COROLLARY. *For any k and n such that $4 \cdot k \leqq n$, the nearest insertion and cheapest insertion methods can result in a k-optimal tour such that*

$$\frac{\text{INSERT}}{\text{OPTIMAL}} = 2 \cdot \left(1 - \frac{1}{n}\right).$$

Proof. We have just shown that the example used to establish Theorem 5 is also k-optimal if $4 \cdot k \leqq n$.

REFERENCES

[1] M. BELLMORE AND G. L. NEMHAUSER, *The traveling salesman problem: A survey*, Operations Res., 16 (1968), pp. 538–558.

[2] N. CHRISTOFIDES, *Worst-case analysis of a new heuristic for the traveling salesman problem*, Symp. on New Directions and Recent Results in Algorithms and Complexity (April 1976), Carnegie–Mellon University, Pittsburgh.

[3] G. A. CROES, *A method for solving traveling salesman problems*, Operations Res., 6 (1958), pp. 791–812.

[4] R. FLOYD, *Algorithm 97, Shortest path*, Comm. ACM, 5 (1962), p. 345.

[5] J. GAVETT, *Three heuristic rules for sequencing jobs to a single production facility*, Management Sci., 11 (1965), pp. 166–176.

[6] W. W. HARDGRAVE AND G. L. NEMHAUSER, *On the relation between the traveling salesman and the longest path problem*, Operations Res., 10 (1962), pp. 647–657.

[7] L. L. KARG AND G. L. THOMPSON, *A heuristic approach to solving traveling salesman problems*, Management Sci., 10 (1964), pp. 225–248.

[8] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–103.

[9] J. B. KRUSKAL, *On the shortest spanning subtree of a graph and the traveling salesman problem*, Proc. Amer. Math. Soc., 2 (1956), pp. 48–50.

[10] S. LIN, *Computer solution of the traveling salesman problem*, Bell System Tech. J., 44 (1965), pp. 2245–2269.

[11] S. LIN AND B. W. KERNIGHAN, *An effective heuristic algorithm for the traveling salesman problem*, Operations Res., 21 (1973), pp. 498–516.

[12] T. A. J. NICHOLSON, *A sequential method for discrete optimization problems and its application to the assignment, traveling salesman, and three machine scheduling problems*, J. Inst. Math. Appl., 3 (1967), pp. 362–375.

[13] R. C. PRIM, *Shortest connection networks and some generalizations*, Bell System Tech. J., 36 (1957), pp. 1389–1401.

[14] S. REITER AND G. SHERMAN, *Discrete optimizing*, J. Soc. Indust. Appl. Math., 13 (1965), pp. 864–889.

[15] S. M. ROBERTS AND B. FLORES, *An engineering approach to the traveling salesman problem*, Management Sci., 13 (1966), pp. 269–288.

[16] S. SAHNI AND T. GONZALES, *P-complete problems and approximate solutions*, IEEE Fifteenth Ann. Symp. on Switching and Automata Theory (1974), pp. 28–32.

SIAM J. COMPUT.
Vol. 6, No. 3, September 1977

FAST ALGORITHMS FOR PARTIAL FRACTION DECOMPOSITION*

H. T. KUNG† AND D. M. TONG‡

Abstract. The partial fraction decomposition of a proper rational function whose denominator has degree n and is given in general factored form can be done in $O(n \log^2 n)$ operations in the worst case. Previous algorithms require $O(n^3)$ operations, and $O(n \log^2 n)$ operations for the special case where the factors appearing in the denominator are all linear.

Key words. fast algorithms, partial fraction decomposition, computational complexity

1. Introduction.

Let

$$\frac{P(x)}{\prod_{i=1}^{k} Q_i^{l_i}(x)}$$

be a given fraction, where the P, Q_i are polynomials and the l_i are positive integral exponents such that

1. $\deg P < \sum_{i=1}^{k} l_i \cdot \deg Q_i = n$, and
2. $Q_1, \cdots, Q_k$ are relatively prime.

The *general partial fraction decomposition problem* (*general* PF *problem*) is to compute the coefficients of the polynomials $C_{i,j}$ for $i = 1, \cdots, k$ and $j = 1, \cdots, l_i$ such that

$$\frac{P(x)}{\prod_{i=1}^{k} Q_i^{l_i}(x)} = \sum_{i=1}^{k} \sum_{j=1}^{l_i} \frac{C_{i,j}(x)}{Q_i^j(x)}$$

with $\deg C_{i,j} < \deg Q_i$ for all i, j. The existence and uniqueness of the polynomials $C_{i,j}$ are well known (see, e.g., van der Waerden (1953, § 29). There are enormous applications of partial fractions in applied mathematics and in network theory (see, e.g., Henrici (1974, Chap. 7) and Weinberg (1962)). This paper gives fast algorithms for solving the general partial fraction decomposition problem when n is large.

Previous algorithms for the problem usually involve solving systems of linear equations (see Henrici (1974, Chap. 7) for a nice summary). Hence they take $O(n^3)$ arithmetic operations, or $O(n^{2.81})$ operations if Strassen's method (Strassen (1969)) is used. For the special case that the Q_i have either degree one or two, many algorithms were known: see, e.g., Schwatt (1924, Chap. VIII), Turnbull (1927), Hazony and Riley (1959), Pottle (1964), Pessen (1965), Brugia (1965), Moad (1966), Valentine (1967), Wehrhahn (1967), Karni (1969) and Linnér (1974). But these algorithms still take $O(n^2)$ or more operations.

Recently Chin and Ullman (1975) showed that in case that all the Q_i have degree one the problem can be done in $O((n \log n)^{3/2})$ operations. This bound

* Received by the editors February 19, 1976. This work was supported in part by the National Science Foundation under Grant MCS75-222-55 and the Office of Naval Research under Contract N00014-76-C-0370, NR044-422.

† Department of Computer Science, Carnegie–Mellon University, Pittsburgh, Pennsylvania 15213.

‡ Department of Mathematics, Kalamazoo College, Kalamazoo, Michigan 49001.

was further improved by Chin in his thesis (Chin (1975)). He showed that if the Q_i are all linear, then the problem can be done in $O((\log k)\cdot(n \log n))$ operations. However, the assumption that the Q_i are all linear factors is crucial in his methods. Hence the problem of solving the general PF problem (without assuming that the Q_i are linear) in $O(n^2)$ operations is stated as an unsolved problem in his thesis. Note that the general PF problem does occur frequently in practice. For example, if we work over the field of real numbers, then the factors Q_i certainly can have either degree one or two. (See also Grau (1971) and Henrici (1971) for more examples.) *In this paper, we show that the general* PF *problem can be done in* $O((\log n)\cdot M(n))$ *operations in the worst case.* $M(n)$ *is any upper bound on the number of operations needed to multiply two n-th degree polynomials, which satisfies some mild regularity condition* (see § 2). In particular, if an FFT algorithm is used for polynomial multiplication (see, e.g., Knuth (1969, § 4.6.4), Borodin and Munro (1975)), then we have $M(n)=O(n \log n)$, which satisfies the regularity condition, and hence the general PF problem can be done in $O(n \log^2 n)$ operations. Moreover, we note that for the special case where the Q_i are all linear, our approach will lead to Chin's $O((\log k)\cdot(n \log n))$ algorithm.

Basic assumptions and preliminary lemmas used in this paper are introduced in § 2. In § 3, the solution of the general PF problem is reduced to the solution of two simpler problems, Problem P1 and Problem P2, and precise statements of the main results of the paper are given. An algorithm, based on a *new theorem* (Theorem 4.1), for solving Problem P1 is presented in § 4. Section 5 contains an algorithm for solving Problem P2. Finally, an important special case of Problem P2 is solved in § 6.

2. Basic assumptions and preliminary lemmas. We assume throughout the paper that polynomials are over some field K, are denoted by upper case letters, and are given in the form $P(x)=\sum p_i x^i$ where $p_i \in K$. To compute P or $P(x)$ means to find all the coefficients of P. We assume that $M(n)$ is an upper bound on the number of operations needed to multiply two nth degree polynomials. Given relatively prime polynomials A_1, A_2 with $\deg A_1$, $\deg A_2 \leqq n$, let $F(n)$ be an upper bound on the number of operations to find polynomials F_1, F_2 such that

$$F_2\cdot A_1+F_1\cdot A_2=1$$

with $\deg F_1<\deg A_1$ and $\deg F_2<\deg A_2$. The existence and uniqueness of F_1 and F_2 are well-known (see, e.g., van der Waerden (1953, § 29)).

Let Z^+ be the set of all nonnegative integers and let $G: Z^+\rightarrow Z^+$ be a nondecreasing function. We say G satisfies *Condition* C, if

$$G(n)=n\cdot H(n)$$

for some nondecreasing function H: $Z^+\rightarrow Z^+$. *We assume that M satisfies Condition* C. Similar regularity conditions are usually assumed (see, e.g., Aho, Hopcroft and Ullman (1974, p. 280), Brent and Kung (1976) and Moenck (1973b)). There are many algorithms for polynomial multiplication. For example, the classical algorithm gives $M(n)=c_1n^2$, binary splitting multiplication gives $M(n)=c_2n^{1.585}$, and if the field K is algebraically closed, then FFT multiplication gives $M(n)=c_3n \log n$, where c_1, c_2, c_3 are positive constants (see e.g., Fateman (1974)). In all cases M satisfies Condition C. In fact all we need in this paper are some consequences of Condition C. Hence it is possible to weaken our assumption on M, if one wishes to do so.

Let $D(n)$ be the number of operations needed to divide a polynomial of degree $2n$ by a polynomial of degree n. Then using Newton's method and the fact that M satisfies Condition C, one can show the following lemma (see, e.g., Borodin and Munro (1975) and Kung (1974)).

LEMMA 2.1. $D(n)=O(M(n))$.

Using the algorithm EGCD in Moenck (1973a), which is a generalization of an algorithm due to Schöenhage (1971) for integer GCDs, one can show the following lemma.

LEMMA 2.2. $F(n)=O((\log n)\cdot M(n))$.

We shall assume that F satisfies the condition that

$$\sum F(n_i) \leqq F\left(\sum n_i\right)$$

for any $n_i \in Z^+$. Clearly, if $F(n)=c\cdot(\log n)\cdot M(n)$ for some positive constant c as in Lemma 2.2, then F satisfies the condition. In fact, the required condition in F is satisfied as long as F satisfies Condition C.

3. Problems P1, P2 and statement of results. Consider the following two instances of the general PF problem defined in § 1.

PROBLEM P1. (This is the general PF problem with $l_i=1$ for all i.) Given the fraction $P/\prod_{i=1}^k R_i$ where the R_i are relatively prime and

$$\deg P < \sum_{i=1}^{k} \deg R_i = n,$$

compute the polynomials $C_1, \cdots, C_k$ such that

$$\frac{P(x)}{\prod_{i=1}^k R_i(x)} = \sum_{i=1}^{k} \frac{C_i(x)}{R_i(x)} \tag{3.1}$$

with $\deg C_i < \deg R_i$ for all i.

The decomposition (3.1) is called the incomplete partial fraction decomposition by Henrici (1971), (1974, Chap. 7). Note also that efficient algorithms for solving Problem P1 will furnish efficient procedures for factoring polynomials, as observed by Grau (1971).

PROBLEM P2. (This is the general PF problem with $k=1$.) Given the fraction P/Q^l where $\deg P < l\cdot \deg Q$ compute the polynomials $C_1, \cdots, C_l$ such that

$$\frac{P(x)}{Q^l(x)} = \sum_{j=1}^{l} \frac{C_j(x)}{Q^j(x)}$$

with $\deg C_j < \deg Q$ for all j.

The following lemma essentially shows that fast algorithms for Problems P1 and P2 will lead to fast algorithms for the general PF problem. *Define* $T(k,n)$, $T_1(k,n)$ *and* $T_2(l, \deg Q)$ *to be the number of operations needed to solve the general* PF *problem, Problem* P1 *and Problem* P2, *respectively.*

LEMMA 3.1.

$$T(k, n) \leqq T_1(k, n) + \sum_{i=1}^{k} [T_2(l_i, \deg Q_i) + O(M(l_i \cdot \deg Q_i))].$$

Proof. The result follows from the observation that general PF problem can be solved in the following way:

1. Multiply $Q_i^{l_i}(x)$ out for $i = 1, \cdots, k$. Let the expansion of $Q_i^{l_i}(x)$ be $R_i(x)$ for all i.

2. Solve Problem P1 for the fraction $P/\prod_{i=1}^{k} R_i$ and obtain the polynomials C_i satisfying (3.1).

3. Solve Problem P2 for the fractions $C_i/Q_i^{l_i}$, $i = 1, \cdots, k$.

Note that each $Q_i^{l_i}(x)$ can be computed in $O(M(l_i \cdot \deg Q_i))$ operations by an algorithm in Brent (1976, § 13). □

We summarize our results on $T_1(k, n)$ and $T_2(l, \deg Q)$ in the following:

(i) $T_1(k, n) \leqq F(n) + O((\log k) \cdot M(n))$. (Theorem 4.2)

(ii) $T_1(k, n) = O((\log k) \cdot (n \log n))$, when the $R_i(x)$ is given in the form $(x - z_i)^{m_i}$ for all i. (Theorem 4.3)

(iii) $T_2(l, \deg Q) = O((\log l) \cdot M(l \cdot \deg Q))$. (Theorem 5.1)

(iv) $T_2(l, \deg Q) = O(l \log l)$, when $\deg Q \leqq 2$. (Theorems 6.1 and 6.2)

We have the following results for the general partial fraction decomposition problem.

THEOREM 3.1. *The general* PF *problem can be done in* $F(n) + O((\log k) \cdot M(n)) + O((\log l) \cdot M(n))$ *operations, where* $l = \max(l_1, \cdots, l_k)$.

Proof. Note that

$$\begin{aligned}\sum_{i=1}^{k} (\log l_i) \cdot M(l_i \cdot \deg Q_i) \\ &\leqq (\log l) \sum_{i=1}^{k} l_i \cdot \deg Q_i \cdot H(l_i \cdot \deg Q_i) \\ &\leqq (\log l) \cdot n \cdot H(n) = (\log l) \cdot M(n).\end{aligned}$$

The result follows from (i), (iii) and Lemma 3.1. □

COROLLARY 3.1. *The general* PF *problem can be done in* $O(n \log^2 n)$ *operations.*

Proof. Note that in Theorem 3.1, $k \leqq n$ and $l \leqq n$. The result follows from the theorem and Lemma 2.2 by letting $M(n) = O(n \log n)$. □

$O(n \log^2 n)$ is the best asymptotic bound known for the general PF problem.

THEOREM 3.2. *The general* PF *problem can be done in* $O((\log k) \cdot (n \log n))$ *operations, if* $Q_i(x) = x - z_i$, *for* $i = 1, \cdots, k$.

Proof. The result follows from (ii), (iv) and Lemma 3.1. □

The bound in Theorem 3.2 was obtained previously by Chin (1975). We include it here just to show that his result will emerge as a special case in our general approach. See the remarks at the end of § 4.

4. An algorithm for problem P1.

We first assume that $P(x) \equiv 1$ in Problem P1. Thus we want to find $A_1, \cdots, A_k$ such that

$$\frac{1}{\prod_{i=1}^{k} R_i(x)} = \sum_{i=1}^{k} \frac{A_i(x)}{R_i(x)}$$

with $\deg A_i < \deg R_i$ for all i. Note that

$$1 = \sum_{i=1}^{k} \left[A_i(x) \prod_{\substack{j=1\\ j\neq i}}^{k} R_j(x) \right]. \tag{4.1}$$

Define

$$R(x) = \sum_{i=1}^{k} \left(\prod_{\substack{j=1\\ j\neq i}}^{k} R_j(x) \right),$$

and for each $i = 1, \cdots, k$, define B_i, D_i by

$$R(x) = B_i(x)R_i(x) + D_i(x) \tag{4.2}$$

where $\deg D_i < \deg R_i$. Note that $D_i(x) \not\equiv 0$, since the R_i are relatively prime. Suppose that $\deg D_i \geqq 1$, i.e., $D_i(x)$ is not a constant. Then (4.2) implies that D_i and R_i are relatively prime, since R_i and R are relatively prime. Hence there exist unique polynomials $\tilde{A}_i$ and E_i such that

$$\tilde{A}_i(x)D_i(x) + E_i(x)R_i(x) = 1 \tag{4.3}$$

with $\deg \tilde{A}_i < \deg R_i$ and $\deg E_i < \deg D_i$. The following theorem appears to be new.

THEOREM 4.1. *For $i = 1, \cdots, k$, if $D_i(x) \equiv d_i$ for some constant d_i, then A_i is the constant $1/d_i$; else $A_i = \tilde{A}_i$.*

Proof. We classify the zeros of R_i according to their multiplicities. Let Z_m be the set of zeros of R_i which have multiplicity m. (The zeros exist in an algebraically closed extension field of K.) Clearly, we have that

$$\sum m \cdot |Z_m| = \deg R_i, \quad \text{where } |Z_m| \text{ is the number of elements in } Z_m, \tag{4.4}$$

and that if $z \in Z_m$ then

$$R_i^{(h)}(z) = 0 \quad \text{for } h = 0, \cdots, m-1. \tag{4.5}$$

Taking derivatives of (4.1) and (4.3), and using (4.5), one can easily show that

$$\sum_{q=0}^{h} \binom{h}{q} A_i^{(q)}(z) \cdot \left(\prod_{\substack{j=1\\ j\neq i}}^{k} R_j \right)^{(h-q)} (z) = \delta_{0,h}, \tag{4.6}$$

$$\sum_{q=0}^{h} \binom{h}{q} \tilde{A}_i^{(q)}(z) \cdot D_i^{(h-q)}(z) = \delta_{0,h} \tag{4.7}$$

for $z \in Z_m$ and $h = 0, \cdots, m-1$, where $\delta_{0,h} = 1$ if $h = 0$ and $\delta_{0,h} = 0$ otherwise. (Derivatives are represented by the superscripts.) Note that by (4.2) and (4.5),

$$\begin{aligned} D_i^{(h-q)}(z) &= R^{(h-q)}(z) \\ &= \left(\prod_{\substack{j=1\\ j\neq i}}^{k} R_j \right)^{(h-q)} (z) \end{aligned} \tag{4.8}$$

for $z \in Z_m$, $h = 0, \cdots, m-1$ and $q = 0, \cdots, h$. Suppose that $D_i(x) \equiv d_i$ for some constant d_i. Then by (4.6) and (4.8),

$$A_i^{(h)}(z) \cdot d_i = \delta_{0,h} \tag{4.9}$$

for $z \in Z_m$, $h = 0, \cdots, m-1$. Since $d_i \neq 0$ and $\deg A_i < \deg R_i$, A_i is uniquely determined by the Hermite interpolation problem defined by (4.9). Hence $A_i(x) \equiv 1/d_i$. On the other hand, suppose that deg $D_i \geqq 1$. Because the R_i are relatively prime,

$$D_i(z) = \left(\prod_{\substack{j=1 \\ j \neq i}}^{k} R_j \right)(z) \neq 0 \tag{4.10}$$

for $z \in Z_m$. By (4.6), (4.7), (4.8) and (4.10) it is easy to see that A_i and $\tilde{A}_i$ are deteremined by the *same* Hermite interpolation problem. This implies that $A_i = \tilde{A}_i$. □

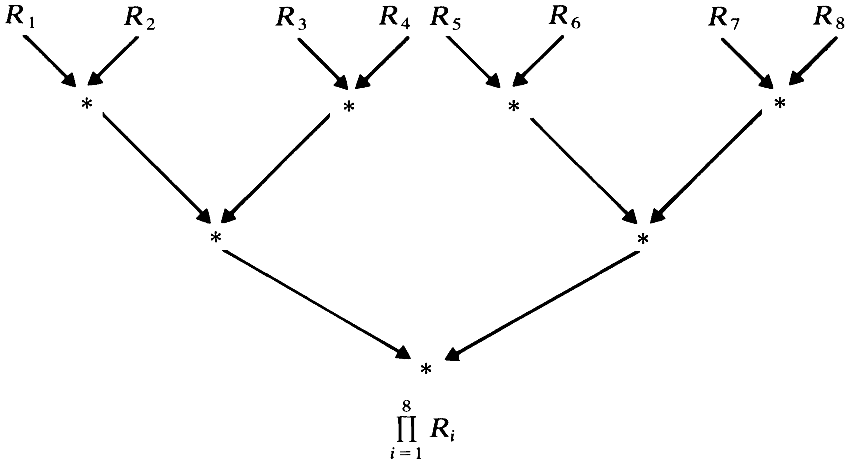
By Theorem 4.1 the following algorithm can be used for computing $A_i(x)$ for $i = 1, \cdots, k$.

ALGORITHM 4.1.

1. Compute $R(x)$.
2. Compute $D_i(x)$ for $i = 1, \cdots, k$.
3. For $i = 1, \cdots, k$, if $D_i(x) \equiv d_i$ for some constant d_i then set $A_i(x) \leftarrow 1/d_i$ else compute $A_i(x)$ by solving (4.3).

In the following we study the number of operations needed by the algorithm.

It is well known that $\prod_{i=1}^{k} R_i(x)$ can be computed by using a binary splitting scheme, which is illustrated as follows for the case $k = 8$:



LEMMA 4.1. *By using the binary splitting, $\prod_{i=1}^{k} R_i(x)$ and all the intermediate results such as $\prod_{i=1}^{2} R_i(x)$ and $\prod_{i=5}^{8} R_i(x)$ can be computed in $O((\log k) \cdot M(n))$ operations.*

Proof. Note that the sum of the degrees of all the polynomials at any level of the tree is n. Hence each level takes $M(n)$ operations, since M satisfies Condition C. The result then follows from the fact that the height of the tree is $\lceil \log_2 k \rceil$. □

LEMMA 4.2. *$R(x)$ can be computed in $O((\log k)\cdot M(n))$ operations.*

Proof. We shall again use the binary splitting technique. We may assume that k is a power of 2. It is easy to check that

$$\sum_{i=1}^{k}\Bigg(\prod_{\substack{j=1\\ j\neq i}}^{k} R_j\Bigg)=\Bigg(\prod_{j=k/2+1}^{k} R_j\Bigg)\cdot\sum_{i=1}^{k/2}\Bigg(\prod_{\substack{j=1\\ j\neq i}}^{k/2} R_j\Bigg)+\Bigg(\prod_{j=1}^{k/2} R_j\Bigg)\cdot\sum_{i=k/2+1}^{k}\Bigg(\prod_{\substack{i=k/2+1\\ j\neq i}}^{k} R_j\Bigg).$$

This gives us a recursive algorithm for computing R. By Lemma 4.1, we may assume that all the products such as $\prod_{j>k/2} R_j$ and $\prod_{j\leqq k/2} R_j$ needed by the algorithm have been precomputed. The result again follows from the fact that the sum of the degrees of all polynomials at any level of the associated binary tree is n. □

LEMMA 4.3. *$D_1(x),\cdots,D_k(x)$ can be computed in $O((\log k)\cdot M(n))$ operations.*

Proof. We may assume that k is a power of 2. Note that if we use divisions to obtain V_1 and V_2 such that

$$R(x)=U_1(x)\cdot\prod_{j=1}^{k/2} R_i(x)+V_1(x),$$

$$R(x)=U_2(x)\cdot\prod_{j=k/2+1}^{k} R_i(x)+V_2(x),$$

where $\deg V_1<\deg\prod_{j=1}^{k/2} R_i$ and $\deg V_2<\deg\prod_{j=k/2+1}^{k} R_i$, then the problem of computing D_i from R for $i=1,\cdots,k$ is reduced to the problems of computing D_i from V_1 for $i=1,\cdots,k/2$ and computing D_i from V_2 for $i=k/2+1,\cdots,k$. This again gives us a recursive procedure. Using the fact that $D(n)=O(M(n))$ (Lemma 2.1), the lemma can be proved by the same argument as used in the proofs of Lemmas 4.1 and 4.2. □

LEMMA 4.4. *$A_1(x),\cdots,A_k(x)$ can be computed in $F(n)$ operations.*

Proof. Since $\deg D_i<\mathit{deg}\, R_i$, the $A_i(x)$ and $E_i(x)$ satisfying (4.3) can be computed in $F(\deg R_i)$ operations. Hence all the A_i can be computed in

$$\sum_{i=1}^{k} F(\deg R_i)\leqq F\Bigg(\sum_{i=1}^{k}\deg R_i\Bigg)=F(n)$$

operations. □

By Lemmas 4.2, 4.3 and 4.4, we know that Algorithm 4.1 can be done in $F(n)+O((\log k)\cdot M(n))$ operations. After the A_i have been computed, we can solve Problem P1 without assuming $P(x)\equiv 1$ in $O((\log k)\cdot M(n))$ operations by the following method: For $i=1,\cdots,k$,

1. compute $K_i(x)$ such that

$$P(x)=J_i(x)R_i(x)+K_i(x)$$

with $\deg K_i<\deg R_i$, for some J_i,

2. compute $L_i(x)=K_i(x)\cdot A_i(x)$ and $C_i(x)$ such that

$$L_i(x)=N_i(x)R_i(x)+C_i(x)$$

with $\deg C_i<\deg R_i$, for some N_i.

Note that

$$\begin{aligned}\frac{P}{\prod R_i}&=\sum\left(\frac{P}{R_i}\right)A_i\\ &=\sum\left(J_i+\frac{K_i}{R_i}\right)A_i\\ &=\sum J_iA_i+\sum\frac{L_i}{R_i}\\ &=\sum J_iA_i+\sum N_i+\sum\frac{C_i}{R_i}\end{aligned}$$

Since $P/\prod R_i$ is a proper fraction, $\sum J_iA_i+\sum N_i$ must be zero. Therefore the C_i are the desired solution. Since $\deg P<n$, by the same argument as used in the proof of Lemma 4.3, $K_i(x)$ for $i=1,\cdots,k$ can be computed in $O((\log k)\cdot M(n))$ operations. A_i and K_i have degree at most $\deg R_i$, so $C_i(x)$ can be computed in $O(M(\deg R_i))$ operations. This implies that $C_1(x),\cdots,C_k(x)$ can be computed in $O(M(n))$ operations. Therefore, we have shown the following

THEOREM 4.2. *Problem* P1 *can be done in*

$$F(n)+O((\log k)\cdot M(n))$$

operations.

We now consider the special case where the $R_i(x)$ is given in the form $(x-z_i)^{m_i}$ for $i=1,\cdots,k$. In this case the A_i satisfying (4.3), i.e.,

$$A_i(x)D_i(x)+E_i(x)(x-z_i)^{m_i}=1,$$

can be computed easily in the following way. Let $\hat{A}_i(x)=A_i(x+z_i)$, $\hat{D}_i(x)=D_i(x+z_i)$, etc. Then

$$\hat{A}_i(x)\hat{D}_i(x)+\hat{E}_i(x)x^{m_i}=1.$$

This implies that

$$\hat{A}_i(x)\hat{D}_i(x)\equiv 1 \pmod{x^{m_i}}. \tag{4.11}$$

Hence we have the following algorithm for computing A_i:

ALGORITHM 4.2.

1. Compute $\hat{D}_i(x)$ such that $\hat{D}_i(x)=D_i(x+z_i)$.
2. Compute $\hat{A}_i(x)$ from (4.10).
3. Compute $A_i(x)$ such that $A_i(x)=\hat{A}_i(x-z_i)$.

Step 1 is equivalent to evaluating D_i and all its derivatives at z_i. Aho, Steiglitz and Ullman (1975) and Vari (1974) have independently shown that this can be done in $O(m_i\log m_i)$ operations. Similarly, step 3 can be done in $O(m_i\log m_i)$ operations. Step 2 involves a division, which is $O(m_i\log m_i)$ by Lemma 2.1. Since $\sum_{i=1}^k m_i\log m_i=O(n\log n)$, by Theorem 4.2 with $M(n)=O(n\log n)$ we have proved the following

THEOREM 4.3. *Problem* P1 *with* $R_i(x)$ *given by* $(x-z_i)^{m_i}$ *for* $i=1,\cdots,k$ *can be done in*

$$O((\log k)\cdot(n\log n))$$

operations.

Suppose that we solve the general PF problem for $1/\prod_{i=1}^k (x-z_i)^{l_i}$ by solving Problem P1 for $1/\prod_{i=1}^k R_i(x)$ with $R_i(x)=(x-z_i)^{l_i}$. Then we need not perform step 3 of Algorithm 4.2 since the solution of the general PF problem is given by the coefficients of the $\hat{A}_i$. It turns out that this is exactly Chin's $O((\log k)\cdot(n\log n))$ algorithm for solving the general PF problem for $1/\prod_{i=1}^k (x-z_i)^{l_i}$. A similar observation can also be made for the case of solving the general PF problem for $P/\prod_{i=1}^k (x-z_i)^{l_i}$ with $P(x)\not\equiv 1$.

5. An algorithm for Problem P2. Note that using division, we have

$$\begin{aligned}\frac{P}{Q^l}&=\frac{1}{Q^{\lceil l/2\rceil}}\cdot\frac{P}{Q^{\lfloor l/2\rfloor}}\\&=\frac{1}{Q^{\lceil l/2\rceil}}\cdot\left(P_1+\frac{P_2}{Q^{\lfloor l/2\rfloor}}\right)\\&=\frac{P_1}{Q^{\lceil l/2\rceil}}+\frac{1}{Q^{\lceil l/2\rceil}}\cdot\left(\frac{P_2}{Q^{\lfloor l/2\rfloor}}\right).\end{aligned}$$

where $\deg P_1<\lceil l/2\rceil\cdot\deg Q$ and $\deg P_2<\lfloor l/2\rfloor\cdot\deg Q$. Thus, to solve Problem P2 for the fraction P/Q^l, it suffices to do the following:

1. Divide P by $Q^{\lfloor l/2\rfloor}$ and obtain the quotient P_1 and the remainder P_2.
2. Solve Problem P2 for the fractions $P_1/Q^{\lceil l/2\rceil}$ and $P_2/Q^{\lfloor l/2\rfloor}$.

This gives us a recursive prodedure for solving Problem P2. Assume that the expansion of the power such as $Q^{\lceil l/2\rceil}(x)$ and $Q^{\lfloor l/2\rfloor}(x)$ required by the recursive procedure have been precomputed. Let $X(l)$ be the number of operations needed to solve Problem P2. Then the recursive procedure gives

$$X(l)\leqq X(\lceil l/2\rceil)+X(\lfloor l/2\rfloor)+D(l\cdot\deg Q)$$

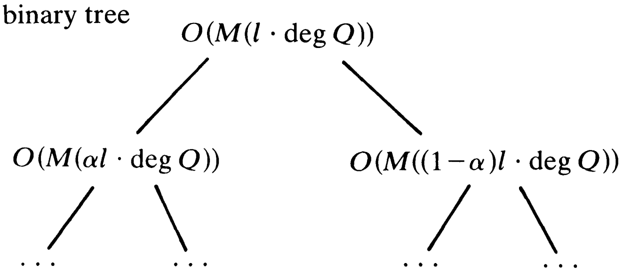
for $l>1$ and $X(1)=0$. Note that

$$\frac{1}{2}\leqq\frac{\lceil l/2\rceil}{l}\leqq\frac{2}{3}$$

for any integer $l\geqq 2$, and that by Lemma 2.1, $D(l\cdot\deg Q)=O(M(l\cdot\deg Q))$. We have

$$X(l)\leqq X(\alpha l)+X((1-\alpha)l)+O(M(l\cdot\deg Q))$$

where α is a variable with its values in $[1/2, 2/3]$. The expansion of the recurrence corresponds to a binary tree



such that $X(l)$ is bounded above by the total value of the nodes inside the tree. Using the fact that M satisfies Condition C, one can easily show that the sum of the values of the nodes at each level is $O(M(l \cdot \deg Q))$. Since $\alpha \in [1/2, 2/3]$, the height of the tree is at most $\lceil \log_{3/2} l \rceil$. Hence

$$X(l) = O((\log l) \cdot M(l \cdot \deg Q)).$$

Now we examine how to compute all the required powers of Q. This can be done by using a recursion based on

$$Q^l = Q^{\lceil l/2 \rceil} \cdot Q^{\lfloor l/2 \rfloor}.$$

The number of operations needed here clearly satisfies the same recurrence as X, and hence is $O((\log l) \cdot M(l \cdot \deg Q))$. We have proved the following

THEOREM 5.1. *Problem* P2 *can be done in*

$$O((\log l) \cdot M(l \cdot \deg Q))$$

operations.

6. A special case for problem P2. The following theorem can be found in Chin and Ullman (1975).

THEOREM 6.1. *Problem* P2 *can be solved in* $O(l \log l)$ *operations if* $\deg Q = 1$.

In this section we extend the theorem to the case that $\deg Q = 2$. Our result is of interest when the underlying field K is the field of real numbers, for in this case irreducible factors can have either degree one or two. We may assume that Q is monic, since this will affect only $O(l)$ operations. Let

$$Q(x) = x^2 + ax + b.$$

By completing the square and letting $y = x + a/2$ and $c = b - a^2/4$, we have

$$\frac{P(x)}{Q^l(x)} = \frac{P(y - a/2)}{(y^2 + c)^l}.$$

Write

$$\begin{aligned} P\left(y - \frac{a}{2}\right) &= \sum_{i=0}^{2l-1} p_i y^i \\ &= (p_0 + p_2 y^2 + \cdots p_{2l-2} y^{2l-2}) \\ &\quad + y(p_1 + p_3 y^2 + \cdots + p_{2l-1} y^{2l-2}) \\ &= P_1(y^2) + y \cdot P_2(y^2), \end{aligned}$$

where $\deg P_1 \leqq l - 1$ and $\deg P_2 \leqq l - 1$. Then

$$\frac{P(x)}{Q^l(x)} = \frac{P_1(y^2)}{(y^2 + c)^l} + y \cdot \frac{P_2(y^2)}{(y^2 + c)^l}.$$

Hence we can solve Problem P2 for $P(x)/Q^l(x)$ by performing the following steps:

1. Compute $p_0, \cdots, p_{2l-1}$.
2. Form $P_1(z) = p_0 + p_2 z + \cdots + p_{2l-2} z^{l-1}$ and $P_2(z) = p_1 + p_3 z + \cdots + p_{2l-1} z^{l-1}$.

Solve Problem P2 for the fractions $P_1(z)/(z+c)^l$ and $P_2(z)/(z+c)^l$, and obtain

$$\frac{P_1(z)}{(z+c)^l}=\sum_{i=1}^{l}\frac{f_i}{(z+c)^i},\qquad \frac{P_2(z)}{(z+c)^l}=\sum_{i=1}^{l}\frac{e_i}{(z+c)^i}.$$

3. Since

$$\begin{aligned}\frac{P(x)}{Q^l(x)}&=\sum_{i=1}^{l}\frac{f_i}{Q^i(x)}+y\cdot\sum_{i=1}^{l}\frac{e_i}{Q^i(x)}\\&=\sum_{i=1}^{l}\frac{f_i}{Q^i(x)}+\left(x+\frac{a}{2}\right)\cdot\sum_{i=1}^{l}\frac{e_i}{Q^i(x)}\\&=\sum_{i=1}^{l}\frac{e_ix+f_i+ae_i/2}{Q^i(x)},\end{aligned}$$

we set $C_i(x)\leftarrow e_ix+f_i+\frac{ae_i}{2}$ for $i=1,\cdots,l$.

By the result of Aho, Steiglitz and Ullman (1975) and Vari (1974) step 1 can be done in $O(l\log l)$ operations. By Theorem 6.1, step 2 can be done in $O(l\log l)$ operations. Step 3 clearly uses $O(l)$ operations. Thus, we have shown the following theorem.

THEOREM 6.2. *Problem* P2 *can be solved in* $O(l\log l)$ *operations if* $\deg Q=2$.

It is an open problem whether Theorem 6.2 holds if $\deg Q>2$.

REFERENCES

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

A. V. AHO, K. STEIGLITZ AND J. D. ULLMAN (1975), *Evaluating polynomials at fixed sets of points*, this Journal, 4, pp. 533–539.

A. BORODIN AND I. MUNRO (1975), *The Computational Complexity of Algebraic and Numerical Problems*, American Elsevier, New York.

R. P. BRENT (1976), *Multiple-precision zero-finding methods and complexity of elementary function evaluation*, Analytic Computational Complexity, J. F. Traub, ed., Academic Press, New York, pp. 151–176.

R. P. BRENT AND H. T. KUNG (1976), *Fast algorithms for manipulating formal power series*, Rep., Computer Sci. Dept. Carnegie–Mellon Univ., Pittsburgh, PA.

O. BRUGIA (1965), *Noniterative method for the partial expansion of a rational function with high order poles*, SIAM Rev. 7, pp. 381–387.

F. Y. CHIN (1975), *Complexity of numerical algorithms for polynomials*, Ph.D. thesis, Dept. of Electrical Engineering, Princeton Univ., Princeton, NJ, October 1975.

F. Y. CHIN AND J. D. ULLMAN (1975), *Asymptotic complexity of partial fraction expansion*, Rep., Computer Sci. Laboratory, Dept. of Electrical Engineering, Princeton Univ., Princeton, NJ.

R. J. FATEMAN (1974), *Polynomial multiplication, powers and asymptotic analysis: Some comments*, this Journal, 3, pp. 196–213.

A. A. GRAU (1971), *The simultaneous Newton improvement of a complete set of approximate factors of a polynomial*, SIAM J. Numer. Anal. 8, pp. 425-438.

D. HAZONY AND J. RILEY (1959), *Evaluating residues and coefficients of high order poles*, IRE Trans. Automatic Control, AC-4, pp. 132–136.

P. HENRICI (1971), *An algorithm for the incomplete decomposition of a rational function into partial fraction*, Z. Angew. Math. Phys., 22, pp. 751–755.

——— (1974), *Applied and Computational Complex Analysis*, vol. 1, Wiley-Interscience, New York.

S. KARNI (1969), *Easy partial fraction expansion with multiple poles*, Proc. IEEE (letters), 57, pp. 231–232.

D. E. KNUTH (1969), *The art of Computer Programming*, vol. 2, Addison-Wesley, Reading, MA.

H. T. KUNG (1974), *On computing reciprocals of power series*, Numer. Math., 22, pp. 341–348.

L. J. P. LINNÉR (1974), *The computation of the kth derivative of polynomials and rational functions in factored form and related matters*, IEEE Trans. Circuits and Systems, CAS-21, pp. 233–236.

M. F. MOAD (1966), *On rational function expansion*, Proc. IEEE (letters), 54, pp. 899–900.

R. T. MOENCK (1973a), *Fast computation of GCDs*, Proc. 5th Annual ACM Symposium on Theory of Computing, May 1973, pp. 142–151.

——— (1973b), *Studies in fast algebraic algorithms*, Ph.D. thesis, Dept. of Computer Science, Univ. of Toronto.

D. W. PESSEN (1965), *Time-saving method for partial fraction expansion of function with one pair of conjugate complex roots*, Proc. IEEE (correspondence), 53, p. 1266.

C. POTTLE (1964), *On the partial fraction expansion of a rational function with multiple poles by digit computer*, IEEE Trans. Circuit Theory (correspondence), CT-11, pp. 161–162.

A. SCHÖNHAGE (1971), *Schnelle Berechnung von Kettenbruchentwicklugen*, Acta Informat., 1, pp. 139–144.

I. J. SCHWATT (1924), *An Introduction to Operations with Series*, Chelsea Publishing Co, New York.

V. STRASSEN (1969), *Gaussian elimination is not optimal*, Numer. Math., 13, pp. 354–356.

H. W. TURNBULL (1927), *Note on partial fractions and determinants*, Proc. Edinburgh Math. Soc. 1, no. 2, pp. 49–54.

C. W. VALENTINE (1967). *A method for partial fraction decomposition*, SIAM Rev., 9, pp. 232–233.

B. L. VAN DER WAERDEN (1953), *Modern Algebra*, vol. 1, Frederick Ungar, New York.

T. M. VARI (1974), *Some complexity results for a class of Toeplitz matrices*, Rep., Dept. of Computer Sci. and Math., York Univ., Toronto.

L. WEINBERG (1962), *Network Analysis and Synthesis*, McGraw-Hill, New York.

E. WEHRHAHN (1967), *On partial fraction expansion with high-order poles*, IEEE Trans. Circuit Theory (correspondence), CT-14, pp. 346–347.

SIAM J. COMPUT.
Vol. 6, No. 3, September 1977

LOCATION OF A POINT IN A PLANAR SUBDIVISION AND ITS APPLICATIONS*

D. T. LEE† AND F. P. PREPARATA‡

Abstract. Given a subdivision of the plane induced by a planar graph with n vertices, in this paper we consider the problem of identifying which region of the subdivision contains a given test point. We present a search algorithm, called point-location algorithm, which operates on a suitably preprocessed data structure. The search runs in time at most $O((\log n)^2)$, while the preprocessing task runs in time at most $O(n \log n)$ and requires $O(n)$ storage. The methods are quite general, since an arbitrary subdivision can be transformed in time at most $O(n \log n)$ into one to which the preprocessing procedure is applicable. This solution of the point location problem yields interesting and efficient solutions of other geometric problems, such as spatial convex inclusion and inclusion in an arbitrary polygon.

Key words. planar region identification, computational geometry, point location, computational complexity, analysis of algorithms, spatial convex inclusion, inclusion in polygon

1. Introduction. We shall consider the following problem referred to as "point location": A given planar straight line graph on n vertices induces a subdivision of the plane; given a test point P, find which region of this subdivision contains P. We shall assume that the graph is originally given as the collection of the edge-lists of its n vertices.

To solve this problem, one must produce an algorithm and its associated data structure, obtained by preprocessing the original data structure. Thus, from a computational viewpoint, any solution to the point location problem should be evaluated with respect to the following three measures: (i) the *search time*, that is, the number of operations required to locate the test point in the subdivision; (ii) the *preprocessing time*, that is, the number of operations required to construct the data structure postulated by the search algorithm; (iii) the amount of *storage* required by the preprocessed data structure.

Some workers have recently considered this problem. For example Ketelsen [1] proposed an algorithm whose search time is $O(n)$. The most recent and interesting results are due to Shamos [2]. His approach is an adaptation of a nearest-neighbor algorithm, also developed by him. It consists of tracing a sheaf of parallel lines through each of the n points, thereby slicing the plane in strips, called "slabs", each of which contains no vertex of the graph and is subdivided by transversal segments in at most $O(n)$ regions. Since each of these slab-regions belongs to a unique plane region, $O(\log n)$[1] comparisons are sufficient to locate the slab and additional $O(\log n)$ comparisons are sufficient to locate the region.

* Received by the editors December 11, 1975, and in revised form December 27, 1976.
This work was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy, and U.S. Air Force) under Contract DAAB-07-72-C-0259.

† Coordinated Science Laboratory and Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801.

‡ Coordinated Science Laboratory and Department of Electrical Engineering, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801.

[1] Here and hereafter "log" denotes logarithm in base 2.

Thus, Shamos's search algorithm has running time $O(\log n)$, but, as is easily realized, both preprocessing time and storage are $O(n^2)$. Shamos also outlines another possible procedure [2], based on recursively splitting the planar subdivision by means of polygonal lines, so that at each step half of the regions to be subdivided lie on either side of a polygonal line. Such an algorithm could achieve a search time $O((\log n)^2)$, but the polygonal lines must be "star-shaped" (in his terminology) and the existence of such lines is an open question even for the special case of a triangulation. In this paper we show that, if one does not insist on recursively halving sets of regions, a suitable set of polygonal lines can be found within a preprocessing time $O(n \log n)$; the resulting data structure can be stored in $O(n)$ memory locations and the search time is at most $O((\log n)^2)$.

The point location problem occurs in a number of applications in operations research, pattern recognition, job scheduling, etc. We shall show in this paper how it can also be applied to solve a number of other problems, among which is spatial convex inclusion.

This paper is organized as follows. In the next section we shall introduce some useful geometric constructs, specifically, monotone complete sets of chains of a planar graph, and justify their application to the solution of the point location problem. In § 3 we shall give a procedure for the construction of such sets of chains for an important subclass of planar subdivisions and illustrate the resulting data structure. In § 4 we shall illustrate in detail the point location algorithm. In § 5 we shall extend the results to arbitrary planar subdivisions. Finally, in § 6 we shall describe further applications of the outlined algorithms to problems in computational geometry.

2. Complete sets of chains.

2. Complete sets of chains. We begin by recalling some standard geometric definitions and introducing some useful notions.

A connected planar straight line (PSL) graph G with a finite number of vertices subdivides the plane into nonempty regions of which exactly one is infinite. These regions form a *planar subdivision* and will be also referred to as the regions of G.

DEFINITION 1. A *chain* $C=(u_1, \cdots, u_p)$ is a PSL graph with vertex set $\{u_1, \cdots, u_p\}$ and edge set $\{(u_i, u_{i+1}) | i=1, \cdots, p-1\}$.

DEFINITION 2. A chain $C=(u_1, u_2, \cdots, u_p)$ is said to be *monotone* with respect to a straight line l if the orthogonal projections $\{l(u_1), \cdots, l(u_p)\}$ of the vertices of C on l are ordered as $(l(u_1), \cdots, l(u_p))$.

Example. The chain in Fig. 1a is monotone with respect to l whereas there is no straight line with respect to which the chain in Fig. 1b is monotone.

It is convenient to extend a chain $(u_1, \cdots, u_p)$ which is monotone with respect to a line l with half-lines parallel to l beyond each terminal point u_1 and u_p. Any such extended chain thus divides the plane into two regions. In the sequel, a monotone chain will always be thought of as being extended in this manner.

DEFINITION 3. Given a PSL G, a set of chains $\mathscr{C}$ with the properties

(i) each edge of G belongs to at least one chain of $\mathscr{C}$;

(ii) for any two chains C_1 and C_2 of $\mathscr{C}$, the vertices of C_1 which are not vertices of C_2 lie on the same side of C_2,

is said to be a *complete set of chains* of G.

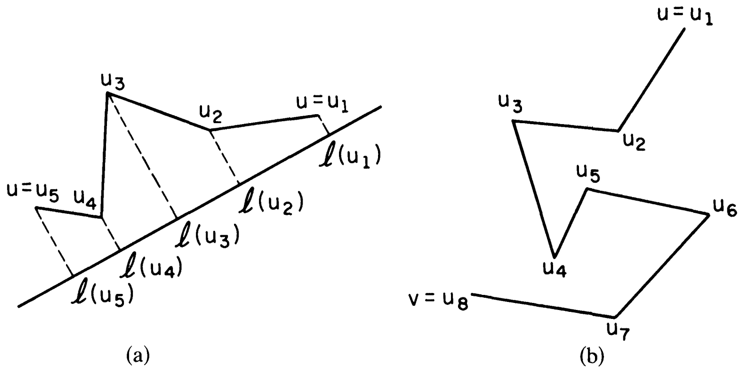


FIG. 1. *Examples of chains*

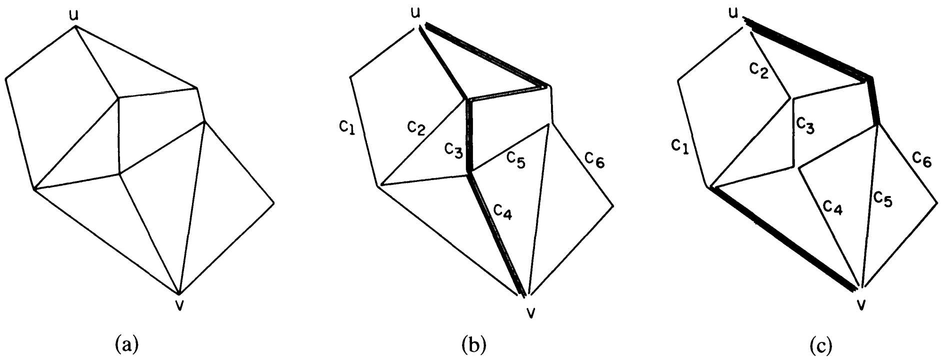


FIG. 2. *A* PSL *graph G and two complete sets of chains of G*

Given a set of chains $\mathscr{C}$ of a PSL graph, it is natural to introduce a binary relation "$<$" on $\mathscr{C}$ as follows: for $C_1, C_2 \in \mathscr{C}$, the notation $C_1 < C_2$ means that C_1 lies on a selected side of C_2 with respect to a fixed observer. It is then obvious that condition (ii) in Definition 3 implies that $<$ on a complete set $\mathscr{C}$ is a total ordering, i.e., $\mathscr{C}$ is a sequence $(C_1, C_2, \cdots, C_r)$. As an example, in Fig. 2b and 2c we exhibit two complete sets of chains of the PSL graph G given in Fig. 2a.

DEFINITION 4. A complete set of chains of a PSL graph G is said to be *monotone* if all of its members are monotone with respect to the same straight line.

Referring to the preceding example, the set in Fig. 2c is monotone, whereas the set in Fig. 2b is not.

We shall now motivate our interest in monotone complete sets of chains. Let C be a chain with vertices $u_1, \cdots, u_s$ which is monotone with respect to a straight line l and let P be a test point. We shall call a *discrimination* (of P against C) the operation of deciding on which side of C the point P lies. As Shamos pointed out [2], such discrimination can be performed with $O(\log s)$ comparisons of coordinates. In fact (see Fig. 3), the projections of the vertices of C on l form a sequence of points $(l(u_1), \cdots, l(u_s))$. With a binary search, i.e., with $\lceil \log (s+1) \rceil$ compari-

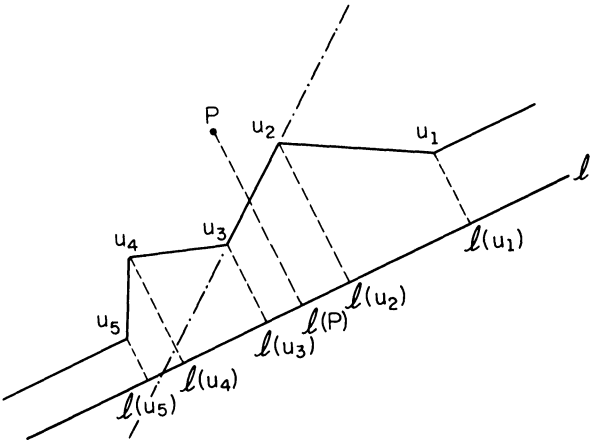


FIG. 3. *Illustration of a "discrimination" of a point P against a chain C*

sons of coordinates, we can determine a unique interval $(l(u_i), l(u_{i+1}))$ of l which contains the projection $l(P)$ of P on l. Next, with a fixed number of arithmetic operations and a single comparison we can determine on which side of the straight line containing $\overline{u_iu_{i+1}}$ the point P lies. This also determines on which side of C the point P lies. Suppose now that a complete monotone set of chains $\mathscr{C} = (C_1, C_2, \cdots, C_r)$ is given and let s be the maximum number of vertices in any chain of this set. Applying binary search to the set of chains $\mathscr{C}$, with $\lceil \log (r+1) \rceil$ discriminations we can determine a unique pair of consecutive chains C_j and C_{j+1} of $\mathscr{C}$ which enclose the given point P. Since the set $\mathscr{C}$ is also complete, by property (i) in Definition 3 the portion of plane contained between two consecutive members of $\mathscr{C}$ is a concatenation of regions of G and (possibly) of chain segments (see Fig. 4). Recall now that the discriminations of P against a chain C_j entails the identification of a unique edge $e^{(j)}$ of C_j. Thus, if P has been located between two consecutive chains C_j and C_{j+1}, the two edges $e^{(j)}$ and $e^{(j+1)}$ uniquely identify the

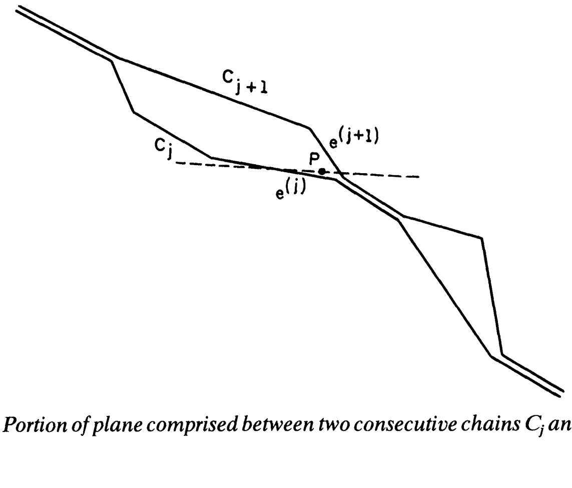


FIG. 4. *Portion of plane comprised between two consecutive chains C_j and C_{j+1}*

region of the graph G to which P belongs, and the point location problem is solved.

This justifies the usefulness of monotone complete sets of chains of a PSL graph G. The preceding discussion also indicates that at most $O(\log s) \cdot O(\log r)$ comparisons are needed for point location. Notice now that, if G has n vertices, each chain of a complete monotone set $\mathcal{C}$ on G has at most n vertices; moreover, by property (i) in Definition 3, there are at most $O(n)$ chains in $\mathcal{C}$. It follows that point location requires at most $O((\log n)^2)$ comparisons.

The remaining crucial question is whether an arbitrary PSL graph always admits of a monotone complete set of chains. The answer is in general negative. However, we shall constructively show that a PSL graph G admits of a monotone complete set of chains, provided it satisfies a rather weak requirement. In addition, we shall show that an arbitrary PSL graph can be easily embedded into one to which the chain construction procedure is applicable. This embedding creates some new "artificial" regions, with no harm, however, to the effective solution of the point location problem.

3. Preprocessing algorithm (construction of complete sets of chains). Let G be a PSL graph in the plane (x, y) with vertex set $\{v_1, v_2, \cdots, v_n\}$. Each vertex v_i is given by a pair of coordinates (x_i, y_i) and the graph G is described by its edge-lists, i.e., by a collection $\{L_1, L_2, \cdots, L_n\}$, where L_j, the edge-list for vertex v_j, is the set of indices of vertices to which v_j is connected. The set of chains we shall construct will be monotone with respect to the y-axis.

We initially arrange the original data structure by sorting the y-coordinates of the vertices and renaming the indices so that $y_1 \geqq y_2 \geqq \cdots \geqq y_n$; this processing requires $O(n \log n)$ operations. Next we construct a two-dimensional list, called the *standard representation of* G, representing a modified connection matrix A of G, where $A[i, j] = 1$ if and only if there is an edge (v_i, v_j) in G and either $y_i > y_j$ or, if $y_i = y_j$, then $i < j$. Notice that the row-lists thus obtained describe a directed graph G', obtained from G by assigning to each edge e of G a direction so that the projection of e on the y-axis runs opposite to this axis. For ease of reference, we shall say that an edge of G is "incoming" or "outgoing" depending upon its direction in G'. The construction of the standard representation of G is a simple operation. We scan the original data structure and process each edge (v_i, v_j) as follows: If either $y_i > y_j$ or $y_i = y_j$ and $i < j$, we assign the edge to the row-list of v_i and to the column-list of v_j; otherwise, we ignore the edge. It is clear that the running time of this construction is proportional to the number of edges, i.e., since G is planar, it is $O(n)$. Finally, the edges of each row- and column-list are sorted according to counterclockwise ascending slope: this sorting operation requires time at most $O(n \log n)$. The two-dimensional list data structure which gives the standard representation of the graph of Fig. 5a is illustrated in Fig. 5b, along with the formats of the vertex and edge nodes. The denominations of the fields of the two formats are now explained. For a vertex v_i, PRED$[v_i]$ and SUCC$[v_i]$ point to the locations of the nodes v_{i-1} and v_{i+1}, respectively, while COL$[v_i]$ and ROW$[v_i]$ point to the first members of the column- and row-lists, respectively; as to the value fields, $\#[v_i] = i$ and $I[v_i]$, the column degree of v_i, is the number of incoming edges of v_i, i.e., the cardinality of its column-list. For an edge $e = (v_i, v_j)$, with

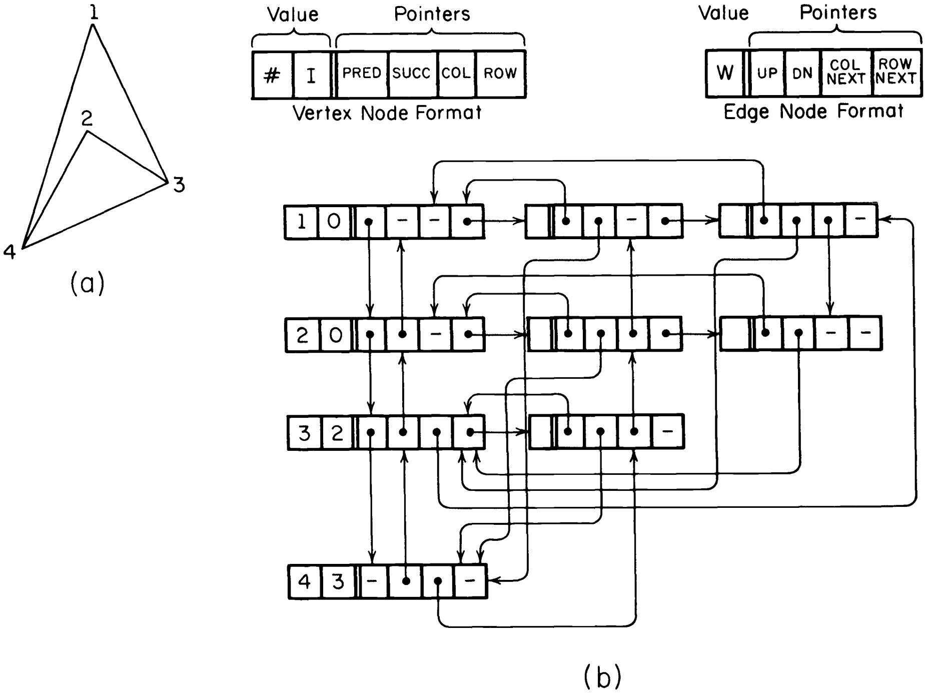


FIG. 5. *A* PSL *graph and the corresponding data structure*

$y_i > y_j$, UP[e] and DN[e] point to v_i and v_j, respectively, while COLNEXT[e] and ROWNEXT[e] point to the next members of the column- and row-lists, respectively; $W[e]$, the weight of e, will be used to denote the number of chains containing the edge e.

DEFINITION 5. A vertex v_j of a PSL graph G is said to be ***regular*** if there are integers $i < j < k$ such that (v_i, v_j) and (v_j, v_k) are edges of G. Graph G is said to be ***regular*** if each v_j is regular for $1 < i < n$ (i.e., with the exception of the two extreme vertices v_1 and v_n).

We shall now show that a regular PSL graph admits of a complete set of chains monotone with respect to the y-axis.

First we show, by induction, that for any j in the range from 2 to n, there is a chain from v_1 to v_j which is monotone with respect to the y-axis. This is trivial for $j = 2$. Assume the statement is true for $k < j$. Since v_j is regular, by Definition 5, there is some $i < j$ such that (v_i, v_j) is an edge of G. But, by the inductive hypothesis, there is a chain from v_1 to v_i monotone with respect to the y-axis; clearly, the concatenation of C and (v_i, v_j) is a chain from v_1 to v_j which is monotone with respect to the y-axis. This extends the inductive hypothesis. To complete the proof, we must show that a set of chains can be constructed, so that properties (i) and (ii) in Definition 3 are satisfied. But this is achieved if we succeed in assigning weights to edges so that 1) each edge has positive weight and 2) for each vertex v_i the sum $W_{\text{in}}(v_i)$ of the weights of its incoming edges equals the sum $W_{\text{out}}(v_i)$ of the weights of its outgoing edges. In fact, condition 1) ensures that each edge belongs to at least one chain (property (i)), and condition 2) ensures that the $W_{\text{in}}(v_i)$ chains passing through a given vertex v_i can be chosen so that they do not cross (property (ii)).

The conditions $W_{\text{in}}(v_i) = W_{\text{out}}(v_i)$, for $i = 2, \cdots, n-1$, referred to as "weight balancing", can be simply achieved with two passes over the previously described data structure. In the first pass we proceed from v_n to v_1 and assign the edge weights so that, for each nonterminal v_i, $W_{\text{in}}(v_i) \geqq W_{\text{out}}(v_i)$. The second pass, in turn, proceeds from v_1 to v_n and modifies the weights so that, $W_{\text{in}}(v_i) \leqq W_{\text{out}}(v_i)$, for every nonterminal v_i, thereby achieving the desired balancing.

During the latter pass we can also explicitly construct the chain data structure. It may appear at first glance that more than $O(n)$ locations are needed to store the chains, since there are $O(n)$ chains and there is no obvious way to bound the number of edges each of them may contain. However, we can obtain a very compact representation of this set using the fact that the chain set is to be used in a binary search. Indeed, it is well-known that a binary search algorithm on a totally ordered set S induces a natural hierarchy on S as follows: each member of S is assigned to a vertex of a binary tree $T(S)$ through a mapping $\tau: S \rightarrow T(S)$, and an actual search operation corresponds to tracing a path from the root to a leaf of $T(S)$. Given two chains C_i and C_j in $\mathscr{C}$ we shall say that C_i is *higher than* C_j, and denote it by $C_i > C_j$, if the path from the root of $T(\mathscr{C})$ to $\tau(C_j)$ contains $\tau(C_i)$. Assume now that $C_i > C_j$ and that C_i and C_j share an edge e. Since, when performing binary search, the discrimination of a point P against C_j is preceded by the discrimination of P against C_i, then clearly edge e may be assigned to C_i only. Thus, as a general rule, an edge of the graph G will be stored only once and will be assigned to the highest chain of the hierarchy which contains it.

With a negligible loss of search efficiency we shall adopt a *standard hierarchy* for the set $\mathscr{C} = \{C_1, C_2, \cdots, C_r\}$ which is independent of the number of chains, as follows: for $i \neq j$, $C_i > C_j \Leftrightarrow j \in [i - 2^{p_i} + 1, i + 2^{p_i} - 1]$, where 2^{p_i} is the largest power of 2 which is a factor of the integer i. For example, for $n = 20$ we have the hierarchy illustrated in Fig. 6. As we shall show, the use of the standard hierarchy will permit the assignment of edges to chains concurrently with the weight balancing operation.

The data structure describing the set $\mathscr{C}$ of chains will be a list representation of the tree $T(\mathscr{C})$, linking the chain nodes; in turn, the node for chain C_i is the header of a list $L(C_i)$, which gives the sequence of the edges assigned to C_i according to the previously outlined criterion. For reasons to become apparent later (§ 4) each edge $e = (v_i, v_j)$ is labeled with two pairs of integers $(I_{\min}[e], I_{\max}[e])$ and $(L[e], R[e])$. The integers $I_{\min}[e]$ and $I_{\max}[e]$ are respectively the minimum and the maximum value of j such that $e \in C_j$. The integers $L[e]$ and $R[e]$ are the labels of the plane regions respectively to the left and to the right of the edge e directed so that its y-projection is concurrent with the y-axis. The formats of the chain and the edge nodes are illustrated in Fig. 7.

We can now give a formal description of the algorithms which accomplish the construction of the set $\mathscr{C}$.

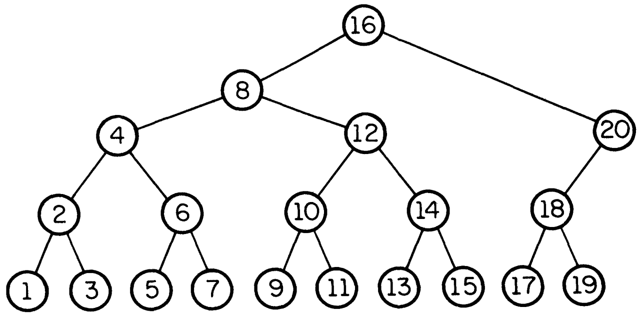


FIG. 6. *Hierarchy for n* = 20

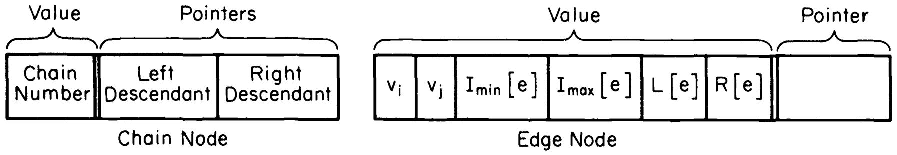


FIG. 7. *Formats of chain and edge nodes in the chain data structure*

CHAIN CONSTRUCTION ALGORITHM.

First pass. This pass accepts a PSL graph G given by its standard representation and assigns the weight of each edge so that for each nonterminal vertex v_i we have $W_{\text{in}}(v_i) \geqq W_{\text{out}}(v_i)$.

```
1. For each edge $e$ in column-list of $v_j$, for $j = 1, \cdots, n$, set $W[e] \leftarrow 1$.
2. Set $i \leftarrow n - 1$.
3. While $1 < i < n$ do:
     4. Set $W_{\text{out}} \leftarrow$ sum of weights in row-list of $v_i$. (Comment: due to the
        ordering of the vertices each edge considered in this step has already
        the weight it will have at the end of the first pass.)
     5. Set $d \leftarrow$ first edge in column-list of $v_i$.
     6. If $W_{\text{out}} > I[v_i]$, set $W[d] \leftarrow W_{\text{out}} - I[v_i] + 1$.
     7. Set $i \leftarrow i - 1$.
8. Halt.
```

Comment. All edges are assigned weight 1 except for the first member of each column-list, which is assigned the value $W_{\text{out}}(v_i) - W_{\text{in}}(v_i) + 1$. In this manner, the condition $W_{\text{in}}(v_i) \geqq W_{\text{out}}(v_i)$ is achieved for each nonterminal vertex. Computationally, the running time is $O(n)$, since each edge of the graph is scanned a fixed number of times (steps 1, 4, and 5) and there are $O(n)$ edges.

Second pass. This pass accepts the output of the first pass and performs the following functions: 1) it balances the weight of each nonterminal vertex of G; 2) it constructs a complete set of chains for G; 3) it labels the regions of the subdivision induced by G. The assignment of an edge to a chain makes use of a special arithmetic function, called PREDECESSOR(k, l), for two integers k and l

with $k \leqq l$, which determines the common predecessor of both k and l which is the lowest in the standard hierarchy; this function, which will not be described in detail, can operate directly on the binary representations of the integers k and l and is easily seen to require a number of operations proportional to $\log n$.

```
Comment. Steps 1–3 initialize for the special vertex $v_1$.
1. Set $A \leftarrow 1, r \leftarrow 2, L \leftarrow 1, R \leftarrow 1$.
2. Set $W_{\text{in}} \leftarrow$ sum of weights in row-list of $v_1$. (Comment: This is the total
   number of chains in $\mathscr{C}$.)
3. Set $i \leftarrow 1$.
4. While $1 \leqq i < n$ do:
    5. Set $W_{\text{out}} \leftarrow$ sum of weights in row-list of $v_i$.
    6. Set $a \leftarrow W_{\text{in}} - W_{\text{out}}$.
    7. Set $e \leftarrow$ first edge in row-list of $v_i$.
    8. While there are edges in row-list of $v_i$ do: (Comment: Steps 9–12
       assign an edge to a chain; steps 13–14 label the regions separated by
       an edge.)
        9. Set $I_{\min}[e] \leftarrow A$.
        10. Set $I_{\max}[e] \leftarrow A + a + W[e] - 1$.
        11. Compute $c \leftarrow$ PREDECESSOR($I_{\min}[e], I_{\max}[e]$).
        12. Assign $e$ to list of chain $c$.
        13. If $e$ is first edge of row-list, set $L[e] \leftarrow L$; else set $L[e] \leftarrow r$ and
            $r \leftarrow r + 1$.
        14. If $e$ is last edge of row-list, set $R[e] \leftarrow R$; else set $R[e] \leftarrow r$.
        15. Set $a \leftarrow 0, A \leftarrow I_{\max}[e] + 1$.
        16. Set $e \leftarrow$ next edge in row-list of $v_i$.
    17. Set $i \leftarrow i + 1$.
    18. Set $W_{\text{in}} \leftarrow$ sum of weights in column-list of $v_i$.
    19. Set $d_1 \leftarrow$ first edge in column-list of $v_i$.
    20. Set $R \leftarrow R[d_1]$.
    21. Set $d_2 \leftarrow$ last edge in column-list of $v_i$.
    22. Let $L \leftarrow L[d_2], A \leftarrow I_{\min}[d_2]$.
23. Halt.
```

Comment. The second pass contains two nested loops. The primary loop (steps 5–22) scans the vertices, the secondary loop (steps 9–16) scans the edges in the row-list of a vertex. For each vertex v_i, the weight balance is achieved by modifying, if necessary, the weight of the first member of the row-list (steps 5, 6, 7, 9, and 10). Each edge is processed only once, but the function PREDECESSOR requires time $O(\log n)$; thus the number of operations required is $O(n \log n)$.

In summary, since the initial sorting, the first balancing pass and the second balancing pass run in times $O(n \log n)$, $O(n)$, and $O(n \log n)$, respectively, the entire preprocessing tasks can be accomplished in time $O(n \log n)$.

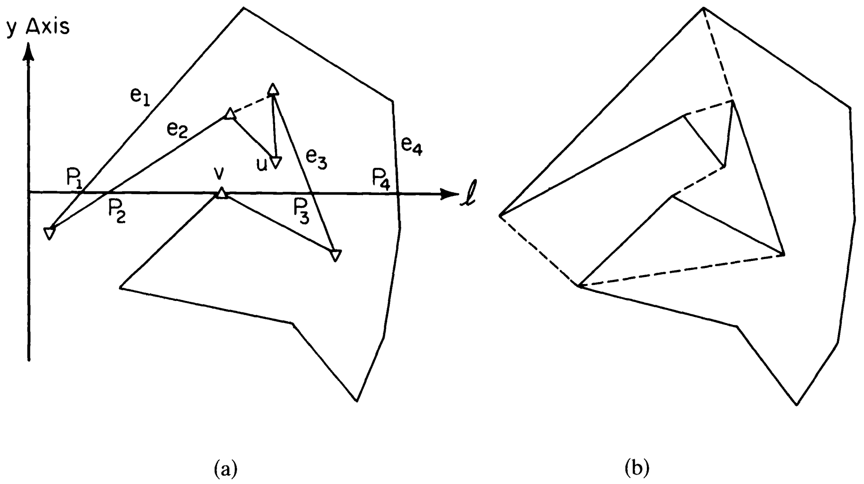
4. Point location algorithm (search). We shall now describe in detail the point location algorithm which we have sketched in § 2. The algorithm accepts the chain data structure described in § 3 and a test point $P \equiv (x, y)$ and determines the planar region R to which P belongs in at most $O((\log n)^2)$ steps. The integer m denotes the number of chains. The search is characterized by a pair of integers

(l, r), with $l < r$, to denote that the point P is contained between chains C_l and C_r; the algorithm terminates when $r - l = 1$ and the region is determined from the data associated with the edges.

1. If $y \geqq y_1$or $y \leqq y_n$, then set $R \leftarrow 1$. (Comment: P belongs to the infinite region of the plane) and halt.
2. Set $g \leftarrow$ root of $T(\mathscr{C})$, $l \leftarrow 0$, $r \leftarrow m + 1$.
3. Set $j \leftarrow$ index of the chain associated with g.
4. If $l \geqq j$, set $g \leftarrow$ right descendant of g and go to step 3.
5. If $r \leqq j$, set $g \leftarrow$ left descendant of g and go to step 3.
6. In the edge-list headed by g, find an edge (v_i, v_k) such that $y_k \leqq y \leqq y_i$ and set $e \leftarrow (v_i, v_k)$. (Comment: This step is the binary search of the edge against which P is to be discriminated.)
7. If P lies to the right of e, set $l \leftarrow I_{\max}[e]$ and $R \leftarrow R[e]$; else set $r \leftarrow I_{\min}[e]$ and $R \leftarrow L[e]$. (Comment: This step discriminates P against the edge e and makes a tentative region assignment; this assignment is final if $r - l = 1$, as indicated by step 8.)
8. If $r - l = 1$, halt; else go to step 3.

5. Regularization of an arbitrary planar polygon. The preprocessing procedure described in § 3 is applicable to regular PSL graphs. However, a PSL graph is not regular in general, since it may contain regions delimited by arbitrary polygons. In this section, we shall illustrate a procedure which transforms an arbitrary polygon with n vertices into a regular PSL graph and runs in time $O(n \log n)$. This procedure will be referred to as *regularization* of a planar polygon. With the aid of this additional procedure, our point-location algorithm and its associated preprocessing are applicable to arbitrary planar subdivisions.

We shall briefly illustrate the idea of the regularization procedure. In a simple planar polygon not all vertices are regular with respect to a selected line (the y-axis): for example, in Fig. 8a, vertices marked with "∇" require an outgoing edge and vertices marked with "$\triangle$" require an incoming edge. Consider now in Fig. 8a the horizontal line l (orthogonal to the y-axis) through a vertex v requiring, for example, an incoming edge. This line l intersects a set of edges $\{e_1, e_2, e_3, e_4\}$ in points P_1, P_2, P_3, and P_4, respectively. Therefore vertex v falls in a unique part of the partition of l determined by these points (in our example, in segment P_2P_3). Associated with the pair of edges (e_2, e_3) there is a vertex of minimum ordinate (in our case, vertex u): thus we can introduce an auxiliary edge (v, u) which is guaranteed not to cross any edge of the polygon and which satisfies the requirement of v for an incoming edge. Thus the regularization procedure will consist of two passes: in the first pass (descending pass) we scan the vertices in order of decreasing y-coordinate and satisfy the requirements of vertices in need of incoming edges; in the second pass (ascending pass) the reverse process occurs. Each stage in the execution of the algorithm is characterized by an ordered set of edges (a sequence of edges). For every newly reached vertex v, the current edge sequence is updated as follows: if v is regular, a new edge replaces a previous edge, if v requires an incoming edge (an outgoing edge), two new edges are inserted (are deleted). Therefore, the natural data structure for the edge sequence is a list, realized by a balanced tree (an AVL tree; see [3, § 6.2.3]).



(a) (b)

FIG. 8. *A simple polygon and its regularized version*

We shall now give a detailed description of the regularization algorithm and, for obvious reasons, we shall confine ourselves to the descending pass. The polygon G is originally given as a sequence of vertices $u_1, u_2, \cdots, u_n$, corresponding to the edge sequence $(u_1, u_2), (u_2, u_3), \cdots, (u_n, u_1)$. Preliminarily, we construct the standard representation of G (see § 3); that is, we sort the vertices in order of decreasing y-coordinate, we relabel them as $v_1, \cdots, v_n$ so that $y(v_1) \geqq \cdots \geqq y(v_n)$ and associate with each vertex the row- and column-lists of its outgoing and incoming edges, respectively. In our case, the total number of edges in the row- and column-lists is equal to 2, since G is a closed polygon.

The data structure describing the current edge sequence is an AVL tree $\mathscr{E}$. Specifically, $\mathscr{E}$ is a sequence $[e_0, w_0], [e_1, w_1], \cdots, [e_k, w_k]$, for some integer k, where e_j is an edge and w_j is a minimum ordinate vertex between e_j and e_{j+1}. The algorithm described below refers to a simple polygon G surrounded by the infinite region of the plane; there are some obvious modifications when G borders other simple polygons.

Descending pass. This algorithm accepts the standard representation of G and creates an auxiliary list of edges satisfying the requirements of vertices in need of an *incoming* edge. We assume that standard operations: INSERT, DELETE, REPLACE are available for the AVL tree $\mathscr{E}$. Use is also made of an artificial edge e_0 corresponding to the line $x = -\infty$. If a vertex v has two outgoing edges, the latter are denoted as $e'(v)$ and $e''(v)$, with $e'(v)$ forming a counterclockwise convex angle with $e''(v)$. The integer $I(v)$ denotes the number of incoming edges of a vertex v.

1. Set $w_0 \leftarrow v_1, e_1 \leftarrow e'(v_1), e_2 \leftarrow e''(v_1), w_1 \leftarrow v_1, w_2 \leftarrow v_1$ and INSERT $[e_0, w_0]$, $[e_1, w_1]$, and $[e_2, w_2]$ into $\mathscr{E}$. ($\mathscr{E}$ is initially empty.)
2. Set $i \leftarrow 2$.
3. While $i < n$ do:

```
  4. Set $j \leftarrow$ smallest integer so that $v_i$ is not to the left of $e_j$.
  5. If $I(v_i)=2$, set $w_{j-1} \leftarrow v_i$, $w_{j+2} \leftarrow v_i$; then DELETE $[e_j, w_j]$ and
     $[e_{j+1}, w_{j+1}]$ from $\mathscr{C}$. (Comment: This case occurs when $v_i$ requires an
     outgoing edge.)
  6. If $I(v_i)=1$, set $w_j \leftarrow v_i$, $w_{j-1} \leftarrow v_i$, $e_j \leftarrow$ outgoing edge of $v_i$. (Comment: $v_i$
     is a regular vertex.)
  7. If $I(v_i)=0$, INSERT $[e'(v_i), v_i]$ and $[e''(v_i), v_i]$ between $[e_j, w_j]$ and
     $[e_{j+1}, w_{j+1}]$ in $\mathscr{C}$. Add the edge $(v_i, w_j)$ to the auxiliary list and set
     $w_j \leftarrow v_i$. (Comment: In this case $v_i$ requires an incoming edge.)
  8. Set $i \leftarrow i+1$.
9. Halt.
```

The ascending pass is the same algorithm, once the signs of the ordinates $y(v_1), \cdots, y(v_n)$ have been changed. At the completion of the two passes the edges in the auxiliary list must be added to the standard representation and the resulting PSL graph is regular with respect to the y-axis.

We now analyze the regularization algorithm. The initial sorting pass of the ordinates requires $O(n \log n)$ operations. Step 4 involves tracing a path in the AVL tree $\mathscr{C}$ and requires at most $O(\log n)$ operations; thus the total amount of work involved in step 4 is $O(n \log n)$. Each update, deletion or insertion (see steps 5, 6, and 7) requires computational work of at most $O(\log n)$; however, there is a fixed maximum number of such operations per vertex, whence the total number of operations required by these steps is $O(n \log n)$. We conclude that the algorithm for regularizing a simple polygon has running time at most $O(n \log n)$.

In Fig. 8b, we illustrate the result of the regularization procedure applied to the polygon in Fig. 8a. New regions have been created, with no difficulty however for the effective solution of the point location problem.

As a final observation, we consider the case in which we must regularize a given general PSL graph G with n vertices and N edges. For every region of G with s vertices, the regularization algorithm runs in time $O(s \log s)$. Assuming that the subdivision induced by G consists of regions $R_1, R_2, \cdots, R_J$ with respective number of vertices $s_1, s_2, \cdots, s_J$, since $s_1+\cdots+s_J \leqq 2N$, we conclude $s_1 \log s_1+\cdots+s_J \log s_J \leqq 2N \log 2N$; but N is $O(n)$, whence the regularization of a general PSL graph with n vertices can also be accomplished in time $O(n \log n)$.

6. Applications. As we indicated in § 1, the point location problem occurs as a subproblem in a number of important applications. Thus our fast point location algorithm provides interesting solutions for other problems. In particular, we shall explicitly consider:

(i) spatial convex inclusion;
(ii) inclusion in an arbitrary planar polygon;
(iii) location in the planar subdivision determined by n straight lines.

6.1. Spatial convex inclusion. Let S be a convex polyhedron with n vertices in 3-space (x, y, z) and let P be a test point; we must determine whether P is inside or outside S. We shall proceed as follows. Let P_1 be a vertex of S with largest z coordinate. Clearly, the stereographic projection of S from P_1 onto the plane (x, y) is a planar graph S'. Since the regions induced by S' are projections of faces of S, and the latter are convex, each region of the planar subdivision is convex. It follows then that S' is regular, since the nonterminal vertices on the boundary are obviously regular and each internal vertex has at least three edges forming convex angles. Thus, with the previous point location algorithm, we can identify a unique region R' of S' which contains a point P', which is the projection onto (x, y) from P_1 of a given test point P in 3-space. Since R' of S' corresponds to a face R of S, the testing for convex inclusion of P in S is accomplished by determining on which side of R the point P lies. Since the operation of constructing the data structure for S' from the data structure of S runs in time $O(n)$, we conclude that, with a preprocessing time at most $O(n \log n)$, spatial convex inclusion can be tested in at most $O((\log n)^2)$ steps.

6.2. Inclusion in an arbitrary planar polygon. Let G be a planar polygon with n vertices and let P be a test point; we must determine whether P is inside or outside G. Shamos [2] provides an algorithm for solving this problem based on the "slab" notion, which has search time $O(\log n)$, with a preprocessing time and storage both $O(n^2)$. By contrast, our algorithm has search time $O((\log n)^2)$, preprocessing time $O(n \log n)$ and storage $O(n)$. We initially regularize the polygon G and transform it into a regular PSL graph G': the regions of the resulting subdivision are partitioned depending on whether they are inside or outside G. Thus the point location algorithm is applied to G' and the problem is solved.

6.3. Location in straight line planar subdivision. Dobkin and Lipton originally suggested the problem of locating a point in the planar subdivision induced by n straight lines in general position [4]. In this case the regions (finite or infinite) are all convex, and therefore the planar graph on $O(n^2)$ vertices is regular. Some minor modification of the algorithms are required and it is straightforward to show that $O((\log n)^2)$ search time and $O(n^2)$ storage can be attained, as conjectured by Shamos [2].

REFERENCES

[1] M. Ketelsen, *Triangular tile identification*, CS 389 course project, Dept. of Computer Sci., Univ. of Illinois, Urbana, IL, Dec. 1973.

[2] M. J. Shamos, *Problems in computational geometry*, Dept. of Computer Sci., Yale Univ., New Haven, CT, May 1975.

[3] D. E. Knuth, *The Art of Computer Programming, Vol.* 3: *Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[4] D. P. Dobkin and R. J. Lipton, *The complexity of searching lines in the plane*, Dept. of Computer Sci., Yale Univ., New Haven, CT, 1975.

SIAM J. COMPUT.
Vol. 6, No. 4, December 1977

INTERNAL FRAGMENTATION IN A CLASS OF BUDDY SYSTEMS*

DAVID L. RUSSELL†

Abstract. A general class of buddy systems is defined and the overallocation of memory due to internal fragmentation is examined. It is shown that, for a uniform distribution of requests, the overallocation varies between $2x_1/(x_1+1)$ and $\frac{1}{2}(x_1+1)$, where $x_1>1$ is the largest real root of the characteristic equation of the particular buddy system. Bounds are found for the overallocation for request distributions characterized by a parameterization of Zipf's law. The expected value of the overallocation is independent of the request distribution within wide values of the parameter, and is given by $(x_1-1)/\ln x_1$. For the binary buddy system $x_1=2$; the overallocation varies between 1.33 and 1.50 and the expected value is ≈ 1.44.

Key words. dynamic storage allocation, buddy system, fragmentation, Fibonacci buddy system

1. Introduction. Buddy systems are algorithms for dynamic storage allocation. Storage is provided in fixed size blocks; an available space list is maintained to keep track of available blocks. When a request for a block of a particular size cannot be satisfied from an available block, then a larger block is split into smaller blocks, called buddies. One of these buddies is used to satisfy the request, and the others are placed on the available space list. (The request procedure may be performed iteratively to obtain the "larger" block, if it is not already on the available space list.) When a block is released, its buddies are examined. If all the buddies are free, then they are recombined into the original parent block. This recombination continues as long as all buddy descendants of a parent block are free.

Inefficient use of memory may arise in two ways. *Internal fragmentation* occurs because blocks may be allocated in fixed, predetermined sizes only; thus, a request for memory must be satisfied by a block of the next larger size. *External fragmentation* results when sufficient memory exists to satisfy a request, but it is found in blocks which are not buddies and therefore may not be combined.

The original buddy system studied by Knowlton [2] and Knuth [3] used blocks of size 2^n, and will be referred to as the *binary buddy system.* Hirschberg [1] and Sedgewick [10] studied *Fibonacci buddy systems,* where the block sizes are Fibonacci numbers F_n. Hirschberg [1] and Peterson and Norman [8] have generalized these schemes to include a wider class of block sizes.

In this paper, internal fragmentation is measured by the overallocation ratio, (memory allocated)/(memory requested). A class of buddy systems, called *simple buddy systems,* is defined and the block sizes of simple buddy systems are characterized. An expression for the overallocation of a simple buddy system is derived, assuming that the sizes of requested blocks are uniformly distributed. This is generalized to request distributions described by a parameterized Zipf's law. Bounds for the overallocation are determined, and the expected value for the overallocation is found.

* Received by the editors February 20, 1975, and in revised form July 16, 1976.

† Computer Science Department, University of Southern California, University Park, Los Angeles, California 90007. This work was supported in part by the National Science Foundation under Grant GJ41644 while the author was at Stanford University.

2. Simple buddy systems. The buddy systems studied in this paper may be defined by a linear homogeneous recurrence equation with constant coefficients:

$$u_n = a_1 u_{n-1} + a_2 u_{n-2} + \cdots + a_r u_{n-r}. \tag{1}$$

(As is common with equations of this type, $a_r \neq 0$; otherwise the order of the difference equation could be reduced.) This represents the splitting of a block of size u_n into a_1 buddies of size u_{n-1}, a_2 buddies of size u_{n-2}, etc. The initial conditions are

$$\begin{aligned} u_n &= 0, \qquad n < 0, \\ u_0 &= 1. \end{aligned} \tag{2}$$

DEFINITION. A *simple buddy system* is a buddy system where the block sizes u_n satisfy (1) and (2), the coefficients a_i of (1) are nonnegative integers, and for large enough n, the sequence u_n is monotonically increasing.

For example, the recurrence relation for the binary buddy system is $u_n = 2u_{n-1}$, and the recurrence relation for the Fibonacci buddy system is $u_n = u_{n-1} + u_{n-2}$. The recurrence relation $u_n = u_{n-1} + u_{n-2} + u_{n-3}$ would represent a buddy system where each larger block is split into three buddies. Note that the class of simple buddy systems does not include all useful buddy schemes. For instance, the weighted buddy system of Shen and Peterson [11] is defined by a system of *two* recurrence relations and is not a simple buddy system.

From the theory of recurrence relations (see, for example, Liu [6]) it is known that the solutions of (1) are determined by the initial conditions and by the zeros of the characteristic polynomial

$$f(z) = z^r - a_1 z^{r-1} - a_2 z^{r-2} - \cdots - a_r. \tag{3}$$

LEMMA. *The characteristic polynomial* (3) *has a real zero* $z_1 > 1$ *of multiplicity* 1, *and* z_1 *is the sole dominant zero.*

Proof. The sum $\sum_{1 \leqq i \leqq r} a_i$ must be strictly greater than 1; if not, the u_n would never be monotonically increasing. But then $f(1) < 0$. Since $f(z) > 0$ for sufficiently large positive z there is a zero $z_1 > 1$. The derivative $f'(z)$ satisfies

$$\begin{aligned} f'(z) &= rz^{r-1} - a_1(r-1)z^{r-2} - \cdots - a_{r-1} \\ &> rz^{r-1} - a_1 rz^{r-2} - \cdots - ra_{r-1} \\ &= \frac{r[f(z) + a_r]}{z} \\ &> \frac{rf(z)}{z} \quad \text{for all } z > 0. \end{aligned}$$

Thus $f'(z_1) > rf(z_1)/z_1 = 0$, and z_1 is a single zero. Furthermore, since $f'(z) > rf(z)/z \geqq 0$ for $z > z_1$, $f(z)$ is monotonically increasing for $z > z_1$, and there are no real roots of magnitude greater than z_1.

By a simple application of inequalities for complex numbers $f(z)$ is seen to satisfy

$$\begin{aligned}
|f(z)| &\geqq |z^r| - |a_1 z^{r-1} + a_2 z^{r-2} + \cdots + a_r| \\
&\geqq |z^r| - |a_1 z^{r-1}| - |a_2 z^{r-2}| - \cdots - |a_r| \\
&= |z|^r - |a_1||z|^{r-1} - |a_2||z|^{r-2} - \cdots - |a_r| \\
&= f(|z|),
\end{aligned} \tag{4}$$

since the a_i are nonnegative and satisfy $|a_i| = a_i$.

In relation (4) equality can hold only if each term $a_i z^{r-i}$ is a real number, and this happens only when $f(z) = g(z^p)$, so that z is a pth root of a real zero of the polynomial g. This would imply, however, that the defining equation is really

$$u_n = a_p u_{n-p} + a_{2p} u_{n-2p} + \cdots + a_{mp} u_{n-mp}, \qquad mp = r.$$

The initial conditions would then give a sequence

$$1, 0, 0, \cdots, u_p, 0, 0, \cdots, u_{2p}, 0, 0, \cdots.$$

Since the sequence u_n is required to be monotonically increasing for large n, this situation cannot occur, and equality never holds in relation (4) above; therefore, $|f(z)| > f(|z|)$.

Now, if a root $z_2 \neq z_1$ exists, where $|z_2| \geqq z_1$, then $|f(z_2)| > f(|z_2|) \geqq f(z_1) = 0$, since $f'(z) > 0$ for $z \geqq z_1$. But then z_2 cannot be a zero of $f(z)$, and thus z_1 is the sole dominant zero. Q.E.D.

COROLLARY. *The block sizes of a simple buddy system satisfy*

$$u_n = c z_1^n + O(n^{p-1}|z_2|^n)$$

where z_2 is the zero of next to largest modulus, p is the multiplicity of z_2, and c is an appropriate constant.

Proof. This follows from the theory of recurrence equations (see, for example, Liu [6]) and the fact that z_1, the sole dominant zero, is of multiplicity 1.

3. Overallocation in a simple buddy system. Suppose that a request for a block of memory of size i results in the allocation of a block of memory of size s_i. Unless $s_i = i$ for all sizes of requests i, some memory will be wasted. If blocks of size L_n may be allocated, then $s_i = L_k$ such that $L_{k-1} < i \leqq L_k$ (assuming that the L_n are monotonically increasing). Given a distribution of request sizes, the average overallocation may be calculated.

If a distribution of requests which is uniform from 1 to m is assumed, then the overallocation may be defined as b_m/a_m, where

$$\begin{aligned}
a_m &= 1 + 2 + 3 + \cdots + m = \tfrac{1}{2}m(m+1), \\
b_m &= s_1 + s_2 + s_3 + \cdots + s_m.
\end{aligned}$$

Since s_j is constant for $L_{k-1} < j \leqq L_k$ the sum b_m, for $L_k \leqq m \leqq L_{k+1}$, may be expressed as

$$b_m = L_1^2 + \sum_{2 \leqq i \leqq k} L_i(L_i - L_{i-1}) + \alpha(L_{k+1} - L_k)L_{k+1},$$

where $m = L_k + \alpha(L_{k+1} - L_k)$, and $0 \leqq \alpha \leqq 1$. The overallocation is then defined as $R(\alpha) = \lim_{m\to\infty} b_m/a_m$.

In a simple buddy system the block sizes L_k are determined by the numbers u_k. Since the u_k may not be monotonically increasing for *small k*, it is not possible to directly set $L_k = u_k$. (Consider, for example, the simple buddy system $u_n = u_{n-1} + u_{n-3}$, which generates the sequence $1, 1, 1, 2, 3, 4, \cdots$ or the system $u_n = u_{n-2} + u_{n-3}$ and its sequence $1, 0, 1, 1, 1, 2, 2, 3, \cdots$.) Since the u_k eventually *do* become monotonic, however, it is true that $L_k = c_1 x_1^k + O(k^{p-1}|z_2|^k)$, for large enough k, say $k \geqq k_0$, and for an appropriate constant c_1. Then m satisfies

$$m = L_k + \alpha(L_{k+1} - L_k) = c_1 x_1^k(1 + \alpha(x_1 - 1)) + O(k^{p-1}|z_2|^k).$$

Furthermore, b_m satisfies

$$\begin{aligned} b_m &= B_1 + \sum_{k_0 < i \leqq k} L_i(L_i - L_{i-1}) + \alpha(L_{k+1} - L_k)L_{k+1} \\ &= B_1 + \sum_{k_0 < i \leqq k} [c_1^2 x_1^{2i-1}(x_1 - 1) + O(x_1^i i^{p-1}|z_2|^i)] + c_1^2 \alpha x_1^{2k+1}(x_1 - 1) \\ &\quad + O(x_1^k k^{p-1}|z_2|^k) \\ &= B_1 + \frac{c_1^2(x_1^{2k+1} - B_2)}{x_1^2 - 1}(x_1 - 1) + c_1^2 \alpha x_1^{2k+1}(x_1 - 1) + O(x_1^k k^{p-1}|z_2|^k). \end{aligned}$$

In this equation, B_1 represents the contribution from the terms with small block size, where the sequence of u_k may not be monotonic, and B_2 represents the contribution due to the bottom limit of the summation; B_1 and B_2 are both constant. The exact error term depends on the magnitude of z_2. (It is, in fact, given by the expression $O(1 + x_1^k k^{p-1}|z_2|^k)$.) In any case, however, the limit of b_m/x_1^{2k} is well-defined and

$$\lim_{\substack{m\to\infty \\ k\to\infty}} \frac{b_m}{x_1^{2k}} = \frac{c_1^2 x_1}{x_1 + 1}(1 + \alpha(x_1^2 - 1)).$$

Similarly, when a_m is calculated, the exact error term depends on the magnitude of z_2. The limit of a_m/x_1^{2k}, though, is well-defined and

$$\lim_{\substack{m\to\infty \\ k\to\infty}} \frac{a_m}{x_1^{2k}} = \frac{1}{2} c_1^2 (1 + \alpha(x_1 - 1))^2.$$

The overallocation of a simple buddy system is given by

$$(5) \qquad R(\alpha) = \lim_{m\to\infty} \frac{b_m}{a_m} = \lim_{\substack{m\to\infty \\ k\to\infty}} \frac{b_m/x_1^{2k}}{a_m/x_1^{2k}} = \frac{2x_1}{x_1 + 1} \frac{1 + \alpha(x_1^2 - 1)}{(1 + \alpha(x_1 - 1))^2}.$$

The minimum value of this function occurs when $\alpha = 0$ or $\alpha = 1$. It is easily seen that

$$(6) \qquad R_{\min} = R(0) = R(1) = \frac{2x_1}{x_1 + 1}.$$

The function $R(\alpha)$ has an extremal value at α_m when α_m is chosen to make $R'(\alpha_m)$ vanish. $R'(\alpha_m)$ will be zero when

$$(1+\alpha_m(x_1-1))^2(x_1^2-1)-(1+\alpha_m(x_1^2-1))(2(1+\alpha_m(x_1-1))(x_1-1))=0$$

Upon simplification, this reduces to $\alpha_m = 1/(x_1+1)$ and thus

$$(7)\quad R_{\max}=R\left(\frac{1}{x_1+1}\right)=\frac{2x_1}{x_1+1}\,\frac{x_1}{[1+(x_1-1)/(x_1+1)]^2}=\frac{2x_1^2(x_1+1)}{(2x_1)^2}=\frac{1}{2}(x_1+1).$$

An "average" value R_{avg} for the overallocation of a buddy system may be found by letting the memory size be a random variable over the positive integers. Since every particular memory size corresponds to a particular value of α, it is tempting to take the integral of $R(\alpha)$ from $\alpha=0$ to $\alpha=1$ as the average value R_{avg}.

This implies that α is uniformly distributed in $[0, 1]$. However this is not true. It is well known that the leading digits of the normalized fraction parts of random numbers are not uniformly distributed [4]. In particular, the probability that a randomly chosen real number written in base b has leading digits $<r$, where $1<r<b$, is $\log_b r$. Thus the leading digits are logarithmically distributed.

In the present case, $m \approx c_1 x_1^n(1+\alpha(x_1-1))$ and the quantity $1+\alpha(x_1-1)$ acts as a "leading digit." The probability distribution function of α is $F(\alpha)=\log_{x_1}(1+\alpha(x_1-1))$, and thus the probability density function for a particular value α is given by

$$f(\alpha)=\frac{d}{d\alpha}(\log_{x_1}(1+\alpha(x_1-1)))=\frac{(x_1-1)}{(\ln x_1)(1+\alpha(x_1-1))}.$$

This may be used as a weight function and multiplied by $R(\alpha)$ before taking the integral to get R_{avg}. The integral may be calculated using elementary calculus and yields the following expression:

$$(8)\quad \begin{aligned} R_{\text{avg}} &= \int_0^1 \frac{(x_1-1)R(\alpha)}{(\ln x_1)(1+\alpha(x_1-1))}\,d\alpha \\ &= \frac{2x_1(x_1-1)}{(x_1+1)(\ln x_1)}\int_0^1 \frac{(1+\alpha(x_1^2-1))}{(1+\alpha(x_1-1))^3}\,d\alpha \\ &= \frac{2x_1(x_1-1)}{(x_1+1)(\ln x_1)}\left\{\frac{-1}{2(x_1-1)(1+\alpha(x_1-1))^2}\right. \\ &\qquad +(x_1+1)\left[\frac{-1}{(x_1-1)(1+\alpha(x_1-1))}\right. \\ &\qquad\qquad \left.\left.+\frac{1}{2(x_1-1)(1+\alpha(x_1-1))^2}\right]\right\}_0^1 \\ &= \frac{2x_1(x_1-1)}{(x_1+1)(\ln x_1)}\left[\frac{x_1+1}{2x_1^2}+(x_1+1)\frac{x_1-1}{2x_1^2}\right] \\ &= \frac{x_1-1}{\ln x_1}. \end{aligned}$$

An alternative derivation of R_{avg} can be obtained by expressing m in terms of a parameter β, so that $m = L_k(L_{k+1}/L_k)^\beta \approx c_1 x_1^{k+\beta}$. Then a random value of m leads to a value of β which is uniformly distributed in [0, 1]. The parameter β satisfies $x_1^\beta = 1+\alpha(x_1-1)$, and therefore $\alpha = (x_1^\beta - 1)/(x_1-1)$. $R(\alpha)$ may be rewritten as

$$(9) \qquad R(\beta) = \frac{2x_1}{x_1+1}\frac{1+(x_1^\beta-1)(x_1+1)}{x_1^{2\beta}}$$

and R_{avg} may be calculated as

$$\begin{aligned} \mathrm{R}_{\text{avg}} &= \int_0^1 R(\beta)\,d\beta \\ &= \frac{2x_1}{x_1+1}\int_0^1 \frac{1+(x_1^\beta-1)(x_1+1)}{x_1^{2\beta}}\,d\beta \\ &= \frac{2x_1}{x_1+1}\left\{(x_1+1)\frac{x_1^{-\beta}}{-\ln x_1} - x_1\frac{x_1^{-2\beta}}{-2\ln x_1}\right\}_0^1 \\ &= \frac{2x_1}{(x_1+1)(\ln x_1)}\left\{(x_1+1)\left(1-\frac{1}{x_1}\right) - \frac{1}{2}x_1\left(1-\frac{1}{x_1^2}\right)\right\} \\ &= \frac{x_1-1}{\ln x_1}. \end{aligned}$$

That these two integrals are actually formally the same can be seen by noting that

$$x_1^\beta \ln x_1\, d\beta = (x_1-1)\,d\alpha, \quad \text{or} \quad d\beta = \frac{(x_1-1)}{(\ln x_1)(1+\alpha(x_1-1))}\,d\alpha.$$

Thus the two expressions for R_{avg} are equivalent.

4. Generalized Zipf distribution. In § 3 it was assumed that the sizes of requested blocks were uniformly distributed from 1 to m. The method can be generalized to find the overallocation of a simple buddy system with other distributions of request sizes. Suppose that the probability of a request of size i is p_i. (Since this is a probability function, $\sum p_i = 1$.) The quantity $a = \sum ip_i$ gives the mean request size, and the quantity $b = \sum p_i s_i$ gives the mean allocated size. The overallocation is then given by $R = b/a$. For the uniform distribution already discussed, $p_i = 1/m$ for $1 \leqq i \leqq m$, and 0 otherwise; the expressions reduce to their previous values.

The calculations can also be easily made for a class of distributions related to Zipf's law. Let $p_i = c/i^\theta$ for $1 \leqq i \leqq m$, and $p_i = 0$ otherwise. If $\theta = 0$, this is the uniform distribution; if $\theta = 1$, it is Zipf's law. The constant $c = 1/H_m^{(\theta)}$ is a normalizing constant chosen to make $\sum p_i = 1$; $H_m^{(\theta)}$ is the mth harmonic number of order θ, defined by

$$H_m^{(\theta)} = \sum_{1 \leqq i \leqq m} \frac{1}{i^\theta}.$$

The mean request size is given by

$$a_m = \sum_{1 \leqq i \leqq m} ip_i = \sum_{1 \leqq i \leqq m} ci\frac{1}{i^\theta} = \sum_{1 \leqq i \leqq m} c\frac{1}{i^{\theta-1}} = cH_m^{(\theta-1)}.$$

The mean allocated block size is calculated as

$$b_m = L_1 \sum_{1 \leqq j \leqq L_1} p_j + \sum_{2 \leqq i \leqq k} L_i \sum_{L_{i-1} < j \leqq L_i} p_j + L_{k+1} \sum_{L_k < j \leqq m} p_j$$

$$b_m/c = L_1 H_{L_1}^{(\theta)} + \sum_{2 \leqq i \leqq k} L_i (H_{L_i}^{(\theta)} - H_{L_{i-1}}^{(\theta)}) + L_{k+1}(H_m^{(\theta)} - H_{L_k}^{(\theta)}).$$

The asymptotic expression for $H_m^{(\theta)}$, $\theta \neq 1$, is [5]

$$H_m^{(\theta)} = \zeta(\theta) + m^{1-\theta}/(1-\theta) + \tfrac{1}{2}m^{-\theta} - \cdots,$$

where $\zeta(\theta)$ is a constant (Reimann's zeta function). Thus, if $\theta \neq 1$, the following expressions are valid:

$$H_{L_i}^{(\theta)} - H_{L_{i-1}}^{(\theta)} \approx \frac{1}{(1-\theta)}((c_1 x_1^i)^{1-\theta} - (c_1 x_1^{i-1})^{1-\theta})$$

$$= \frac{c_1^{1-\theta} x_1^{(i-1)(1-\theta)}}{(1-\theta)}(x_1^{1-\theta} - 1),$$

$$H_m^{(\theta)} - H_{L_k}^{(\theta)} \approx \frac{c_1^{1-\theta} x_1^{k(1-\theta)}}{(1-\theta)}(x_1^{\beta(1-\theta)} - 1).$$

(As before, the block sizes L_i are given by $L_i \approx c_1 x_1^i$ and m satisfies $m = L_k + \alpha(L_{k+1} - L_k) \approx c_1 x_1^k(1 + \alpha(x_1 - 1)) = c_1 x_1^{k+\beta}$.)

Completing the calculation for b_m/c (again ignoring the constant contribution from small block sizes and the bottom limit of the summation) leads to the following expression:

$$b_m/c \approx \frac{c_1^{2-\theta}}{1-\theta}\left\{\sum_{i \leqq k} x_1^{(i-1)(1-\theta)+i}(x_1^{1-\theta} - 1) + x_1^{k+1} x_1^{k(1-\theta)}(x_1^{\beta(1-\theta)} - 1)\right\}$$

$$= \frac{c_1^{2-\theta}}{1-\theta}\left\{\frac{x_1^{1-\theta} - 1}{x_1^{2-\theta} - 1} x_1^{k(1-\theta)+k+1} + x_1^{k+1+k(1-\theta)}(x_1^{\beta(1-\theta)} - 1)\right\}$$

$$= \frac{c_1^{2-\theta}}{1-\theta} x_1^{2k+1-k\theta}\left\{\frac{x_1^{1-\theta} - 1}{x_1^{2-\theta} - 1} + x_1^{\beta(1-\theta)} - 1\right\}.$$

The asymptotic value of a_m, if $\theta \neq 2$, is given by

$$a_m/c = H_m^{(\theta-1)} \approx \frac{(c_1 x_1^{k+\beta})^{2-\theta}}{2-\theta} = \frac{c_1^{2-\theta}}{2-\theta} x_1^{2k-\theta k+\beta(2-\theta)}.$$

Therefore, the overallocation as a function of β can be given by

$$(10) \qquad R(\beta) = \frac{(2-\theta)x_1}{(1-\theta)} \frac{\{(x_1^{1-\theta} - 1)/(x_1^{2-\theta} - 1) + x_1^{\beta(1-\theta)} - 1\}}{x_1^{\beta(2-\theta)}}.$$

When $\theta = 0$ this expression reduces to the previously determined formula (9) for the overallocation under a uniform request distribution. When $\theta = 1$ or $\theta = 2$, however, the expression is invalid.

If $\theta=1$ the expression is indeterminate and its value may be calculated by l'Hôpital's rule:

$$R_{\theta=1}(\beta)=\frac{x_1\ln x_1}{x_1-1}\,\frac{1+\beta(x_1-1)}{x_1^{\beta}}. \tag{11}$$

Alternatively, the proper asymptotic series for $H_m^{(1)}$ may be used [3]: $H_m^{(1)}=\ln m+\gamma+1/(2m)-1/(12m^2)+\cdots$. Then the following expressions hold:

$$H_{L_i}^{(1)}-H_{L_{i-1}}^{(1)}\approx\ln(c_1x_1^i)-\ln(c_1x_1^{i-1})=\ln x_1,$$
$$H_m^{(1)}-H_{L_k}^{(1)}\approx\ln(c_1x_1^{k+\beta})-\ln(c_1x_1^k)=\beta\ln x_1.$$

Thus, b_m/c and a_m/c are given by the following expressions:

$$\frac{b_m}{c}\approx\frac{c_1x_1^{k+1}}{x_1-1}\ln x_1+c_1x_1^{k+1}\beta\ln x_1,$$

$$\frac{a_m}{c}=\sum_{1\leqq i\leqq m}i\frac{1}{i}=m\approx c_1x_1^{k+\beta},$$

and this also leads to (11).

When $\theta=2$, use of l'Hôpital's rule gives

$$R_{\theta=2}(\beta)=\frac{x_1-1}{\ln x_1}. \tag{12}$$

Alternatively, it is easily shown that, when $\theta=2$, the following expressions are satisfied:

$$H_{L_i}^{(2)}-H_{L_{i-1}}^{(2)}\approx-(c_1x_1^i)^{-1}+(c_1x_1^{i-1})^{-1}=(c_1x_1^i)^{-1}(x_1-1),$$
$$\sum_k L_i(H_{L_i}^{(2)}-H_{L_{i-1}}^{(2)})\approx\sum_k(x_1-1)=\text{const.}+k(x_1-1).$$

Thus, b_m/c and a_m/c are given by the expressions

$$b_m/c=\text{const.}+k(x_1-1)-x_1(x_1^{-\beta}-1),$$
$$a_m/c=H_m^{(1)}=\ln m+\gamma+\cdots\approx(k+\beta)\ln x_1,$$

and these expressions lead also to (12). (Note that for $\theta=2$ the overallocation is not a function of β.)

If $\theta>2$ then a_m converges to a finite limit, and the methods of this paper are not directly applicable; the overallocation will depend not on the asymptotic behavior of the block sizes, but rather on the actual beginning values of the sequence L_i.

The minimum value of $R(\beta)$ satisfies $R(0)=R(1)=R_{\min}$ and is given by the following formulas:

$$R_{\min}=\frac{(2-\theta)x_1}{1-\theta}\,\frac{x_1^{1-\theta}-1}{x_1^{2-\theta}-1}=\frac{2-\theta}{1-\theta}\,\frac{1-x_1^{\theta-1}}{1-x_1^{\theta-2}},$$

$$R_{\min}(\theta=1)=\frac{x_1\ln x_1}{x_1-1}, \tag{13}$$

$$R_{\min}(\theta=2)=\frac{x_1-1}{\ln x_1}.$$

The maximum value of $R(\beta)$ is more difficult to obtain. The maximum is obtained at a value β_m given by $R'(\beta_m)=0$. Evaluating this expression shows that β_m satisfies

$$x_1^{\beta_m(1-\theta)}=\left\{1-\frac{x_1^{1-\theta}-1}{x_1^{2-\theta}-1}\right\}(2-\theta).$$

By substituting this expression in the formula for $R(\beta)$, it is seen that $R_{\max}$ is given by

(14) $$R_{\max}=R(\beta_m)=x_1/x_1^{\beta_m}=x_1\left\{\left[1-\frac{x_1^{1-\theta}-1}{x_1^{2-\theta}-1}\right](2-\theta)\right\}^{-1/(1-\theta)}.$$

When $\theta=1$ the following expressions are obtained:

$$\beta_m=\frac{1}{\ln x_1}-\frac{1}{x_1-1},$$

$$R_{\max}(\theta=1)=\frac{x_1}{e}x_1^{1/(x_1-1)}.$$

When $\theta=2$, $R_{\max}$ is independent of β and $R_{\max}=(x_1-1)/\ln x_1$.

The integral from $\beta=0$ to $\beta=1$ gives an average value for the overallocation

$$\begin{aligned}R_{\text{avg}}&=\int_0^1 R(\beta)\,d\beta\\&=\frac{(2-\theta)x_1}{(1-\theta)}\left\{\left[\frac{x_1^{1-\theta}-1}{x_1^{2-\theta}-1}-1\right]\int_0^1 x_1^{-\beta(2-\theta)}\,d\beta+\int_0^1 x_1^{-\beta}\,d\beta\right\}\\&=\frac{(2-\theta)x_1}{(1-\theta)\ln x_1}\left\{\frac{[(x_1^{2-\theta}-1)-(x_1^{1-\theta}-1)](x_1^{\theta-2})(x_1^{2-\theta}-1)}{(x_1^{2-\theta}-1)(\theta-2)}+\frac{x_1-1}{x_1}\right\}\\&=\frac{(2-\theta)(x_1-1)}{(1-\theta)\ln x_1}\left\{\frac{1}{\theta-1}+1\right\}\\&=\frac{x_1-1}{\ln x_1}.\end{aligned}$$

Thus the expected value of the overallocation ratio is *independent* of the parameter θ.

5. Discussion of analytic results. The overallocation in a simple buddy system depends on the largest real root x_1 of the corresponding characteristic equation, on the size m of the maximum request (through the parameters α and β), and on the distribution of request sizes (through the parameter θ).

For a particular simple buddy system, i.e., a given value of x_1, expressions (6) and (7) give $R_{\min}$ and $R_{\max}$, the minimum and maximum amounts of overallocation due to internal fragmentation. Whether the actual overallocation is given by either of these bounds, or instead by some intermediate value, depends on the size m of the maximum request. If the value of m is unknown, it is reasonable to choose m randomly; then the expected value of overallocation is given by expression (8) for R_{avg}. The value of R_{min}, R_{max}, and R_{avg} are plotted in Fig. 1 as a function of x_1. Note that x_1 need never be larger than 2, since the binary buddy system ($u_n = 2u_{n-1}$, or $x_1 = 2$) has less overallocation than any simple buddy scheme with $x_1 > 2$.

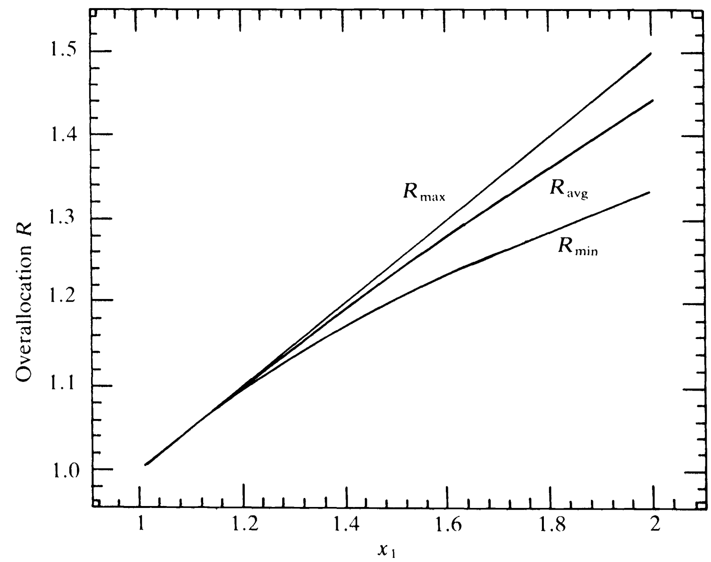


FIG. 1. R_{min}, R_{max}, *and* R_{avg} *vs.* x_1

As x_1 approaches 1, the overallocation also approaches 1. Intuitively, as x_1 approaches 1, there is a larger number of block sizes that are available to be allocated and there is therefore a higher probability of being able to allocate a block of exactly the requested size; the waste due to assigning a block that is larger than requested is reduced. The expressions for R_{min}, R_{max}, and R_{avg} can be expanded around the point $x_1 = 1$ to better illustrate the limiting behavior as x_1 approaches 1.

$$R_{\mathrm{min}}(\theta = 0) = \frac{2x_1}{x_1 + 1}$$

$$= 1 + \tfrac{1}{2}(x_1 - 1) - \frac{(x_1 - 1)^2}{4} + \frac{(x_1 - 1)^3}{8} - \frac{(x_1 - 1)^4}{16} + \cdots,$$

$$R_{\mathrm{avg}}(\text{all } \theta) = \frac{x_1 - 1}{\ln x_1}$$

$$= 1 + \tfrac{1}{2}(x_1 - 1) - \frac{(x_1 - 1)^2}{12} + \frac{(x_1 - 1)^3}{24} - \frac{19(x_1 - 1)^4}{720} + \cdots,$$

$$R_{\mathrm{max}}(\theta = 0) = \tfrac{1}{2}(x_1 + 1) = 1 + \tfrac{1}{2}(x_1 - 1).$$

The functions plotted in Fig. 1 represent the widest bounds that can be placed on overallocation. This occurs for a uniform request distribution, when $\theta = 0$ in the parameterized Zipf distribution. The dependence of the overallocation on the maximum request size for different values of θ, i.e., different request distributions, is shown in Fig. 2. Figure 2a gives a family of curves showing the overallocation for a binary buddy system ($x_1 = 2$) for various values of θ. The x-axis represents the maximum size of a requested block and is the parameter β; the y-axis shows the resulting overallocation. The curve with the widest excursions is the plot for a uniform distribution of requests; the minimum value is $1.33 \cdots$ and the maximum value is 1.5. As θ increases the minimum value of overallocation increases and the maximum value decreases. When $\theta = 2$ the overallocation is constant and is equal to R_{avg}. Fig. 2b shows the same function for a Fibonacci buddy system ($x_1 = \varphi = 1.618 \cdots$). The two graphs are almost identical; note, however, the difference in vertical scale between the graphs for the binary buddy system and the Fibonacci buddy system.

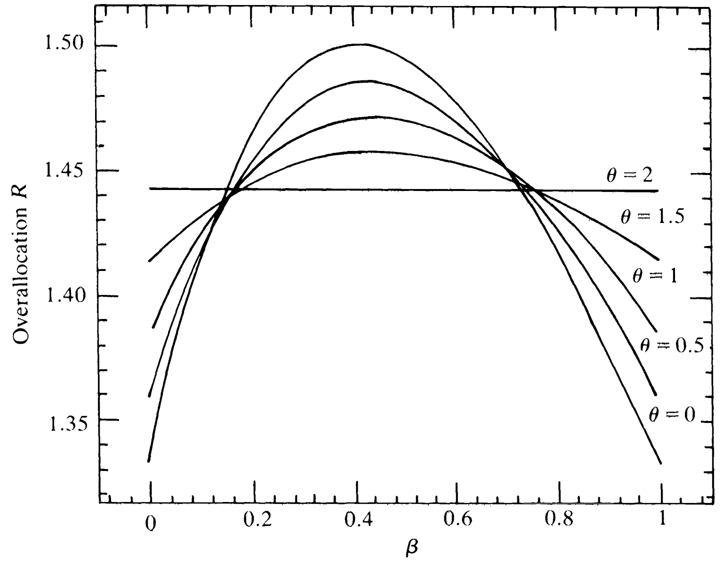


FIG. 2a. *Overallocation in the binary buddy system* ($x_1 = 2$)

In Fig. 3, the values of $R_{\min}$ and $R_{\max}$ are plotted as a function of θ, for θ in the range $0 \leqq \theta \leqq 2$, for three separate simple buddy systems: binary buddy, Fibonacci buddy, and a hypothetical buddy system with $x_1 = 1.3$. (This approximates the simple buddy system defined by $u_n = u_{n-1} + u_{n-6}$, for which $x_1 = 1.285 \cdots$, and which has block sizes $1, 2, 3, 4, 5, 6, 7, 9, 12, 16, 21, 27, 34, 43, 55, \cdots$). Again it is clear that, as θ approaches the value 2, $R_{\min}$ and $R_{\max}$ both approach the expected value R_{avg}.

6. Conclusions. Whether or not the analytic results presented in this paper have any practical significance depends on several factors.

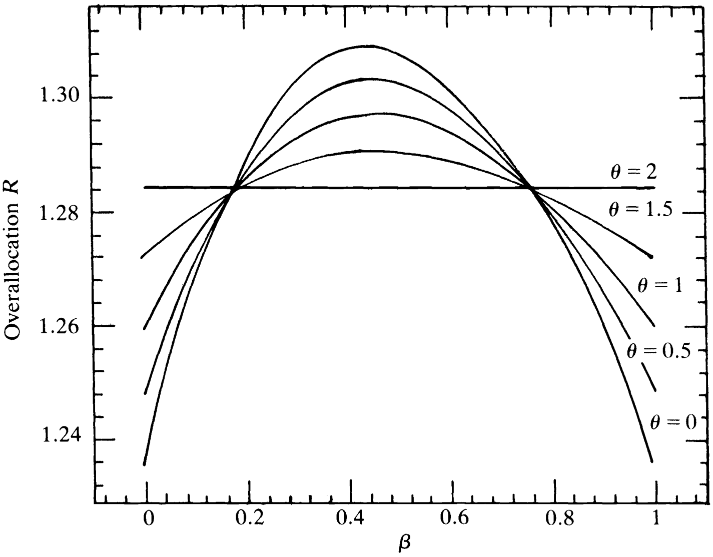


FIG. 2b. *Overallocation in the Fibonacci buddy system* ($x_1 = \varphi = 1.618 \cdots$)

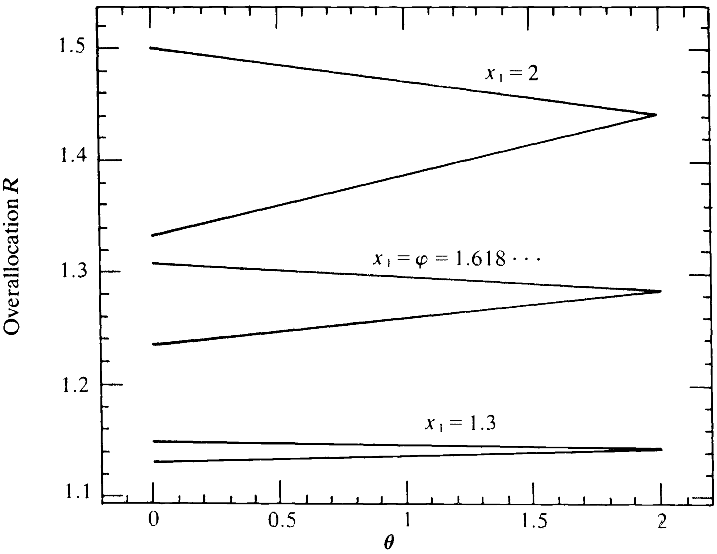


FIG. 3. *$R_{\min}$ and $R_{\max}$ for three simple buddy systems*

One of the most important of these factors concerns the assumed distribution of request sizes. The parameterized Zipf distribution was used in the analysis largely because it was mathematically tractable. At the same time, it represents a large family of request distributions with the not-unreasonable property that the frequency of requests decreases as the size of the requested block increases. The most remarkable aspect of the Zipf request distribution, however, is that *the expected value of the overallocation is independent of the parameter θ.* Regardless of the rapidity with which the frequency of requests decreases with increasing block size, the expected value of overallocation remains the same. Intuitively, this would seem to indicate a relative insensitivity of simple buddy systems to the actual request distribution. Furthermore, it indicates that execution time variations in the actual request distribution may not be overly significant. Thus, in many cases it may be possible to estimate the overallocation due to internal fragmentation without possessing detailed information about the precise usage patterns of the system.

University of Maryland		Brigham Young University		CP-67	
Size	CDF(%)	Size	CDF(%)	Size	PDF(%)
2	0.0	3	0.0	1	11.1
8	36.0	16	6.4	2	0.2
10	44.0	32	16.8	3	3.7
15	54.0	48	27.6	4	24.8
25	84.0	64	40.0	5	21.9
30	94.0	80	45.8	6	0.3
35	96.5	96	62.7	7	0.6
40	97.5	112	82.6	8	11.2
50	98.5	128	94.9	9	2.0
70	99.3	144	95.3	10	4.1
100	99.6	160	95.7	11	0.2
200	100.0	176	96.1	12	0.2
		192	96.4	17	0.9
		208	97.0	18	1.9
		224	98.3	21	0.2
		256	99.4	23	0.3
		272	99.6	27	0.1
		304	99.8	29	15.6
		352	99.9	31	0.4
		511	100.0	50	0.3

FIG. 4. *Actual request distributions*

To test the analytic results of this paper against more practical situations, published data for three actual request distributions was used. These distributions, listed in Fig. 4, were the distribution of buffer requests for the UNIVAC 1108 Exec 8 system at the University of Maryland [1], the distribution of partition size requests on the IBM 360/65 OS MVT system at Brigham Young University [8], and the distribution of memory requests on an IBM CP-67 system [7], [8]. The Maryland and BYU distributions are described by cumulative distribution functions; the probability density function is assumed to be linear between tabulated points. The CP-67 request distribution is described by a probability density function; the probability of a request for a block whose size is not tabulated is zero.

In each case, the actual internal fragmentation, as represented by the actual overallocation, was calculated by the formula $R = \sum p_i s_i / \sum i p_i$ where the probabilities p_i were determined by the request distributions of Fig. 4 and the s_i are the block sizes for the particular buddy system. Three buddy systems were tested:

binary buddy, Fibonacci buddy, and a simple buddy system defined by $u_n = u_{n-1} + u_{n-3}$ (denoted F-2 below). The actual overallocation and the overallocation predicted by the analytic approximations of this paper are shown in Fig. 5.

The predicted analytic values for the Maryland and BYU distributions are not far from, and in fact somewhat overestimate, the actual values for overallocation. Note that the actual overallocation, as well as the predicted overallocation, decreases as x_1 becomes closer to 1, i.e., in the order binary buddy, Fibonacci buddy, F-2. On the other hand, the CP-67 distribution, which does not even approximate a decreasing probability of block request with increasing block size, is not well described by the predicted values.

		Overallocation			
			Predicted		
Buddy system	Distribution	Actual	$R_{\min}$	R_{avg}	$R_{\max}$
Binary	U. Maryland BYU CP-67	1.38 1.29 1.22	1.33	1.44	1.50
Fibonacci	U. Maryland BYU CP-67	1.25 1.28 1.15	1.24	1.29	1.39
F-2	U. Maryland BYU CP-67	1.18 1.19 1.27	1.19	1.22	1.23

FIG. 5. *Overallocation using actual distributions*

Of course, internal fragmentation is only one aspect of the problem of dynamic storage allocation. External fragmentation can also lead to the loss of usable memory. Previous work by Randell [9] and Knuth [3] seems to show that internal fragmentation is often more important than external fragmentation in determining the memory utilization of a dynamic storage algorithm. On the other hand, simulation studies by Peterson and Norman [8] on buddy systems characterized by $u_n = u_{n-1} + u_{n-p}$ indicate that, at least in some cases, as memory lost to internal fragmentation *decreases*, the amount of memory lost to external fragmentation *increases*. According to their results, as much as 35–40% of memory may remain unusable due to a combination of both internal and external fragmentation.

Current analyses of external fragmentation have not yet been sufficiently developed to determine in what circumstances the above observation of constant total fragmentation may occur. Until they are, the bounds and estimates on overallocation due to internal fragmentation presented here will help act as guidelines in designing buddy systems for dynamic storage allocation.

REFERENCES

[1] D. S. HIRSCHBERG, *A class of dynamic memory allocation algorithms*, Comm. ACM, 16 (1973), pp. 615–618.

[2] K. C. Knowlton, *A fast storage allocator*, Ibid., 8 (1965), pp. 623–625.

[3] D. E. Knuth, *The Art of Computer Programming, Volume* 1: *Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1968, pp. 111, 435–455.

[4] ———, *The Art of Computer Programming, Volume* 2: *Seminumerical Algorithms*, Addison-Wesley, Reading, MA, 1969, pp. 219–229.

[5] ———, *The Art of Computer Programming, Volume* 3: *Sorting and Searching*, Addison-Wesley, Reading, MA, 1973, pp. 397–398, 666.

[6] C. L. Liu, *Introduction to Combinatorial Mathematics*, McGraw-Hill, New York, 1968.

[7] B. H. Margolin, R. P. Parmelee and M. Schatzoff, *Analysis of free-storage algorithms*, IBM System J., 10 (1971), pp. 283–304.

[8] J. L. Peterson and T. A. Norman, *Buddy systems*, Comm. ACM, 20 (1977), pp. 421–431.

[9] B. Randell, *A note on storage fragmentation and program segmentation*, Ibid., 12 (1969), pp. 365–369, 372.

[10] R. Sedgewick, *A Fibonacci buddy system*, unpublished, (1972), 18 pp.

[11] K. K. Shen and J. L. Peterson, *A weighted buddy method for dynamic storage allocation*, Comm. ACM, 17 (1974), pp. 558–562.

SIAM J. COMPUT.
Vol. 6, No. 4, December 1977

A NEW ALGORITHM FOR MINIMUM COST BINARY TREES*

ADRIANO M. GARSIA AND MICHELLE L. WACHS†

Abstract. A new algorithm for constructing minimum cost binary trees in $O(n \log n)$ time is presented. The algorithm is similar to the well-known Hu–Tucker algorithm. Our proof of validity is based on finite variational methods and is therefore quite different and somewhat simpler than the proof for the Hu–Tucker algorithm. Our proof also yields some additional information about the structure of minimum cost binary trees. This permits a linear time implementation of our algorithm in a special case.

Key words. algorithms, binary trees, minimum cost trees

Introduction. We shall be concerned here with the class $\mathcal{T}$ of binary trees with positive real numbers appended at their terminal nodes. Examples of such trees are shown in Fig. 1.

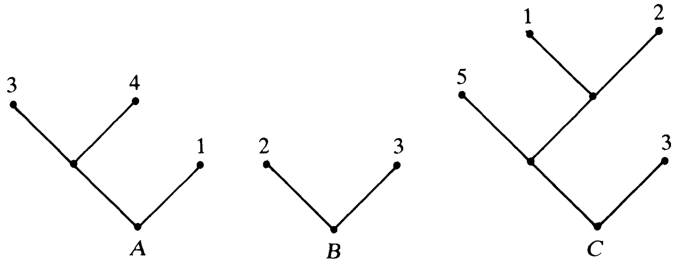


FIG. 1

The class $\mathcal{T}$ can be defined as the smallest class which satisfies the following:

1) Each positive real is in $\mathcal{T}$.

2) If T_1 and T_2 are in $\mathcal{T}$ then $T_1 \vee T_2$ is in $\mathcal{T}$.

To each tree T in $\mathcal{T}$ we associate a weighted path length or cost which we will refer to as $w(T)$ and which is defined by the formula

$$w(T) = \sum_{\nu=1}^{n} h_\nu p_\nu$$

where $p_1, p_2, \cdots, p_n$ are the numbers appearing at the terminal nodes[1] of T and $h_1, h_2, \cdots, h_n$ are their respective "levels". More precisely h_ν is by definition equal to the number of nonterminal nodes we encounter in the path leading from the root of T to the terminal node of p_ν.

For instance, for the trees in Fig. 1 we have

$$w(A) = 2 \cdot 3 + 2 \cdot 4 + 1 \cdot 1 = 15,$$

$$w(B) = 1 \cdot 2 + 1 \cdot 3 = 5,$$

$$w(C) = 2 \cdot 5 + 3 \cdot 1 + 3 \cdot 2 + 1 \cdot 3 = 22.$$

* Received by the editors December 15, 1975, and in final revised form December 27, 1976.

† Department of Mathematics, University of California, San Diego, La Jolla, California 92037.

[1] See [6] for definition.

A problem which occurs in applications is that of constructing a tree which is of minimum cost among all $T \in \mathcal{T}$ which have given positive reals $p_1, p_2, \cdots, p_n$ (in the given order) at their terminal nodes. We will refer to these minimum cost trees as minimal trees.

Several algorithms have been devised for constructing such minimal trees (see [1], [2] and [4]). The most widely accepted of these is due to Hu and Tucker (see [5, pp. 433–447]).

In this paper, by a systematic use of the "variational" technique we obtain some information about the structure of minimal trees. As a consequence we derive a new algorithm closely related to the Hu–Tucker algorithm but easier to justify. To do this we develop a language that is somewhat more amenable to computations than drawings of trees.

The paper is divided into five sections. In the first section we describe the algorithm. In the second we introduce the language and some of the notation. In the third section we state the main results. In the fourth section we give the proofs. Finally, in the last section we derive some further consequences, including a few comments on how the algorithm can be implemented.

1. The algorithm. The algorithm is composed of two parts. In the first part starting from the given numbers

$$p_1, p_2, \cdots, p_n$$

we construct a binary tree $T_B \in \mathcal{T}$ which has a permutation

$$p_{\sigma_1}, p_{\sigma_2}, \cdots, p_{\sigma_n}$$

of the given numbers at the terminal nodes. In general this permutation is not the identity, so T_B is bad (not a solution).

In the second part we construct the good tree $T_G \in \mathcal{T}$ which has $p_1, p_2, \cdots, p_n$ at its terminal nodes (in the right order) and where each p_ν appears in T_G at the same level it appeared in T_B.

The reader familiar with the Hu–Tucker algorithm will find that this second part is the same in both algorithms. In addition, it can be shown that the tree T_B produced by the first part of our algorithm is very closely related to that given by the first part of the Hu–Tucker algorithm. The tree T_B is obtained in steps and each step is identical. We start with a list of given numbers

$$p_1, p_2, \cdots, p_n$$

and produce a new list of $n-1$ numbers by the following procedure.

1) We locate the right-most minimal sum pair of adjacent entries. Let that be p_{i-1}, p_i.[2]
2) We next locate the first entry (if any) to the right of p_i that is greater than or equal to $p_{i-1}+p_i$. Let that be p_{i+k+1}. Then the new list is

$$p_1, p_2, \cdots, p_{i-2}, p_{i+1}, \cdots, p_{i+k}, (p_{i-1}+p_i), p_{i+k+1}, \cdots, p_n.$$

[2] Actually any of the R.M. pairs defined in (3.1) will do; however, here we mean the one that is farthest to the right.

3) If no such entry exists then the new list is

$$p_1, p_2, \cdots, p_{i-2}, p_{i+1}, \cdots, p_n, (p_{i-1}+p_i).$$

Each of these steps generates a node of T_B. After $n-1$ steps we are left with just one number in the list and the first part of the algorithm is completed. To obtain the desired tree T_B from these lists, assuming that we do it by hand, we draw *dotted* lines joining our entries (to keep track of them as they move about) and *solid* lines joining p_{i-1} to $p_{i-1}+p_i$, and p_i to $p_{i-1}+p_i$.

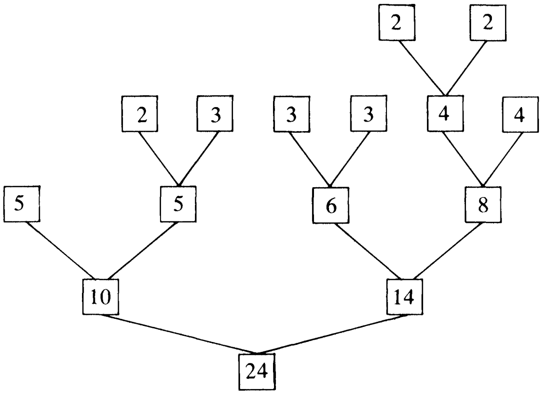
Once we are finished the tree T_B can be immediately read off the resulting picture. This is best understood when carried out on a specific example. For instance, if we start with the numbers

$$5\,3\,3\,2\,2\,2\,3\,4$$

and follow our recipe we end up with Fig. 2.

This is actually a badly drawn picture of the binary tree T_B which is to be the result of the first part of our construction.

Our tree T_B is, indeed, none other than the one obtained by untangling the lines in Fig. 2, i.e. T_B is the tree:



The good tree T_G we are seeking is then the unique tree whose terminal nodes labeled

(1.1) $$5\,3\,3\,2\,2\,2\,3\,4$$

appear at the same level as they do in T_B. (It is part of the burden of proof that such a tree does indeed exist).

To obtain these levels we need not (in fact, it is better not to) draw the untangled picture of T_B. They can simply be read off the Fig. 2 by counting for each entry the number of solid lines encountered as we go along the tree from that entry down to the root.

In the present example, for the entries in (1.1) we obtain the respective levels

(1.2) $$2\,3\,3\,3\,4\,4\,3\,3.$$

A simple construction then yields that in this case the desired tree T_G is Fig. 3.

Perhaps the simplest way to obtain T_G from the list of levels is to draw dotted lines to represent each of the levels. Then place the entries one after the other at their respective levels. Then level by level, (starting from the highest level, and proceeding on down) from left to right we join pairs of roots of previously obtained subtrees until we are left with just one tree. Thus Fig. 3 can be viewed as the end product of the following sequence of figures (Figs. 4–6).

2. The language. To simplify our presentation we shall represent our trees as words of a language $\mathscr{L}$ whose syntax follows very closely the syntax of our family $\mathscr{T}$ of binary trees.

The precise definition of $\mathscr{L}$ can be given as follows:

1) *Alphabet.* (a) The 3 symbols "(", "$\bigwedge$" and ")" are in the alphabet.
(b) Every positive real is in the alphabet.

2) *Syntax.* (a) Every positive real is a word in $\mathscr{L}$.
(b) If T_1 and T_2 are words in $\mathscr{L}$ then $(T_1 \bigwedge T_2)$ is a word in $\mathscr{L}$.

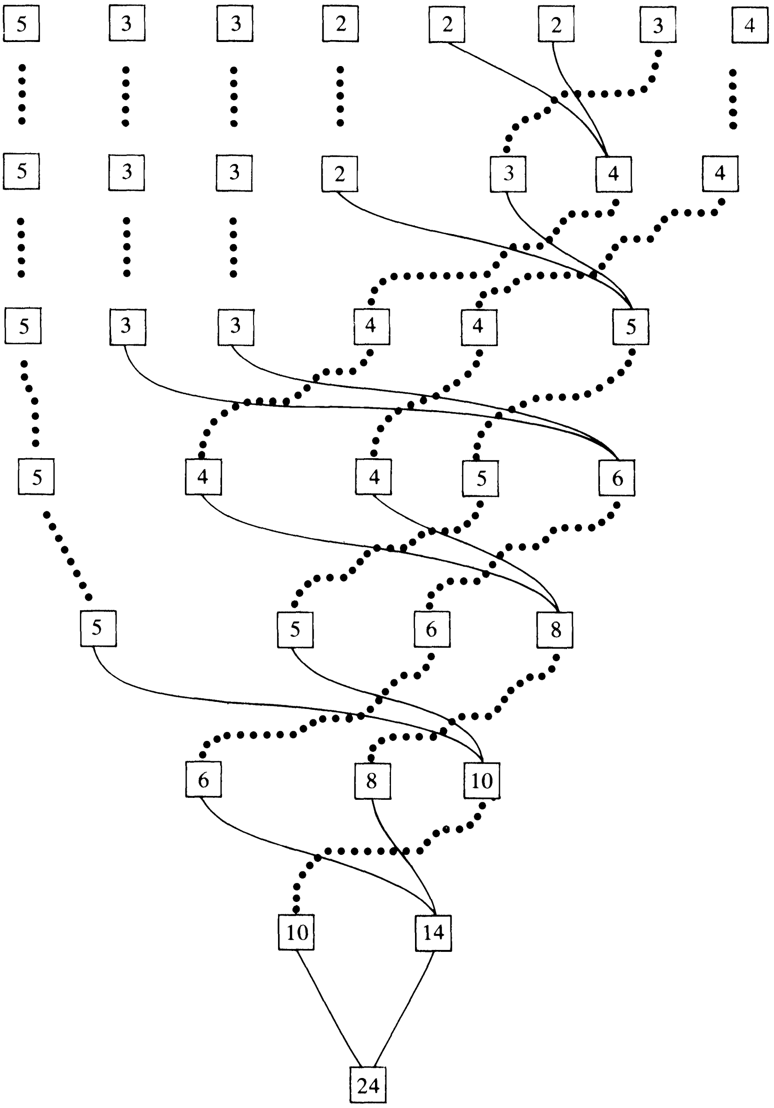


FIG. 2

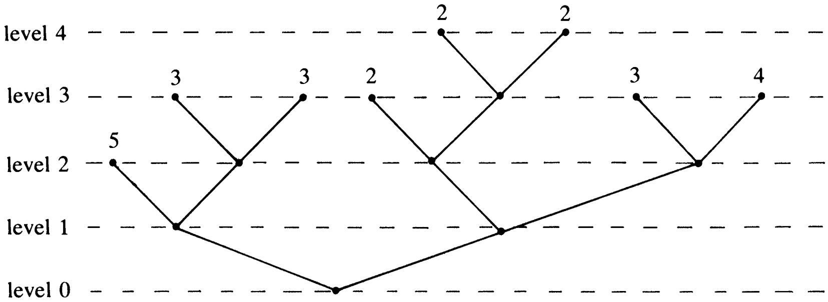


FIG. 3

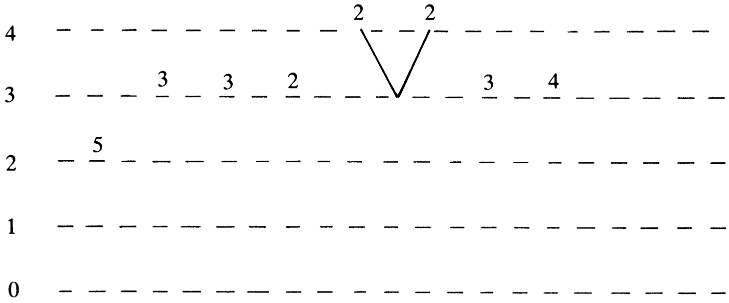


FIG. 4

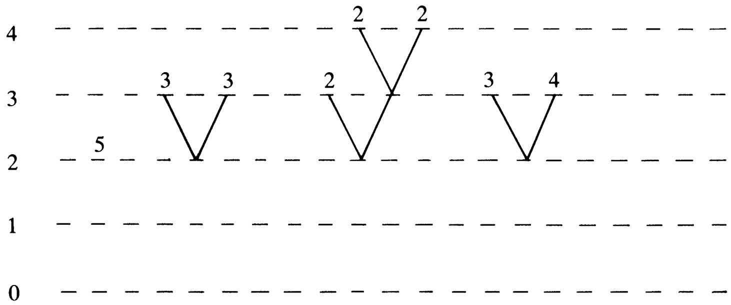


FIG. 5

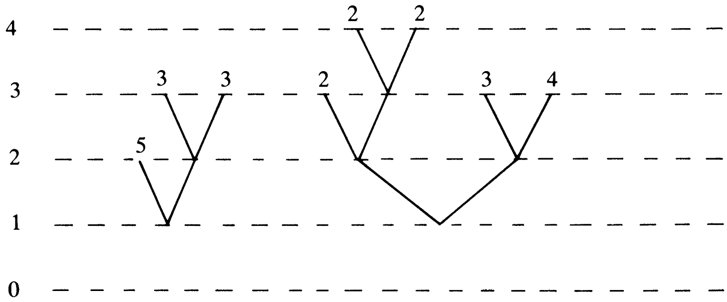


FIG. 6

For instance, if this language is used the trees A, B and C of Fig. 1 are represented by the words in $\mathscr{L}$:

$$A=((3\wedge 4)\wedge 1),\quad B=(2\wedge 3),\quad C=((5\wedge(1\wedge 2))\wedge 3).$$

Sometimes it will be convenient to use an abbreviation in writing parentheses. To this end the symbol "${}_h[$", where h is a nonnegative integer, is to mean a string of h consecutive left parentheses "(".

Similarly "$]_h$" is to mean a string of h consecutive right parentheses ")".

For instance, instead of

$$((3\wedge(1\wedge 2))\wedge((5\wedge 6)\wedge 3))$$

we can also write

$${}_2[3\wedge(1\wedge 2]_2\wedge{}_2[5\wedge 6)\wedge 3]_2.$$

Before we can state our results we need to introduce some further notation.

DEFINITION 2.1. If $T_1, T_2, \cdots, T_n$ are words in $\mathscr{L}$, by $\mathscr{S}(T_1, T_2, \cdots, T_n)$ we mean the set of all words in $\mathscr{L}$ that can be obtained by inserting the symbols "(", "$\wedge$" and ")" in between $T_1, T_2, \cdots, T_n$.

For instance,

$$\mathscr{S}(T_1, T_2)=\{(T_1\wedge T_2)\};$$

$$\mathscr{S}(T_1, T_2, T_3)=\{(T_1\wedge(T_2\wedge T_3)), ((T_1\wedge T_2)\wedge T_3)\};$$

$$\begin{aligned}\mathscr{S}(T_1, T_2, T_3, T_4)=\{&(((T_1\wedge T_2)\wedge T_3)\wedge T_4), ((T_1\wedge(T_2\wedge T_3))\wedge T_4),\\ &((T_1\wedge T_2)\wedge(T_3\wedge T_4)), (T_1\wedge((T_2\wedge T_3)\wedge T_4)),\\ &(T_1\wedge(T_2\wedge(T_3\wedge T_4)))\}.\end{aligned}$$

DEFINITION 2.2. If T is a word, the sum of all the reals appearing in T will be denoted by "$|T|$".

DEFINITION 2.3. If T is a word in $\mathscr{L}$ the "weight" of T will be denoted by $w(T)$ and is recursively defined as follows:

1) If T is a real then $w(T)=0$.

2) If $T=(T_1\wedge T_2)$ then $w(T)=|T|+w(T_1)+w(T_2)$.

It is not difficult to see that the weight of a word T in $\mathscr{L}$ is precisely the same as the weighted path length of the tree corresponding to T. As a matter of fact, from our definition we can easily derive that if $p_1, p_2, \cdots, p_n$ are positive reals and $T\in\mathscr{S}(p_1, p_2, \cdots, p_n)$ then

$$w(T)=\sum_{i=1}^{n} h_i p_i \tag{2.1}$$

where h_i is equal to the number of left parentheses minus the number of right parentheses that are to the left of p_i in T.

Here and in the following we shall refer to the integer h_i appearing in (2.1) as the "*level*" of p_i in T.

DEFINITION 2.4. Let U and V be words in $\mathcal{S}(p_1, p_2, \cdots, p_n)$ and $\mathcal{S}(q_1, q_2, \cdots, q_n)$ resectively. Let h_i be the level of p_i in U and k_i be the level of q_i in V. We shall say that U is a "*rearrangement*" of V if and only if there is a permutation $(\sigma_1, \sigma_2, \cdots, \sigma_n)$ of $(1, 2, \cdots, n)$ such that

$$p_i = q_{\sigma_i}, \qquad h_i = k_{\sigma_i}, \qquad i = 1, 2, \cdots, n.$$

Finally, if $\mathcal{S}$ is a set of words we shall set

$$w(\mathcal{S}) = \min_{T \in \mathcal{S}} w(T).$$

We shall also say that T is "*minimal*" for $\mathcal{S}$ if and only if

$$w(T) = w(\mathcal{S}).$$

We are now ready to state our results.

3. The basic theorem.

Let $p_1, p_2, \cdots, p_n$ be given positive reals. We shall say that the pair p_{i-1}, p_i is "right minimal" (briefly R.M.) if and only if we have

$$(3.1) \qquad \begin{aligned} &\text{(a)} \quad p_{i-2} + p_{i-1} \geqq p_{i-1} + p_i \quad (\text{when } i > 2), \\ &\text{(b)} \quad p_{i-1} + p_i < p_{j-1} + p_j \qquad \forall j > i. \end{aligned}$$

In particular we see that if $i + 1 \leqq n$ we must also have

$$(3.2) \qquad p_{i-2} + p_{i+1} > p_{i-1} + p_i.$$

Here and in the following we shall set, for a fixed i,

$$\begin{aligned} \mathcal{S} &= \mathcal{S}(p_1, p_2, \cdots, p_n), \\ \mathcal{S}_0 &= \mathcal{S}(p_1, p_2, \cdots, p_{i-2}, (p_{i-1} \wedge p_i), p_{i+1}, \cdots, p_n), \\ \mathcal{S}_k &= \mathcal{S}(p_1, p_2, \cdots, p_{i-2}, p_{i+1}, \cdots, p_{i+k}, (p_{i-1} \wedge p_i), p_{i+k+1}, \cdots, p_n). \end{aligned}$$

Our basic result can be stated as follows:

THEOREM 3.1. *Let the pair p_{i-1}, p_i be right minimal and $k \geqq 0$ be such that*

$$(3.3) \qquad p_{i+j} < p_{i-1} + p_i \quad \text{for } 0 \leqq j \leqq k.$$

Then, if

$$(3.4) \qquad i + k = n \quad \text{or} \quad p_{i+k+1} \geqq p_{i-1} + p_i$$

we have

$$w(\mathcal{S}) = w(\mathcal{S}_k)$$

and every minimal word in $\mathcal{S}_k$ has a rearrangement in $\mathcal{S}$.

This theorem is an immediate consequence of the following four lemmas.

LEMMA 3.1. *Let p_{i-1}, p_i be R.M. and $k \geqq 0$ be such that*

$$(3.5) \qquad p_{i+j} < p_{i-1} + p_i \quad \text{for } 0 \leqq j \leqq k;$$

then

$$w(\mathcal{S}_k) \geqq w(\mathcal{S}). \tag{3.6}$$

Furthermore, when equality holds, every minimal word in $\mathcal{S}_k$ *has a rearrangement in* $\mathcal{S}$.

LEMMA 3.2. *Let* p_{i-1}, p_i *be R.M. and for some* $k \geqq 1$,

$$p_{i+k} < p_{i-1} + p_i. \tag{3.7}$$

Then, if $w(\mathcal{S}_{k-1}) < w(\mathcal{S}_k)$ *we must have*:

(a) $i+k+1 \leqq n$ *and* $p_{i+k+1} < p_{i-1} + p_i$.

(b) *Every minimal word of* $\mathcal{S}_{k-1}$ *is of the form*

$$T = \alpha(p_{i-1} \wedge p_i)\underset{h}{]}\wedge\underset{h}{[}(p_{i+k} \wedge p_{i+k+1})\beta,^{3}$$

where $h \geqq 0$.

LEMMA 3.3. *Let* $p_1, p_2, \cdots, p_n$ *be positive reals such that for some* i

$$\begin{aligned} &\text{(a)} \quad p_{i-2} + p_{i-1} \geqq p_{i-1} + p_i, \\ &\text{(b)} \quad p_{i-1} + p_i \leqq p_i + p_{i+1} \quad (\textit{if } i+1 \leqq n), \\ &\text{(c)} \quad p_{i-1} + p_i \leqq p_{i+1} + p_{i+2} \quad (\textit{if } i+2 \leqq n) \end{aligned} \tag{3.8}$$

and

$$w(\mathcal{S}) < w(\mathcal{S}_0). \tag{3.9}$$

Then we must necessarily have $i+1 \leqq n$, $p_{i+1} < p_{i-1} + p_i$ *and every minimal word in* $\mathcal{S}(p_1, p_2, \cdots, p_n)$ *is of the form*

$$T = \alpha(A \wedge p_{i-1})\underset{h}{]}\wedge\underset{h}{[}(p_i \wedge p_{i+1})\beta.$$

LEMMA 3.4. *Let* p_{i-1}, p_i *be R.M. and* $k \geqq 1$ *be such that*

$$p_{i+j} < p_{i-1} + p_i \quad \textit{for } 0 \leqq j \leqq k;$$

then

$$w(\mathcal{S}) = \{\min w(\mathcal{S}_{k-1}), w(\mathcal{S}_k)\}. \tag{3.10}$$

Assume that $p_1, p_2, \cdots, p_n$ satisfy the hypotheses of Theorem 3.1. Then:

1) For $k \geqq 1$, Lemma 3.2 gives that we cannot have $w(\mathcal{S}_{k-1}) < w(\mathcal{S}_k)$. Thus Lemma 3.4 gives $w(\mathcal{S}) = w(\mathcal{S}_k)$. But then Lemma 3.1 gives that every minimal word in $\mathcal{S}_k$ has a rearrangement in $\mathcal{S}$.

2) For $k = 0$, Lemma 3.3 gives that we cannot have $w(\mathcal{S}) < w(\mathcal{S}_0)$. Thus $w(\mathcal{S}) = w(\mathcal{S}_0)$ and since $\mathcal{S}_0 \subset \mathcal{S}$ there is nothing to add in this case.

We therefore only need to establish the four lemmas to prove Theorem 3.1. This will be done in the next section.

[3] Here α and β represent whatever is needed to complete the expression into a word in $\mathcal{L}$.

4. Proofs.

1) *Proof of Lemma* 3.1. The result is trivially true for $k=0$ since $\mathcal{S}_0=\mathcal{S}(p_1,\cdots,p_{i-2},(p_{i-1}\wedge p_i),\cdots,p_n)$ is a subset of $\mathcal{S}=\mathcal{S}(p_1,\cdots,p_n)$. By induction, let us then assume it to be true up to $k-1$.

Let T be a minimal word in $\mathcal{S}_k$. Then

$$T=\alpha p_{i+k}\underset{h_1}{]}\wedge\underset{h_2}{[}(p_{i-1}\wedge p_i)\beta.$$

Case A. If $h_2\geqq h_1$ set

$$T'=\alpha(p_{i-1}\wedge p_i)\underset{h_1}{]}\wedge\underset{h_2}{[}p_{i+k}\beta.$$

It is easy to see that

$$w(T)-w(T')=(h_2-h_1)[p_{i-1}+p_i-p_{i+k}]\geqq 0. \tag{4.1}$$

Since $T'\in\mathcal{S}_{k-1}$ we get (using the induction hypothesis)

$$w(\mathcal{S}_k)=w(T)\geqq w(T')\geqq w(\mathcal{S}_{k-1})\geqq w(\mathcal{S}). \tag{4.2}$$

Furthermore, if $w(\mathcal{S}_k)=w(\mathcal{S})$ the equality sign must hold in (4.1) and (4.2). But, in view of (3.5) this can only happen if $h_1=h_2$. But then T' is a rearrangement of T and the induction hypothesis gives that T' has a rearrangement in $\mathcal{S}$.

Case B. Let $h_1>h_2$ or equivalently $h_1\geqq h_2+1$. In this case we can write T in the form

$$T=\alpha'(Q\wedge p_{i+k})\underset{h_1-1}{]}\wedge\underset{h_2}{[}(p_{i-1}\wedge p_i)\beta$$

where Q is a word in $\mathcal{L}$ and α' is whatever is needed to complete T.

We then let

$$T'=\alpha' Q\underset{h_1-1}{]}\wedge\underset{h_2}{[}(p_{i+k}\wedge(p_{i-1}\wedge p_i))\beta.$$

Since $T'\in\mathcal{S}_k$ and T is minimal for $\mathcal{S}_k$ we must have $w(T')\geqq w(T)$. This gives

$$0\geqq w(T)-w(T')=|Q|+(h_1-h_2-1)p_{i+k}-(p_{i-1}+p_i).$$

Now Q must have an entry equal to p_{i+k-1} if $k>1$ and to p_{i-2} if $k=1$. Thus we get

(a) (if $k>1$) $p_{i+k-1}+(h_1-h_2-1)p_{i+k}\leqq p_{i-1}+p_i<p_{i+k-1}+p_{i+k}$,

(b) (if $k=1$) $p_{i-2}+(h_1-h_2-1)p_{i+1}\leqq p_{i-1}+p_i<p_{i-2}+p_{i+1}$.[4]

We see then that in either case we must have $h_1=h_2+1$, and Q must be equal to p_{i+k-1} for $k>1$. In other words, we have

(a) (if $k>1$) $T=\alpha'(p_{i+k-1}\wedge p_{i+k})\underset{h_2}{]}\wedge\underset{h_2}{[}(p_{i-1}\wedge p_i)\beta$,

(b) (if $k=1$) $T=\alpha'(Q\wedge p_{i+1})\underset{h_2}{]}\wedge\underset{h_2}{[}(p_{i-1}\wedge p_i)\beta$.

[4] See (3.2).

In the first case we note that the word

$$T''=\alpha'(p_{i-1}\wedge p_i)\underset{h_2}{]}\wedge\underset{h_2}{[}(p_{i+k-1}\wedge p_{i+k})\beta$$

is a rearrangement of T in $\mathscr{S}_{k-2}$, so the induction hypothesis yields $w(\mathscr{S}_k \geqq w(\mathscr{S}_{k-2}) \geqq w(\mathscr{S})$ as well as the statement about the rearrangements. The second case is even simpler because the word

$$T'''=\alpha'(Q\wedge p_{i-1})\underset{h_2}{]}\wedge\underset{h_2}{[}(p_i\wedge p_{i+1})\beta$$

is already the rearrangement of T in $\mathscr{S}$. Q.E.D.

2) *Proof of Lemma* 3.2. Let T be a minimal word in $\mathscr{S}_{k-1}$. Then

$$T=\alpha(p_{i-1}\wedge p_i)\underset{h_1}{]}\wedge\underset{h_2}{[}p_{i+k}\beta.$$

Note then that the word

$$T'=\alpha p_{i+k}\underset{h_1}{]}\wedge\underset{h_2}{[}(p_{i-1}\wedge p_i)\beta$$

is in $\mathscr{S}_k$, so if $w(\mathscr{S}_k)>w(\mathscr{S}_{k-1})$ we must necessarily have $w(T')>w(T)$. Thus

$$0>w(T)-w(T')=(h_1-h_2)[p_{i-1}+p_i-p_{i+k}],$$

and (3.7) then gives us that $h_2>h_1$. Since this implies that $h_2\geqq 1$ we must conclude that $i+k<n$ and that T can also be written in the form

$$T=\alpha(p_{i-1}\wedge p_i)\underset{h_1}{]}\wedge\underset{h_2-1}{[}(p_{i+k}\wedge R)\beta$$

where R is a word in $\mathscr{L}$ whose first numerical entry is p_{i+k+1}. This given, let

$$T'=\alpha(p_{i+k}\wedge(p_{i-1}\wedge p_i))\underset{h_1}{]}\wedge\underset{h_2-1}{[}R\beta',$$

and note that

$$w(T)-w(T')=|R|+(h_2-h_1-1)p_{i+k}-p_{i-1}-p_i.$$

Since again $T'\in\mathscr{S}_k$ we must have $w(T)<w(T')$; thus

$$p_{i+k+1}+(h_2-h_1-1)p_{i+k}\leqq|R|+(h_2-h_1-1)p_{i+k}<p_{i-1}+p_i.$$

The right minimality of p_{i-1}, p_i implies that $p_{i-1}+p_i<p_{i+k+1}+p_{i+k}$. Thus $h_2-h_1-1\geqq 1$ is impossible and we must conclude that

(a) $h_2=h_1+1$,

(b) $R=p_{i+k+1}$,

(c) $p_{i+k+1}<p_{i-1}+p_i$.

All this brings T to the desired form

$$T=\alpha(p_{i-1}\wedge p_i)\underset{h_1}{]}\wedge\underset{h_1}{[}(p_{i+k}\wedge p_{i+k+1})\beta'$$

and the proof of the lemma is complete.

3) *Proof of Lemma* 3.3. Let T be a minimal word in S. Then

$$T=\alpha p_{i-1}\underset{h_1}{]}\bigwedge\underset{h_2}{[}p_i\beta.$$

Clearly we cannot have $h_1=h_2=0$ for otherwise $T\in\mathscr{S}_0$, and this would contradict (3.9).

Case A. Say $h_1\geqq 1$. Then T can be written in the form

$$T=\alpha'(A\bigwedge p_{i-1})\underset{h_1-1}{]}\bigwedge\underset{h_2}{[}p_i\beta$$

where A is a word in $\mathscr{L}$ whose last numerical entry is p_{i-2}. Let

$$T'=\alpha' A\underset{h_1-1}{]}\bigwedge\underset{h_2}{[}(p_{i-1}\bigwedge p_i)\beta.$$

Since $T'\in\mathscr{S}_0$ we must have $w(T')>w(T)$; thus

$$0>w(T)-w(T')=|A|+(h_1-h_2-1)p_{i-1}-p_i, \tag{4.3}$$

and $|A|\geqq p_{i-2}$ gives (using (3.8) (a))

$$p_{i-2}+(h_1-h_2-1)p_{i-1}<p_i\leqq p_{i-2}.$$

This means $h_1<h_2+1$; thus $h_1\leqq h_2$. In particular also $h_2\geqq 1$.

Case B. Say $h_2\geqq 1$. Then T can be written in the form

$$T=\alpha p_{i-1}\underset{h_1}{]}\bigwedge\underset{h_2-1}{[}(p_i\bigwedge B)\beta'.$$

Let

$$T'=\alpha(p_{i-1}\bigwedge p_i)\underset{h_1}{]}\bigwedge\underset{h_2-1}{[}B\beta'.$$

Again, since $T'\in\mathscr{S}_0$ we must have $w(T')>w(T)$. This gives (using (3.8) (b))

$$|B|+(h_2-h_1-1)p_i<p_{i-1}\leqq p_{i+1} \tag{4.4}$$

from which we easily deduce $h_2\leqq h_1$. So again $h_1\geqq 1$. Combining the results of the two cases with (4.3) and (4.4) we deduce that we must have

(a) $h_1=h_2\geqq 1$,
(b) $|A|<p_{i-1}+p_i$,
(c) $|B|<p_{i-1}+p_i$.

The last of these inequalities (with (3.8) (c)) then forces $B=p_{i+1}$ leaving T in the form asserted by the lemma.

We can now proceed to the proof of our last lemma. This will be carried out by induction on k. Indeed, the result just shown gives us the first step in the induction argument that will prove (3.10). For under the hypotheses of Lemma 3.4 if $w(\mathscr{S})<w(\mathscr{S}_0)$ every minimal word in $\mathscr{S}$ must be of the form

$$T=\alpha(A\bigwedge p_{i-1})\underset{h}{]}\bigwedge\underset{h}{[}(p_i\bigwedge p_{i+1})\beta.$$

But then the word

$$T'=\alpha(A\bigwedge p_{i+1})\underset{h}{]}\bigwedge\underset{h}{[}(p_{i-1}\bigwedge p_i)\beta$$

is a rearrangement of T in $\mathcal{S}_1$. This gives

$$w(\mathcal{S})=w(T)=w(T')\geqq w(\mathcal{S}_1).$$

So Lemma 3.1 gives $w(\mathcal{S})=w(\mathcal{S}_1)$.

Assume that we have carried this through up to $k-1$ ($\geqq 1$). This given, if

$$w(\mathcal{S})<w(\mathcal{S}_{k-1})$$

then we must have

$$w(\mathcal{S}_{k-2})=w(\mathcal{S})<w(\mathcal{S}_{k-1}).$$

So Lemma 3.2 gives that every minimal word in $\mathcal{S}_{k-2}$ is of the form

$$T=\alpha(p_{i-1}\wedge p_i)\underset{h}{]}\wedge\underset{h}{[}(p_{i+k-1}\wedge p_{i+k})\beta.$$

But then the word

$$T'=\alpha(p_{i+k-1}\wedge p_{i+k})\underset{h}{]}\wedge\underset{h}{[}(p_{i-1}\wedge p_i)\beta$$

is a rearrangement of T in $\mathcal{S}_k$. This gives

$$w(\mathcal{S})=w(T)=w(T')\geqq w(\mathcal{S}_k)$$

and Lemma 3.1 gives

$$w(\mathcal{S})=w(\mathcal{S}_k).$$

This completes our proofs.

5. Further consequences. There are some interesting consequences of our arguments that are worthwhile mentioning here. To this end given a pair p_{i-1}, p_i we shall say that "$i+k$" is in the "*right range*" of p_{i-1}, p_i if and only if we have

$$p_{i-1}+p_i\geqq p_j \quad \text{for } j=i+1, i+2, \cdots, i+k. \tag{5.1}$$

Similarly we shall say that "$i-h$" is in the "*left range*" of p_{i-1}, p_i if and only if

$$p_{i-1}+p_i\geqq p_j \quad \text{for } j=i-1, \cdots, i-h. \tag{5.2}$$

Now, the arguments of the last section yield also the following remarkable fact.

THEOREM 5.1. *Let $i-h$ and $i+k$ be in the left and right ranges of p_{i-1}, p_i, where $h>2$ and $k>1$, and suppose that*

$$p_{i-1}+p_i\leqq p_{j-1}+p_j \quad \textit{for } i-h<j\leqq i+k. \tag{5.3}$$

Then there is a minimal word in $\mathcal{S}(p_1, p_2, \cdots, p_n)$ with p_{i-1}, p_i at the same level and each p_j ($i-h\leqq j\leqq i+k$) falling at the same or one level below that of p_{i-1}, p_i.

Proof. Let us assume first that $p_1, p_2, \cdots, p_n$ have the following properties for some $h>2$, $k>1$ and $i>1$.

$$\begin{aligned}&\text{(a)}\quad p_{i-1}+p_i>p_j \quad \text{for } i-h\leqq j\leqq i+k,\\ &\text{(b)}\quad p_{i-1}+p_i<p_{j-1}+p_j \quad \text{for } i-h<j\leqq i+k \ (j\neq i),\\ &\text{(c)}\quad \forall T_1, T_2\in\mathcal{S}(p_1, \cdots, p_n),\ w(T_1)\neq w(T_2).\end{aligned} \tag{5.4}$$

This given, it is not difficult to check that the proofs of Lemmas 3.1, 3.2, 3.3 and 3.4 remain valid also under these assumptions.[5]

Thus, with the notation of § 3, we must have

$$w(\mathcal{S})=\min\{w(\mathcal{S}_{\nu-1}),\ w(\mathcal{S}_\nu)\} \tag{5.5}$$

for $\nu=1, 2, \cdots, k$.

Consequently, if for a given ν

$$w(\mathcal{S}_\nu)=w(\mathcal{S})$$

then the proof of Lemma 3.1 gives that a minimal word in $\mathcal{S}_\nu$ has to be of the form

$$T=\alpha p_{i+\nu}\underset{h}{]}\wedge\underset{h}{[}(p_{i-1}\wedge p_i)\beta$$

or

$$(5.6)\qquad \begin{aligned} &\text{(a)}\quad (\text{if } \nu>1)\ T=\alpha(p_{i+\nu-1}\wedge p_{i+\nu})\underset{h}{]}\wedge\underset{h}{[}(p_{i-1}\wedge p_i)\beta,\\ &\text{(b)}\quad (\text{if } \nu=1)\ T=\alpha(Q\wedge p_{i+1})\underset{h}{]}\wedge\underset{h}{[}(p_{i-1}\wedge p_i)\beta. \end{aligned}$$

We see that in the first case $p_{i+\nu}$ is one level below the pair p_{i-1}, p_i and in the second case $p_{i+\nu}$ is at the same level as the pair p_{i-1}, p_i.

On the other hand, if

$$w(\mathcal{S}_\nu)>w(\mathcal{S})$$

then by (5.5) we must have

$$w(\mathcal{S})=w(\mathcal{S}_{\nu-1})<w(\mathcal{S}_\nu).$$

But then Lemma 3.2 gives that a minimal word for $\mathcal{S}_{\nu-1}$ has to be of the form

$$T=\alpha(p_{i-1}\wedge p_i)\underset{h}{]}\wedge\underset{h}{[}(p_{i+\nu}\wedge p_{i+\nu+1})\beta,$$

in which case $p_{i+\nu}$ is at the same level as the pair p_{i-1}, p_i.

Because of Lemma 3.1 and the uniqueness of the minimal word in $\mathcal{S}$ (given by (5.4) (c)) all of these words are rearrangements of the same word in $\mathcal{S}(p_1, p_2, \cdots, p_n)$. This proves that the minimal word in $\mathcal{S}(p_1, p_2, \cdots, p_n)$ has the desired properties to the right of the pair p_{i-1}, p_i. By symmetry the same must hold to the left.

We shall complete the proof by reducing the general case to the case so far considered. Note that every result so far proved holds not only for p_j which are positive real numbers, but also for p_j selected from a real finite-dimensional vector space with lexicographic ordering on vectors. By the hypothesis of the theorem we have the following inequalities:

$$(5.7)\qquad \begin{aligned} &\text{(a)}\quad p_{i-1}+p_i\geqq p_j \quad \text{for } i-h\leqq j\leqq i+k,\\ &\text{(b)}\quad p_{i-1}+p_i\leqq p_{j-1}+p_j \quad \text{for } i-h<j\leqq i+k\ (j\neq i). \end{aligned}$$

[5] This is essentially due to the fact that the right minimality of p_{i-1}, p_i was never fully used there.

By replacing the positive reals p_j with appropriately defined vectors p'_j, we can make these inequalities strict and also satisfy condition (5.4) (c).

Let R_{n+2} be the real $(n+2)$-dimensional vector space with lexicographic ordering of vectors. For $1 \leqq j \leqq n$, let $p'_j \in R_{n+2}$ be defined as follows:

$$p'_j = (p_j, 1, 0, \cdots, \underset{\substack{\uparrow \\ \text{position } j+2}}{1}, 0, \cdots, 0) \quad \text{if } j < i-1,$$

$$p'_j = (p_j, 1, 0, \cdots, 0, 1, 0) \qquad \text{if } j = i-1,$$

$$p'_j = (p_j, 1, 0, \cdots, 0, 1) \qquad \text{if } j = i,$$

$$p'_j = (p_j, 1, 0, \cdots, \underset{\substack{\uparrow \\ \text{position } j}}{1}, 0, \cdots, 0) \quad \text{if } j > i.$$

Note the following facts:

1) Any word T minimal for vectors p'_j is also minimal for reals p_j.

2) The definition of the first two components of the vectors p'_j implies, using (5.7) (a), that

$$(5.8) \qquad p'_{i-1} + p'_i > p'_j \quad \text{for } i-h \leqq j \leqq i+k.$$

3) The definition of the first component and the last n components the vectors p'_j implies, using (5.7) (b), that

$$(5.9) \qquad p'_{i-1} + p'_i < p'_{j-1} + p'_j \quad \text{for } i-h < j \leqq i+k \ (j \neq i).$$

4) The weight of any word T for vectors p'_j is a vector which encodes, in the last n components, the levels of all p'_j in T. Thus any two distinct words have different weights with respect to the vectors p'_j.

It follows that the unique minimal word for vectors p'_j satisfies the theorem, and since this word is minimal for reals p_j, the theorem holds for reals p_j.

It is interesting to note that we can also prove Theorem 5.1 by varying all the numbers p_j by some small ε to make (5.4) (a), (b), (c) hold and then taking the limit as ε tends to zero. In the proof we have actually given, the unit vectors $(0, \cdots, 0, 1, 0, \cdots, 0)$ behave essentially as infinitesimals of various sizes.

LEMMA 5.1. *Suppose $p_i \leqq p_{i+1}$ for all $1 \leqq i < n$. Then some minimal word in $\mathscr{L}$ has the property that the level of p_i is at least as large as the level of p_{i+1}, for $1 \leqq i < n$.*

Proof. We assume $p_i < p_{i+1}$ for all $1 \leqq i < n$ and prove that *any* minimal word has the desired property. The lemma then follows by using a suitably defined collection of vectors in place of positive reals, as in the proof of Theorem 5.1.

Let T be a minimal word. Then

$$T = \alpha p_i \underset{h}{]} \wedge \underset{k}{[} p_{i+1} \beta.$$

If $k \doteq 0$ the lemma holds for i; thus assume $k > 0$. That is

$$T = \alpha p_i \underset{h}{]} \wedge \underset{k-1}{[} (p_{i+1} \wedge A) \beta.$$

Consider

$$T' = \alpha (p_i \wedge p_{i+1}) \underset{h}{]} \wedge \underset{k-1}{[} A\beta.$$

Since T is minimal, $w(T)-w(T')\leqq 0$, that is,

$$-p_i+(k-h-1)p_{i+1}+|A|\leqq 0.$$

But $|A|\geqq p_{i+2}>p_i$; thus $(k-h-1)<0$, and $k\leqq h$. Thus the lemma holds for i.

To complete the proof, suppose $p_i\leqq p_{i+1}$ for all $1\leqq i<n$. Let R_2 be the real two-dimensional vector space, with vectors ordered lexicographically. For $1\leqq j\leqq n$, let $p'_j\in R_2$ be defined by

$$p'_j=(p_j, j).$$

Then $p'_i<p'_{i+1}$ for all $1\leqq i<n$, and any minimal word for vectors p'_j satisfies the lemma. But such a word must also be minimal for reals p_j. Q.E.D.

Now we present a linear-time algorithm to construct a minimum cost tree for the case when the minimum sum pair p_{i-1}, p_i satisfies $p_{i-1}+p_i\geqq p_j$ for all j.

Let $\nu=n-2^k$, where 2^k is the largest power of two no greater than n.

The algorithm consists of two steps.

Step 1. Find the largest i such that $p_{i-2}+p_{i-1}\geqq p_{i-1}+p_i$. Delete the pair p_{i-1}, p_i. Repeat this step until either all elements are paired or only a singleton remains.

Step 2. Select the smallest ν pairs among those pairs found by Step 1. Place the elements in these pairs at level $k+1$ of the minimal tree and place the remaining elements at level k.

Proof of correctness. To verify that the algorithm is correct, we must show that a tree with levels as specified by the algorithm actually exists and that such a tree is minimal. The proof depends mainly on Theorem 5.1 and Lemma 5.1. To deal with ties, we must introduce vectors as in the previous two proofs.

Suppose we run the algorithm. Let $(p_{i_1}, p_{j_1}), \cdots, (p_{i_m}, p_{j_m})$ be the pairs (in increasing order of $p_{i_l}+p_{j_l}$) found by Step 1 of the algorithm; here $m=\lfloor n/2\rfloor$. Let $(p_{i_1}, p_{j_1}), \cdots, (p_{i_\nu}, p_{j_\nu})$ be the pairs selected by Step 2 of the algorithm. Let R_{n+2} be the real $(n+2)$-dimensional vector space, and let $p'_k\in R_{n+2}$ for $1\leqq k\leqq n$ be defined as follows:

$$p'_k=(p_k, 1, 0, \cdots, 0, 1, 0, \cdots, 0),$$

where the position of the last 1 in p'_k depends on the value of k. Values of k from 1 through n are associated with components 3 through $n+2$ of the vector space as follows: if $k=i_l$ for some $1\leqq l\leqq m$, k is associated with component $m+3-l$. The remaining values of k are associated with components $m+3$ through $n+2$ in increasing order of k.

The following facts imply the validity of the algorithm.

1) $p_{i-1}+p_i\geqq p_j$ for all i, j by hypothesis, so $p'_{i-1}+p'_i>p'_j$, considering vector components one and two.

2) The cost of any tree with respect to the vectors p'_k encodes the levels of all terminal nodes, so each tree has a distinct cost.

3) By 1), 2), and Theorem 5.1, the unique minimal tree for vectors p'_k has at most two levels. This tree is also minimal for reals p_k.

4) $p_{i_l}+p_{j_l}\leqq p_{i_{l+1}}+p_{j_{l+1}}$ for $1\leqq l<m$ by hypothesis, so $p'_{i_l}+p'_{j_l}<p'_{i_{l+1}}+p'_{j_{l+1}}$ for $1\leqq l<m$, considering vector components one through $m+2$.

5) Consider the first pair $p_{i-1}+p_i$ found by Step 1 of the algorithm. Note that we must have $p_k<p_{k+2}$ for $i-1\leqq k\leqq n-2$ and $p_{i-2}\geqq p_i$. Thus $p'_k<p'_{k+2}$ for

$i-1 \leqq k \leqq n-2$; and $p'_{i-2} > p'_i$, since p_i is *not* the left half of some pair found by Step 1.

6) It follows from 5) by induction that Step 1 applied to vectors p'_k finds exactly the same pairs as Step 1 applied to reals p_k. From 4) it then follows that the algorithm selects exactly the same ν pairs when applied to vectors p'_k as it did when applied to reals p_k.

7) Suppose we apply the original algorithm to the list of vectors $p'_1, \cdots, p'_n$ and let the algorithm run until at most one of the original vectors p'_k is not yet combined with other items. It is easy to see that the result will be the list $q_1 = p'_{i_1} + p'_{j_1}, q_2 = p'_{i_2} + p'_{j_2}, \cdots, q_m = p'_{i_m} + p'_{j_m}$ if $n = 2m$, and the list $q_0, q_1 = p'_{i_1} + p'_{j_1}, \cdots, q_m = p'_{i_m} + p'_{j_m}$ if $n = 2m+1$, where q_0 is a unique unpaired vector p'_l.

8) By 1) and 4), $q_k < q_{k+1}$ for $1 \leqq k < m$ if n is even, $q_k < q_{k+1}$ for $0 \leqq k < m$ if n is odd. By Lemma 5.1, the unique minimal tree T' for vectors q_k has the levels of the terminal nodes in nonincreasing order. But substituting $(p_{i_k} \wedge p_{j_k})$ for each q_k must give a rearrangement of the minimal tree T for the original vectors p'_k. This tree has only two levels. It follows that, in T, items from the pairs $(p'_{i_1}, p'_{j_1}), \cdots, (p'_{i_\nu}, p'_{j_\nu})$ must be at the higher level, and the remaining items must be at the lower level. But T is also a minimal tree for reals p_k. T is the tree constructed by the algorithm for vectors p'_k, and by 6) T is also constructed for reals p_k. Thus the algorithm is valid.

One of the interesting consequences of this result is that, at least when the minimal pair sum exceeds every p_j, finding a minimal word in $\mathscr{S}(p_1, \cdots, p_n)$ depends essentially on selecting the smallest ν of a given list of numbers. Using the techniques in [5, pp. 216–218] an algorithm for producing such a minimal tree can be implemented on the computer in linear time. This is carried out in full detail in the Appendix.

It is easy to show that if we apply this linear time algorithm to arbitrary $p_1, \cdots, p_n$, we will construct a word of minimum weight among all the words in which the levels of the p_i's differ by at most one.

Since there are $O(4^n)$ possible words in $\mathscr{S}(p_1, p_2, \cdots, p_n)$ it is quite plausible that a linear time implementation is possible in general. To this end some further results of the type given in Theorem 5.1 might be helpful. Indeed, by studying the "local" minimum pair sums p_{i-1}, p_i and how their "right" and "left" ranges overlap additional information concerning minimal words may be obtained. This suggests some interesting directions for future research.

Using the results of the present paper it is not difficult to put together an algorithm for the general case which requires no more than $O(n \log n)$ comparisons.

This can be roughly described as follows. Given $p_1, p_2, \cdots, p_n$ we first find the last i such that $p_{i-2} + p_{i-1} \geqq p_{i-1} + p_i$. Next we locate the first p_j with $j > i$ such that $p_j > p_{i-1} + p_i$. Finally we remove p_{i-1}, p_i and insert $p_{i-1} + p_i$ immediately before p_j (or at the end if such p_j does not exist).

Then proceed in the same way on the new list of numbers. Note now we will have

$$p_{j-2} < p_j$$

for $j = i+1, \cdots, n$. Thus the second step using binary insertion [5, pp. 406–414] requires only $O(\log n)$ comparisons. Since in $n-1$ passes the procedure terminates and the total number of comparisons required by the first step is $O(n)$ it is easy to see that no more than $O(n \log n)$ comparisons are needed by this procedure.

Quite recently R. Tarjan has put together an implementation of our algorithm which requires $O(n \log n)$ total time including data manipulations. This is presented in the following Appendix.

Finally, we would like to mention that using Lemmas 3.1, 3.2 and 3.3 an entirely new proof of the Hu–Tucker algorithm can be obtained. However, this proof is not significantly simpler than the more recent one derived by Hu in [3] and thus will be omitted.

Appendix. Implementation (by R. E. Tarjan). In this Appendix we present implementations of the algorithms discussed in the paper. We implement algorithms for the following problems.

1) Given a binary tree, compute the depth $d(i)$ of each leaf i, $1 \leqq i \leqq n$.

2) Given a set of depths $d(i)$, $1 \leqq i \leqq n$, construct a binary tree whose leaves have depths $d(i)$.

3) Given a list of weights $p(i)$ satisfying the conditions of Theorem 5.1, divide the weights into sets which are on the higher and lower levels of some minimal tree of two levels.

4) Given an arbitrary list of weights $p(i)$, construct a binary tree which is a rearrangement of a minimal tree for weights $p(i)$.

Using algorithms for 3) and 2), we can construct a minimal tree if the conditions of Theorem 5.1 are satisfied. Using algorithms for 4), 1), and 2), we can construct a minimal tree for an arbitrary list of weights.

We represent a binary tree of n leaves by using integers 1 through $2n-1$ to label the vertices. If i is a vertex, $l(i)$ is the left child of i and $r(i)$ is the right child of i. We assume that vertices 1 through n are the leaves, $n+1$ through $2n-1$ the internal vertices, and $2n-1$ the root.

Let T be a tree represented in this fashion. Algorithm DEPTH below will compute the depth of each leaf in T. DEPTH uses a recursively programmed depth-first search to explore T and compute depths as it goes. For further discussion of the depth-first search technique, see [7].

We present the algorithm in ALGOL-like notation. The algorithm is simple enough that its working should be obvious. Variable "b" stores the current value of the depth. The algorithm requires $O(n)$ running time and storage space.

```
DEPTH: begin
    procedure DFS1(v);
        if $v \leqq n$ then $d(v) := b$
        else begin
            $b := b+1$;
            DFS1($l(v)$);
            DFS1($r(v)$);
            $b := b-1$;
        end DFS1;
        $b := 0$;
        $r := 2n-1$;
        $DFS(r)$;
    end DEPTH;
```

To construct a tree with leaves at particular depths, we can also use a depth-first search. This search builds the tree from the root, stopping the construction process for a particular leaf when the correct depth is reached. Algorithm TREE below carries out this construction. The algorithm is straightforward. Variable b keeps track of the depth, i of the last constructed leaf, and w of the last constructed internal vertex. The algorithm requires $O(n)$ time and space. It is interesting to compare this algorithm (efficient for the computer) with the algorithm of § 1 (efficient for hand computation).

```
TREE: begin
    procedure $DFS2(v)$; begin
        $b := b+1$;
        if $b = d(i)$ then begin
            $l(v) = i$;
            $i := i+1$;
        end else begin
            $w := w-1$;
            $l(v) := w$;
            $DFS2(w)$;
        end;
        if $b = d(i)$ then begin
            $r(v) := i$;
            $i := i+1$;
        end else begin
            $w := w-1$;
            $r(v) := w$;
            $DFS2(w)$;
        end;
        $b := b-1$;
        end $DFS2$;
        for $i := 1$ until $n$ do $l(i) := r(i) := 0$;
        $i := 1$;
        $r := w := 2n-1$;
        $b := 0$;
        $DFS2(r)$;
end TREE;
```

The next algorithm, TWOLEVEL, implements Steps 1 and 2 of the algorithm based on Theorem 5.1. The algorithm examines the weights $p(k)$ from right-to-left looking for the rightmost minimal sum pair. As weights are examined, they are moved into an array $q(i)$ (to avoid gaps caused by the deletion of minimal pairs). The variable $index(i)$ records the position of $q(i)$ in the original list of weights $p(k)$. The lth minimal pair p_{i_l}, p_{j_l} deleted by Step 1 is represented in arrays LH, RH, SUM by $LH(l)=i_l$, $RH(l)=j_l$, $SUM(l)=p_{i_l}+p_{j_l}$. The algorithm assumes that a special entry $p(0)$ exists satisfying $p(0)>p(i)$ for all $1\leqq i\leqq n$. The selection of the ν smallest-sum pairs requires $O(n)$ time using techniques discussed in [5, pp. 209–219]. The rest of the algorithm clearly requires $O(n)$ time (and space).

```
TWOLEVEL: begin
    $l := i := 0$;
    $k := n$;
    while $k \geqq 0$ do if ($i \geqq 2$ and $p(k) \geqq q(i-1)$) then begin
        $l := l+1$;
        $LH(l) := index\ (i)$;
        $RH(l) := index(i-1)$;
        $SUM(l) := q(i-1)+q(i)$;
        $i := i-2$;
    end else begin
        $i := i+1$;
        $q(i) := p(k)$;
        $index(i) := k$;
        $k := k-1$;
    end;
    $k := \lfloor \log_2 n \rfloor$;
    $\nu := n-2^k$
    select the smallest $\nu$ values of $SUM(l)$;
    assign depth $k+1$ to the corresponding leaves $LH(l)$, $RH(l)$;
    assign depth $k$ to all remaining leaves;
end TWOLEVEL;
```

The last program implements the main part of the algorithm for arbitrary weights. It is the most complicated algorithm we consider, and it requires a special data structure. We need a data structure in which to store lists $i_1, i_2, \cdots, i_k$ satisfying the property

$$(*)\qquad p(i_j)<p(i_{j+2})\quad \text{for } 1\leqq j\leqq k-2.$$

If this property holds, note that $q(i_j)<q(i_{j-1})$ for $1\leqq j\leqq k-1$, where $q(i_1)=p(i_1)$, $q(i_j)=\max\{p(i_{j-1}), p(i_j)\}$ for $2\leqq j\leqq k$. By storing the list in a balanced binary tree, we can carry out each of the following operations in $O(\log n)$ time [5, pp. 451–470).

(a) If each item i in *list* 1 satisfies $p(i)<p(j)$ for each item j in *list* 2, form a new *list* by concatenating *list* 1 and *list* 2. (If *list* 1 and *list* 2 satisfy (*), then so does the new list.)

This operation includes the operation of adding a new element to the front or back of any list. We denote the operation by "*list* := *list* 1‖*list* 2".

(b) Given a value x, split a *list* into the two lists *list* 1 and *list* 2 such that every element in *list* 1 satisfies $p(i)<x$ and the first element j of *list* 2 satisfies $p(j)\geqq x$. We denote this operation by "split *list* on x into *list* 1, *list* 2".

(c) Locate the jth element of a list.

(d) Delete the first element of a list.

Algorithm MINTREE below computes a rearrangement of a minimum binary tree using the algorithm discussed in this paper. For convenience, the algorithm assumes that $p(0)$ and $p(n+1)$ are special values such that $p(n+1) > p(0) > p(i)$ for $1 \leqq i \leqq n$. When the algorithm locates a new rightmost minimal sum pair $p(i), p(j)$, it deletes i and j from the list of items and forms a new item k with weight $p(k)=p(i)+p(j)$.

```
MINTREE: begin
    procedure COMBINE(x, list 2); begin
        let $i$ be the second element of list (if any);
        while $i$ exists and $(x \geqq p(i))$ do begin
            let $j$ be the first element of list;
            delete $j$, $i$ from list;
            $k := k+1$;
            $p(k) := p(j)+p(i)$;
            $l(k) := j$; $r(k) := i$;
            split list on $p(k)$ into list 3, list;
            add $k$ to end of list 3;
            COMBINE($p(k)$, list 3);
            let $i$ be the second element of list (if any);
        end;
        list := list 2||list 1;
    end COMBINE;
    list := $(n, n+1)$;
    $k := n$;
    for $j := 1$ until $n$ do $l(j) := r(j) := 0$;
    for $j := n-1$ step $-1$ until 0 do COMBINE($p(j)$, $(j)$);
end MINTREE;
```

The crucial observation which validates this algorithm is the following. After the rightmost minimal pair $p_{i-1}+p_i$ is found and inserted in front of some p_{i+k}, the new rightmost minimal pair is either $p_{i+k+1}+p_{i+k+2}$, $p_{i+1}+p_{i+2}$, or involves some p_j with $j<i-1$. Procedure *COMBINE*, after inserting $p_{i-1}+p_i$ in front of p_k, checks all these possibilities, calling itself recursively to check $p_{i+k+1}+p_{i+k+2}$ for minimality. Since only $O(n)$ rightmost minimal pairs are ever formed, MINTREE requires $O(n)$ time plus time for $O(n)$ operations on lists, or $O(n \log n)$ time total. The storage required is $O(n)$.

Acknowledgment. The authors are indebted to S. G. Williamson and J. Evans; to the former for introducing us to the subject and to both for several stimulating conversations. Finally, last but not least we are grateful to R. E. Tarjan for carefully reading the original version of the manuscript and suggesting substantial improvements. Indeed, the last section has been almost entirely rewritten by Tarjan himself.

REFERENCES

[1] E. N. GILBERT AND E. F. MOORE, *Variable length binary encodings*, Bell Systems Tech. J., 38 (1959), pp. 933–968.

[2] T. C. HU AND C. TUCKER, *Optimum computer search trees*, SIAM J. Appl. Math., 21 (1971), pp. 514–532.

[3] T. C. HU, *A new proof of the T-C algorithm*, Ibid., 25 (1971), pp. 83–94.

[4] D. E. KNUTH, *Optimum binary search trees*, Acta Informat. 1 (1971), pp. 14–25.

[5] ———, *The Art of Computer Programming, Vol.* 3. *Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[6] ———, *The Art of Computer Programming, Vol.* 1. *Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1973.

[7] R. TARJAN, *Depth-first search and linear graph algorithms*, this Journal, 1 (1972), pp. 146–160.

SIAM J. COMPUT.
Vol. 6, No. 4, December 1977

DETERMINING THE STABILITY NUMBER OF A GRAPH*

V. CHVÁTAL†

Abstract. We formalize certain rules for deriving upper bounds on the stability number of a graph. The resulting system is powerful enough to (i) encompass the algorithms of Tarjan's type and (ii) provide very short proofs on graphs for which the stability number equals the clique-covering number. However, our main result shows that for almost all graphs with a (sufficiently large) linear number of edges, proofs within our system must have at least exponential length.

Key words. random graphs, stable sets, independent sets, vertex packings, binary trees

1. Introduction. By a *graph*, we shall mean what is sometimes called a Michigan graph: one that is finite, undirected, without loops and multiple edges. A set S of vertices in a graph G is called *independent* or *stable* if no two vertices in S are adjacent; the largest cardinality $\alpha(G)$ of a stable set in G is called the *stability number* of G. Now, let G be a graph and let t be a positive integer such that

$$\alpha(G) \leqq t; \tag{1.1}$$

how laborious is it to verify a proof of (1.1)? Of course, this question has a direct bearing on the conjecture that $P \neq NP$; in particular, the celebrated theorem of Cook [2] suggests that it is extremely time-consuming to verify proofs of (1.1). We shall refrain from elaborating on this interesting point; instead, we direct the reader to [2], [14] and [1]. As for *evaluating* $\alpha(G)$, the best available algorithm is due to Tarjan and Trojanowski [20]: its running time on a graph of order n is $O(2^{n/3})$.

The framework of the present paper is quite modest: restricting the intuitive notion of a proof rather drastically, we shall study the resulting system of "recursive proofs". This system remains powerful enough to

(i) encompass a certain class of algorithms that includes the Tarjan–Trojanowski algorithm,

(ii) provide very short proofs of (1.1) for every graph G whose set of vertices can be covered by $\alpha(G)$ cliques.

Nevertheless, we shall show that there are valid inequalities (1.1) whose proofs must be excessively long. More explicitly, for every sufficiently large d there is a positive ε with the following property: for an overwhelming majority of all graphs G with n vertices and dn edges there are valid inequalities (1.1) whose recursive proofs must have length at least $(1+\varepsilon)^n$. (The assumption that the number of edges of G grows *linearly* with n is crucial: in fact, the conclusion fails as soon as d is allowed to grow beyond every bound. For details, see Proposition 4.1.)

* Received by the editors December 3, 1976.

† Computer Science Department, Stanford University, Stanford, California 94305. This research was supported in part by National Science Foundation Grant MCS 72-03752 A03 and by the Office of Naval Research at Stanford University, and by National Research Council Grant A9211 at University of Montreal.

At this moment, it may be worth pointing out two shortcomings that practitioners sometimes find in results on computational complexity: the worst case criterion and the asymptotic point of view. The first of these objections does not apply to our result at all but the second one certainly does: the numerical values of ε are very small. (One could improve on them by taking a little more care in the computations but even then they probably would not be very impressive.)

In § 2, we point out those properties of random graphs which appear in the proof of the main result: looking at small subgraphs of G, and then extrapolating in a straightforward way, one would expect $\alpha(G)$ to be much larger than it actually is. In that sense, $\alpha(G)$ is very much a "global parameter". And it is precisely this global character which makes the proofs of (1.1) so long. In § 3, we describe a certain class of crude algorithms for evaluating $\alpha(G)$ and then touch briefly upon the more sophisticated algorithm of Tarjan and Trojanowski. That section provides the motivation for the definition of a recursive proof presented in § 4. The exponential that appears in our main result originates from an upper bound on the tail of the hypergeometric distribution; it finds its way into the theorem via a lemma on binary trees which we set aside in § 5.

In the context of another NP-complete problem (namely, that of satisfiability of Boolean expressions), there are many results similar in spirit to ours; most of them can be found in [3]. In particular, the proof system investigated recently by Galil [11] is very much like ours; however, the similarity does not extend beyond the superficial level.

2. Random graphs. In this section, we shall deal with graphs whose vertices are labeled as $v_1, v_2, \cdots, v_n$. Two such graphs may be distinct even if they are isomorphic; hence their total number is $2^{n(n-1)/2}$. If P is a property which a graph may or may not have then we shall denote by $t(P, n)$ the number of those graphs with n vertices which do have the property. Finally, we shall say that *almost all* graphs have the property P if the ratio $t(P, n)/2^{n(n-1)/2}$ tends to one as n tends to infinity. A typical statement of this kind appears in the following lemma. The lemma itself seems to be a part of the graph-theoretical folklore. It appears at least implicitly in a 1947 paper by Erdös [5]; further refinements can be found in works of Matula [17], Grimmett and McDiarmid [12], Erdös and Bollobás [7] and perhaps others.

LEMMA 2.1. *Almost all graphs G of order n have the property that* $\alpha(G) < 2\log n/\log 2$.

Proof. Denote $2\log n/\log 2$, rounded up to the nearest integer, by $k(n)$. Clearly, the number of those graphs of order n for which $\alpha \geqq k$, divided by the number of *all* graphs of order n, does not exceed

$$\binom{n}{k}2^{-k(k-1)/2}. \tag{2.1}$$

By elementary estimations, (2.1) is at most

$$\left(\frac{en}{k}2^{-(k-1)/2}\right)^k. \tag{2.2}$$

For all sufficiently large n, we have

$$\frac{en}{k}2^{-(k-1)/2} \leqq e2^{1/2}/k < .99$$

and so (2.1) tends to zero as n tends to infinity. □

In the theory of random graphs developed by Erdös and Rényi [8], [9], [10], one investigates graphs with n vertices and m edges. Clearly, the number of such graphs is

$$\binom{\binom{n}{2}}{m}. \tag{2.3}$$

We shall denote by $t(P, n, m)$ the number of those graphs with n vertices and m edges which have some property P. If m is a function of n such that each $m(n)$ is a nonnegative integer not exceeding $n(n-1)/2$ and if the ratio of $t(P, n, m)$ to (2.3) tends to one as n tends to infinity then we shall say that *almost all* graphs with n vertices and m edges have the property P. The following lemma has been used by Erdös in [6] and elsewhere. (Throughout the paper, log denotes the natural logarithm.)

LEMMA 2.2. *If $m(n) \geqq 16n$ for all sufficiently large n then almost all graphs G with n vertices and m edges have the property that*

$$\alpha(G) < \frac{n^2}{m}\log\frac{m}{n}. \tag{2.4}$$

Proof. Denote the right-hand side of (2.4), rounded up to the nearest integer by $k(n)$; note that $k(n) \to \infty$ as $n \to \infty$. Clearly, the number of those graphs with n vertices and m edges for which $\alpha \geqq k$, divided by the number of *all* graphs with n vertices and m edges, does not exceed

$$\frac{\binom{n}{k}\binom{\binom{n}{2}-\binom{k}{2}}{m}}{\binom{\binom{n}{2}}{m}}. \tag{2.5}$$

By elementary estimations, (2.5) does not exceed

$$\left(\frac{en}{k}\right)^k\left(1-\frac{k(k-1)}{n(n-1)}\right)^m < \left(\frac{en}{k}\exp\left(-\frac{m(k-1)}{n(n-1)}\right)\right)^k.$$

In addition, we have

$$\frac{en}{k}\exp\left(-\frac{m(k-1)}{n(n-1)}\right) < \frac{em}{n\log(m/n)}\exp\left(-\log\frac{m}{n}+\frac{m}{n^2}\right).$$

Since the last quantity becomes smaller than .99 for all sufficiently large n, we conclude that (2.5) tends to zero as n tends to infinity. □

Next, let us digress a little. When m, n, s are nonnegative integers such that $m \leqq n$ and when t is a positive real number, we shall set $p = m/n$, denote by$\sum^*$ the summation over all integers $j \geqq s(p+t)$ and define

$$B(m,n,s,t)=\sum{}^* \binom{s}{j}\left(\frac{m}{n}\right)^j\left(\frac{n-m}{n}\right)^{s-j},$$

$$H(m,n,s,t)=\sum{}^* \frac{\binom{m}{j}\binom{n-m}{s-j}}{\binom{n}{s}}.$$

Thus B is the familiar "tail of the binomial distribution" and H is the "tail of the hypergeometric distribution". The well-known interpretation of these quantities goes as follows. Imagine a barrel containing n apples, exactly m of which are rotten; take a random sample of s apples. Technically, the sampling can be done in at least two ways. We might pick and examine the apples one by one, each time throwing the apple back into the barrel before reaching in again: this is called sampling *with replacement.* Or we might just grab the s apples at the same time: that is called sampling *without replacement.* Whichever method we use, we should expect about ps rotten apples in the sample. The quantities B and H give the probability that at least $(p+t)s$ rotten apples will appear in the sample with and without replacement, respectively.

An elegant argument (apparently due to S. N. Bernstein) shows that

$$B(m,n,s,t)\leqq\left(\left(\frac{p}{p+t}\right)^{p+t}\left(\frac{1-p}{1-p-t}\right)^{1-p-t}\right)^s.$$

A similar bound for H seems to be far more difficult to establish. A special case of a theorem of Hoeffding ([13, Thm. 4]) states that

$$H(m,n,s,t)\leqq\left(\left(\frac{p}{p+t}\right)^{p+t}\left(\frac{1-p}{1-p-t}\right)^{1-p-t}\right)^s. \tag{2.6}$$

It is a routine matter to convert (2.6) into weaker but more tractable bounds; we are about to do that for $t=p$.

LEMMA 2.3. $H(m,n,s,m/n)\leqq e^{-ms/4n}$.

Proof. If $p>1/2$ then the left-hand side vanishes. If $p<1/2$ then (2.6) implies

$$\frac{1}{s}\log H(m,n,s,p)\leqq 2p\log\frac{1}{2}+(1-2p)\log\left(1+\frac{p}{1-2p}\right)$$
$$\leqq 2p\log\frac{1}{2}+p<-\frac{p}{4}$$

which is the desired conclusion. □

Upper bounds on H are useful in proving statements about random graphs, such as the following one.

LEMMA 2.4. *Almost all graphs G with n vertices and m edges* ($m \geqq 3n$) *have the following property: every subgraph of G induced by s vertices such that*

$$s \geqq \frac{4n^2}{m} \log \frac{m}{n} \tag{2.7}$$

has fewer than $2ms^2/n^2$ edges.

Proof. Clearly, the number of those graphs which do *not* have the property, divided by the number of *all* graphs with n vertices and m edges, does not exceed

$$\sum_s \binom{n}{s} H\left(\binom{s}{2}, \binom{n}{2}, m, \binom{s}{2}\Big/\binom{n}{2}\right). \tag{2.8}$$

By Lemma 2.3, this quantity does not exceed

$$\sum_s \binom{n}{s} \exp\left(-\frac{ms(s-1)}{4n(n-1)}\right) < \sum_s \left(\frac{en}{s} \exp\left(-\frac{m(s-1)}{4n^2}\right)\right)^s.$$

By (2.7), we have

$$\frac{en}{s} \exp\left(-\frac{m(s-1)}{4n^2}\right) \leqq \frac{em}{4n \log(m/n)} \exp\left(\frac{m}{4n^2} - \log \frac{m}{n}\right) < .99.$$

Hence (2.8), being bounded from above by

$$\sum_s (.99)^s < 100(.99)^{4n^2 \log(m/n)/m},$$

tends to zero as n tends to infinity. □

3. Algorithms. In this section, we shall first describe a class of crude algorithms for finding a largest stable set in a graph and point out that by the use of appropriate data structures, the running time of these algorithms can be cut down considerably. Then we shall briefly outline a class of more sophisticated algorithms which we shall call Tarjan algorithms.

Let us suppose that, given a graph $G = (V, E)$ and a subset S of V, we wish to find a largest stable subset A of S. We may begin by choosing a vertex $v \in S$; the desired set A either does not contain v or it does contain v. In the first case, A is a largest stable subset of the set $S_1 = S - \{v\}$; in the second case, $A - \{v\}$ is the largest stable subset of the set S_2 obtained from S by deleting v with all of its neighbors in S. We shall denote S_1 by $S - v$ and S_2 by $S * v$; with this notation, we have

$$\alpha(S) = \max(\alpha(S - v), 1 + \alpha(S * v)).$$

Thus we have reduced the original problem into two similar, but smaller, subproblems: one for $S - v$ and the other for $S * v$.

Now, an algorithm for finding a largest stable set in G suggests itself: begin with $S = V$, do what we have just done and then simply iterate away. One may visualize a binary tree with nodes labeled by subsets of V. The root is labeled by V itself; if a node is labeled by a nonempty set S then its left son is labeled by $S - v$ and its right son is labeled by $S * v$ for some $v \in S$. If G has n vertices altogether and if each vertex has fewer than d neighbors then the tree will have at least $2^{n/d}$ nodes. Of course, that does not mean that the algorithm will create at least $2^{n/d}$

subproblems: different nodes of the tree may have the same label. (To take an extreme example, note that all the leaves of the tree will be labeled by $\varnothing$.)

We shall describe a possible implementation of the algorithm. For definiteness, let us assume that we have a fixed "choice function" f which assigns to each nonempty subset S of V a vertex $f(S) \in S$. Such a function gives rise to an algorithm which we shall call the *f-driven algorithm.*

In its first phase, the algorithm creates a list of certain subsets of V, which will be called *subproblems*. It will be convenient to keep the list ordered, with larger subproblems preceding the smaller ones; within each group of subproblems of the same size, the order may be lexicographic. At each moment, we shall have a partial list of subproblems, with a pointer at one of them. At the very beginning, V will be the only subproblem on the list; the first phase will terminate as soon as the pointer gets to $\varnothing$. When the pointer is at a nonempty set S, we define $S_1 = S - f(S)$ and $S_2 = S * f(S)$. Then we add S_1 and S_2 on the list (unless they are already present), shift the pointer to the successor of S and iterate.

In the second phase, we pass through the list in a reverse order (from $\varnothing$ to V) and evaluate $\alpha(G)$ for each subproblem S. To begin with, we have $\alpha\ (\varnothing) = 0$; for each nonempty subproblem S, we have $\alpha(S) = \max\,(\alpha(S_1), 1 + \alpha(S_2))$.

In the third phase, we shall find a largest stable set A in G. To begin with, let us set $A = \varnothing$ and $S = V$. With each iteration, the set S will shrink; when it will become empty, A will be the desired largest stable set in G. Each iteration is simple. If $\alpha(S) = \alpha(S_1)$ then we replace S by S_1; otherwise $\alpha(S) = 1 + \alpha(S_2)$ in which case we add $f(S)$ to A and replace S by S_2.

It is crucial to use the appropriate data structures when implementing the first phase. Trivially, the number of subproblems on the list never exceeds 2^n. If we implement the list as a *balanced tree* (see [15] or [1]) then each of the look-ups and insertions can be handled within a number of set-comparisons proportional to n. If each $f(S)$ can be evaluated within a steps and if the total number of subproblems is b then the running time of the algorithm is $O(abn^2)$. For at least a few choices of f that come to mind, a is polynomial in n. In that case, b threatens to be the decisive factor in the upper bound.

Needless to say, the number of subproblems depends on the choice function f; for most functions f, that number seems difficult to estimate. To simplify the situation, we shall restrict ourselves to very special choice functions: when the vertices of G are ordered as $v_1, v_2, \cdots, v_n$, the function f chooses that vertex of S which has the smallest subscript. The resulting f-driven algorithm will be called an *order-driven* algorithm.

The following proposition and its corollaries (Propositions 3.2–3.5) are due to Szemerédi. In its statement, $N(k)$ denotes the number of stable subsets of $\{v_1, v_2, \cdots, v_k\}$. Here and later on, we shall find it convenient to denote by $S * T$ the subset of S resulting when all the vertices in T and all their neighbors are deleted.

PROPOSITION 3.1. *The order-driven algorithm applied to a graph with vertices* $v_1, v_2, \cdots, v_n$ *creates at most*

$$1 + \sum_{k=0}^{n-1} \min\,(N(k), 2^{n-k-1})$$

subproblems.

Proof. For each subproblem S, let k be the largest subscript such that $\{v_1, v_2, \cdots, v_k\} \cap S = \varnothing$. It is not difficult to see that

$$S = \{v_{k+1}, v_{k+2}, \cdots, v_n\} * B$$

for some stable subset B of $\{v_1, v_2, \cdots, v_k\}$. Hence for each fixed k, there are at most $N(k)$ subproblems S. In addition, if $k < n$ then there are only 2^{n-k-1} subsets S of $\{v_{k+1}, \cdots, v_n\}$ such that $v_{k+1} \in S$. □

PROPOSITION 3.2. *The order-driven algorithm applied to a graph G of order n such that $\alpha(G) \leqq n/2$ creates at most*

$$n^2\binom{n}{\alpha(G)}$$

subproblems.

Proof. Trivially, we have

$$N(k) \leqq \sum_{i=0}^{\alpha(G)} \binom{k}{i} \leqq n\binom{n}{\alpha(G)}$$

for each k; the rest follows from Proposition 3.1. □

PROPOSITION 3.3. *For almost all graphs G of order n, the order-driven algorithm creates at most*

$$n^{2(1+\log n/\log 2)}$$

subproblems.

The proof follows immediately from Proposition 3.2 and Lemma 2.1.

PROPOSITION 3.4. *If $m(n)/n \to \infty$ then almost all graphs G with n vertices and m edges have the following property: for every constant $c > 1$, the order-driven algorithm on G creates $o(c^n)$ subproblems.*

Proof. By Lemma 2.2, we have $\alpha(G) = o(n)$ for almost all graphs with n vertices and m edges; the rest follows from Proposition 3.2. □

PROPOSITION 3.5. *For every graph with n vertices, the order-driven algorithm creates at most $3 \cdot 2^{(n-1)/2} - 1$ subproblems.*

Proof. We have

$$\sum_{k=0}^{n-1} \min(N(k), 2^{n-k-1}) \leqq \sum_{k=0}^{n-1} \min(2^k, 2^{n-k-1}) \leqq 3 \cdot 2^{(n-1)/2} - 2;$$

the rest follows from Proposition 3.1. □

Note that the bound of Proposition 3.5 is sharp: it is attained by the graph with vertices $v_1, v_2, \cdots, v_{2m+1}$ and edges $v_1v_{2m+1}, v_2v_{2m}, \cdots, v_mv_{m+2}$. Nevertheless, if we can *choose* the ordering of the vertices then the bound can be improved.

PROPOSITION 3.6. *Every graph with n vertices can be ordered in such a way that the order-driven algorithm creates $O(n2^{3n/7})$ subproblems.*

Proof. We shall first describe the ordering and then we shall show that it has the desired property. Suppose that we have already constructed the initial segment $v_1, v_2, \cdots, v_{4t}$ for some $t \geqq 0$. If the graph $H = G - \{v_1, v_2, \cdots, v_{4t}\}$ contains a path $w_1w_2w_3w_4$ then we set $v_{4t+i} = w_i$ for $1 \leqq i \leqq 4$ and iterate. Otherwise each component of H is a star or a triangle. In that case, we denote $4t$ by m and enumerate the vertices of H as $v_{m+1}, v_{m+2}, \cdots, v_n$ in such a way that

(i) the vertices of each component of order j are enumerated as $v_{i+1}, v_{i+2}, \cdots, v_{i+j}$ for some i,

(ii) if that component is a star then v_{i+1} is its center.

It is not difficult to verify that $N(k) \leqq 2^{(3k+1)/4}$ for each $k = 1, 2, \cdots, m$. If $m \geqq 4n/7$ then

$$\sum_{k=0}^{n-1} \min (N(k), 2^{n-k-1}) = O(n 2^{3n/7}).$$

If $m < 4n/7$ then we resort to another argument: note that each subproblem has the form $\{v_{k+1}, v_{k+2}, \cdots, v_n\} * B$ such that $1 \leqq k \leqq n$ and B is a stable subset of $\{v_1, v_2, \cdots, v_m\}$. Since $N(m) \leqq 2^{(3m+1)/4}$, the total number of subproblems is $O(n 2^{3n/7})$. □

It is not unlikely that the bound of Proposition 3.6 can be improved. Let us call a number c ***admissible*** if every graph with n vertices can be ordered in such a way that the order-driven algorithm creates $O(c^n)$ subproblems; let c_0 denote the infimum of all admissible c. By Proposition 3.6, we have $c_0 \leqq 2^{3/7}$; on the other hand, the main result of this paper implies that $c_0 > 1$. What is the exact value of c_0? Similar questions apply to the wider class of f-driven algorithms and to the even wider class of *Tarjan algorithms* which we are about to outline.

As pointed out at the beginning of this section, every f-driven algorithm applied to a graph gives rise to a binary tree whose nodes are labeled by subproblems: if a node x is labeled by a nonempty subproblem S then the left son of x is labeled by $S - v$ and the right son of x is labeled by $S * v$ for some $v \in S$. Elimination of duplications on the list of subproblems amounts to *pruning* the tree: we simply omit nodes whose presence would result in duplicated labels. The idea of Tarjan [19] leads to pruning of a different kind. In an f-driven algorithm, each subproblem S is generated in the form $(V - A) * B$ such that B is a stable set; eventually, such a subproblem yields a stable set of size $\alpha(S) + |B|$. If another subproblem S_1 is generated in the form $(V - A_1) * B_1$ such that $S_1 \subseteq S$ and $|B_1| \leqq |B|$ then S_1 can be discarded: in a sense, S_1 is *dominated* by S. In terms of the binary tree, we might index each node x by the number r of right-hand turns on the path from the root to x; a branch rooted at a node x_1 (labeled by S_1 and indexed by r_1) may be pruned off whenever there is another node x (labeled by S and indexed by r) such that $S_1 \subseteq S$ and $r_1 \leqq r$.

Now we have arrived at two kinds of pruning: these might be called "duplication pruning" and "dominance pruning", the former being (in a sense) a special case of the latter. An f-driven algorithm with the option of using both duplication pruning and dominance pruning to eliminate subproblems will be called a *Tarjan algorithm*. Of course, *systematic* use of dominance pruning may shorten the list of subproblems quite considerably. In terms of running time, however, the means could defeat the purpose: in general, it may take a very long time to decide whether the subproblem that has been just created is dominated by at least one of the subproblems already on the list. Thus it may be wise to pass up the option of (possible) dominance pruning in most cases, resorting to it only in those simple situations where the dominating subproblem is almost staring at us. Such a strategy led Tarjan [19] to an algorithm whose worst-case running time for a graph with n vertices is $O(1.286^n)$. Later on, Tarjan and Trojanowski [20] designed an improved version of that algorithm with running time $O(2^{n/3})$. It may be worth pointing out that these upper bounds come out of rather rudimentary applications of dominance pruning only: the argument does not take duplication pruning into account at all. Thus, it is not inconceivable that (with the subproblems kept in a balanced tree, so that duplication pruning is easy to implement) the worst-case running time of the Tarjan–Trojanowski algorithm is even better than $O(2^{n/3})$. Nevertheless, the main result of this paper implies the existence of a constant $c>1$ and arbitrarily large graphs G with n vertices such that *every* Tarjan algorithm applied to G must create at least c^n different subproblems. (In fact, almost all graphs with n vertices and dn edges have this property as long as d is sufficiently large.)

One more comment: from the practical point of view, the Tarjan–Trojanowski algorithm might be preferable even to (hypothetical) f-driven algorithms creating c^n subproblems for c fairly close to 1. The point is that the *space* requirements of such algorithms would be roughly nc^n whereas the space required by the Tarjan–Trojanowski algorithm is only polynomial in n.

4. Recursive proofs. For the moment, let us deal with an arbitrary but fixed graph $G=(V,E)$. By a *statement*, we shall mean an ordered pair (S,t) such that S is a subset of V and t is a nonnegative integer. (Such a statement is to be interpreted as the inequality $\alpha(S)\leqq t$ which, of course, may be true or false.) By a *recursive proof* of a statement (S,t) over G, we shall mean a sequence of statements

$$(4.1)\qquad (S_i, t_i), \qquad i=0,1,\cdots,m,$$

such that $(S_0, t_0)=(\emptyset, 0)$, $(S_m, t_m)=(S, t)$ and such that each statement (S_k, t_k) with $k\geqq 1$ can be derived from the previous statements (S_i, t_i), $0\leqq i\leqq k$, by at least one of the following two rules.

1. The dichotomy rule: from $(S-v, t_i)$ and $(S*v, t_j)$ we can derive $(S, \max(t_i, 1+t_j))$.

2. The monotone rule: from (S,t) we can derive (S',t') whenever $S'\subseteq S$ and $t'\geqq t$.

Clearly, if (4.1) is a recursive proof of (S,t) then $\alpha(S_i)\leqq t_i$ for every i; in particular, $\alpha(S)\leqq t$. Conversely, if $\alpha(S)\leqq t$ then there is a recursive proof of (S,t). In order to see that, consider the family F of subproblems created by some f-driven algorithm that has just found a largest stable subset of S. Enumerate all the ordered pairs $(S^*, \alpha(S^*))$ with $S^*\in F$ as (4.1) in such a way that $|S_i|\leqq|S_{i+1}|$ for every i. Clearly, the resulting sequence constitutes a recursive proof of $(S,\alpha(S))$; if $t>\alpha(S)$ then one additional application of the monotone rule completes a recursive proof of (S,t).

It will be convenient to define the *length* of (4.1) as m. Now, Propositions 3.1–3.6 yield direct corollaries in terms of recursive proofs. We shall state explicitly only one of them.

PROPOSITION 4.1. *If $c>1$ and if $m(n)/n\to\infty$ then, for almost all graphs $G=(V,E)$ with n vertices and m edges, there are recursive proofs of $(V,\alpha(G))$ of length $o(c^n)$.*

In addition, every Tarjan algorithm applied to $G=(V,E)$ yields a recursive proof of $(V, \alpha(G))$. Hence for every graph $G=(V,E)$ of order n, there is a recursive proof of $(V,\alpha(G))$ of length $O(2^{n/3})$.

Now, we shall show that for a certain class of graphs $G=(V, E)$, there exist very short recursive proofs of $(V, \alpha(G))$. This class consists of all those graphs G for which $\alpha(G)$ equals $\theta(G)$, the smallest number of cliques whose union is V. (Trivially, we have $\alpha(G) \leqq \theta(G)$ for every graph G.) It may be instructive to split the argument into three easy propositions.

PROPOSITION 4.2. *If $G=(V, E)$ is a complete graph of order n then there is a recursive proof of $(V, 1)$ whose length is n.*

Proof. Enumerating the vertices of G as $v_1, v_2, \cdots, v_n$, define $S_i=\{v_1, v_2, \cdots, v_i\}$. Trivially, the sequence $(\varnothing, 0)$, $(S_1, 1), \cdots, (S_n, 1)$ constitutes a recursive proof. □

PROPOSITION 4.3. *Let $G_1=(V_1, E_1)$ and $G_2=(V_2, E_2)$ be graphs such that $V_1 \cap V_2=\varnothing$; let $G_1 \cup G_2$ denote the graph $(V_1 \cup V_2, E_1 \cup E_2)$. If there are recursive proofs of $(V_j, \alpha(G_j))$ of length m_j for each $j=1,2$ then there is a recursive proof of $(V_1 \cup V_2, \alpha(G_1)+\alpha(G_2))$ whose length does not exceed m_1+m_2.*

Proof. If (S_i, t_i) with $i=0,1, \cdots, m$ is a recursive proof of $(V_2, \alpha(G_2))$ then a recursive proof of $(V_1, \alpha(G_1))$, followed by the sequence

$$(V_1 \cup S_i, \alpha(G_1)+t_i), \qquad i=1,2, \cdots, m_2,$$

constitutes a recursive proof of $(V_1 \cup V_2, \alpha(G_1)+\alpha(G_2))$. □

PROPOSITION 4.4. *Let F be a subgraph of G and let* (4.1) *be a recursive proof* over F. *Then there is recursive proof of* (S_m, t_m) over G *whose length does not exceed* $2m$.

Proof. We shall create the desired proof over G from (4.1) by inserting a new statement immediately before each (S_k, t_k) that has been obtained from the previous statements by the dichotomy rule. For every such (S_k, t_k), there are subscripts i, j and a vertex $v \in S_k$ such that $i<k$, $j<k$, $t_k=\max(t_i, 1+t_j)$ and $S_i=S_k-v$, $S_j=S_k * v$ *in* F. The statement to be inserted immediately before (S_k, t_k) is (S_k', t_k-1) such that $S_k'=S_k * v$ *in* G. Clearly, (S_k', t_k-1) follows from (S_j, t_j) by the monotone rule whereas (S_k, t_k) follows from (S_i, t_i) and (S_k', t_k-1) by the dichotomy rule. □

PROPOSITION 4.5. *For every graph $G=(V, E)$ of order n there is a recursive proof of $(V, \theta(G))$ whose length does not exceed $2n$.*

Proof. Consider the subgraph F of G consisting of $\theta(G)$ cliques whose union equals V. By Proposition 4.2. and by repeated applications of Proposition 4.3, there is a recursive proof of $(V, \theta(G))$ over F whose length equals n. The rest follows from Proposition 4.4. □

We shall close this section with another easy observation which will be handy later. The proof can be left to the reader.

PROPOSITION 4.6. *If* (4.1) *is a recursive proof over $G=(V, E)$ and if $W \subseteq V$ then*

$$(S_i \cap W, t_i), \qquad i=0,1, \cdots, m,$$

is a recursive proof over the subgraph of G induced by W.

5. A lemma on binary trees. Let a, b, r, s be nonnegative integers. A binary tree whose nodes are colored red and blue will be called *(a, b, r, s)-constrained* if, along each path from the root to a leaf,

(i) exactly a nodes are followed by their left sons and exactly b nodes are followed by their right sons,

(ii) at most r nodes are red,

(iii) at least s red nodes are followed by their right sons.

If, for some choice of integers a, b, r and s, there is at least one (a, b, r, s)-constrained tree then we denote by $f(a, b, r, s)$ the largest possible number of leaves in such a tree; otherwise we set $f(a, b, r, s)=0$. Trivially, we have

$$f(a, b, r, s) \leqq \binom{a+b}{b}$$

and

$$f(a, b, r, s)=0 \quad \text{whenever } s>b \text{ or } s>r.$$

The purpose of this section is to derive the following upper bound on $f(a, b, r, s)$.

LEMMA 5.1. *If $s \geqq 2br/(a+b-1)$, $s \geqq r-a+1$ and $s \geqq 1$ then*

$$f(a, b, r, s) \leqq \binom{a+b}{b} \frac{ab}{a+b} e^{-br/(4(a+b))}.$$

First of all, we shall establish a simple recursive bound.

FACT 5.2. $f(a, b, r, s) \leqq \max \begin{cases} f(a-1, b, r, s)+f(a, b-1, r, s) \\ f(a-1, b, r-1, s)+f(a, b-1, r-1, s-1). \end{cases}$

Proof. Let T be an (a, b, r, s)-constrained tree. If its root is blue then the left subtree is either empty or $(a-1, b, r, s)$-constrained and the right subtree is either empty or $(a, b-1, r, s)$-constrained. If the root is red then the left subtree of T is either empty or $(a-1, b, r-1, s)$-constrained and the right subtree of T is either empty or $(a, b-1, r-1, s-1)$-constrained; hence the desired conclusion. □

Next, for every choice of nonnegative integers a, b, r, s such that

$$s \leqq b, \quad s \leqq r, \quad s \geqq r-a+1$$

we define

$$F(a, b, r, s)=\sum_{i=0}^{b-s} \binom{r+i}{s+i} \binom{a+b-r-1-i}{b-s-i}.$$

It is easy to verify that

$$F(a, b, r, 0)=\binom{a+b}{b},$$

$$F(a, b, r, b)=\binom{r}{b},$$

$$F(a, b, r, r)=\binom{a+b-r}{a},$$

$$F(a, b, r, r-a+1)=\binom{a+b}{a}-\binom{r}{a},$$

$$F(a-1, b, r, s)+F(a, b-1, r, s)=F(a, b, r, s),$$

$$F(a-1, b, r-1, s)+F(a, b-1, r-1, s-1)=F(a, b, r, s)$$

whenever the left-hand side terms are defined.

FACT 5.3. *We have $f(a,b,r,s) \leqq F(a,b,r,s)$ whenever the right-hand side is defined.*

This inequality can be proved by induction on $a+b$ in a straightforward way; we omit the tedious details. It is not unlikely that there is a direct combinatorial proof of Fact 5.3. Furthermore, it is not difficult to show that $f(a,b,r,s) = F(a,b,r,s)$ whenever the right-hand side is defined; however, that is irrelevant for our purpose.

Proof of Lemma 5.1. We may assume $s \leqq b$ and $s \leqq r$ for otherwise the left-hand side vanishes. Then, by Fact 5.3,

$$f(a,b,r,s) \leqq F(a,b,r,s).$$

Since $r \geqq s \geqq 2br/(a+b-1)$, we have $2b/(a+b-1) \leqq 1$ and

$$s+i \geqq 2b\frac{r+i}{a+b-1}$$

for every nonnegative i. Hence, with the notation of § 2,

$$\binom{r+i}{s+i}\binom{a+b-r-1-i}{b-s-i} \leqq H(r+i, a+b-1, b, (r+i)/(a+b-1)) \cdot \binom{a+b-1}{b}.$$

By Lemma 2.3, we have

$$f(a,b,r,s) \leqq \sum_{i=0}^{b-s} \binom{a+b}{b}\frac{a}{a+b}e^{-br/(4(a+b))}$$

which implies the desired result.

6. The main result.

A graph G of order n will be called (d, ε)*-sparse* if

(i) every vertex of G has degree less than d,

(ii) every subgraph of G induced by m vertices such that $m \geqq \varepsilon n$ has fewer than dm^2/n edges.

THEOREM 6.1. *Let n, t be positive integers and let d, ε be positive reals such that*

$$n \leqq 10td,$$
$$n \geqq 500t^{2/3}d,$$
$$n \geqq 100t^{3/4}d^{3/4},$$
$$n \geqq 2000t,$$
$$\varepsilon < n^2/(1810t^2d^2);$$

let $G = (V, E)$ be a (d, ε)-sparse graph of order n. Then every recursive proof of (V, t) has length at least

$$\frac{90d}{n}\exp\frac{n}{20d^2}. \tag{6.1}$$

Proof. We shall set

$$a = \lfloor n^3/(45000t^2d^2)\rfloor, \qquad b = \lfloor n^2/(900td^2)\rfloor$$

and show that every recursive proof of (V, t) has length at least

$$\frac{a+b}{ab}\exp\frac{b^2}{a+b}. \tag{6.2}$$

The reader may easily verify that $a \geqq b \geqq 200$ and so

$$a > \frac{200}{201}\cdot\frac{n^3}{45000t^2d^2}, \qquad b > \frac{200}{201}\cdot\frac{n^2}{900td^2}.$$

Then it follows that (6.2) is at least (6.1).

Let us outline our strategy. With each recursive proof of (V, t), we shall associate a binary tree T whose nodes will be labeled by statements from the proof. The assignment of labels to nodes will not be one-to-one (to take an extreme example, all the leaves of T will be labeled by $\emptyset$) and so the number of nodes of T may be much greater than the length of the proof. We shall find a node z with a certain convenient property and then we shall construct a new binary tree T^*. Even though T^* will not be a subtree of T in a strict sense, its nodes will come from T; in particular, z will be the root of T^*. Finally, we shall show that within the set N of leaves of T^*, no label is repeated too often. More precisely, for each subset S of V we shall define

$$N(S) = \{x \in N : x \text{ is labeled by } (S, t') \text{ for some } t'\}$$

and prove that

$$|N(S)| \leqq |N| \cdot \frac{ab}{a+b}\exp\left(-\frac{b^2}{a+b}\right). \tag{6.3}$$

Since N will be nonempty, (6.3) will imply the desired result: indeed, the number of those sets S for which $N(S) \neq \emptyset$ must be at least (6.2).

Before going into the details, the reader may welcome a preview of the idea behind the proof of (6.3), however vague such a preview may have to be. Let (W, t^*) be the statement that labels z. In the absence of the monotone rule, the tree T^* is constructed in such a way that every subproblem S labeling a leaf of T^* is obtained from W by simply deleting a vertices and by deleting b vertices with their neighbors. If we had our way, the subgraph H induced by $W-S$ would consist of a isolated vertices and b disjoint stars: in that case, we could reconstruct the two sets of vertices, proving that $|N(S)| = 1$. Actually, we shall be content even if things are not all that clear-cut, as long as we can *approximately* reconstruct the two sets. That will be the case as long as H is reasonably large. (If H is large then *most* of the b vertices must have large degrees. At the same time, the second defining property of a (d, ε)-sparse graph implies that the average degree in H is rather small. Hence the vertices of large degrees are quite conspicuous.) In order to *guarantee* that H will be large, we have to choose z appropriately. In general, the rules for constructing T^* are designed to neutralize the desultory effects of the monotone rule. Now that the poor reader is properly confused, we can proceed to the details.

Constructing T, we shall find it convenient to call certain statements in the proof *eligible*: a statement will be called eligible if it is $(\emptyset, 0)$ or if it follows from some two earlier statements by the dichotomy rule. Only the eligible statements, with a possible exception of (V, t), will be used to label the nodes of T. The construction of T is recursive; the root of T is labeled by (V, t). Suppose that we have constructed a node x labeled by a statement (S_k, t_k) and having no sons at this moment. If $(S_k, t_k)=(\varnothing, 0)$ then x will be a leaf of T. Otherwise there are eligible statements (S_i, t_i), (S_j, t_j) and a vertex $v \in S_k$ such that

$$i, j \leqq k, \quad S_i \supseteq S_k - v, \quad S_j \supseteq S_k * v, \quad t_k \geqq \max(t_i, 1+t_j).$$

In that case, we shall create both sons of x, label the left one by (S_i, t_i) and label the right one by (S_j, t_j). For further reference, we shall set $S(x)=S_k$, $t(x)=t_k$ and $v(x)=v$. It will be useful to note that

$$(6.4) \qquad t_i \leqq t_k \quad \text{and} \quad t_j \leqq t_k - 1.$$

Next, we shall find the special node z. A node y will be called a *descendant* of a node x if there is a path $x_0, x_1, \cdots, x_k$ such that $x=x_0$, $y=x_k$ and each x_{i+1} is a son of x_i. If, in addition, exactly b nodes x_i are followed by their *right*sons x_{i+1} then y will be called a *b-descendant* of x. Repeated applications of (6.4) show that

$$(6.5) \qquad \text{if } y \text{ is a } b\text{-descendant of } x \text{ then } t(y) \leqq t(x) - b.$$

We claim that

$$(6.6) \qquad \begin{array}{l} \text{there is a node } z \text{ such that } S(z) \geqq n/2 \text{ and such that} \\ |S(y)| < |S(z)| - bn/(2t) \text{ for every } b\text{-descendant } y \text{ of } z. \end{array}$$

A node with this property can be found by constructing a certain sequence $y_0, y_1, \cdots$ of nodes of T such that y_0 is the root of T. If the most recently constructed y_i has a b-descendant y such that $|S(y)| \geqq |S(y_i)| - bn/(2t)$ then set $y_{i+1}=y$; otherwise stop. By (6.5) and by the construction of the sequence, we have

$$|S(y_i)| \geqq n(1 - bi/(2t)), \qquad t(y_i) \leqq t - bi$$

for every i. Since $t(y_i) \geqq 0$, we must have $i \leqq t/b$ and so $|S(y_i)| \geqq n/2$ for every i. In particular, the very last y_i in the sequence has the properties required of z. We shall denote $S(z)$ by W.

With each descendant x of z, we shall associate two subsets $A(x)$, $B(x)$ of V: considering the path $x_0, x_1, \cdots, x_k$ from $x_0 = z$ to $x_k = x$ we shall define

$$A(x) = \{v(x_i): 0 \leqq i < k \text{ and } x_{i+1} \text{ is the left son of } x_i\},$$

$$B(x) = \{v(x_i): 0 \leqq i < k \text{ and } x_{i+1} \text{ is the right son of } x_i\}.$$

Clearly, we have

$$(6.7) \qquad |S(x) \cap W| \geqq |W| - |(A(x) - B(x)) \cap W| - \sum_{v \in B(x)} (1 + d(v))$$

for every descendant x of z. In particular,

$$(6.8) \qquad \text{if } |(A(x) - B(x)) \cap W| \leqq a \text{ and } |B(x)| \leqq b \text{ then } S(x) \neq \varnothing:$$

just observe that

$$a + b(1+d) \leqq 2a + bd < 3bd < n/2.$$

Before proceeding to construct T^*, we shall associate a node x^* with each descendant x of z such that

$$|(A(x)-B(x))\cap W|\leqq a \quad \text{and} \quad |B(x)|\leqq b.$$

Consider the path $x_0, x_1, \cdots, x_k$ such that $x_0=x$, x_k is a leaf of T and each x_{i+1} is the left son of x_i. Note that $B(x_i)=B(x)$ for every i. There must be at least one i such that $0\leqq i<k$ and

$$v(x_i)\in W, \qquad v(x_i)\notin A(x)\cup B(x) \tag{6.9}$$

for otherwise

$$(A(x_k)-B(x_k))\cap W=(A(x)-B(x))\cap W \quad \text{and} \quad B(x_k)=B(x)$$

but $S(x_k)=\varnothing$, contradicting (6.8). We shall denote the first x_i satisfying (6.9) by x^*; note that

$$(A(x^*)-B(x^*))\cap W=(A(x)-B(x))\cap W.$$

At last, we are ready to construct T^*. Each of its nodes x will come from T and satisfy

$$v(x)\in W, \qquad v(x)\notin A(x)\cup B(x),$$
$$|(A(x)-B(x))\cap W|\leqq a, \quad |B(x)|\leqq b, \quad B(x)\subseteq W.$$

The construction of T^* is recursive; the root of T^* is z. Suppose that we have already constructed some node x of T^*; let x_{L} denote the left son of x in T and let x_{R} denote the right son of x in T. If $|(A(x)-B(x))\cap W|=a$ then x will have no left son in T^*; otherwise we shall make x_{L}^* the left son of x in T^*. If $|B(x)|=b$ then x will have no right son in T^*; otherwise we shall make x_{R}^* the right son of x in T^*. It will be useful to make note of the following property of T^*:

(6.10)
along each path from the root to a leaf,
exactly a nodes are followed by their left sons,
exactly b nodes are followed by their right sons,
and these $a+b$ nodes x give rise to distinct vertices $v(x)$.

Finally, we shall prove the inequality (6.3). Without loss of generality, we may assume that $N(S)\neq\varnothing$ and so $S=S(y)$ for some $y\in N$. Denote by H the subgraph of G induced by $W-S(y)$ and denote by m the order of H. Since y is a b-descendant of z in T, (6.6) implies that

$$m>bn/(2t).$$

On the other hand, (6.7) implies that

$$m\leqq a+b(1+d)\leqq 2a+bd<3bd.$$

Enumerate the vertices of H as $u_1, u_2, \cdots, u_m$ in such a way that

$$d_H(u_1)\geqq d_H(u_2)\geqq\cdots\geqq d_H(u_m).$$

Since $bn/(2t)>n^3/(1809t^2d^2)$ and since G is (d,ε)-sparse, the graph H has fewer than dm^2/n edges. That is,

$$\sum_{i=1}^{m} d_H(u_i) < 2dm^2/n.$$

It is now easy to see that, for every positive integer r, we have

$$(6.11) \qquad d_H(u_i) < 2dm^2/(nr) \quad \text{whenever } i > r.$$

We shall use (6.11) with

$$r = \lfloor am/(4bd) \rfloor.$$

Let us note at once that $r \geqq 200$ and so

$$r > \frac{200}{201} \cdot \frac{am}{4bd}.$$

It will be also useful to note that

$$\frac{2rbd}{a} \leqq \frac{m}{2},$$

$$\frac{2dm^2b}{nr} \leqq \frac{201}{200} \cdot \frac{8b^2d^2}{an} \cdot m \leqq \left(\frac{201}{200}\right)^2 \cdot \frac{4m}{9},$$

$$(6.12) \qquad r \leqq a,$$

$$(6.13) \qquad \frac{2br}{a+b-1} \geqq \frac{br}{a} \geqq 1.$$

And while we are at it, let us also verify that

$$a < \frac{201}{200} \cdot \frac{bn}{50t} < \frac{201}{200} \cdot \frac{m}{25},$$

$$b \leqq \frac{1}{1000} \cdot \frac{bn}{2t} < \frac{m}{1000}.$$

So much for high-school algebra. Now, we shall set $R = \{u_1, u_2, \cdots, u_r\}$ and prove that

$$(6.14) \qquad |B(x) \cap R| \geqq \frac{2rb}{a+b-1}$$

for every $x \in N(S)$. To begin with, (6.7), (6.11) and $B(x) \subseteq W$ imply

$$m \leqq a + b + d|B(x) \cap R| + 2dm^2b/(nr).$$

If (6.14) failed then we would have

$$m \leqq a + b + 2rbd/a + 2dm^2b/(nr).$$

However, the right-hand side of this inequality is at most

$$m\left(\frac{201}{200} \cdot \frac{1}{25} + \frac{1}{1000} + \frac{1}{2} + \left(\frac{201}{200}\right)^2 \cdot \frac{4}{9}\right) < m.$$

Hence (6.14) must hold.

The rest is easy. Consider the subtree of T^* consisting of all the paths from the root z to leaves in $N(S)$; color each of its nodes x red if $v(x)\in R$ and blue otherwise. By (6.10) and (6.14), this tree is $(a, b, r, 2br/(a+b-1))$-constrained. Because of (6.12) and (6.13), Lemma 5.1 applies and shows that

$$|N(S)|\leqq\binom{a+b}{b}\cdot\frac{ab}{a+b}\exp\left(-\frac{b^2}{a+b}\right).$$

By virtue of (6.10), this is the desired inequality (6.3).

THEOREM 6.2. *Let m be a function of n such that $m(n)=o((n/\log n)^2)$ but $m(n)\geqq 10^{10}n$ for all sufficiently large n. Then, for almost all graphs $G=(V, E)$ with n vertices and m edges, every recursive proof of $(V, \alpha(G))$ has length at least*

$$\frac{360m}{n^2}\exp\frac{n^2}{m(350\log m/n)^3}.$$

Proof. By Lemma 2.2 and by Lemma 2.4, almost all graphs with n vertices and m edges have the following two properties:

(P1) $$\alpha(G)<\frac{n^2}{m}\log\frac{m}{n},$$

(P2) every subgraph induced by s vertices such that

$$s\geqq\frac{4n^2}{m}\log\frac{m}{n}$$

has at most $2ms^2/n^2$ edges.

We shall show that all sufficiently large graphs with these two properties satisfy the conclusion of Theorem 6.2. Let us define

$$k(n)=\left\lfloor\frac{n^2}{m}\left(50\log\frac{m}{n}\right)^3\right\rfloor,$$

$$d(n)=4km/n^2,$$

$$\varepsilon(n)=\frac{4n^2}{km}\log\frac{m}{n}.$$

Firstly, we shall show that every graph with n vertices and $m\geqq 10^{10}n$ edges satisfying (P2) contains an induced (d, ε)-sparse subgraph of order k. To begin with, $2k(n)\leqq n$. Next, an easy averaging argument shows that G contains an induced subgraph H_0 with $2k$ vertices and at most $4mk^2/n^2$ edges. Beginning with H_0, we shall construct a sequence $H_0, H_1, \cdots$ of induced subgraphs of H_0 as follows: if the last constructed H_i has a vertex v of degree at least d then set $H_{i+1}=H_i-v$, otherwise stop. Clearly, if an H_i gets constructed then H_0 had at least di edges and so $i\leqq k$. In particular, the very last H_j in the sequence has at least k vertices; in H_j, we shall choose an induced subgraph H of order k. Let W denote the set of vertices of H. By (P2), every subgraph of H with $s\geqq\varepsilon k=4n^2\log(m/n)/m$ vertices has at most $2ms^2/n^2<ds^2/k$ edges. Hence H is (d, ε)-sparse.

Next, by (P1), we have

$$\alpha(G) < \frac{n^2}{m} \log \frac{m}{n}.$$

On the other hand, we have

$$\alpha(G) \geqq \alpha(H) \geqq k/(d+1).$$

For all sufficiently large n, Theorem 6.1 asserts that every proof of $(W, \alpha(G))$ has length at least

$$\frac{90d}{k} \exp \frac{k}{20d^2} > \frac{360m}{n^2} \exp \frac{n^2}{m(350 \log m/n)^3}.$$

By Proposition 4.6, this is also a lower bound on the length of every recursive proof of $(V, \alpha(G))$. □

Let us state a special case of Theorem 6.2 in a compact form.

THEOREM 6.3. *For every sufficiently large integer d there is a constant $c > 1$ with the following property: for almost all graphs $G = (V, E)$ with n vertices and dn edges, every recursive proof of $(V, \alpha(G))$ has length at least c^n.*

In closing, two remarks may be in order. Firstly, it would be interesting to *construct* an infinite class $\mathscr{C}$ of graphs for which there is a constant $c > 1$ with the following property: for every graph $G = (V, E)$ in $\mathscr{C}$ and with n vertices, every recursive proof of $(V, \alpha(G))$ has length at least c^n. Secondly, it is somewhat frustrating that Theorem 6.2 does not apply to graphs with cn^2 edges. Perhaps the following is true.

CONJECTURE 6.4. *There is a positive constant c with the following property: for almost all graphs $G = (V, E)$ with n vertices, every recursive proof of $(V, \alpha(G))$ has length at least $n^{c \log n}$.*

7. Concluding remarks. There are many "natural" proof systems which extend our system of recursive proofs; we shall mention just a few. It would be interesting to strengthen our results by proving their analogues for the extended proof systems.

To begin with, one might adjoin the following inference rule:

from (S_1, t_1) and (S_2, t_2) we can deduce $(S_1 \cup S_2, t_1 + t_2)$.

Proposition 4.3 shows that, to some extent, this rule is implicit in the system of recursive proofs. Nevertheless, its addition just might make the system considerably more powerful. Along this line, further extensions lead to the system of *cutting plane proofs* which we are about to describe briefly. Let us consider a graph $G = (V, E)$ with vertices $v_1, v_2, \cdots, v_n$ none of which is isolated. A cutting plane proof of (V, t) is a sequence of inequalities

$$\sum_{j=1}^{n} a_{ij} x_j \leqq b_i \qquad i = 1, 2, \cdots, m,$$

such that

(i) all the numbers a_{ij} and b_i are nonnegative integers,

(ii) for every $k = 1, 2, \cdots, m$, either the kth inequality reads $x_r + x_s \leqq 1$ for some edge v_rv_s or else there are nonnegative multipliers $y_1, y_2, \cdots, y_{k-1}$ such that

$$\sum_{i=1}^{k-1} y_ia_{ij} \geqq a_{kj}, \qquad \left\lfloor \sum_{i=1}^{k-1} y_ib_i \right\rfloor \leqq b_k,$$

(iii) the last inequality reads $\sum_{j=1}^{n} x_j \leqq t$.

It is not difficult to see that $\alpha(G) \leqq t$ whenever there is a cutting plane proof of (V, t). The converse is easy as well: in fact, every recursive proof of (V, t) can be converted into a cutting plane proof of (V, t). (The details are left to the reader.)

An inference rule which strengthens the monotone rule and is *not* subsumed in the notion of a cutting plane proof goes as follows. Let us write $S_1 < S_2$ if there is a one-to-one mapping $f: S_1 \to S_2$ such that $f(u)$ and $f(v)$ are adjacent only if u and v are. Clearly,

from (S, t) we can derive (S', t') whenever $S' < S$ and $t' \geqq t$.

Again, it would be interesting to find out whether the addition of this inference rule makes the system of recursive proofs considerably stronger.

Colin McDiarmid [18] investigated a proof system, similar to our system of recursive proofs, for deriving lower bounds on the chromatic number of graphs. We shall describe his system very briefly. Let G be a graph whose vertices are labeled by nonempty and pairwise disjoint subsets of $\{1, 2, \cdots, m\}$; let u and v be two vertices of G. We shall denote by G' the graph obtained from G by adding the edge uv; we shall denote by G'' the graph obtained from G by identifying u with v (in which case the label of the new vertex is the union of the labels of u and v). As usual, $\omega(G)$ denotes the order of the largest clique in G. By a recursive proof of $[G_m, t_m]$, we shall mean a sequence

$$[G_i, t_i], \qquad i = 1, 2, \cdots, m,$$

such that, for each k, either $t_k \leqq \omega(G_k)$ or else there are subscripts $i, j < k$ such that $G_i = G'_k$, $G_j = G''_k$ and $t_k = \min(t_i, t_j)$. Clearly, if there is a recursive proof of $[G_m, t_m]$ then $\chi(G_m) \geqq t_m$. McDiarmid has proved that, for almost all graphs with n vertices, every recursive proof of $[G, \chi(G)]$ has length at least

$$\exp(.157\, n\, (\log n)^{1/2}).$$

His result implies that the average running time of the Corneil–Graham algorithm for finding the chromatic number of a graph[4] grows faster than every exponential. On the other hand, Lawler [16] has designed an algorithm for finding the chromatic number of a graph of order n within $O(2.45^n)$ steps. (Of course, these facts are of an asymptotic nature and imply nothing about the relative merits of the two algorithms applied to graphs with, say, 200 vertices.)

Acknowledgment. Finally, I wish to thank several friends for their help with my work on this paper. To Colin McDiarmid and Endre Szemerédi I am indebted for many stimulating conversations. Persi Diaconis told me about Hoeffding's paper [13]. David Avis, Don Knuth, Ivo Rosenberg, and Bob Tarjan read various versions of the manuscript and made many helpful suggestions to improve the presentation.

REFERENCES

[1] A. E. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] S. A. Cook, *The complexity of theorem-proving procedures*, Proceedings of Third Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1971, pp. 151–158.

[3] S. A. Cook and R. A. Reckhow, *On the lengths of proofs in the propositional calculus*, Proceedings of Sixth Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1974, pp. 135–148.

[4] D. G. Corneil and B. Graham, *An algorithm for determining the chromatic number of a graph*, this Journal, 2 (1973), pp. 311–318.

[5] P. Erdös, *Some remarks on the theory of graphs*, Bull. Amer. Math. Soc., 53 (1947), pp. 292–294.

[6] ———, *On circuits and subgraphs of chromatic graphs*, Mathematika, 9 (1962), pp. 170–175.

[7] P. Erdös and B. Bollobás, *Cliques in random graphs*, Math. Proc. Cambridge Philos. Soc., 80 (1976), pp. 419–427.

[8] P. Erdös and A. Rényi, *On random graphs, I*, Publ. Math. Debrecen, 6 (1959), pp. 290–297.

[9] ———, *On the evolution of random graphs*, Magyar Tud. Akad. Mat. Kut. Int. Közl., 5 (1960), pp. 17–61.

[10] ———, *On the strength of connectedness of a random graph*, Acta Math. Acad. Sci. Hungar., 12 (1961), pp. 261–267.

[11] Z. Galil, *On enumeration procedures for theorem proving and for integer programming*, IBM report RC 5719 (# 24648), 1975.

[12] G. R. Grimmett and C. J. H. McDiarmid, *On colouring random graphs*, Math. Proc. Cambridge Philos. Soc., 77 (1975), pp. 313–324.

[13] W. Hoeffding, *Probability inequalities for sums of bounded random variables*, Amer. Statist. Assoc., 58 (1963), pp. 13–29.

[14] R. M. Karp, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.

[15] D. E. Knuth, *The Art of Computer Programming, Vol.* 3: *Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[16] E. L. Lawler, *A note on the complexity of the chromatic number problem*, Information Processing Lett., 5 (1976), pp. 66–67.

[17] D. W. Matula, *On the complete subgraphs of a random graph*, Proc. 2nd Conf. Combinatorial Theory and Appl., Chapel Hill, NC, 356–369.

[18] C. J. H. McDiarmid, *Determining the chromatic number of a graph*, this Journal, submitted.

[19] R. E. Tarjan, *Finding a maximum clique*, Tech. Rep. 72–123, Computer Science Dept., Cornell University, Ithaca, NY, 1972.

[20] R. E. Tarjan and A. E. Trojanowski, *Finding a maximum independent set*, this Journal, 6 (1977), pp. 537–546.

SIAM J. COMPUT.
Vol. 6, No. 4, December 1977

MINIMIZING GARBAGE COLLECTION AS A FUNCTION OF REGION SIZE*

RICHARD G. LARSON†

Abstract. Under a virtual storage operating system, the time spent by a list processing language on garbage collection varies in a complex manner with the amount of memory allocated. This dependence is investigated and strategies for minimizing the time spent on garbage collection are given.

Key words. garbage collection, computation time, optimal storage allocation, list processing languages

1. Introduction. Higher-level list processing languages such as SNOBOL4 permit the convenient solution of relatively complex problems. However, programs written in such languages usually require large amounts of storage. If a virtual storage operating system is used, careful programming can produce programs with good locality, especially if the language uses a compacting garbage collection algorithm [2], [3]. However, the process of garbage collection necessarily has poor locality. One factor that the programmer often can control is the amount of virtual storage allocated to the program. A small region results in many fast storage regenerations; a large region results in a few slow storage regenerations.

It is possible to estimate the amount of time taken by garbage collection [3, p. 419]. If some simple assumptions are made about the average time needed to access a randomly chosen memory location as a function of region size (this is not constant with a virtual storage operating system), an estimate can be made of the total time required for garbage collection. This estimate is given in Proposition 1. The estimate is adequate to suggest strategies for choosing the amount of storage which minimizes garbage collection time. Such strategies are suggested by Propositions 2 and 3. These strategies depend on the amount of active data which must be preserved during the garbage collection, and on the amount of real memory available.

2. Garbage collection time. In this section we give an estimate of the number of garbage collections and the total time of garbage collection which is needed for a given computation.

One way to measure the course of a computation is by the amount of time spent computing since the beginning of the computation. Another way is by the amount of data produced. For example, in LISP one could count the number of CONSs. For the next proposition, the second method is used. At the point in a computation where X units of data have been produced, denote the amount of active data which must be preserved by garbage collection by $A(X)$. A garbage collection in a region of size R with amount of active data A takes $aA+bR$ time units [3].

The term aA in this expression is the time necessary for the garbage collection algorithm to mark the active nodes; in the case of a compacting garbage

* Received by the editors January 23, 1976, and in revised form August 15, 1976.

† Department of Mathematics, University of Illinois at Chicago Circle, Chicago, Illinois 60680. This work was supported in part by the National Science Foundation under Grant MPS71-02790 A05.

collection such as described in [2], it also includes the time necessary to move the active data and to adjust the pointers. The term bR is the time necessary to make a sequential pass through memory, collecting the unmarked nodes. Because marking and moving the active data are rather complicated procedures, in general the quantity a will be substantially larger than b.

PROPOSITION 1. *Suppose a computation is done which produces amount of data D, using a region for storing data of size R. Then the number of garbage collections needed for this computation is approximately*

$$\int_0^D \frac{dX}{R-A(X)}-1.$$

The total time spent doing garbage collection during this computation is approximately

$$\int_0^D \frac{(aA(X)+bR)\,dX}{R-A(X)}-bR.$$

Proof. Let $X_0=0$, and let X_i be the amount of data which have been produced by the time of the ith garbage collection. Since $R-A(X_i)$ is the amount of storage recovered by the ith garbage collection, we have the following relation:

$$X_{i+1}-X_i=R-A(X_i).$$

The computation continues until $X \geqq D$. Let n be the smallest integer such that $X_{n+1}>D$. Then n is the number of garbage collections necessary for the computation. Let

$$\Delta X_i=X_{i+1}-X_i=R-A(X_i).$$

Note that

$$n=\left(\sum_{i=0}^n 1\right)-1=\sum_{i=0}^n \frac{\Delta X_i}{R-A(X_i)}-1.$$

The sum on the right-hand side of this equation is a Riemann sum for the integral

$$\int_0^D \frac{dX}{R-A(X)}.$$

Therefore

$$n \approx \int_0^D \frac{dX}{R-A(X)}-1.$$

The time spent on garbage collection is

$$\sum_{i=1}^n (aA(X_i)+bR)=\sum_{i=0}^n (aA(X_i)+bR)-bR,$$

since $A(X_0)=A(0)=0$. But

$$\sum_{i=0}^{n} aA(X_i)+bR=\sum_{i=0}^{n}(aA(X_i)+bR)\frac{\Delta X_i}{R-A(X_i)}$$

$$\approx\int_0^D\frac{(aA(X)+bR)\,dX}{R-A(X)}.$$

This completes the proof of the proposition.

3. Strategies. In this section we use Proposition 1 to study the effect on garbage collection time of varying region size. We first consider the case where a and b are independent of R.

PROPOSITION 2. *Suppose that a and b do not depend on R. Then garbage collection time for a given computation is a decreasing function of R.*

Proof. Since

$$T=\int_0^D\frac{(aA+bR)\,dX}{R-A}-bR,$$

it follows that

$$\frac{dT}{dR}=-\int_0^D\frac{(a+b)A\,dX}{(R-A)^2}-b$$

which is negative. This completes the proof of the Proposition.

The situation where a and b vary with R is much more complicated. Suppose that there is a fixed amount H of fast memory, and that a and b depend on the proportional amounts of fast and slow memory available. That is, assume

$$a=\begin{cases}a_0, & R\leqq H,\\ a_0H/R+a_1(1-H/R), & R>H,\end{cases}$$

$$b=\begin{cases}b_0, & R\leqq H,\\ b_0H/R+b_1(1-H/R), & R>H.\end{cases}$$

We assume that $a_1>a_0$ and that $b_1>b_0$. At the beginning of § 2 we gave a justification for the assumption that $a\gg b$. This allows us to assume that $a_1\gg b_1$. We also assume that D is large, so that the term $-bR$ is negligible compared to the integral in the time estimate in Proposition 1.

Denote $\bar{a}=a_1-a_0$ and $\bar{b}=b_1-b_0$. Note that our assumptions imply that $(\bar{a}+\bar{b})/(a_1+b_1)$ is a positive number somewhat less than 1, and that $\bar{b}/(a_1+b_1)$ is a positive number substantially less than 1. Assume that $\bar{b}<\bar{a}$. In the following proposition we consider conditions on $A=A(X)$. These are to be interpreted as requiring that A be in the indicated range for all values of X.

PROPOSITION 3. *Under the above assumptions,*

(a) *if*

$$A/H>(\bar{a}+\bar{b})/(a_1+b_1),$$

then the garbage collection time is a decreasing function of R;

(b) *if*

$$(a+b)/(a_1+b_1) > A/H \gg \bar{b}/(a_1+b_1),$$

then the garbage collection time is a decreasing function of R *for* $R<H$ *and for* $R>2H-A/2$;

(c) *if*

$$A/H \gg \bar{b}/(a_1+b_1)$$

is false, then the garbage collection time is a decreasing function of R *for* $R<H$ *and is an increasing function of* R *for not excessively large* $R>H$.

The strategic implications of this Proposition are as follows: if A is not substantially less than H, then garbage collection time is minimized by taking R as large as possible; if A is very small compared with H, then garbage collection time is minimized by taking $R=H$. Of course, in practical situations, other considerations such as possible charges for space as well as time and scheduling priorities will influence the choice of R.

Proof. It follows from Proposition 2 that garbage collection time is a decreasing function of R for $R<H$. Note that

$$\frac{da}{dR}=\bar{a}H/R^2, \qquad R>H,$$

and

$$\frac{db}{dR}=\bar{b}H/R^2, \qquad R>H.$$

We consider

$$\frac{d}{dR}\int_0^D \frac{(aA+bR)\,dX}{R-A}=\int_0^D \frac{d}{dR}\left(\frac{aA+bR}{R-A}\right)dX.$$

The integrand on the right-hand side equals

$$\frac{(\bar{b}H-(a_1+b_1)A)R^2+2\bar{a}AHR-\bar{a}A^2H}{R^2(R-A)^2}.$$

This has the same sign as

$$\begin{aligned} Q(R)&=(\bar{b}H-(a_1+b_1)A)R^2+2\bar{a}AHR-\bar{a}A^2H\\ &=C_2R^2+C_1R+C_0. \end{aligned}$$

To find the sign of $Q(R)$, we first find the roots. The discriminant is

$$\begin{aligned} \Delta&=C_1^2-4C_0C_2\\ &=(2\bar{a}AH)^2-4(\bar{b}H-(a_1+b_1)A)(-\bar{a}A^2H)\\ &=4\bar{a}A^2H(\bar{a}H+\bar{b}H-(a_1+b_1)A)\\ &=4\bar{a}^2A^2H^2\left(1+\frac{\bar{b}H-(a_1+b_1)A}{\bar{a}H}\right). \end{aligned}$$

Note that the discriminant is negative if and only if

$$A/H>(\bar{a}+\bar{b})/(a_1+b_1).$$

We see that $Q(0)<0$. If the discriminant is negative, $Q(R)$ cannot change sign. Therefore, if

$$A/H>(\bar{a}+\bar{b})/(a_1+b_1),$$

the derivative is always negative. This proves part (a) of the proposition.

To prove parts (b) and (c), we consider the location of the roots of $Q(R)$.

Suppose now that

$$A/H<(\bar{a}+\bar{b})/(a_1+b_1).$$

This implies that

$$-1<\frac{\bar{b}H-(a_1+b_1)A}{\bar{a}H}.$$

The assumption that $\bar{b}<\bar{a}$ implies that

$$\frac{\bar{b}H-(a_1+b_1)A}{\bar{a}H}<1.$$

Therefore

$$\left|\frac{\bar{b}H-(a_1+b_1)A}{\bar{a}H}\right|<1.$$

Use the Taylor series expansion for $\sqrt{1+x}$ to get that

$$\sqrt{\Delta}\approx 2\bar{a}AH\left(1+\frac{\bar{b}H-(a_1+b_1)A}{2\bar{a}H}\right).$$

Therefore the roots of $Q(R)$ are

$$z_1=\frac{-C_1+\sqrt{\Delta}}{2C_2}\approx\frac{A}{2},$$

$$z_2=\frac{-C_1-\sqrt{\Delta}}{2C_2}\approx\frac{2\bar{a}AH}{(a_1+b_1)A-\bar{b}H}-\frac{A}{2}.$$

If $A/H \gg \bar{b}/(a_1+b_1)$, then

$$z_2\approx\frac{2\bar{a}H}{(a_1+b_1)}-\frac{A}{2}.$$

Since $C_2=\bar{b}H-(a_1+b_1)A$ is negative in this case, $Q(R)$ is negative for R a large positive or large negative number. The fact that C_2 is negative also implies that $z_1<z_2$. Since

$$\frac{\bar{a}}{a_1+b_1}<1,$$

it follows that $z_2 < 2H - A/2$. Since $Q(R)$ is negative for $R > z_2$, and since dT/dR has the same sign as $Q(R)$ for $R > H$, it follows that T is a decreasing function of R for $R > 2H - A/2$. Part (b) of the proposition follows.

The proof of part (c) breaks down into two subcases:

(c1) Suppose A/H is slightly larger than $\bar{b}/(a_1 + b_1)$. The argument in the proof of part (b) shows that the derivative is positive for

$$H < R < \frac{2\bar{a}AH}{(a_1+b_1)A - \bar{b}H} - \frac{A}{2}.$$

Since A/H is only slightly larger than $\bar{b}/(a_1 + b_1)$, the denominator of the first term on the right-hand side of the inequality is small, so the right-hand end of the inequality is large.

(c2) Suppose $A/H < \bar{b}/(a_1 + b_1)$. Then C_2 is positive. The argument in the proof of part (b) shows that the two roots are approximately

$$-\frac{2\bar{a}AH}{\bar{b}H - (a_1+b_1)A} - \frac{A}{2}$$

which is negative, and $A/2$ which is $< H$. Therefore the derivative is positive for all $R > H$. This completes the proof of the proposition.

Acknowledgment. Computing services used in this research were provided by the Computer Center of the University of Illinois at Chicago Circle. This assistance is gratefully acknowledged.

REFERENCES

[1] P. J. Denning, *Virtual memory*, Comput. Surveys, 2 (1970), pp. 153–189.

[2] R. E. Griswold, *The Macro Implementation of SNOBOL4*, W. H. Freeman, San Francisco, 1972.

[3] D. Knuth, *The Art of Computer Programming*, Vol. I, Addison-Wesley, Reading, MA, 1968.

[4] J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin, *LISP* 1.5 *Programmer's Manual*, MIT Press, Cambridge, MA, 1965.

SIAM J. COMPUT.
Vol. 6, No. 4, December 1977

ON THE WORST-CASE BEHAVIOR OF STRING-SEARCHING ALGORITHMS*

RONALD L. RIVEST†

Abstract. Any algorithm for finding a pattern of length k in a string of length n must examine at least $n-k+1$ of the characters of the string in the worst case. By considering the pattern $00\cdots 0$, we prove that this is the best possible result. Therefore there do not exist pattern matching algorithms whose worst-case behavior is "sublinear" in n (that is, linear with constant less than one), in contrast with the situation for average behavior (the Boyer–Moore algorithm is known to be sublinear on the average).

Key words. string-searching, pattern matching, text editing, computational complexity, worst-case behavior

1. Introduction. Let $s=s_1s_2\cdots s_n$ denote a string of length n over some finite alphabet Σ, and similarly let $p=p_1p_2\cdots p_k$ denote a pattern of length k over the same alphabet. The "string-searching problem" is to determine if the pattern occurs in the string—that is, if

$$(\exists j)(1\leqq j\leqq n-k+1)\wedge(p_1p_2\cdots p_k=s_js_{j+1}\cdots s_{j+k-1}).$$

We denote this occurrence as $p\leqq s$.

Several efficient algorithms exist for determining whether $p\leqq s$, given a pattern p of length k and a string s of length n. For example, the algorithm of Knuth, Morris and Pratt [3], [4] first constructs (in time $O(k)$) a finite state automaton to recognize the regular set $\Sigma^*p\Sigma^*$ (see [1] also). Then $p\leqq s$ iff the automaton accepts s, which can be determined in time $O(n)$. The entire algorithm runs in time $O(n+k)$. As an example (which we shall use later), for $p=0101$ the automaton of Fig. 1 would be constructed. Here we assume that $\Sigma=\{0, 1\}$. State 1 is the initial state and state 5 is the only accepting state.

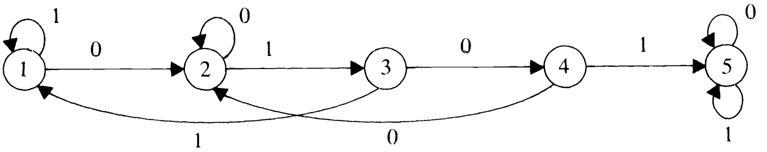


FIG. 1

Recently, Boyer and Moore published an algorithm [2] which is significantly faster than the Knuth–Morris–Pratt algorithm on the average. The latter algorithm examines every character in s exactly once, whereas the Boyer–Moore algorithm looks at only some fraction $c<1$ of the characters on the average; a

* Received by the editors June 15, 1976, and in revised form October 5, 1976.

† Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. This report was prepared with the support of the National Science Foundation Grant GJ-43634X, Contract DCR74-12997-A01, and Grant MCS76-14294.

typical value for c might be .24 when p is a five-letter English word. The worst-case behavior of the algorithm is nonlinear in n and k, although a slight modification of their algorithm due to B. Kuipers results in a linear worst-case time algorithm as well. (Knuth [5] has shown that the average number of times a character in s is examined by the modified algorithm is bounded above by 6; the proof, however, is *very* complicated.) The Boyer–Moore algorithm requires that the string s be stored in some sort of random-access memory in order to achieve any savings. Their procedure examines s_k, then s_{k-1}, and so on, until an s_j such that $s_j \neq p_j$ is found. Then some of the initial characters of s may be deleted and the process repeated with the shorter string s. If the examined (matching) subsection $s_{j+1} \cdots s_k$ of s occurs nowhere else in p, the first k characters of s may be skipped, even though only $k-j+1$ of them have been examined. Otherwise some smaller number may be discarded, reflecting the next possible alignment of $s_{j+1} \cdots s_k$ with some subsection of p. Another heuristic is also used: the latest occurrence of s_j in p (hopefully preceding p_j) is used to determine how many characters from s can be deleted before s_j aligns with some character in p. In the best case we find that $s_k \neq p_k$ and that s_k occurs nowhere in p; then k characters of s can be skipped at the cost of examining just one.

The focus of this paper is on the worst-case behavior of such pattern-matching algorithms. We answer (in the negative) the conjecture that a pattern-matching algorithm can exist whose worst-case behavior is "sublinear" in the same sense that the Boyer–Moore algorithm is sublinear in its average behavior. More precisely, we show that for every pattern p and for every correct algorithm A which determines if $p \leqq s$ for arbitrary strings s, there exists a string s which causes A to examine at least $|s|-|p|+1$ characters of s. This result is given in § 2 of this paper. In § 3 we show that this lower bound is the best possible by considering an algorithm for the pattern $p = 00 \cdots 0$.

2. The worst-case lower bound. The approach models the method Rivest and Vuillemin used to prove the Aanderaa–Rosenberg conjecture [5]. Fix the pattern p and let A_p be any algorithm for determining whether $p \leqq s$ for any string s. Let $w(A_p, n)$ denote the maximum number of characters in s examined by algorithm A_p for any string s in Σ^n; $w(A_p, n)$ is the worst-case cost function for algorithm A.

We assume that $w(A_p, n) \leqq w(A_p, n+1)$ for all A_p and all n. Otherwise if $w(A_p, n) > w(A_p, n+1)$ for some n an improved algorithm A'_p can be derived from A_p by letting A'_p behave on inputs s just as A_p does whenever $|s| \neq n$ and letting A'_p behave on the strings s of length n just as A_p would behave on the string sz where $z \neq p_k$ (simulating the query of the $(n+1)$st character z). Since $(p \leqq s) \Leftrightarrow (p \leqq sz)$, we have that $w(A'_p, n) \leqq w(A_p, n+1)$, and $w(A'_p, m) = w(A_p, m)$ for $m \neq n$. Thus $w(A'_p, n) \leqq w(A'_p, n+1)$; repeating this procedure as necessary yields an improved procedure A'_p such that $w(A'_p, n) \leqq w(A_p, n)$ for all n and $w(A'_p, n) \leqq w(A'_p, n+1)$ for all n.

THEOREM 1. $(\forall p)(\forall A_p)(\forall n)(w(A, n) \geqq n-k+1)$, *where* $k = |p|$.

Proof. We shall in fact prove that $w(A_p, n) = n$ for infinitely many n, such that these values of n occur not more than k apart. Using our assumption that $w(A_p, n) \leqq w(A_p, n+1)$ then yields the theorem.

Let $f(p, n)$ denote $|\{s \mid s \in \Sigma^n \wedge p \leqq s\}|$, the number of strings of length n which contain p as a substring. The following result is immediate from [5].

LEMMA 1. *If* $f(p, n) \not\equiv 0 (\mathrm{mod}\, |\Sigma|)$, *then* $w(A_p, n) = n$.

The proof of Lemma 1 will not be given here; we only remark that it follows from a calculation of $f(p, n)$ using a decision-tree representation of A_p. If $w(A_p, n) < n$ then $f(p, n) \equiv 0 \pmod{|\Sigma|}$ follows.

In order to calculate $f(p, n)$ we make use of the finite state automaton (fsa) constructed by the Knuth–Morris–Pratt algorithm for recognizing $\Sigma^* p \Sigma^*$. Let the states of this fsa be numbered so that state 1 is the initial state, state i (for $1 \leqq i \leqq k$) is arrived at whenever a string ending in $p_1 p_2 \cdots p_{i-1}$ has been read (and this is the largest such i), and state $k+1$ is the accepting state. There is a transition labeled p_i from state $i-1$ to state i (for $1 \leqq i \leqq k$); all other transitions leaving state $i-1$ arrive at some state numbered strictly less than i.

Let $g_p(n, i)$ denote $|\{s \mid s \in \Sigma^n$ and the fsa on s ends in state $i\}|$. Then $g_p(n, k+1) = f(p, n)$. The fsa will be used to derive a set of linear recurrences for the vector $\bar{g}_n = (g_p(n, 1), g_p(n, 2), \cdots, g_p(n, k+1))$. In fact $\bar{g}_{n+1} = T \cdot \bar{g}_n$, where T is a $k+1$ by $k+1$ matrix whose (i, j) entry is the number of symbols in Σ which cause a transition from state j to state i. For example, for $p = 0101$ the corresponding matrix $T = \{t_{ij}\}$ is

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 \end{bmatrix}.$$

In general, the sum of each column is $|\Sigma|$, $t_{i,i-1} = 1$ for $2 \leqq i \leqq k+1$, and $t_{ij} = 0$ if $j < i-1$. Also, $t_{k+1,k+1} = |\Sigma|$. To initialize the recurrence we have $\bar{g}_0 = (1, 0, 0, \cdots, 0)$.

Since we are interested in $f(p, n) = g_p(n, k+1)$ only with respect to its residue modulo $|\Sigma|$, we consider the reduced recurrence $\bar{g}_{n+1} = T' \cdot \bar{g}_n \pmod{|\Sigma|}$, where the entries of T' are those of T reduced modulo $|\Sigma|$. In fact T' is just T with the $(k+1, k+1)$ entry replaced by 0. We now observe that $g_p(n, k+1) \equiv g_p(n-1, k)$, so we will concentrate on the parity of $g_p(n, k)$ from now on. The k by k upper left submatrix T'' of T' maps $(g_p(n, 1) \pmod{|\Sigma|}, \cdots, g_p(n, k) \pmod{|\Sigma|})$ onto $\bar{g}'_{n+1}$.

Now T'' induces a mapping from $\Gamma = \{0, 1, \cdots, |\Sigma| - 1\}^k$ to itself. Furthermore, T'' is easily seen to be invertible; sequentially adding row i to row $i+1$ for $i = 1, 2, \cdots, k$ will reduce T'' to an upper triangular form with $|\Sigma| - 1$ along the main diagonal (we assume that $|\Sigma| > 1$).

Since T'' is invertible, the directed graph G, whose vertices are elements of Γ and whose edges (x, y) are present whenever $T''x = y$, consists of a set of disjoint cycles. We need to show that the cycle containing $\bar{g}'_0 = (1, 0, 0, \cdots, 0)$ has a vertex whose kth coordinate is nonzero at least once every k steps.

We first observe that the all-zero vector 0^k is not an element of the cycle, since it belongs to a one-element cycle (it is fixed by the linear mapping T''), and $\bar{g}'_0$ is not the zero vector.

Next we observe that for any vector $x \in \Gamma$ such that $x_i \neq 0$ and $x_j = 0$ for $j > i$, the vector $y = (T'')^{k-i}x$ has $y_k \neq 0$. In general, if $x \neq 0^k$, $x_k = 0$ and i is the largest integer such that $x_i \neq 0$, then $(T''x)_{i+1} = x_i$ since the lower diagonal portion of T'' is zero except for the subdiagonal, which consists entirely of ones.

This completes the proof of

LEMMA 2. $(\forall n > k)(\exists j)(0 \leqq j < k)(w(A_p, n-j) = n-j)$.

Theorem 1 now follows directly from Lemma 1 and our assumption. □

3. An upper bound on the worst-case. The lower bound of $|s| - |p| + 1$ proved in the last section may seem weak at first; one's first guess might be that $w(A_p, n) = n$ as long as $n \geqq |p|$. This, however, turns out to be false, as we demonstrate in this section by a careful analysis of an algorithm for the pattern $p = 0^k$.

THEOREM 2. $(p = 0^k) \Rightarrow (\exists A_p)(w(A_p, n) = n - \mu(n))$, *where*

$$\mu(n) = \begin{cases} 0 & \textit{if } n^p \equiv 0 \pmod{k+1} \textit{ or } n \equiv k \pmod{k+1}, \\ n \pmod{k+1} & \textit{otherwise}. \end{cases}$$

Proof. The algorithm A_p works in a fashion similar to the Boyer–Moore algorithm. It is given below.

```
ALGORITHM $A_p$ for $p = 0^k$.
Input: a string $s_1 s_2 \cdots s_n$.
Local variables: $r, i, j$
Procedure:
    $r := 0; i := 0; j := 0;$
    repeat if $r + k > n$ then
            begin print ("$p \not\leqq s$"); exit end;
            if $s_{r+k-j} = 0$ then $j := j + 1$
            else begin $r := r + k - j$;
                       $i := j$;
                       $j := 0$
            end;
    until $i + j = k$;
    print ("$p \leqq s$ at position", $r + 1$).
```

Inductively the algorithm knows at the top of the repeat loop that positions $s_{r+1}, s_{r+2}, \cdots, s_{r+i}$ and positions $s_{r+k-j+1}, \cdots, s_{r+k}$ are all zero; it next tests position s_{r+k-j} and adjusts r, i, and j accordingly. Let $c(m, i, j)$ denote the maximum number of characters in s that A_p needs to examine, starting from some instant when $m = n - r$ and i and j define the state of A_p's knowledge about s as above. Thus $w(A_p, n) = c(n, 0, 0)$ by definition. Furthermore, we have by construction that

$$(*) \qquad c(n, i, j) = \begin{cases} 0 & \text{if } i + j = k \text{ or } n < k, \\ \max(c(n, i, j+1), c(n-k+j, j, 0)) + 1 & \text{otherwise.} \end{cases}$$

Define for integers m and i, $0 \leqq i \leqq k-1$, $0 \leqq m \leqq k$,

$$\beta(m, i) = \begin{cases} 0 & \text{if } m = k, \\ m + 1 & \text{if } i > m \text{ and } m < k, \\ m - i & \text{if } i \leqq m \text{ and } m < k. \end{cases}$$

LEMMA.

$$c(n,i,j)=\begin{cases}0 & \text{if } i+j=k \text{ or } n<k,\\ n-i-j-\beta(m,i) & \text{otherwise, where } m=n \pmod{k+1}.\end{cases}$$

Proof. By induction, as in the definition (*) of $c(n,i,j)$. The lemma is clearly correct if $i+j=k$ or $n<k$. Henceforth, assume $i+j<k\leqq n$. There are two cases to consider. Let m denote $n \pmod{k+1}$.

Case 1. $c(n,i,j)=c(n,i,j+1)+1$. Here the lemma follows directly as long as $i+j+1\neq k$; otherwise $c(n,i,j+1)\leqq c(n-k+j,j,0)$, so here we can appeal to Case 2.

Case 2. $c(n,i,j)=c(n-k+j,j,0)+1$.

Case 2a. $n-k+j<k$. Here we know that $c(n,i,j+1)\geqq c(n-k+j,j,0)$ so the lemma holds by Case 1. (If both $n-k+j<k$ and $i+j+1=k$ then the lemma follows by the definition of β).

Case 2b. $n-k+j\geqq k$.

Case 2b(1). $i+j+1=k$. Here we need to show that

$$n-i-j-\beta(m,i)=n-k+1-\beta(n-k+j \pmod{k+1},j),$$

or

$$(**)\qquad \beta(m,i)=\beta(n-i-1 \pmod{k+1},k-1-i).$$

Case 2b(1)i. $m=k$. Here both sides of (**) are 0, since $n-i-1\equiv k-i-1 \pmod{k+1}$.

Case 2b(1)ii. $i>m$ and $m<k$. Both sides of (**) are $m+1$, since $n-i-1 \pmod{k+1}>k-i-1$.

Case 2b(1)iii. $i\leqq m$ and $m<k$. Both sides of (**) are $m-i$, since $0\leqq m-i-1<k-i-1$.

Case 2b(2). $i+j+1<k$. Here it suffices to show that

$$n-i-j-1-\beta(m,i)\geqq n-k+j-h-\beta(n-k+j \pmod{k+1},j),$$

that is, that $c(n,i,j+1)\geqq c(n-k+j,j,0)$, so that we may appeal to Case 1. Since $m+1\equiv n-k \pmod{k+1}$, this is equivalent to

$$(***)\qquad k-i-j\geqq 1+\beta(m,i)-\beta(m+j+1 \pmod{k+1},j).$$

Note that the left-hand side of (***) is strictly greater than one, since we are in Case 2b(2).

Case 2b(2)i. $m=k$. Here the right side of (***) is at most one.

Case 2b(2)ii. $i>m$. The right side of (***) equals 1 since $m+j+1<i+j+1<k$.

Case 2b(2)iii. $i\leq m$. If $m+j+1<k$, then the right hand side of (***) is $-i$. If $m+j+1=k$ then it is $1+m-i=k-i-j$. If $m+j+1>k$ then if $m<k$ it is $1+(m-i)-(m+j+1-k-1+1)=k-i-j$; otherwise it is one.

This completes the proof of the lemma. Theorem 2 follows since $\beta(n \pmod{k+1},0)=\mu(n)$. □

We conclude from Theorem 2 that when searching for the pattern 0^k in a string $s\in\Sigma^n$, we only need to examine at most $n-k+1$ characters of s if $n\equiv k-1 \pmod{k+1}$. The uniform lower bound of theorem 1 can therefore not be improved. Note that the use of the pattern 0^k in Theorem 2 means that Theorem 1 is best possible even for a binary alphabet.

Conclusions. We have shown that pattern matching in strings is inherently linear (with constant 1) in the length of the string. An open problem is to prove the equivalent of Theorem 2 for all patterns:

$$(\forall p)(\exists A_p)(\overset{\infty}{\exists} n)(w(A_p, n) = n - |p| + 1).$$

Acknowledgment. I would like to thank Donna Brown for several helpful discussions, during one of which a weak version of Theorem 2 was observed. I would also like to thank Leo Guibas and Robert Floyd for communications regarding improved proofs of Theorems 1 and 2, respectively.

REFERENCES

[1] A. V. Aho, and M. J. Corasick, *Fast pattern matching; An aid to bibliographic search*, Comm. ACM, 18 (1975), pp. 333–340.

[2] R. S. Boyer and J. Strother Moore, *A fast string searching algorithm*, Tech. Rep. 3, Stanford Res. Inst., Mar. 1976. To appear, Comm. ACM.

[3] D. E. Knuth, J. H. Morris and V. R. Pratt, *Fast pattern matching in strings*, Computer Science Department Tech. Rep. CS-74-440, Stanford Univ., Stanford, CA, 1974.

[4] ———, *Fast pattern matching in strings*, this Journal, 6 (1977), pp. 323–350.

[5] R. L. Rivest and J. Vuillemin, *A generalization and proof of the Aanderaa-Rosenberg conjecture*, Proc. 7th SIGACT Symp. on Theory of Computing, pp. 6–11.

SIAM J. COMPUT.
Vol. 6, No. 4, December 1977

COMPUTATIONAL COMPLEXITY OF PROBABILISTIC TURING MACHINES*

JOHN GILL†

Abstract. A probabilistic Turing machine is a Turing machine with the ability to make decisions based on the outcomes of unbiased coin tosses. The partial function computed by a probabilistic machine is defined by assigning to each input the output which occurs with probability greater than $\frac{1}{2}$. With this definition, only partial recursive functions are probabilistically computable. The run time and tape of probabilistic machines are defined. A palindrome-like language is described that can be recognized faster by one-tape probabilistic Turing machines than by one-tape deterministic Turing machines. It is shown that every nondeterministic machine can be simulated in the same space by a probabilistic machine with small error probability. Several classes of languages recognized probabilistically in polynomial time are defined and compared with *NP*.

Key words. probabilistic Turing machines, probabilistic algorithms, probabilistic computation, computational complexity, Turing machines, Markov chains, crossing sequences, polynomial reducibility, polynomial-complete languages

1. Introduction. Probabilistic methods in computation have usually been used for problems arising from probabilistic considerations [5]. Monte Carlo methods are often used to obtain qualitative information about the behavior of large systems, especially for systems subject to random disturbances. As another example, Monte Carlo integration techniques are most applicable to multidimensional integrals, which typically are the probabilities of events that are complex combinations of simpler events.

Recently certain problems have been shown to be solvable by probabilistic algorithms that are faster than the known deterministic algorithms for solving these problems. Rabin [20] describes a probabilistic algorithm for finding the nearest pair of a set of n points in k-space that executes in average time $O(n)$, which is faster than the $n \log n$ steps required by the best known deterministic algorithms. Solovay and Strassen [26] and Rabin [20] present probabilistic algorithms for recognizing prime numbers in polynomial time with a small probability of error. These results suggest that probabilistic algorithms may be useful for solving other deterministically intractable problems.

In this paper we study a formal model for probabilistic algorithms, the probabilistic Turing machine. A probabilistic Turing machine is a computer with the ability to make random decisions. The output of a probabilistic machine is not in general uniquely determined by the input, but we can define in a natural way the partial function computed by a probabilistic machine (Definition 2.1). It has been shown that the ability to make random decisions does not increase the ultimate computational power of Turing machines [4], [23], [8]. However, it is natural to ask if probabilistic machines can be proven to require less time or tape than deterministic machines. This paper provides a partial answer to this question.

In § 2 we define probabilistic Turing machines (PTMs) and the functions computed by probabilistic machines. We show that every probabilistically computable function is partial recursive, and in fact that any standard enumeration of

* Received by the editors June 1, 1976, and in revised form February 18, 1977.

† Department of Electrical Engineering, Stanford University, Stanford, California 94305. This research was supported in part by National Science Foundation under Grant GK-43121.

PTMs yields an acceptable Gödel numbering of the partial recursive functions. We discuss briefly the relationship between our definitions of PTMs and the more general notion of Santos [23].

In § 3 we show that maximum and average run time are unsuitable as complexity measures for the entire class of PTMs. We introduce a more convenient concept of probabilistic run time, and show that every probabilistic machine can be simulated deterministically in at most exponentially more time.

In § 4 we exhibit a palindrome-like language that can be recognized by a fixed one-tape probabilistic Turing machine faster for infinitely many inputs than by any one-tape deterministic Turing machine.

In § 5 we prove elementary facts about several classes of languages recognizable probabilistically in polynomial time. Simon [25] has shown that the largest of these classes, *PP*, contains *NP* and is identical with the class of languages accepted by polynomial bounded threshold machines. The set of formulas of propositional calculus satisfied by a majority of interpretations is proven to be a polynomial complete language for *PP*.

In § 6 we define tape for probabilistic machines and prove that it is a Blum complexity measure. We show that the output length of a PTM is at most an exponential of the tape; that the run time is at most a double exponential of the tape; that every PTM can be simulated deterministically in at most exponentially more tape; and that every language accepted by a nondeterministic machine can be recognized probabilistically using the same amount of tape.

2. Probabilistic Turing machines. The model of probabilistic computation studied in this paper, the probabilistic Turing machine, is obtained from the usual Turing machine model by allowing machines access to the simplest type of randomness. A probabilistic Turing machine is a coin-tossing computer which can make decisions based on the outcomes of fair coin tosses.

We assume a standard Turing machine model [11]. A *multitape Turing machine* consists of a finite control unit equipped with a read-only input tape, a write-only output tape, and a finite number of read-write worktapes. A Turing machine is *deterministic* if the current state of the machine uniquely determines the next action of the machine; otherwise the machine is *nondeterministic*.

DEFINITION 2.1. A *probabilistic Turing machine* (PTM) is a Turing machine with distinguished states called coin-tossing states. For each coin-tossing state, the finite control unit specifies two possible next states. The computation of a probabilistic Turing machine is deterministic except that in coin-tossing states the machine tosses an unbiased coin to decide between the two possible next states.

Formal definitions of PTMs in terms of state-transition functions are given in [8] and [23] and are omitted here.

The probabilistic Turing machine model can be extended by relaxing the requirement that unbiased random decisions are made. It can be shown that the resulting model has the same computational power as the PTM defined by Santos [23]. At the end of this section, we state a result characterizing the functions computable by PTMs with unrestricted coin biases.

The computation of a PTM is determined by its input and the outcomes of the coin tosses performed by the machine. It will be convenient to describe probabilistic machines in terms of partial functions of two variables. One variable is the usual input to the machine. The other variable is the random input, a binary sequence representing the outcomes of the coin tosses. For simplicity we suppose that the machine tosses a coin at each step of its computation, although the result of the coin toss is ignored except when the machine is in a coin-tossing state. The possible computations of a probabilistic machine M_i with tape alphabet Σ are represented by a partial function $\phi_i: \Sigma^* \times \{0, 1\}^* \to \Sigma^*$. The value of $\phi_i(x; \alpha)$ is the output of the computation of M_i on input x with coin tosses specified by the binary sequence α. By convention, $\phi_i(x; \alpha)$ is undefined if the computation requires more than $|\alpha|$ steps, where $|\alpha|$ denotes the length of α.

In the remainder of this paper $M_i(x)$ will denote either the possible computations of M_i with input x (a random process) or the output of the computation (a random variable). We assume a standard enumeration $\{M_i\}$ of the probabilistic Turing machines.

In general, a PTM computes a random function [23]; for each input x, the machine M_i produces output y with probability $\Pr\{M_i(x)=y\}$. We define the deterministic partial function computed by any PTM as follows.

DEFINITION 2.2. The partial function ϕ_i computed by probabilistic Turing machine M_i is defined by

$$\phi_i(x)=\begin{cases} y & \text{if } \Pr\{M_i(x)=y\}>\frac{1}{2}, \\ \text{undefined} & \text{if no such } y \text{ exists.}\end{cases}$$

A partial function is *probabilistically computable* if it is computed by some PTM.

PROPOSITION 2.3. *Every function computed by a probabilistic Turing machine is partial recursive. Therefore the class of probabilistically computable functions is the same as the class of partial recursive functions.*

Proof. Every partial recursive function is probabilistically computable, since deterministic Turing machines are special cases of probabilistic Turing machines. Conversely, let M_i be a PTM. To compute $\phi_i(x)$ it suffices to find an output y such that $\Pr\{M_i(x)=y\}>\frac{1}{2}$. For any n we can compute $\Pr\{M_i(x)=y \text{ in time } n\}$, the probability that $M_i(x)$ outputs y in some computation of at most n steps. This probability equals $m2^{-n}$, where m is the number of coin toss sequences α of length n such that $\phi_i(x; \alpha)=y$. If $\Pr\{M_i(x)=y\}>\frac{1}{2}$ for some y, then $\Pr\{M_i(x)=y \text{ in time } n\}>\frac{1}{2}$ for all large enough n. Therefore to compute $\phi_i(x)$, we systematically evaluate $\Pr\{M_i(x)=y \text{ in time } n\}$ until we find some y and n such that this probability exceeds $\frac{1}{2}$. We then set $\phi_i(x)=y$. □

The next two propositions imply that $\{\phi_i\}$ is an acceptable Gödel numbering [21] of the partial recursive functions. We assume a standard computable pairing function $\langle i, x\rangle$.

PROPOSITION 2.4 (Universal PTM). *There is a universal probabilistic Turing machine M_u such that $\phi_u(\langle i, x\rangle; \alpha)=\phi_i(x; \alpha)$ for every binary string α. In particular,* $\Pr\{M_u\langle i, x\rangle=y\}=\Pr\{M_i(x)=y\}$ *for every i, x, and y, and therefore $\phi_u\langle i, x\rangle=\phi_i(x)$.*

PROPOSITION 2.5 (S-m-n Theorem). *There is a recursive function s such that $\phi_{s(e,i)}(x; \alpha)=\phi_e(\langle i, x\rangle; \alpha)$ for every binary string α. In particular,* $\Pr\{M_{s(e,i)}(x)=y\}=\Pr\{M_e\langle i, x\rangle=y\}$ *for every e, i, x, and y, and therefore $\phi_{s(e,i)}(x)=\phi_e\langle i, x\rangle$.*

The proofs of Propositions 2.4 and 2.5 use simulations essentially identical to those used for deterministic Turing machines. As a corollary of these two propositions, the recursion theorem holds for $\{\phi_i\}$. In fact, the standard proof of the recursion theorem [22, p. 180] can be modified to yield the following stronger result.

PROPOSITION 2.6 (Recursion Theorem). *For every total recursive function f there is an index e such that $\phi_e(x;\alpha)=\phi_{f(e)}(x;\alpha)$ for every binary string α. In particular,* $\Pr\{M_e(x)=y\}=\Pr\{M_{f(e)}(x)=y\}$ *for every x and y, and therefore $\phi_e = \phi_{f(e)}$.*

In the later sections of this paper we shall be interested in the time and tape used by probabilistic machines. Another important aspect of a PTM is its reliability.

DEFINITION 2.7. The *error probability* of M_i is the function e_i defined by

$$e_i(x)=\begin{cases}\Pr\{M_i(x)\neq\phi_i(x)\} & \text{if } \phi_i(x) \text{ is defined,}\\ \text{undefined} & \text{if } \phi_i(x) \text{ is undefined.}\end{cases}$$

Clearly $e_i(x)<\frac{1}{2}$ whenever $e_i(x)$ is defined. In general $e_i(x)$ is not computable, as can be shown using the recursion theorem of Proposition 2.6. However, $e_i(x)$ can be effectively approximated from above.

A useful probabilistic algorithm should have small probability of error. At the very least, the error probability should be uniformly bounded below $\frac{1}{2}$ for all inputs.

DEFINITION 2.8. M_i computes ϕ_i with *bounded error probability* if there is a constant $\varepsilon<\frac{1}{2}$ such that $e_i(x)\leqq\varepsilon$ for every ε in the domain of ϕ_i.

A PTM which computes with bounded error probability corresponds to a probabilistic automaton with an isolated cutpoint at $\frac{1}{2}$ [16], [18].

The probabilistic Turing machine model of Definition 2.1 restricts the randomness accessible by the computer to be of a very simple type—independent equiprobable bits. The more general probabilistic Turing machines of Santos [23] can be simulated by coin-tossing machines, provided that coins with arbitrary biases are allowed. The computational power of coin-tossing machines with biased coins is characterized by the following proposition, part of which appears implicitly in [24].

PROPOSITION 2.9. *Let $p_1, p_2, \cdots, p_k$ be real numbers such that $0\leqq p_i\leqq 1$. The partial functions computable probabilistically by PTMs which make random decisions with biases drawn from $\{p_1, p_2, \cdots, p_k\}$ are exactly the functions partial recursive in the binary representations of $p_1, p_2, \cdots, p_k$.*

3. Time. In the last section we showed that only partial recursive functions are probabilistically computable, and so no new functions can be computed by probabilistic algorithms. The natural question is whether probabilistic machines can compute more efficiently than deterministic machines, that is, using less time or tape. We study time in this and the following two sections and tape in the final section.

First we must agree on the definition of run time for PTMs. The obvious measure of probabilistic run time is the average time. (In some situations the maximum running time is more appropriate.) Although average run time is the first choice as measure of computation time for PTMs, it has theoretical drawbacks. Average run time is not a Blum complexity measure for the entire class of PTMs.

We recall that $\{\phi_i, \Phi_i\}$ is a Blum measure of computational complexity if $\{\phi_i\}$ is an acceptable Gödel numbering and $\{\Phi_i\}$ is a sequence of partial recursive functions satisfying the two axioms [1]:

(B1) $\phi_i(x)$ is defined iff $\Phi_i(x)$ is defined;

(B2) $\Phi_i(x) \leqq n$ is a decidable predicate of i, x, and n.

Suppose that ϕ_i denotes the partial recursive function computed by the probabilistic Turing machine M_i. Let us write $\bar{T}_i(x)$ for the least integer greater than or equal to the average run time of $M_i(x)$. Then $\{\phi_i, \bar{T}_i\}$ is not a complexity measure for several reasons:

(i) $\bar{T}_i$ is not in general a computable function. However, $\bar{T}_i$ is approximable from below; that is, there is a recursive function $g(i, x, n)$, nondecreasing in n, such that $\bar{T}_i(x) = \lim_{n\to\infty} g(i, x, n)$. In fact, *every* partial function ψ such that $\psi(x) \geqq 3$ and $\psi(x) = \lim_{n\to\infty} g(x, n)$ for some recursive function g nondecreasing in n is the average run time of some probabilistic algorithm.

(ii) $\{\phi_i, \bar{T}_i\}$ does not satisfy axiom (B1). On the one hand, $\bar{T}_i(x)$ may be finite even if $\phi_i(x)$ is undefined, for example, if $\Pr\{M_i(x) = 0\} = \Pr\{M_i(x) = 1\} = \frac{1}{2}$. A similar situation occurs for tape and tape reversal measures for deterministic Turing machines. This difficulty could be removed by redefining $\bar{T}_i(x)$ to be infinite whenever $\phi_i(x)$ is undefined. However, $\bar{T}_i(x)$ may be infinite even when $\phi_i(x)$ is defined.

(iii) $\{\phi_i, \bar{T}_i\}$ does not satisfy axiom (B2). This can be shown directly using the recursion theorem as in [8], or it can be inferred from the following result.

PROPOSITION 3.1. *Every recursively enumerable set is accepted by some probabilistic Turing machine with finite average run time.*

Proof. By definition, the set accepted by a Turing machine (deterministic or probabilistic) is the domain of the function computed by the machine. Let W be any r.e. set, and suppose that W is accepted by a deterministic Turing machine M. A probabilistic machine M' accepting W executes the following program:

```
1 repeat
2   simulate one step of $M(x)$;
3   if $M(x)$ accepted at last step then accept;
4 until cointoss() = heads;
5 if cointoss() = heads then accept else reject.
```

Here cointoss is a Boolean function that returns the result of an unbiased coin toss.

For x not in W, the domain of M, the above procedure can terminate only at line 5, and so it rejects and accepts with probability $\frac{1}{2}$; thus x is not in the domain of M'. On the other hand, if x is in W, then M' accepts on line 3 with positive probability, and so M' accepts x with probability greater than $\frac{1}{2}$. Therefore M' accepts W. A straightforward calculation bounds the average run time of M' by 5. □

Suppose that we modify the flowchart in the proof of Proposition 3.1 by changing line 3 to

```
3' if $M(x)$ accepted at last step then loop forever;
```

Let $\bar{T}(x)$ denote the average run time of M' with input x. Then $\bar{T}(x) = \infty$ for x in W and $\bar{T}(x) \leqq 5$ for x not in W. Thus x belongs to W iff $\bar{T}(x) > 5$. We conclude that $\{\phi_i, \bar{T}_i\}$ does not satisfy axiom (B2), since otherwise every r.e. set would be recursive.

The proof of the following result is similar to that of Proposition 3.1.

COROLLARY 3.2. *Every* 0, 1-*valued recursive function can be computed probabilistically with finite average run time.*

A relatively simple diagonalization shows that the restriction of Corollary 3.2 to 0, 1-valued functions in necessary.

PROPOSITION 3.3. (i) *For every recursive function h there is a* 0, 1, 2-*valued recursive function f requiring average run time more than $h(x)$ a.e.*

(ii) *Every recursive function f requires average run time more than $\frac{1}{2}|f(x)|$ on input x.*

The procedure described in Proposition 3.1 is not a useful method for accepting r.e. sets, since the error probability is very nearly $\frac{1}{2}$. According to Proposition 3.7 below, average run time is a more reasonable cost measure when restricted to PTMs with bounded error probability.

Maximum run time is also inadequate for measuring the time of probabilistic computations, primarily because maximum run time can be infinite even for algorithms with finite average run time. We are thus led to the following "theoretical" definition of probabilistic time. Recall that $\Pr\{M_i(x)=y \text{ in time } n\}$ is the probability that M_i with input x gives output y in some computation of at most n steps.

DEFINITION 3.4. The *Blum run time T_i* of probabilistic Turing machine M_i is defined by

$$T_i(x)=\begin{cases}\text{least } n \text{ such that} & \\ \Pr\{M_i(x)=\phi_i(x) \text{ in times}\}>\frac{1}{2} & \text{if } \phi_i(x) \text{ is defined,}\\ \infty & \text{if } \phi_i(x) \text{ is undefined.}\end{cases}$$

This definition, which also appears in [30], is analogous to the definition of the run time of a nondeterministic Turing machine as the length of the shortest accepting computation.

PROPOSITION 3.5. *$\{\phi_i, T_i\}$ is a Blum complexity measure.*

Proof. To verify the first axiom, we must show that $T_i(x)<\infty$ iff $\phi_i(x)$ is defined. By definition, $T_i(x)=\infty$ if $\phi_i(x)$ is not defined. Conversely, suppose that $\phi_i(x)$ is defined. Then $\Pr\{M_i(x)=\phi_i(x)\}>\frac{1}{2}$, and so $\Pr\{M_i(x)=\phi_i(x) \text{ in time } n\}>\frac{1}{2}$ for all large enough n. Therefore $T_i(x)<\infty$.

The second axiom is also easily verified. For any i, x, and n, we show how to decide deterministically in time $O(n^2 2^n)$ whether $T_i(x)\leqq n$. As in the proof of Proposition 2.3, we can calculate $\Pr\{M_i(x)=y \text{ in time } n\}$ by a straightforward simulation of all possible computations of $M_i(x)$. Similarly we can calculate the probability that y is a prefix of the output of $M_i(x)$ in time n, which we denote by $\Pr\{M_i(x)\sqsupset y \text{ in time } n\}$. These probabilities can be computed in time $O(n2^n)$. Now we proceed by induction on m to determine if the output of $M_i(x)$ in time n is of length m. For $m=0$ we merely check if $\Pr\{M_i(x)=\Lambda \text{ in time } n\}>\frac{1}{2}$. Suppose that for some $m<n$ we have found y of length m such that $\Pr\{M_i(x)\sqsupset y \text{ in time } n\}>\frac{1}{2}$ but $\Pr\{M_i(x)=y \text{ in time } n\}\leqq\frac{1}{2}$. We compute $\Pr\{M_i(x)=y' \text{ in time } n\}$ and $\Pr\{M_i(x)\sqsupset y' \text{ in time } n\}$ for each y' of length $m+1$ that extends y. If $\Pr\{M_i(x)=y' \text{ in time } n\}>\frac{1}{2}$ for some such y' then $T_i(x)\leqq n$. Otherwise either $\Pr\{M_i(x)\sqsupset y' \text{ in time } n\}>\frac{1}{2}$ for some y' of length $m+1$ or $\phi_i(x)$ is undefined and therefore $T_i(x) > n$. Thus either for some $m \leqq n$ we find the output $\phi_i(x)$ of length m and conclude that $T_i(x) \leqq n$ or we verify that $T_i(x) > n$. The entire procedure can be performed in time $O(n) \cdot O(n2^n) = O(n^2 2^n)$. □

PROPOSITION 3.6. *If M is a probabilistic Turing machine with Blum run time T, then M can be simulated by a deterministic Turing machine in time $O(T^2 2^T)$.*

Proof. For $n = 1, 2, 3, \cdots$ we use the procedure outlined in the proof of Proposition 3.5 to decide if $T(x) \leqq n$. For $n = T(x)$ the procedure produces the output of $M(x)$. The total computation time is at most $O(1^2 \cdot 2^1) + O(2^2 \cdot 2^2) + \cdots + O(T(x)^2 2^{T(x)}) = O(T(x)^2 2^{T(x)})$. □

The next result gives evidence that Definition 3.4 is a reasonable notion of probabilistic run time.

PROPOSITION 3.7. *If M_i is a probabilistic Turing machine with bounded error probability, then there is a constant $c > 0$ such that $T_i(x) \leqq c\bar{T}_i(x)$ whenever $\phi_i(x)$ is defined.*

Proof. If M_i has bounded error probability then there is a constant $\varepsilon < \frac{1}{2}$ such that $e_i(x) < \varepsilon$ for every x in the domain of ϕ_i. Let $c = 1/(\frac{1}{2} - \varepsilon)$.

If $\bar{T}_i(x) = \infty$ then there is nothing to prove. So suppose that $\bar{T}_i(x) < \infty$. Obviously Pr {run time of $M_i(x) > c\bar{T}_i(x)\} < 1/c = \frac{1}{2} - \varepsilon$, or equivalently, Pr {run time of $M_i(x) \leqq c\bar{T}_i(x)\} > \frac{1}{2} + \varepsilon$. Since Pr $\{M_i(x) \neq \phi_i(x)$ in time $c\bar{T}_i(x)\} \leqq$ Pr $\{M_i(x) \neq \phi_i(x)\} < \varepsilon$, it follows that Pr $\{M_i(x) = \phi_i(x)$ in time $c\bar{T}_i(x)\} > \frac{1}{2}$. Therefore $T_i(x) \leqq c\bar{T}_i(x)$. □

No reverse inequality is true in general; there are PTMs with bounded error probability for which $\bar{T}_i(x) = \infty$ and $T_i(x) < \infty$. Proposition 3.7 states that for PTMs with bounded error probability, the average run time cannot be much less than the Blum run time, and therefore the average run time does not assign too small a cost to computations. The pathology of Proposition 3.1 cannot occur for PTMs with bounded error probability.

COROLLARY 3.8. *If M is a probabilistic Turing machine with bounded error probability and average run time $\bar{T}$, then M can be simulated by a deterministic Turing machine in time $2^{O(\bar{T})}$.*

Proof. Let $\varepsilon < \frac{1}{2}$ be a uniform error probability bound for M. Then by Propositions 3.6 and 3.7, if $c = 1/(\frac{1}{2} - \varepsilon)$, then M can be simulated deterministically in time $O((c\bar{T})^2 2^{c\bar{T}}) = 2^{O(\bar{T})}$. □

To conclude this section, we list fundamental open problems about the computational power of probabilistic Turing machines. We state these problems as positive conjectures.

Conjecture 1. There is a function computable probabilistically in polynomial time but not computable deterministically in polynomial time.

Conjecture 2. There is a function computable probabilistically with bounded error probability in polynomial time but not computable deterministically in polynomial time.

Conjecture 3. There is a function computable probabilistically with zero error probability in polynomial bounded *average* running time but not computable deterministically in polynomial time.

Conjecture 2 obviously implies Conjecture 1, and it can be shown, as in Proposition 5.2, that Conjecture 3 implies Conjecture 2. In § 5 we observe that Conjecture 1 is true if $P \neq NP$; thus Conjecture 1 is quite plausible.

A function satisfying the conditions of Conjecture 2 would be much more interesting. One candidate is PRIMES, the characteristic function of the set of prime numbers. Solovay and Strassen [26] and Rabin [20] have described efficient Monte Carlo tests for primality, thus showing that the primes can be recognized probabilistically in polynomial time with bounded error probability. (However, there is also the possibility that the primes can be recognized *deterministically* in polynomial time, which is the case if the extended Riemann hypothesis is true [14].) We suggest that other difficult (but not NP-hard) problems might be solvable probabilistically in polynomial time, such as graph isomorphism, linear inequalities, and optimum trees with unequally weighted branches.

In the next section we give an affirmative answer to a much weaker version of Conjecture 2.

4. An example of probabilistic speedup. In this section we exhibit a language that can be recognized more rapidly by one-tape probabilistic Turing machines than by one-tape deterministic Turing machines. The advantage of restricting ourselves to one-tape machines is the availability of crossing sequence techniques [19], [10], [28] for establishing lower bounds on the complexity of some simple computational problems.

We denote the ith symbol of a string w by $w[i]$. For any string w of even length $2n$, let $r(n)$ be the fraction of symbols in the first half of w which equal the corresponding symbols in the latter half of w; that is, $r(w)=m/n$, where m is the number of indices i between 1 and n such that $w[i]=w[n+i]$. For any real number λ let P_λ be the palindrome-like language of binary strings w of even length such that $r(w)\geqq\lambda$. In particular, $P_1=\{ww: w \text{ is a binary string}\}$ is a subset of P_λ for every $\lambda<1$. We denote by P_1^{2n} the strings of P_1 of length $2n$.

THEOREM 4.1. *Suppose that λ is a rational number such that $0<\lambda<1$. There is a one-tape probabilistic Turing machine M that recognizes P_λ with bounded error probability and runs faster for infinitely many inputs than any one-tape deterministic Turing machine that recognizes P_λ. That is, if M' is any one-tape deterministic Turing machine that recognizes P_λ, then the (maximum) run time of M is less than the run time of M' for infinitely many inputs.*

Proof. In Lemma 4.2 we describe a one-tape probabilistic Turing machine M which recognizes P_λ in time $O(n \log n)$ for inputs in P_1. In Lemma 4.3 we prove that for every one-tape deterministic Turing machine M' that recognizes P_λ there is a constant $c>0$ such that the maximum run time of M' on inputs in P_1^{2n} is at least cn^2. Therefore Lemmas 4.2 and 4.3 provide the proof of the theorem. □

LEMMA 4.2. *For every rational number λ such that $0<\lambda<1$ there is a one-tape probabilistic Turing machine M that recognizes P_λ with bounded error probability and maximum run time $O(n \log n)$ for inputs in P_1.*

Proof. For any error probability bound $\varepsilon>0$ let m be an integer large enough that $((1+\lambda)/2)^m<\varepsilon$. Suppose that M_0 is any standard one-tape deterministic Turing machine that recognizes P_λ. (Straightforward one-tape Turing machine programming techniques yield a machine M_0 with run time $O(n^2)$.) Let M be a one-tape probabilistic machine that with input w operates as follows:

(i) M checks that the input length is even, say $2n$, and calculates $\tilde{n}$, the binary representation of n. (By the usual one-tape machine method for converting a unary input to binary [15, p. 123], this stage can be performed in $O(n \log n)$ steps.)

(ii) M randomly selects m numbers $i_1, i_2, \cdots, i_m$ such that $1 \leqq i_j \leqq n$. M selects the index i_j by letting $i_j = 1 + (\alpha_j \bmod n)$, for a randomly chosen bit string α_j of length $|\tilde{n}|$. Note that i_j assumes values of numbers between 1 and n with probabilities $2^{-|\tilde{n}|}$ or $2^{-|\tilde{n}|+1}$. (Obtaining these random samples requires only $O(\log n)$ steps.)

(iii) M compares $w[i_j]$ and $w[n+i_j]$ for $j = 1, 2, \cdots, m$. If $w[i_j] = w[n+i_j]$ for every j, then M accepts w. (Each comparison of $w[i_j]$ with $w[n+i_j]$ can be made in $O(n \log n)$ steps, using a familiar technique for selecting the ith symbol of the tape. In this technique, we load i into a counter stored on the tape, then while decrementing the counter advance the tape head and simultaneously drag the counter along.)

(iv) If $w[i_j] \neq w[n+i_j]$ for some j, then M simulates the deterministic machine M_0 to determine if w belongs to P_λ.

Clearly, inputs in P_1 are accepted by M at the end of stage (iii). Therefore M runs in time $O(n \log n)$ on P_1.

We must show that M recognizes P_λ with error probability less than ε. Because M does not falsely reject strings in P_λ, it is enough to bound above the probability of falsely accepting a string w not in P_λ. Let $2n$ be the length of w. Since w is not in P_λ, more than $(1-\lambda)n$ of the numbers i between 1 and n satisfy $w[i] \neq w[n+i]$. It follows easily that the probability that $w[i_j] \neq w[n+i_j]$ for any one of the random samples i_j is at least $(1-\lambda)/2$, and so the probability that $w[i_j] = w[n+i_j]$ is at most $(1+\lambda)/2$. Since the samples $i_1, i_2, \cdots, i_m$ are independent, the probability that M accepts w, which is the probability that $w[i_j] = w[n+i_j]$ for every j, is at most $((1+\lambda)/2)^m < \varepsilon$. Therefore the error probability of M is less than ε. □

LEMMA 4.3. *Suppose that* $0 < \lambda < 1$. *For every one-tape deterministic Turing machine* M' *that recognizes* P_λ *there is a constant* $c > 0$ *such that the maximum run time of* M' *on inputs in* P_1 *of length* $2n$ *is at least* cn^2.

Proof. We use a crossing sequence argument [10]. Let $\mu = 1-\lambda$ and let ν be any number such that $\mu < \nu < 1$. Suppose that M' is a one-tape deterministic machine that recognizes P_λ. Fix n. For each k such that $\nu n \leqq k \leqq n$ we examine the crossing sequences of M' at boundary k of words in P_1^{2n}.

Suppose that uu and vv are strings in P_1^{2n} with the same crossing sequence at boundary k. Write uu and vv as $u_1u_2u_1u_2$ and $v_1v_2v_1v_2$, where $|u_1| = |v_1| = k$. Since both uu and vv are in P_1 and hence in P_λ, they are both accepted by M'. By the fundamental property of crossing sequences, M' accepts the hybrid string $u_1v_2v_1v_2$, and so $u_1v_2v_1v_2$ is in P_λ. Therefore u_1 and v_1 differ in at most $(1-\lambda)n = \mu n$ positions.

For a fixed crossing sequence ξ and string v_2 of length $n-k$, let $S(\xi, v_2)$ be the set of strings v_1 of length k such that $v_1v_2v_1v_2$ has crossing sequence ξ at boundary k. We shall show that there is a constant $\rho < 1$ such that the number $N(\xi, v_2)$ of elements in $S(\xi, v_2)$ is at most $2^{\rho k}$. In fact,

$$N(\xi, v_2) \leqq \sum_{i=0}^{\mu n/2} \binom{k}{i} \leqq 2^{kH(\mu n/(2k))} \leqq 2^{kH(\mu/(2\nu))}, \tag{4.1}$$

where H is the binary entropy function [7, p. 78] defined by

$$H(x) = -x \log_2 x - (1-x)\log_2 (1-x).$$

The first inequality in (4.1) follows from a theorem of Kleitman [12] on the maximum number of binary sequences of length k, no two of which differ in more than μn positions. The second inequality is an application of the Chernoff bound [2] and can be found in [17, p. 466]. The final inequality holds because $H(x)$ is strictly increasing for $0 \leqq x \leqq \frac{1}{2}$. Let $\rho = H(\mu/(2\nu))$. Then $\rho < 1 = H(\frac{1}{2})$ because $\mu/2\nu < \frac{1}{2}$. Thus inequality (4.1) can be rewritten $N(\xi, v_2) \leqq 2^{\rho k}$.

For a fixed v_2 of length $n-k$, at most $2^{\rho k}$ strings $v_1v_2v_1v_2$ in P_1^{2n} have crossing sequence ξ at boundary k. Summing over v_2, we see that the number $N(\xi)$ of strings in P_1^{2n} with crossing sequence ξ at boundary k is bounded by $2^{n-\delta n}$ for some $\delta > 0$. In fact,

$$N(\xi) \leqq 2^{n-k} \cdot 2^{\rho k} = 2^n \cdot 2^{-(1-\rho)k} \leqq 2^n \cdot 2^{-(1-\rho)\nu n}, \tag{4.2}$$

since $k \geqq \nu n$. We can take $\delta = (1-\rho)\nu$. There are 2^n sequences in P_1^{2n}, and so among all strings in P_1^{2n} at least $2^{\delta n}$ distinct crossing sequences must occur at boundary k.

Next we estimate the average, over all words in P_1^{2n}, of the crossing sequence length at boundary k. If s is the number of states of M', then there are s^i crossing sequences of length i. Define l by

$$\sum_{i=0}^{l} s^i \leqq 2^{\delta n} < \sum_{i=0}^{l+1} s^i. \tag{4.3}$$

(Note that for some string in P_1^{2n} a crossing of length at least l must occur at boundary k.) From (4.3) we infer that $s^l \geqq s^{-2}2^{\delta n}$, and consequently $l \geqq (\delta/\log_2 s)n - 2$. The average crossing sequence length at boundary k is at least

$$2^{-n} \sum_{i=0}^{l} is^i 2^{n-\delta n} \geqq ls^l 2^{-\delta n} \geqq l/s^2 \geqq (\delta/(s^2 \log_2 s))n - 2/s^2, \tag{4.4}$$

because at most $s^i 2^{n-\delta n}$ strings have crossing sequences of length i at boundary k.

The computation time of M' on any input is the sum of the crossing sequence lengths over all boundaries. Summing the average crossing sequence length over all boundaries k such that $\nu n \leqq k \leqq n$, we obtain the following lower bound for the average computation time of M' on inputs in P_1^{2n}:

$$(n - \nu n)[(\delta/(s^2 \log_2 s))n - 2/s^2] = an^2 - bn \tag{4.5}$$

for some constants $a, b > 0$. Therefore the average computation time is at least cn^2 for some $c > 0$, and so the maximum run time of M' on P_1^{2n} is at least cn^2. □

The significance of Theorem 4.1 is that it gives an example of a problem and a machine model for which we can *prove* that probabilistic algorithms are faster than deterministic algorithms. The chief limitations of this result are that the amount of the speedup is small and that the machine model is of limited practical interest.

Vaiser [31] constructed a one-tape probabilistic Turing machine that recognizes the palindromes in *linear* time but with unbounded error probability.

Freivald [6] has improved Theorem 4.1 by showing that P_1 can be recognized by a one-tape probabilistic Turing machine with bounded error probability in time $O(n \log^2 n)$.

5. Probabilistic polynomial languages. We now define several classes of languages computable probabilistically in polynomial time and investigate relationships between these classes.

A probabilistic or nondeterministic Turing machine is *polynomial bounded* if there is a polynomial $p(n)$ such that every possible computation of the machine on inputs of length n halts in at most $p(n)$ steps. A probabilistic machine *recognizes* a language if the machine computes the characteristic function of the language.

DEFINITION 5.1. (i) *PP* is the class of languages recognized by polynomial bounded PTMs.

(ii) *BPP* is the class of languages recognized by polynomial bounded PTMs with bounded error probability.

(iii) *ZPP* is the class of languages recognized by PTMs with polynomial bounded *average* run time and zero error probability.

PROPOSITION 5.2. (i) *ZPP* $\subseteq$ *BPP* $\subseteq$ *PP* $\subseteq$ *PSPACE.*

(i) *PP, BPP, and ZPP are closed under complementation.*

(iii) *BPP and ZPP are closed under union and intersection.*

Proof. (i) It is clear from the proof of Proposition 3.5 that every polynomial bounded Turing machine can be simulated in polynomial space, and so *PP* $\subseteq$ *PSPACE*. By definition, *BPP* $\subseteq$ *PP*. To show that *ZPP* $\subseteq$ *BPP*, suppose that a language L is recognized by a probabilistic machine M with zero error probability and average run time bounded by a polynomial $p(n)$. For any constant $c > 2$, let M' be a PTM that recognizes L by simulating M for up to $cp(n)$ steps on inputs of length n. If the simulated computation of M does not halt within this time, then M' halts with an arbitrary answer. Since M requires more than $cp(n)$ steps with probability less than $1/c$, the error probability of the polynomial bounded machine M' is at most $1/c < \frac{1}{2}$.

Part (ii) is obvious from the definitions.

(iii) Suppose that L_1 and L_2 are recognized with zero error probability by M_1 and M_2 with average run times bounded respectively by $p_1(n)$ and $p_2(n)$. By a standard construction we obtain from M_1 and M_2 a machine that recognizes $L_1 \cup L_2$ with zero error probability and average run time at most $n + p_1(n) + p_2(n)$. Therefore *ZPP* is closed under union.

We note that every language in *BPP* can be recognized by a polynomial bounded PTM with error probability smaller than any desired positive constant, since we can increase the reliability of a probabilistic computer by repeating its computations a sufficiently large number of times and giving as output the majority result. Suppose that L_1 and L_2 belong to *BPP*. For every $\varepsilon > 0$ there exist polynomial bounded machines M_1 and M_2 that recognize L_1 and L_2 with error probability at most $\varepsilon/2$. The standard machine derived from M_1 and M_2 that recognizes $L_1 \cup L_2$ is polynomial bounded and has error probability at most $\varepsilon/2 + \varepsilon/2 = \varepsilon$. Therefore *BPP* is closed under union.

The closure of *ZPP* and *BPP* under intersection follows from closure under union and complementation. □

It is not known whether *PP* is closed under union and intersection. If *PP* is not closed under union, then by Propositions 5.2 and 5.3, $NP \subsetneqq PP \subsetneqq PSPACE$.

PROPOSITION 5.3. $P \subseteq ZPP \subseteq NP \subseteq PP$.

Proof. Every polynomial bounded deterministic Turing machine computes with zero error probability. Therefore $P \subseteq ZPP$.

Suppose that L is in *ZPP* and M is a PTM that recognizes L with zero error probability and polynomial bounded average running time. Then M considered as a nondeterministic Turing machine accepts L in polynomial time, because for every input in L there is at least one accepting computation of M of polynomial bounded length. Therefore L is in *NP*, and so $ZPP \subseteq NP$.

Finally suppose that L is in *NP*. Without loss of generality, we may assume that L is accepted by a polynomial bounded nondeterministic machine M for which each state of the machine permits at most two possible next actions. If M is considered to be a probabilistic machine, then L is the set of strings for which there exists an accepting computation; that is, x is in L iff $\Pr\{M(x) \text{ accepts}\} > 0$. To show that L belongs to *PP*, we replace M by a machine M' such that $\Pr\{M(x) \text{ accepts}\} > 0$ iff $\Pr\{M'(x) \text{ accepts}\} > \frac{1}{2}$. The machine M' tosses a coin at the beginning of its computation and accepts immediately if the result is heads; otherwise M' simulates M probabilistically, accepting iff M accepts.

There remains one small detail. If x is not in L then it is possible that $\Pr\{M'(x) \text{ accepts}\} = \frac{1}{2}$. Thus M' might not compute the characteristic function of L. We must make a final modification to obtain a machine M'' such that $\Pr\{M''(x) \text{ accepts}\} < \frac{1}{2}$ for x not in L.

Let $p(n)$ be a polynomial bounding the run time of M. Every x in L of length n is accepted by M with probability at least $2^{-p(n)}$, since there is at least one accepting computation and every computation of length $p(n)$ has probability at least $2^{-p(n)}$. A probabilistic machine M'' recognizing L operates as follows. At the beginning of its computation, M'' tosses $p(n)+1$ coins and accepts without further computation with probability $\frac{1}{2} - 2^{-p(n)-1}$; otherwise M'' simulates M, accepting iff M accepts. It is easily calculated that M'' rejects inputs not in L with probability $\frac{1}{2} + 2^{-p(n)-1}$ and accepts inputs in L with probability at least $\frac{1}{2} + 2^{-2p(n)-1}$. Therefore M'' recognizes L probabilistically in polynomial time. We conclude that $NP \subseteq PP$. □

The most important question about the classes of probabilistic polynomial languages is whether they properly contain P. We believe that $P \subsetneqq ZPP$, which can be seen to be equivalent to Conjecture 3 of § 3. Furthermore, there is evidence that $P \subsetneqq BPP$ and $P \subsetneqq PP$.

It appears quite likely that $P \subsetneqq PP$, since $NP \subseteq PP$. In fact, Simon [25] has listed a large number of combinatorial problems that are polynomial complete in *PP*. These problems seem to be intermediate in complexity between *NP*-complete problems and *PSPACE*-complete problems, which suggests that $NP \subsetneqq PP$.

PRIMES, the set of prime numbers, is the leading candidate for a language in $BPP-P$. Rabin [20] and Solovay and Strassen [26] have devised probabilistic algorithms that recognize the prime numbers in polynomial time with bounded error probability, and so PRIMES is in *BPP*. (Rabin has used his probabilistic algorithm to discover a 400-bit number that is "very probably" a prime.) If it can be shown that PRIMES cannot be recognized deterministically in polynomial time, then PRIMES is in $BPP-P$.

We can suggest no language in $ZPP-P$. Rabin noted [20] that both Rabin's and Solovay–Strassen's primality testing algorithms always correctly identify prime numbers. If a probabilistic primality testing algorithm can be found which always correctly identifies composite numbers (but may make mistakes on prime numbers), then this algorithm can be combined with Rabin's algorithm to yield a procedure that recognizes prime numbers with zero error probability in polynomial bounded average time. This observation leads to the following definition and proposition.

DEFINITION 5.4. *VPP* is the class of languages recognized by polynomial bounded PTMs which have zero error probability for inputs not in the languages. Equivalently, L is in *VPP* iff L is recognized by a probabilistic Turing machine M such that $\Pr\{M(x) \text{ accepts}\}=0$ for every x not in L.

The composite numbers are an example of a language in *VPP*.

PROPOSITION 5.5. (i) *L is in VPP iff L is accepted by a PTM whose average run time is polynomial bounded on L.*

(ii) $VPP \subseteq NP \cap BPP$.

(iii) [Rabin] *L is in ZPP iff both L and $\bar{L}$ are in VPP.*

We omit the easy proof of Proposition 5.5.

It does not appear that either $BPP \subseteq NP$ or $NP \subseteq BPP$. We note that $NP \subseteq BPP$ if every language in *NP* can be accepted by a polynomial bounded nondeterministic machine such that for inputs in the language accepting computations are a large fraction of possible computations. The following diagram summarizes known relations among the classes *P*, *ZPP*, *VPP*, *BPP*, *NP*, *PP*, and *PSPACE*.

$$P \subseteq ZPP \subseteq VPP \begin{array}{c} \rotatebox{30}{$\subseteq$}\, BPP \,\rotatebox{-30}{$\subseteq$} \\ \rotatebox{-30}{$\subseteq$}\, NP \,\rotatebox{30}{$\subseteq$} \end{array} PP \subseteq PSPACE.$$

None of the inclusions are known to be proper. We also know little about the relation between *PP* and the polynomial arithmetic hierarchy of Meyer and Stockmeyer [13], [27], which is also contained in *PSPACE*.

The following proposition is analogous to the characterization [3] of *NP* as the class of languages L for which there exist a polynomially computable relation $R(x, y)$ and a polynomial $p(n)$ such that $L=\{x: R(x, y)$ holds for some string y of length $\leqq p(|x|)\}$.

PROPOSITION 5.6. *A language L belongs to PP iff there exist a polynomially computable relation $R(x, y)$ and a polynomial $p(n)$ such that $L=\{x: R(x, y)$ holds for a majority of strings y of length $p(|x|)\}$.*

To conclude this section, we describe a simple polynomial complete language for *PP*. A language B is *polynomial m-reducible* to another language A if there is a function f computable in polynomial time such that x is in B iff $f(x)$ is in A. If A and B are polynomial m-reducible to each other, then A and B are *polynomial m-equivalent*. A language A is *polynomial m-complete* in a class if A belongs to that class and every language in that class is polynomial m-reducible to A.

An *interpretation* of a propositional formula $F(x_1, \cdots, x_n)$ is any assignment of truth values to the propositional variables $x_1, \cdots, x_n$ of F. A *satisfying interpretation* of $F(x_1, \cdots, x_n)$ is an interpretation under which F is true. The set of formulas for which there is at least one satisfying interpretation is denoted by SAT. It is well known that SAT is polynomial m-complete for *NP* [3].

DEFINITION 5.7. (i) MAJ is the set of propositional formulas satisfied by a majority of their interpretations; that is, $F(x_1, \cdots, x_n)$ is in MAJ iff $F(x_1, \cdots, x_n)$ is true for more than 2^{n-1} assignments of truth values to $x_1, \cdots, x_n$.

(ii) #SAT is the set of pairs $\langle i, F\rangle$ such that propositional formula F has more than i satisfying interpretations.

Simon [25] has shown that *PP* is the class of languages accepted by polynomial bounded threshold machines and that #SAT and a large number of similar combinatorial problems are polynomial m-complete for threshold machines and hence for *PP*.

LEMMA 5.8 [Simon]. #SAT *is polynomial m-complete for PP.*

LEMMA 5.9. MAJ *and* #SAT *are polynomial m-equivalent.*

Proof. It is clear that MAJ is polynomial m-reducible to #SAT, because $F(x_1, \cdots, x_n)$ is in MAJ iff $\langle 2^{n-1}, F(x_1, \cdots, x_n)\rangle$ is in #SAT.

To show that #SAT is polynomial m-reducible to MAJ, suppose that w is an arbitrary input of the form $\langle i, F(x_1, \cdots, x_n)\rangle$. We may assume that $i < 2^n$, since it is obvious that w is not in #SAT if $i \geqq 2^n$. Let $G(x_1, \cdots, x_n)$ be a formula, computable in polynomial time, that has exactly $2^n - i$ satisfying interpretations,[1] and let x_0 be a propositional variable not occurring in $F(x_1, \cdots, x_n)$. Then the formula $H(x_0, x_1, \cdots, x_n) = x_0F(x_1, \cdots, x_n) \vee \bar{x}_0G(x_1, \cdots, x_n)$ has more than 2^n satisfying interpretations iff $F(x_1, \cdots, x_n)$ has more than i satisfying interpretations. Therefore $\langle i, F\rangle$ is in #SAT iff H is in MAJ. □

PROPOSITION 5.10. MAJ *is polynomial m-complete for PP.*

Proof. This follows immediately from Lemmas 5.8 and 5.9. For completeness, we show that MAJ is in *PP* by describing a probabilistic polynomial time algorithm for recognizing MAJ. With input $F(x_1, \cdots, x_n)$ we select equiprobably one of the 2^n possible interpretations and evaluate $F(x_1, \cdots, x_n)$ for this interpretation. If $F(x_1, \cdots, x_n)$ is false for this interpretation then we reject, while if $F(x_1, \cdots, x_n)$ is true then we accept with probability $1-2^{-n-1}$. It is easily calculated that formulas in MAJ are accepted with probability greater than $\frac{1}{2}+2^{-n-1}$ and formulas not in MAJ are rejected with probability greater than $\frac{1}{2}+2^{-n-2}$. □

We have been unable to construct polynomial complete languages for *ZPP* or *BPP*.

6. Tape. The chief purpose of this section is to show that the definition of probabilistic tape analogous to that of probabilistic time (Definition 3.4) yields a Blum complexity measure. As a corollary of justifying the definition, we see how to simulate deterministically a probabilistic machine in at most exponentially more space. This is the best result now known. At the end of the section we point out that every language accepted by a nondeterministic Turing machine can be recognized in the same tape by a PTM with bounded error probability. This is evidence that probabilistic machines might be more efficient than deterministic machines in the use of tape.

[1] $G(x_1, \cdots, x_n) = x_1 \cdots x_{r_1} \vee x_1 \cdots \bar{x}_{r_1} \cdots x_{r_2} \vee \cdots \vee x_1 \cdots \bar{x}_{r_1} \cdots \bar{x}_{r_{k-1}} \cdots x_{r_k}$, where $2^n - i = 2^{n-r_1} + 2^{n-r_2} + \cdots + 2^{n-r_k}$ and $1 \leqq r_1 < r_2 \cdots < r_k \leqq n$.

DEFINITION 6.1. The *Blum tape* S_i of probabilistic Turing machine M_i is defined by

$$S_i(x)=\begin{cases}\text{least } n \text{ such that} & \\ \Pr\{M_i(x)=\phi_i(x) \text{ in tape } n\}>\frac{1}{2} & \text{if } \phi_i(x) \text{ is defined},\\ \infty & \text{if } \phi_i(x) \text{ is undefined.}\end{cases}$$

The proof that probabilistic tape is a complexity measure is rather involved. We require a preliminary result bounding the length of the output of a PTM. (The corresponding result for deterministic machines is trivial.) We need a lemma about finite state Markov chains.

LEMMA 6.2. *Suppose that $\mathcal{M}$ is a Markov chain with s states. Let $f_{ij}(n)$ be the probability that $\mathcal{M}$, when started in state i at time* 0, *first reaches state j at time n. If $n \geqq s$ and $i \neq j$ then $f_{ij}(n) \leqq \frac{1}{2}$.*

Proof. Let I be the finite set of states and (p_{ij}) the transition probability matrix of $\mathcal{M}$. Fix a state j. Without loss of generality, we can assume that j is a trap state, since the first entry probabilities $f_{ij}(n)$ for $i \neq j$ do not depend on the transition probabilities from state j.

For any set of states H not including j, define $f_{ij}(n; H)$ and $g_i(n; H)$ as follows:

(i) $f_{ij}(n; H)$ is the probability that $\mathcal{M}$, started in i at time 0, first enters j at time n, without passing through any state of H at times $1, 2, \cdots, n-1$;

(ii) $g_i(n; H)$ is the probability that $\mathcal{M}$, started in i at time 0, passes through no state of H at times $1, 2, \cdots, n$.

To establish the lemma, we prove something stronger: If j is a trap state and (a) i is not in H and $n \geqq |I-H|$ or (b) i is in H and $n \geqq |I-H|+1$, then

$$f_{ij}(n; H) \leqq \tfrac{1}{2} g_i(n; H), \tag{6.1}$$

where $|I-H|$ denotes the number of states in $I-H$. The lemma is the special case $H=\varnothing$. The proof of (6.1) is by induction on $|I-H|$.

If $|I-H|=1$ then $H=I-\{j\}$. Since $i \neq j$ implies that i is in H, only case (b) of (6.1) need be considered. Obviously $f_{ij}(n; H)=0$ if $n \geqq 2=|I-H|+1$, and so (6.1) is true for $|I-H|=1$.

Now suppose that $|I-H|>1$. There are two cases.

Case (a). i is not in H and $n \geqq |I-H|$. Since j is a trap state,

$$\sum_{m=1}^{n} f_{ij}(m; H) \leqq g_i(n; H). \tag{6.2}$$

If $f_{ij}(m; H) > \frac{1}{2} g_i(n; H)$ for any $m<n$, then (6.2) implies that $f_{ij}(n; H) < \frac{1}{2} g_i(n; H)$. So in what follows we may assume that $f_{ij}(m; H) \leqq \frac{1}{2} g_i(n; H)$ for $m<n$. By case (b) of the induction hypothesis,

$$f_{ij}(n; H \cup \{i\}) \leqq \tfrac{1}{2} g_i(n; H \cup \{i\}), \tag{6.3}$$

since $|I-(H \cup \{i\})| < |I-H|$ and $n \geqq |I-H| = |I-(H \cup \{i\})|+1$. Next we note that

$$g_i(n; H) = g_i(n; H \cup \{i\}) + \sum_{m=1}^{n} f_{ii}(m; H) g_i(n-m; H), \tag{6.4}$$

and therefore

$$(6.5)\qquad \begin{aligned} g_i(n; H\cup\{i\}) &= g_i(n;H) - \sum_{m=1}^{n} f_{ii}(m;H)g_i(n-m;H)\\ &\leqq g_i(n;H)\left(1-\sum_{m=1}^{n} f_{ii}(m;H)\right), \end{aligned}$$

since $g_i(n;H)\leqq g_i(n-m;H)$. Combining (6.3) and (6.5) we obtain

$$(6.6)\qquad f_{ij}(n;H\cup\{i\})\leqq \tfrac{1}{2}g_i(n;H)\left(1-\sum_{m=1}^{n} f_{ii}(m;H)\right).$$

As in (6.4) we can use (6.6) and the inequalities $f_{ij}(m;H)\leqq\frac{1}{2}g_i(n;H)$ for $m<n$ to bound $f_{ij}(n;H)$:

$$(6.7)\qquad \begin{aligned} f_{ij}(n;H) &= f_{ij}(n;H\cup\{i\}) + \sum_{m=1}^{n} f_{ii}(m;H\cup\{j\})f_{ij}(n-m;H)\\ &\leqq \tfrac{1}{2}g_i(n;H)\left(1-\sum_{m=1}^{n} f_{ii}(m;H)\right) + \sum_{m=1}^{n}\tfrac{1}{2}g_i(n;H)f_{ii}(m;H)\\ &= \tfrac{1}{2}g_i(n;H). \end{aligned}$$

Case (b). i is in H and $n\geqq|I-H|+1$. Then

$$(6.8)\qquad \begin{aligned} f_{ij}(n;H) &= \sum_{k\notin H} p_{ik}f_{kj}(n-1;H)\\ &\leqq \sum_{k\notin H} p_{ik}\tfrac{1}{2}g_k(n-1;H) \text{ by case (a)}\\ &= \frac{1}{2}\sum_{k\notin H} p_{ik}g_k(n-1;H) = \tfrac{1}{2}g_i(n;H). \quad \square \end{aligned}$$

A probabilistic Turing machine M with a fixed input x can be thought of as a discrete time Markov process [23]. The states of this Markov process are the instantaneous descriptions of M with input x, and the transition probabilities are determined by the state-transition probabilities of M. An *instantaneous description* (ID) of $M(x)$ consists of the position of the input tape head, the state of the finite control, the contents of the worktapes, and the positions of the worktape heads. (The contents of the output tape are not included.) The number of instaneous descriptions of $M(x)$ within worktape n is bounded by $s(|x|+2)n^k d^n$, where s is the number of states of the finite control, k is the number of worktapes, and d is the number of symbols in the tape alphabet. For $n\geqq\log|x|$ this bound can be replaced by c^n for some constant c that depends on M. Let us denote by $I(i, x, n)$ the exact number of instantaneous descriptions of $M_i(x)$ that use at most n worktapes squares. For any computation $M_i(x)$ we let BEGIN be the start ID. Without loss of generality we may assume that there is a unique halt ID, denoted END. (Any PTM can be replaced by one using the same workspace which cleans its worktapes and rewinds its input tape before halting.)

PROPOSITION 6.3. *If S_i is the Blum tape of the probabilistic Turing machine M_i, then the length of the output of $M_i(x)$ is no more than the number of instantaneous* *descriptions of $M_i(x)$ in tape $S_i(x)$. In particular, if $S_i(x) \geqq \log |x|$, then there is a constant c depending on M_i such that $|\phi_i(x)| \leqq c^{S_i(x)}$.*

Proof. An ID of $M_i(x)$ is called a *writing* ID if the next action of $M_i(x)$ specified by the ID includes writing a symbol on the output tape. We construct a Markov chain whose states are the writing IDs of $M_i(x)$ in worktape $S_i(x)$ together with the halt ID END. For any two writing IDs I and I' the one-step transition probability $p(I'|I)$ is defined to be the probability that $M_i(x)$ starting in instantaneous description I reaches I' in a finite number of steps without passing through another writing ID, that is, after writing only one output symbol. The transition probability $p(\mathrm{END}|I)$ is the probability that $M_i(x)$ starting in I halts or loops without writing. END is defined to be a trap state, that is, $p(\mathrm{END}|\mathrm{END}) = 1$.

Suppose that the start ID BEGIN is not a writing ID. Let $f(n)$ be the probability that $M_i(x)$ in tape $S_i(x)$ halts with an output of length exactly n. Then

$$f(n) \leqq \sum_I p(I|\mathrm{BEGIN}) f_{I,\mathrm{END}}(n), \tag{6.9}$$

where $p(I|\mathrm{BEGIN})$ is the probability that I is the first writing ID reached by $M_i(x)$, and $f_{I,\mathrm{END}}(n)$ is the probability that the Markov chain defined above, starting in state I, first reaches the trap state END in exactly n steps. By Lemma 6.2 this latter probability is at most $\frac{1}{2}$ when n exceeds the number of states of the process. Therefore if $n \geqq I(i, x, S_i(x))$ then $f(n) \leqq \frac{1}{2}$ and clearly no string of length n can be the output of $M_i(x)$ with probability greater than $\frac{1}{2}$.

If BEGIN is a writing ID, then $f(n) = f_{\mathrm{BEGIN,END}}(n) \leqq \frac{1}{2}$ if $n \geqq I(i, x, S_i(x))$. Therefore $|\phi_i(x)| \leqq I(i, x, S_i(x))$. □

THEOREM 6.4. *$\{\phi_i, S_i\}$ is a Blum complexity measure.*

Proof. The first axiom is verified as in the proof of Proposition 3.5. To establish the second axiom, we sketch a method for deciding $S_i(x) \leqq n$. The procedure uses space bounded by a polynomial in $I(i, x, n)$.

By Proposition 6.3, if $S_i(x) \leqq n$ then $|\phi_i(x)| \leqq I(i, x, n)$. To decide if $S_i(x) \leqq n$ it is sufficient to determine if $\Pr\{M_i(x) = y \text{ in tape } n\} > \frac{1}{2}$ for each y of length up to $I(i, x, n)$. To answer the latter questions, we shall in fact calculate $\Pr\{M_i(x) = y \text{ in tape } n\}$ for all such y.

Fix y of length at most $I(i, x, n)$. We construct a Markov chain $\mathcal{M}$ whose states are a trap state FAIL together with all pairs $\langle I, w\rangle$ where I is an ID of $M_i(x)$ in tape n and w is a prefix of y. The number of states of this Markov chain is no more than $2I(i, x, n)^2$. The transition probabilities of the process are defined so that the Markov process simulates those computations of $M_i(x)$ within tape n which produce output y. The one-step transition probabilities of $\mathcal{M}$ are determined by the transition probabilities of M_i:

$$p(\langle I', w'\rangle|\langle I, w\rangle) = \begin{cases} \Pr\{M_i(x) \text{ starting in } I \text{ enters } I' \text{ in} \\ \quad \text{the next step without writing}\} & \text{if } w' = w \\ \\ \Pr\{M_i(x) \text{ starting in } I \text{ writes } a \text{ and} \\ \quad \text{enters } I' \text{ at the next step}\} & \text{if } w' = wa \end{cases}$$

and

$$p(\mathrm{FAIL}|\langle I, w\rangle) = 1 - \sum_{\langle I', w'\rangle} p(\langle I', w'\rangle|\langle I, w\rangle);$$

$p(\text{FAIL}|\langle I, w\rangle)$ is the probability that $M_i(x)$ starting in I performs in the next step any action inconsistent with giving y as output in a computation within worktape n. Since M_i is a coin-tossing PTM, all transition probabilities are 0, 1, or $\frac{1}{2}$.

The state $\langle\text{BEGIN}, \Lambda\rangle$ of the Markov chain corresponds to the overall state of $M_i(x)$ at the beginning of its computation, and the state $\langle\text{END}, y\rangle$ corresponds to the overall state of $M_i(x)$ at the end of a computation that has produced output y. By the definition of the simulating Markov chain, $\Pr\{M_i(x)=y \text{ in tape } n\}$ equals $\Pr\{\langle\text{BEGIN}, \Lambda\rangle \overset{*}{\vdash} \langle\text{END}, y\rangle\}$, the probability that the simulating process starting in state $\langle\text{BEGIN}, \Lambda\rangle$ eventually reaches state $\langle\text{END}, y\rangle$. We show how to calculate $\Pr\{\langle\text{BEGIN}, \Lambda\rangle \overset{*}{\vdash} \langle\text{END}, y\rangle\}$.

First we determine the states $\langle I, w\rangle$ such that $\Pr\{\langle I, w\rangle \overset{*}{\vdash} \langle\text{END}, y\rangle\}>0$. These states can be found by computing the transitive closure of the directed graph of states of $\mathcal{M}$. (Finding these states can be accomplished in space bounded by a polynomial in $I(i, x, n)$.) Let the states satisfying this condition be $s_1, s_2, \cdots, s_m$. Let p_{jk} be the transition probability of moving in one step from s_j to s_k and let x_j denote $\Pr\{s_j \overset{*}{\vdash} \langle\text{END}, y\rangle\}$. The probabilities x_j satisfy the system of linear equations

$$x_j = \sum_k p_{jk}x_k + p(\langle\text{END}, y\rangle|s_j) \qquad (j=1, 2, \cdots, m), \tag{6.10}$$

which can be rewritten

$$\sum_k 2(p_{jk}-\delta_{jk})x_k = -2p(\langle\text{END}, y\rangle|s_j) \qquad (j=1, 2, \cdots, m), \tag{6.11}$$

a system with coefficients 0, ±1, ±2. By eliminating states $\langle I, w\rangle$ such that $\Pr\{\langle I, w\rangle \overset{*}{\vdash} \langle\text{END}, y\rangle\}=0$, we have guaranteed that this system has a unique solution.

The system above can now be solved for $x_1, x_2, \cdots, x_m$ by brute force. By Cramer's rule, the solution can be written as $N_1/D, N_2/D, \cdots, N_m/D$, where D is the determinant of the coefficient matrix and each N_j is an integer such that $0<N_j\leqq D$. By expanding the coefficient matrix by minors along rows, we see that $D\leqq 4^m$. Therefore the space required for storing trial solutions is $O(m^2)=O(I(i, x, n)^4)$, and so the solution of the linear system (6.10) can be found in space bounded above by a polynomial of $I(i, x, n)$. Once the system is solved, we have the value of $\Pr\{\langle\text{BEGIN}, \Lambda\rangle \overset{*}{\vdash} \langle\text{END}, y\rangle\}$.

We summarize the proof. For each possible output string y, we compute $\Pr\{M_i(x)=y \text{ in tape } n\}$ by solving a linear system of order $O(I(i, x, n)^2)$. Then $S_i(x)\leqq n$ iff $\Pr\{M_i(x)=y \text{ in tape } n\}>\frac{1}{2}$ for some y of length at most $I(i, x, n)$.

COROLLARY 6.5. *The probabilistic Turing machine M_i can be simulated deterministically in tape $O(I(i, x, S_i(x))^4)$. In particular, if $S_i(x)\geqq \log|x|$ there is a constant c depending on M_i such that M_i can be simulated deterministically in tape c^{S_i}.*

Proof. The output $\phi_i(x)$ is discovered as a by-product of the procedure of Theorem 6.4 for deciding $S_i(x)\leqq n$. A deterministic method for simulating M_i consists of deciding $S_i(x)\leqq n$ for $n=1, 2, \cdots$, until $\phi_i(x)$ is found. □

LEMMA 6.6. *If $e_i(x)$ denotes the error probability of $M_i(x)$, then $e_i(x)\leqq \frac{1}{2}(1-16^{-I(i,x,S_i(x))^2})$. In particular, if $S_i(x)\geqq\log|x|$ then there is a constant c depending on M_i such that $e_i(x)\leqq\frac{1}{2}(1-2^{-c^{S_i(x)}})$.*

Proof. From the proof of Theorem 6.4, we can write $\Pr\{M_i(x)=\phi_i(x)$ in tape $S_i(x)\}$ as N/D for some $D \leqq 4^m$, where $m \leqq 2I(i, x, S_i(x))^2$. Thus $D \leqq 16^{I(i,x,S_i(x))^2}$. Since $N/D > \frac{1}{2}$ and N is an integer, $N/D - \frac{1}{2} \geqq 1/(2D)$. Therefore $\Pr\{M_i(x)=\phi_i(x)$ in tape $S_i(x)\} - \frac{1}{2} \geqq \frac{1}{2} \cdot 16^{-I(i,x,S_i(x))^2}$. □

PROPOSITION 6.7. *For each probabilistic Turing machine M_i with run time T_i and tape S_i there is a constant c such that $T_i(x) \leqq c^{I(i,x,S_i(x))}$. In particular, if $S_i(x) \geqq \log|x|$ then there is a constant c such that $T_i(x) \leqq 2^{cS_i(x)}$.*

Proof. For brevity, we write $I(i, x, S_i(x))$ simply as I. It is easily seen that for every k, the probability that $M_i(x)$ halts after more than kI steps is at most $(1-2^{-I})^k$. If $k \geqq (1+\ln 16I^2)2^I$, then $(1-2^{-I})^k < (\frac{1}{2})16^{-I^2}$. Since $M_i(x)$ halts in more than kI steps with probability less than $(\frac{1}{2})16^{-I^2}$, by Lemma 6.6, $\Pr\{M_i(x)= \phi_i(x)$ in time $kI\} > \frac{1}{2}$. Therefore $T_i(x) \leqq kI < c^I$ for some $c > 2$. □

Neither of the bounds of Lemma 6.6 or Proposition 6.7 can be improved significantly. Corollary 6.5 states that deterministic machines require at most exponentially more tape than probabilistic machines. It is an open question whether this exponential bound can be improved. It is also unknown if every probabilistic machine using tape S can be simulated deterministically in tape bounded by a polynomial of S. The next result provides evidence that probabilistic machines might be more efficient in the use of tape than deterministic machines.

PROPOSITION 6.8. *Every language accepted by a nondeterministic Turing machine in tape $S(x) \geqq \log|x|$ can be recognized by a probabilistic Turing machine with bounded error probability in tape $S(x)$.*

Proof. We consider only the case that S is a constructable tape function [11, p. 149]. The general case requires a minor modification [8].

Suppose that L is a language accepted by some nondeterministic machine M in tape $S(x) \geqq \log|x|$. There is a constant c such that every x in L is accepted by some computation of length less than $c^{S(x)}$. Let $d = c+1$. A probabilistic machine M' that recognizes L operates as follows. With input x:

(i) M' marks off $S(x)$ worktape squares. This requires only tape $S(x)$ because S is constructible.

(ii) M' simulates up to $c^{S(x)}$ steps of a computation of M, choosing the next action by a coin toss when there is a nondeterministic choice. If the computation requires more than $c^{S(x)}$ steps, or attempts to use more than $S(x)$ worktape squares, or halts in the allotted time without accepting, then M' goes to (iii). Otherwise, the simulated computation of M was an accepting computation, and so M' accepts x.

(iii) M' tosses $d^{S(x)}$ coins. If all tosses result in heads, then M' halts and rejects. Otherwise, M' clears its worktapes, rewinds its input tape, and goes back to (ii).

Note that steps (ii) and (iii) can be performed within tape $S(x)$.

Obviously M' rejects inputs not in L with probability 1. If x is in L then any single execution of step (ii) will find an accepting computation with probability at least $2^{-cS(x)}$. Therefore for x in L, we calculate easily that

$$(6.12) \qquad \Pr\{M' \text{ accepts } x\} \geqq 2^{-cS(x)} \sum_{i=0}^{\infty} [(1-2^{-cS(x)})(1-2^{-dS(x)})]^i \geqq \tfrac{2}{3}.$$

Therefore M' accepts L with bounded error probability. □

Acknowledgments. This research was supported by National Science Foundation Grant GK-43121 at Stanford University and was completed at the Forschungsinstitut für Mathematik of the Eidgenössische Technische Hochschule, Zürich. Some of the results appear in the author's doctoral dissertation, supervised by Manuel Blum and supported by National Science Foundation Grant GJ-708 at the University of California, Berkeley. A preliminary version of this paper was presented in [9]. The author thanks the referees for their comments and corrections, and thanks Istvan Simon for helpful discussions.

REFERENCES

[1] M. BLUM, *A machine-independent theory of the complexity of recursive functions*, J. Assoc. Comput. Mach., 14 (1967), pp. 322–336.

[2] H. CHERNOFF, *A measure of asymptotic efficiency for tests based on the sums of observations*, Ann. Math. Statist., 23 (1952), pp. 493–507.

[3] S. A. COOK, *The complexity of theorem-proving procedures*, Proc. 3rd ACM Symp. Theory of Computing, Shaker Heights, OH, 1971, pp. 151–158.

[4] K. DE LEEUW, E. F. MOORE, C. E. SHANNON AND N. SHAPIRO, *Computability by probabilistic machines*, Automata Studies, Annals of Mathematics Studies no. 34, Princeton University Press, Princeton, NJ, 1956, pp. 183–212.

[5] W. FREIBERGER AND U. GRENANDER, *A Short Course in Computational Probability and Statistics*, Applied Mathematical Sciences vol. 6, Springer-Verlag, Berlin, 1971.

[6] R. V. FREIVALD, *Fast computation by probabilistic Turing machines*, Theory of Algorithms and Programs, no. 2, Latvian State University, Riga, 1975, pp. 201–205. (In Russian.)

[7] R. G. GALLAGER, *Information Theory and Reliable Communication*, John Wiley, New York, 1968.

[8] J. T. GILL III, *Probabilistic Turing machines and complexity of computation*, Ph.D. dissertation, Dept. of Mathematics, University of California, Berkeley, 1972.

[9] ———, *Computational complexity of probabilistic Turing machines*, Proc. 6th ACM Symp. Theory of Computing, Seattle, WA, 1974, pp. 91–95.

[10] F. C. HENNIE, *One-tape, off-line Turing machine computations*, Information and Control, 8 (1965), pp. 553–578.

[11] J. E. HOPCROFT AND J. D. ULLMAN, *Formal Languages and their Relation to Automata*, Addison-Wesley, Reading, MA, 1969.

[12] D. J. KLEITMAN, *On a combinatorial conjecture of Erdös*, J. Combinatorial Theory, 1 (1966), pp. 209–214.

[13] A. R. MEYER AND L. J. STOCKMEYER, *The equivalence problem for regular expressions with squaring requires exponential tape*, Proc. 13th IEEE Symp. Switching and Automata Theory, College Park, MD, 1972, pp. 125–129.

[14] Ġ. E. MILLER, *Riemann's hypothesis and tests for primality*, Proc. 7th ACM Symp. Theory of Computing, Albuquerque, NM, 1975, pp. 234–239.

[15] M. MINSKY, *Computation: Finite and Infinite Machines*, Prentice-Hall, Englewood Cliffs, NJ, 1967.

[16] A. PAZ, *Introduction to Probabilistic Automata*, Academic Press, New York, 1971.

[17] W. W. PETERSON AND E. J. WELDON, *Error-Correcting Codes*, second edition, MIT Press, Cambridge, MA, 1972.

[18] M. O. RABIN, *Probabilistic automata*, Information and Control, 6 (1963), pp. 230–245; also in *Sequential Machines*, E. F. Moore, ed., Addison-Wesley, Reading, MA, 1964, pp. 98–114.

[19] ———, *Real-time computation*, Israel J. Math., 1 (1963), pp. 203–211.

[20] ———, *Probabilistic algorithms*, Algorithms and Complexity: New Directions and Recent Results, J. F. Traub, ed., Academic Press, New York, 1976, pp. 21–39.

[21] H. ROGERS, JR., *Gödel numberings of partial recursive functions*, J. Symbolic Logic, 23 (1958), pp. 331–341.

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

[23] E. S. SANTOS, *Probabilistic Turing machines and computability*, Proc. Amer. Math. Soc., 22 (1969), pp. 704–710.

[24] ———, *Computability by probabilistic Turing machines*, Trans. Amer. Math. Soc., 159 (1971), pp. 165–184.

[25] J. SIMON, *On some central problems in computational complexity*, Tech. Rep. TR 75-224, Dept. of Computer Sci., Cornell University, Ithaca, NY, 1975.

[26] R. SOLOVAY AND V. STRASSEN, *A fast Monte-Carlo test for primality*, this Journal, 6 (1977), pp. 84–85.

[27] L. J. STOCKMEYER, *The polynomial-time hierarchy*, Theoretical Computer Science, 3 (1977), pp. 1–22.

[28] B. A. TRAKHTENBROT, *Complexity of algorithms and computation*, Novosibirsk State University, Novosibirsk, 1967. (In Russian.)

[29] ———, *Notes on computational complexity of probabilistic machines*, Theory of Algorithms and Mathematical Logic, Computing Center of the USSR Academy of Sciences, Moscow, 1974, pp. 159–176. (In Russian.)

[30] ———, *On problems solvable by successive trials*, Proc. Symp. Mathematical Foundations of Computer Science, Lecture Notes in Computer Science no. 32, J. Becvar, ed., Springer-Verlag, Berlin, 1975.

[31] A. V. VAISER, *Computational complexity and reliable recognition of languages by probabilistic finite automata*, System Management, no. 1, Tomsk, USSR, 1975, pp. 172–181. (In Russian.)

SIAM J. COMPUT.
Vol. 6, No. 4, December 1977

A NOTE ON SPIRA'S ALGORITHM FOR THE ALL-PAIRS SHORTEST-PATH PROBLEM*

JOHN S. CARSON† AND AVERILL M. LAW†

Abstract. We correct some errors in Spira's algorithm for the all-pairs shortest-path problem, and empirically compare his algorithm (with two distinct sorting routines) to Dijkstra's procedure. The results show that Spira's algorithm is only efficient for "large" networks. Furthermore, it is seen that the asymptotic number of additions and comparisons required by two algorithms may be a very poor indicator of their relative running times.

Key words. shortest path, algorithm, computational efficiency

1. Introduction. In [5], Spira presented a new algorithm for finding the shortest path between each pair of nodes in a directed network with nonnegative arc lengths. Previous algorithms [1], [2], [4], [7] have average running time $O(n^3)$, whereas Spira's procedure runs in average time $O(n^2 \log^2 n)$. This note corrects a few minor errors in Spira's statement of the algorithm, and compares the running time of Spira's algorithm to that for Yen's implementation [7] of Dijkstra's procedure [2].

For the classes of networks tested here, we found that when n (the number of nodes) is less than 100, Dijkstra's method runs faster, but for "large" networks (100 nodes or more) Spira's algorithm is superior, provided it is implemented efficiently. Thus, if one must solve a large number of small problems, it is advantageous to use Dijkstra's algorithm.

In what follows, references to Step 1, Step 2, etc. refer to the steps of the algorithm in [5], which we assume is familiar to the reader.

2. The algorithm. Spira's statement of his procedure contains the following errors:

Step 1. The second sentence should read "Set $I(i, j) \leftarrow l$ where d_{il} is the jth element in the sorted set of arcs beginning at node i for all $1 \leqq i \leqq n$, $1 \leqq j \leqq n-1$." (Note that I is an $n \times (n-1)$ array.)

Step 4. Replace $I(k, \text{COUNT})$ by $I(k, p_k)$. Delete "Set $p_k \leftarrow p_k + 1$."

Step 6. It should read "If $p_t = n$, go to 9." (If $p_t = n$, then we know that the shortest distances (from source node i) to the $n-2$ distinct nodes in the list $I(t, 1), I(t, 2), \cdots, I(t, n-2)$ have already been found and that hence $I(t, p_t - 1)$ is the last node whose shortest distance is to be found.)

Step 7. Replace $I(i, p_t)$ by $I(t, p_t)$.

Step 8. Replace $I(i, p_t - 1)$ by $I(t, p_t - 1)$ throughout.

Step 9. Replace $I(i, p_t - 1)$ by $I(t, p_t - 1)$ throughout.

Step 10. Replace $\leqq$ by $<$.

* Received by the editors March 11, 1976.

† Department of Industrial Engineering, University of Wisconsin—Madison, Madison, Wisconsin 53706.

Spira's procedure is distinguished by the fact that it requires a sorting routine which must be able to add a new element to a set S (Step 4), and replace the minimum of S by a new element (Step 7), in such a way that successive minima of S can be efficiently extracted at Step 5. In considering how best to accomplish these purposes, we tested two routines: first, the played binary tree as described by Spira [5], and second, HEAPSORT [6]. We found that HEAPSORT required less storage space and no use of pointers, was easier to implement, and actually ran faster. (See the empirical results in the next section.)

It should also be mentioned that the initial sorting (Step 1) was accomplished by repeated use of either the played binary tree or HEAPSORT. Furthermore, all counts of operations and running times include this initial sort.

3. Empirical results. We programmed the corrected version of Spira's algorithm and Yen's version [7] of Dijkstra's algorithm in FORTRAN and compared their performance on the UNIVAC 1110. Twenty test problems for a 40-node network, and ten test problems for 60-, 80-, 100-, and 120-node networks were randomly generated with all arcs present and arc lengths uniformly distributed on the integers $1, 2, \cdots, 100$. Table 1 gives the average number of operations performed (additions plus comparisons) and the average running times for these test problems, plus the minimum and maximum running times. Dijkstra's algorithm has no variation in number of operations, and essentially none in running time, whereas Spira's algorithm has considerable variation in both. For the 40-, 60-, and 80-node networks, Dijkstra's algorithm was always faster. For the 100-node network, Spira's algorithm (with HEAPSORT) was faster in five out of ten test problems, and had an average running time which was slightly less than that for Dijkstra's algorithm. For the 120-node network, Spira's algorithm (with HEAPSORT) was faster in eight out of ten test problems, and took only about 5% more time in the other two cases. In all cases, HEAPSORT outperformed the played binary tree.

The last column in Table 1 gives average running time divided by $n^2\log^2 n$. These calculations indicate that our implementations of Spira's algorithm actually run in time $O(n^2\log^2 n)$.

We also ran tests with arc lengths distributed uniform $(0, b)$ for $b = 10, 100, 1000$; normal $(50, \sigma^2)$ for $\sigma^2 = 4, 25, 400, 2500$ (truncated below 0); and exponential with mean 50. In additional runs, we allowed 50% and 90% of the arcs to be missing. In all cases except one, the running times were not significantly different from those in Table 1. The exceptional case occured with arcs normal $(50, \sigma^2)$ for σ^2 small, in which case Spira's algorithm was at least three times faster. This will occur for any distribution concentrated for the most part on the interval $(a, a+c)$ with $a > 0$ and c small relative to a, because most of the shortest paths will contain one (or a small number of) arcs.

Note that the number of operations is a poor indicator of the actual running time of an algorithm. For example, when $n = 120$ Dijkstra's procedure required roughly three times as many additions and comparisons, yet it only took about 10% more running time than Spira's algorithm with HEAPSORT.

In conclusion, for the networks tested here the method of Dijkstra and Yen is easier to implement and actually runs faster when the number of nodes is less than

TABLE 1

Performance of the algorithms for networks with arcs uniformly distributed on 1, 2, · · · , 100 *and all arcs present*

Algorithm	Number of nodes	Average number of operations	Average running time (seconds)	Minimum running time (seconds)	Maximum running time (seconds)	$\frac{\text{Avg. running time}}{n^2 \log^2 n} \times 10^5$
Dijkstra	40	93,600	1.54	1.54	1.55	
	60	318,600	5.09	5.07	5.10	
	80	758,400	11.89	11.89	11.91	
	100	1,485,000	23.45	23.44	23.47	
	120	2,570,400	39.56	39.52	39.58	
Spira with played binary tree	40	63,930	3.12	2.65	4.39	6.88
	60	174,336	8.37	7.14	9.55	6.66
	80	331,449	15.78	13.98	19.01	6.17
	100	550,772	25.97	22.42	28.78	5.88
	120	875,518	41.94	36.06	48.14	6.11
Spira with HEAPSORT	40	66,617	2.82	2.41	3.99	6.22
	60	179,806	7.46	6.17	8.74	5.94
	80	338,306	13.96	12.17	17.19	5.46
	100	555,859	23.36	19.78	25.28	5.29
	120	876,491	35.65	29.96	41.22	5.19

100. On the other hand, for networks of 100 or more nodes, Spira's algorithm coupled with an appropriate sort routine may be worth the extra effort needed to program it efficiently.

REFERENCES

[1] G. B. DANTZIG, *All shortest routes in a graph*, Operations Research House, Stanford University, Tech. Rep. 66-3, Stanford, CA, 1966.

[2] E. W. DIJKSTRA, *A note on two problems in connection with graphs*, Numer. Math., 1 (1959), pp. 269–271.

[3] S. E. DREYFUS, *An appraisal of some shortest path algorithms*, Operations Res., 17 (1969), pp. 395–411.

[4] R. W. FLOYD, *Algorithm* 97: *Shortest path*, Comm. ACM, 5 (1962), p. 345.

[5] P. M. SPIRA, *A new algorithm for finding all shortest paths in a graph of positive arcs in average time* $O(n^2 \log^2 n)$, this Journal, 2 (1973), pp. 28–32.

[6] J. W. J. WILLIAMS, *HEAPSORT*, Comm. ACM, 7 (1964), p. 347.

[7] J. Y. YEN, *Finding the lengths of all shortest paths in N-node nonnegative-distance complete networks using $\frac{1}{2}N^3$ additions and N^3 comparisons*, J. Assoc. Comput. Mach., 19 (1972), pp. 423–424.

SIAM J. COMPUT.
Vol. 6, No. 4, December 1977

A UNIFIED TREATMENT OF DISCRETE FAST UNITARY TRANSFORMS*

BERNARD J. FINO† AND V. RALPH ALGAZI‡

Abstract. A set of recursive rules which generate unitary transforms with a fast algorithm (FUT) are presented. For each rule, simple relations give the number of elementary operations required by the fast algorithm. The common Fourier, Walsh–Hadamard (W–H), Haar, and Slant transforms are expressed with these rules. The framework developed allows the introduction of generalized transforms which include all common transforms in a large class of "identical computation transforms". A systematic and unified view is provided for unitary transforms which have appeared in the literature. This approach leads to a number of new transforms of potential interest. Generalization to complex and multidimensional unitary transforms is considered and some structural relations between transforms are established.

Key words. discrete transforms, fast algorithms, fast Fourier transforms, fast generalized transforms, generalized Kronecker product, Haar transform, identical computation transforms, slant transform, unitary transforms and matrices, Walsh–Hadamard transform

Introduction. The dissemination of the fast Fourier transform algorithms, originally introduced by Good [1], and known as Cooley–Tukey [2] and Sande–Tukey [3] algorithms, has resulted in a large extension in the range of applications of the well known Fourier transform. Recently the Walsh–Hadamard transform, also with a fast algorithm [4], has drawn considerable interest [5]. The Haar transform, although closely related to the Walsh–Hadamard transform [6] and potentially of interest [7], has received much less attention. These transforms have been used successfully for signal filtering [8], pattern classification [4], [9], speech signal encoding [10] and above all for picture encoding [11], [12], [13]. An overview of transforms can be found in [37] and [38]. Only a few transforms have been considered in these applications while many other transforms could be of interest. Some workers have considered the definition of generalized transforms and we mention the works by Andrews, et al. [14], [15], [16], Rao, et al. [17], [18] and Harmuth [19, pp. 30–36].

In this paper we present a unified view of discrete unitary transforms with a fast algorithm. A *discrete unitary transform* is characterized by a unitary matrix $[T]$ such that $[T][T^{*t}]=[I]$ where * denotes a conjugate, "t" denotes transpose and $[I]$ is the identity matrix of same order as $[T]$, say N. In mathematics a unitary matrix expresses a rotation of the orthonormal basis and preserves the Euclidean norm, $\|V\|=\mathbf{V}\cdot\mathbf{V}^{*t}$, of any vector $\mathbf{V}$ and all inner products of vectors. In signal representation, this property means energy conservation and an easy expression of the mean square error when some components of the signal are ignored in the new base. The computation of the transformed vector $\mathbf{W}$ of $\mathbf{V}$ by the transform $[T]$ such that $\mathbf{W}=[T]\mathbf{V}$ usually requires N^2 multiplications and $N(N-1)$ additions.

* Received by the editors April 19, 1974, and in revised form June 28, 1976. This research was sponsored in part by the National Aeronautic and Space Administration under Grant NASA-NGL-05-003-404, and in part by the National Science Foundation under Grant NSF-ENG-75-10063. The work reported is part of the doctoral research of Dr. Fino at the University of California, Berkeley, Department of Electrical Engineering and Computer Science.

† Systems Applications, Inc., San Rafael, California.

‡ Department of Electrical Engineering, University of California, Davis, California 95616.

For some specific transforms of interest such as the Fourier, Walsh–Hadamard transforms, a *fast algorithm* has been found which requires fewer elementary operations. The analysis of these fast algorithms has been done by factorization of the matrix $[T]$ into a set of largely sparse matrices, each expressing a stage of computation. This is the approach followed by Good [1] in his original paper which leads to the fast Fourier transform [2], [3] the fast Walsh transform [4] and other known fast transforms.

Such an approach is analytic and determines decomposition matrices for a given fast unitary transform.

Our approach is synthetic and is based on two observations:

a) A few types of fast unitary matrices of small order generate recursively fast unitary transforms of arbitrary order.

b) The same recursive relations between fast unitary transforms lead to simple recursive relations for the number of elementary operations needed in FUT of different order. The first observation has been used by several of the authors cited; we shall exploit it systematically. The second observation has not been exploited in the mathematical or technical literature. By using three recursive rules,[1] we shall discuss systematically the generation of FUT and the number of elementary operations needed for the fast algorithm. Using this framework we shall define a large family of FUT and derive simply a number of old and new results about the FFT algorithms, other known transforms and establish structural properties between transforms.

1. Recursive generative rules. We shall present three basic and elementary rules which generate a new unitary matrix from some original unitary matrices. For each rule we relate the number of elementary operations for the new transform to the number of elementary operations of the same type required by the original transforms. For Rule 1 there is only one original matrix, for Rule 2 two, and for Rule 3 a set of original matrices. These rules will then be used in a constructive and systematic fashion to generate FUT's.

Rule 1. *Operations on the columns of a unitary matrix.* Given a unitary matrix $[T]$, two obvious operations on the columns yield another unitary matrix of some order:

a) *Permutation of the columns*: This operation does not require any computation. In the computational process, this operation can be performed by applying the permutation to the coefficients of the input vector instead of the columns themselves.

b) *Multiplication of a column by a root of unity*: This operation requires a complex multiplication if the root of unity is not ± 1 or $\pm j$ $(j=\sqrt{-1})$.[2]

These operations on the columns may be expressed by a matrix product $[T]$ $[D]$ with $[D]$ such that $D_{ki}=e^{j\theta_i}$, if column k is to be replaced by column i multiplied by the root of unity, $e^{j\theta_i}$, and all other entries of $[D]$ are null.

[1] We denote by "rule" a set of operations performed in a prescribed order. We reserve the term "operation" for the elementary operations such as additions, multiplications, etc., which determine the computational complexity of a transform.

[2] Multiplications by ± 1 and $\pm j$ may be counted as operations if the hardware realization of the algorithm is not able to keep track of them. However, in an error analysis of the algorithm these multiplications, even if they are performed, do not introduce any error.

Rule 2. *Rotation of rows by a unitary matrix.* Consider a unitary matrix $[T]$ of order N. The N row vectors form an orthonormal basis for S_n, the N dimensional space they span. m row vectors of $[T]$ form an orthonormal basis for a subspace S_m. If these m vectors are rotated by a unitary matrix $[U]$ of order m, we obtain a new orthonormal basis $\mathscr{B}$ for S_m. The remaining unchanged $N-m$ rows of $[T]$ are an orthonormal basis of the subspace S_{N-m} orthogonal to S_m and form with $\mathscr{B}$ a new orthonormal basis for S_N. Thus, the matrix $[T']$ obtained after rotation of the m rows by the unitary matrix $[U]$ is unitary.

Some particular cases of interest are:

a) multiplication of the whole matrix by a unitary matrix of the same order

b) permutation of the rows (multiplication by a permutation matrix)

c) multiplication of a row by any root of unity.

Operations b) and c) can be represented by the matrix product $[D][T]$ where $[D]$ is, as before, such that $D_{ik} = e^{j\theta_i}$, if row i of $[T]$ is replaced by row k multiplied by the root of unity, $e^{j\theta_i}$, and all other entries of $[D]$ are null.

Number of elementary operations. If transforms T and U require respectively t and u elementary operations of a specific kind, it is obvious that the transform T' will require at most t' of these operations with

$$t' = t + u. \tag{1}$$

(It may happen that $[T']$ so generated has a simpler algorithm.)

Equation (1) applies independently to any type of elementary operation: additions, real and complex multiplications as well as any other specific operation (e.g. shift, multiplication by $\sqrt{2}$, etc.)

Rule 3. *Generalized Kronecker product.* The two previous rules are quite obvious, but used in combination they result in a powerful tool for the generation of FUT. The matrix Kronecker product, described for instance in [36], and the generalized Kronecker product presented here are both simple and useful composite rules.

These Kronecker products take full advantage of the decomposition of FUT's into *block form* matrices, which reduces the computation and inversion of FUT to processing smaller matrices. In other terminology, Kronecker products imply *separability* and computational separability savings.

Given two sets of unitary matrices, set $\{\mathscr{A}\}$ of m matrices (A^i) $(i = 0, \cdots, m-1)$,[3] all of order n, and set $\{\mathscr{B}\}$ of n matrices (B^i) $(i = 0, \cdots, n-1)$, all of order m, we define the generalized Kronecker product of the sets $\{\mathscr{A}\}$ and $\{\mathscr{B}\}$, denoted $\{\mathscr{A}\}\otimes\{\mathscr{B}\}$ to be the square matrix $[C]$ of order mn such that

$$C_{i,j} = C_{um+w, u'm+w'} = A^{w}_{u,u'} \cdot B^{u'}_{w,w'}, \tag{2}$$

with

$$i = um + w, \qquad u, u' = 0, \cdots, n-1,$$
$$j = u'm + w', \qquad w, w' = 0, \cdots, m-1.$$

In the particular case in which the matrices $[A^i] = [A]$ are all identical, and also the matrices $[B^i] = [B]$, the generalized Kronecker product $\{\mathscr{A}\}\otimes\{\mathscr{B}\}$ reduces to the usual Kronecker product of matrices [14]: $[A]\otimes[B]$.

[3] The index ranges from 0 to $m-1$ to ease further notation and also to conform to common usage.

It is easy to show that $[C]$ is unitary and can be factorized into

$$[C]=[P^t][\mathrm{Diag}\,\{\mathcal{A}\}][P][\mathrm{Diag}\,\{\mathcal{B}\}] \tag{3}$$

where $[\mathrm{Diag}\,\{\mathcal{A}\}]$ and $[\mathrm{Diag}\,\{\mathcal{B}\}]$ are block diagonal matrices formed with the matrices of the sets $\{\mathcal{A}\}$ and $\{\mathcal{B}\}$:

$$[\mathrm{Diag}\,\{\mathcal{A}\}]=\begin{bmatrix}[A^0] & & & \\ & [A^1] & & \\ & & \ddots & \\ & & & [A^{m-1}]\end{bmatrix}$$

and $[P]$ is the perfect shuffle permutation matrix[4] of order mn such that $P_{k,l} = \delta_{v,z'}\delta_{z,v'}$ when $k = vn + z$, $l = v'm + z'$ and z', $v = 0, \cdots, m - 1$;

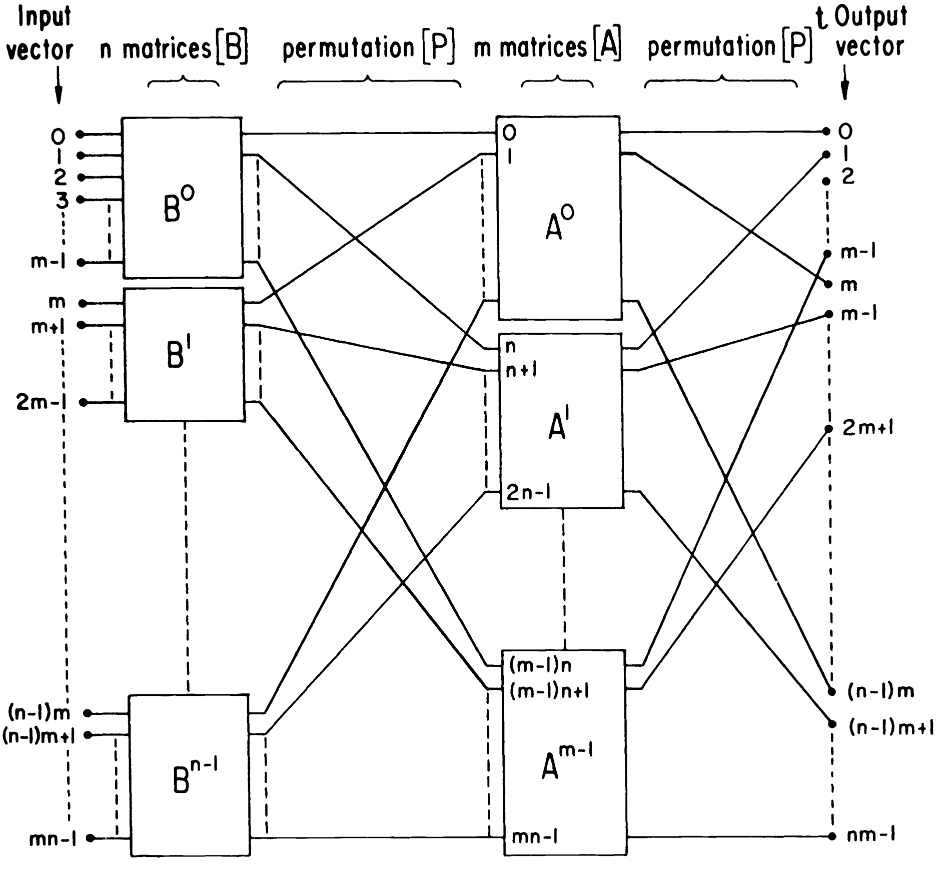


FIG. 1. *Generalized Kronecker product: algorithm.*

[4] The perfect shuffle is defined for example in [21]. $[P_{2,m}]$ corresponds to the usual "shuffle" of two packs of m cards $(a, b, c, \cdots, m)$ and $(a', b', c', \cdots, m)$ into $(a, a', b, b', c, c', \cdots, m, m')$. The perfect shuffle is illustrated in Fig. 1 and corresponds to the symmetric distribution of n packs of m cards into m packs of n cards.

z, $v'=0, \cdots, n-1$, and δ denotes the Kronecker delta. Equation (3) is a generalization of the factorization of a simple Kronecker product into Good matrices [14].

Fast algorithm and number of elementary operations. With the computational blocks corresponding to the transforms $A^0, \cdots, A^{m-1}$ and $B^0, \cdots, B^{n-1}$, the factorization of (3) leads directly to the computational block of the transform C given in Fig. 1.

From the structure of the algorithm of Fig. 1 it is easy to see that if the matrices $[A^i]$ $(i=0, \cdots, m-1)$ and $[B^j]$ $(j=0, \cdots, n-1)$ have algorithms requiring respectively p_n^i and q_m^j elementary operations of a specific type, their generalized Kronecker product $[C]$ will require P_{mn} of these operations with

$$P_{mn} = \sum_{i=0}^{m-1} p_n^i + \sum_{j=0}^{n-1} q_m^j. \tag{4}$$

In the particular case of a simple Kronecker product $p_n^i = p_n$ and $q_m^j = q_m$ so

$$P_{mn} = mp_n + nq_m. \tag{5}$$

Note that the use of Rules 1 and 2 only increases the number of elementary operations while the order of the generated transform does not change. For Rule 3, even if $[A^i]$ and $[B^j]$ do not have fast algorithms and thus require n^2 and m^2 elementary operations, $[C]$ requires a maximum of $(m+n)mn \leqq (mn)^2$ (for $m, n > 1$) elementary operations.

The results of equations (1), (4) and (5) are important: for every transform generated with the recursive rules presented, they give a simple and systematic way to estimate the number of elementary operations.

2. Identical computation (IC) family. The generative rules defined above create a unified framework for the known FUT, introduce new transforms, and allow an easy evaluation of the number of elementary operations required. Additional structure can be introduced which still allows for the generation of all known FUT. One large family of transforms considered now are "identical computation transforms": This family not only has a greatly reduced number of generating matrices, but provides also a uniform treatment of the input vector. It is our belief that most if not all transforms used practically now and probably in the future belong to this family.

Denote by $\{\mathscr{A}\} \otimes [B_q]$ the generalized Kronecker product of a set $\{\mathscr{A}\}$ of q matrices $[A_p^k]$ $(k=0, \cdots, q-1)$, of order p and a set $\{\mathscr{B}\}$ of p identical matrices $[B_q]$ of order q. $[B_q]$ will be called a *core* matrix and $[A_p^k]$ a *parent* matrix. The IC transforms are recursively generated from a unique class, $\mathscr{C}$, of parent matrices of some order f and an original core matrix $[\mathscr{O}]$ of order q. An IC transform of order (qf^n) is then obtained from the original core matrix $[\mathscr{O}]$ by the recursive formulas:

$$\begin{aligned} &[IC_{qf}] = [D_{qf}][\{\mathscr{A}\} \otimes [\mathscr{O}]][D'_{qf}], \\ &[IC_{qf^n}] = [D_{qf^n}][\{\mathscr{A}_n\} \otimes [\mathrm{IC}_{qf^{n-1}}]][D'_{qf^n}] \end{aligned} \tag{6}$$

where the matrices $[D]$ and $[D']$ express respectively a reordering followed by multiplications by roots of unity of the rows and the columns. All parent matrices of $\{\mathscr{A}_i\} \cdots \{\mathscr{A}_n\}$ belong to $\mathscr{C}$.

The common characteristic of all the transforms of the IC family is that their algorithms only use in any computation intermediate results obtained from the input vector through identical computations (so the name of the family). This property provides a uniform treatment of successive components of the input vector if we consider that any parent matrix treats uniformly its input vector. For this family, all the normalizations can be delayed to the last stage of computation.

We consider different choices for the original matrix $[\mathcal{O}]$, the class $\mathcal{C}$ of parent matrices, the matrices $[D]$ and $[D']$ and the sets $\{\mathcal{A}_k\}$, and show that the basic transforms, Fourier, Walsh–Hadamard and Haar, are IC transforms.

3. Basic transforms: Fourier, Walsh–Hadamard, Haar, Slant. With the help of the generative rules, we now examine the well known Fourier, W–H, and Haar transforms. Our approach allows the derivation of some new results concerning the number of multiplications required by a FFT of composite order, a concise presentation of the different definitions of the W–H transform, and simple definitions of the Haar transform. In addition it makes apparent the common structure of these transforms. This will lead in the next section to the definition of families of transforms between the basic transforms.

In the following we emphasize specific orderings for the basic transforms: frequencies for the Fourier transform, zequencies[5] for the W–H transform and ranks[6] for the Haar transform. These orderings have proved to be useful in applications because they usually concentrate the signal energy into the first transform coefficients [22].

3.1. Generalized fast Fourier transform of composite order.

a) DECOMPOSITION THEOREM. *Given the Fourier matrices $[F_p]$ and $[F_q]$ of orders p and q respectively, the matrix $[F_{pq}]$ such that*

$$[F_{pq}]=[\{[F_q^k]\}\otimes[F_p]][P]^t \tag{7}$$

is the Fourier matrix of order pq. The set $\{[F_q^k]\}$, $k=0,\cdots,p-1$, is such that

$$[F_q^k]=[F_q][D_k] \tag{8}$$

where $[D_k]$ is a diagonal matrix with $(D_k)_{u',u'}=\exp[-2\pi jku'/(pq)]$. $[P]^t$ is the perfect shuffle

$$P_{s,t}=\delta_{u,z}\delta_{k,w}$$

$$\textit{with } s=uq+k,\quad t=wp+z,\quad u,z<p,\quad k,w<q.$$

[5] This terminology has been introduced by Yuen [20]. The *zequency* is the number of zero crossings.

[6] We use rank to indicate the usual order of the rows of a Haar matrix; see Fig. 3. This is, of course, a different notion than the rank of a matrix.

Proof. We denote $[\{[F_q^k]\} \otimes [F_p]]$ by $[F'_{pq}]$.

$$(F'_{pq})_{ug+k,u'g+k'} = (F_p^k)_{u,u'} \cdot (F_q)_{k,k'}$$

$$= (F_p)_{u,u'} \cdot e^{-2\pi jku'/(pq)} \cdot (F_q)_{k,k'} = \frac{1}{\sqrt{2^{pq}}} e^{-2j(uu'/p+ku'/(pq)+kk'/q)},$$

$$[F_{pq}] = [F'_{pq}] \cdot [P]^t \Rightarrow (F_{pq})_{uq+k,wp+z} = (F'_{pq})_{uq+k,u'q+k'} \cdot \delta_{z,u'}\delta_{w,k'}$$

$$= e^{-2\pi j(uz/p+kz/(pq)+kw/q)}$$

$$= e^{-2\pi j(uq+k)(wp+z)/(pq)} \quad \text{Q.E.D.}$$

$[F_p][F_q]$ and $[F_{pq}]$ are symmetric, and it is possible to derive, from (3) and (7), new expressions for $[F_{pq}]$:

$$[F_{pq}] = [P][\{[F_q^k]\} \otimes [F_p]],$$

$$(9) \qquad [F_{pq}] = [[F_p] \otimes \{[F_q'^k]\}][P],$$

$$[F_{pq}] = [P^t][[F_p] \otimes \{[F_q'^k]\}] \quad \text{with } [F_2'^k] = [D_k][F_q].$$

If $[F_p]$ and $[F_q]$ require respectively $\mathscr{A}_p$ and $\mathscr{A}_q$ complex additions, $\mathscr{M}_p$ and $\mathscr{M}_q$ complex multiplications, $[F_{pq}]$ will require by application of (4):

$$(10) \qquad \mathscr{A}_{pq} = p\mathscr{A}_q + q\mathscr{A}_p \quad \text{complex additions,}$$

$$(11) \qquad \mathscr{M}_{pq} = p\mathscr{M}_q + q\mathscr{M}_p + C_{p,q} \quad \text{complex multiplications}$$

where $C_{p,q}$ is the number of complex multiplications introduced by (8):

$$C_{p,q} = \begin{cases} pq & \text{if all the factors including } \pm 1, \pm j \text{ are considered;} \\ (p-1)(q-1) & \text{if the factors } \pm 1 \text{ are discarded;} \\ (p-1)(q-1)-1 & \text{if the factors } \pm j \text{ are} \\ & \text{also discarded when } pq \text{ is a power of 2.} \end{cases}$$

b) *Generalized FFT of composite order.* If the order of the Fourier transform is composite, i.e. $N = \rho_1 \cdots \rho_n$, the previous decomposition theorem yields the well known FFT algorithms [2] [3] detailed by Glassman [23] in the most general case. Our approach is similar but more systematic and with more concise notation than those of Kahaner [24] and Drubin [25]. The recursive use of the formulae (10) and (11) gives the number of required operations. In the case of $N = r^n$ we can solve these recursive equations: This is the case of FFT of radix r.

$$(12) \qquad \mathscr{A}_{r^n} = r^{n-1}\mathscr{A}_r + r\mathscr{A}_{r^{n-1}} \quad \text{or} \quad \mathscr{A}_{r^n} = nr^{n-1}\mathscr{A}_r$$

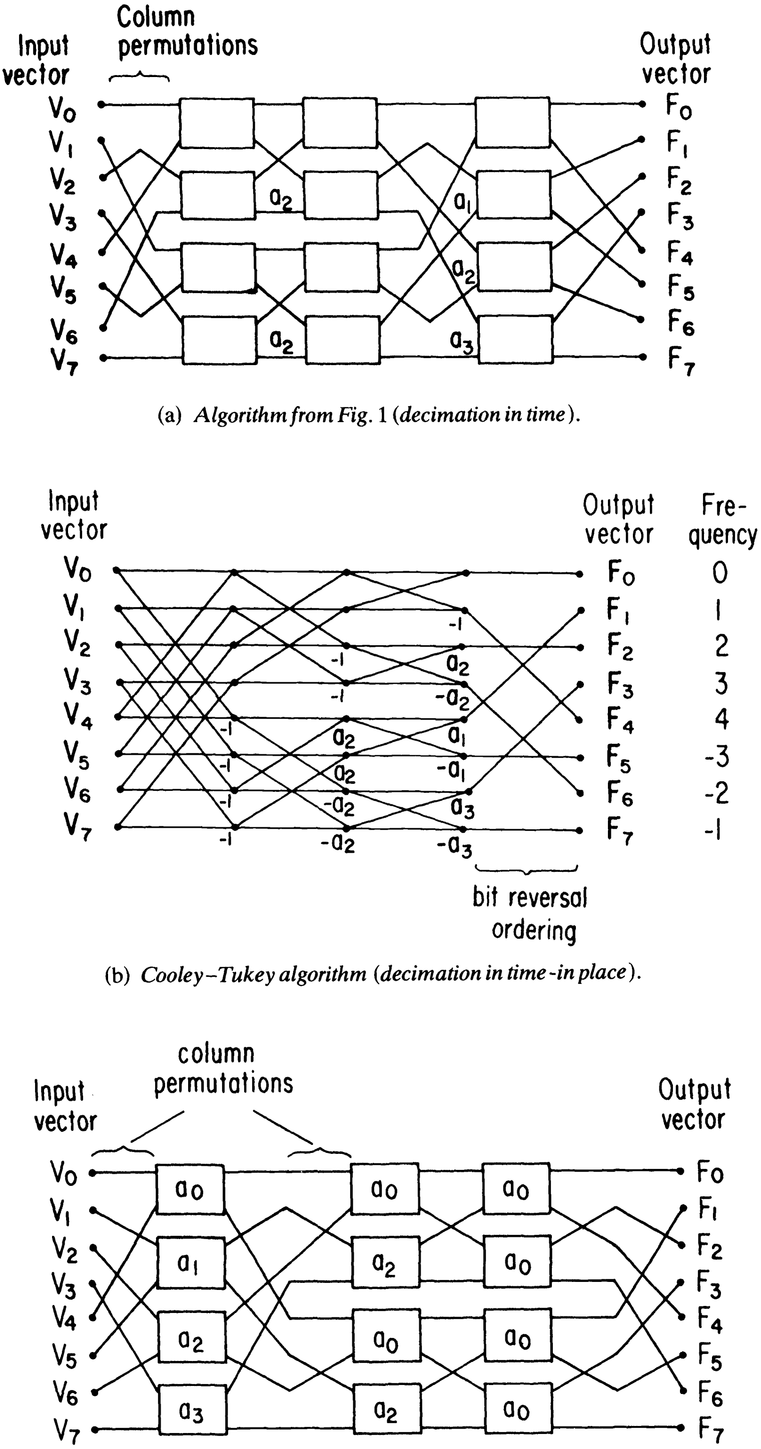
$$\mathscr{M}_{r^n} = r^{n-1}\mathscr{M}_r + r\mathscr{M}_{r^{n-1}} + (r-\alpha)(r^{n-1}-\alpha) - \beta$$

or

$$(13) \qquad \mathscr{M}_{r^n} = nr^{n-1}\mathscr{M}_r + (r-\alpha)\left[(n-1)r^{n-1} - \alpha\frac{(r^{n-1}-1)}{r-1}\right] - \beta\frac{(r^{n-1}-1)}{r-1}$$

(α, β depend on the value of $C_{p,q}$).

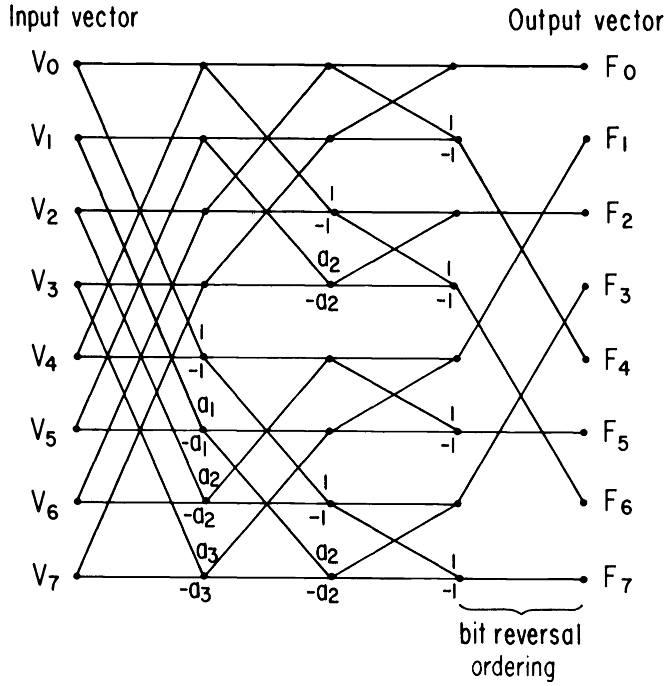
The radices 2, 4, 8 and 16 have been considered in the literature.



(a) *Algorithm from Fig.* 1 (*decimation in time*).

(b) *Cooley–Tukey algorithm* (*decimation in time-in place*).

(c) *Algorithm from Fig.* 1 (*decimation in frequency*).



(d) *Sande–Tukey algorithm (decimation in frequency-in place).*

FIG. 2. *Fast Fourier transform—radix* 2, *order* 8.

For the radix 2, which gives the most popular FFT, the recursive relations given by the decomposition theorem are

$$(14)\qquad [F_{2^n}]=[\{[F_2^k]\}\otimes[F_{2^{n-1}}]][P]^t=[P][\{[F_2^k]\}\otimes[F_{2^{n-1}}]]$$

and

$$(15)\qquad [F_{2^n}]=[[F_{2^{n-1}}]\otimes\{[F_2'^k]\}][P]=[P]^t[[F_{2^{n-1}}]\otimes\{[F_2'^k]\}]$$

with

$$[F_2^k]=\frac{1}{\sqrt{2}}\begin{bmatrix}1 & e^{-2\pi jk/2^n}\\ 1 & -e^{-2\pi jk/2^n}\end{bmatrix}\quad\text{and}\quad [F_2'^k]=\frac{2}{\sqrt{2}}\begin{bmatrix}1 & 1\\ e^{-2\pi jk/2^n} & -e^{-2\pi jk/2^n}\end{bmatrix}$$

The algorithm corresponding to the recursive formula (14) and obtained by recursive use of Fig. 1 is shown in Fig. 2(a); it can be arranged equivalently with all operations "in place" as shown in Fig. 2(b), which is the classical diagram of the Cooley–Tukey [2] algorithm with decimation in time.

The algorithm corresponding to the formula (15) is the Sande–Tukey [3] algorithm with decimation in frequency and is shown in Figs. 2(c) and 2(d).

For these figures the factors are

$$a_0=1,\qquad a_1=\exp(-2\pi j/8),$$
$$a_2=\exp(-4\pi j/8)=-j,\qquad a_3=\exp(-6\pi j/8).$$

We can compare the FFT with radices 2, 4, 8 and 16 for transforms of order $N=2^n=r^{n/\log_2 r}$ (n is then a multiple of 12). The formulas (12) and (13) then give Table 1.

TABLE 1

Radix	$\mathcal{A}_r$	$\mathcal{M}_r$	$\mathcal{A}_{r^n}$	$\mathcal{M}_{r^n}^1$ (all factors)	$\mathcal{M}_{r^n}^2$ (no factors ±1)	$\mathcal{M}_{r^n}^3$ (no factors ±1 ±j)
2	2	0	$n2^n$	$(n-1)2^n$	$n2^{n-1}-2^n+1$	$n2^{n-1}-3\cdot 2^{n-1}+2$
4	8	0		$\left(\frac{n}{2}-1\right)2^n$	$3n2^{n-3}-2^n+1$	$3n2^{n-3}-\frac{13\cdot 2^{n-2}-4}{3}$
8	24	1		$\left(\frac{3n}{8}-1\right)2^n$	$\frac{n2^n}{3}-2^n+1$	$\frac{n2^n}{3}-\frac{57\cdot 2^{n-3}-8}{7}$
16	64	6		$\left(\frac{11n}{32}-1\right)2^n$	$\frac{21n2^n}{64}-2^n+1$	$\frac{21n2^n}{64}-\frac{241\cdot 2^{n-4}-16}{15}$

The column $\mathcal{M}_{r^n}^2$ has been given by Singleton [26].[7] In fact our approach allows the evaluation of the number of elementary operations for any composite order, in particular for mixed radix FFT.

The factors ±1 are easy to track in the algorithms and for most realizations multiplications by ±1 are not performed. The factors $\pm j$ appear in various places in the algorithms and in most realizations multiplications by $\pm j$ are performed; however, in an error analysis these multiplications do not introduce any rounding error and the column $\mathcal{M}_{r^n}^3$ is then of interest. A refinement of little practical interest may be trivially introduced in the count of elementary operations by keeping track of the $\frac{1}{8}$ coefficients $(1+j)/\sqrt{2}$.

3.2. Walsh–Hadamard transform. Recursive relations for the Walsh–Hadamard matrices in "natural" ordering have been known for a long time but the more useful orderings in zequency and Paley's orderings (see [20] for a discussion of these orderings) have not been defined recursively. In a separate note [27] we present recursive relations for the W–H matrices in Paley's and zequency orderings which clarify the reordering procedures and also the various fast W–H algorithms. For completeness, we recall the basic recursive definitions for the three common orderings.

(16) Natural ordering: $(WH_{2^n}\text{nat}]=[WH_2]\otimes[WH_{2^{n-1}}\text{nat}]$,

(17) Paley's ordering: $[WH_{2^n}\text{pal}]=[WH_2]\otimes[WH_{2^{n-1}}\text{pal}][P]^t$,

(18) Zequency ordering: $[WH_{2^n}\text{zeq}]=[\text{Diag}[R]][P][[WH_2]\otimes[WH_{2^{n-1}}\text{zeq}]]$

[7] An earlier publication by G. D. Bergland, *A fast Fourier transform using base 8 iterations*, Math. Comput., 22 (1968), pp. 275–279, presents somewhat differently these results.

with

$$[R]=\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{bmatrix}.$$

3.3. Haar transform. The Haar transform is usually defined from the Haar functions [11]. The Haar matrix of order 8 $[H_8]$ ordered by ranks is as follows:

$$[H_8]=\frac{1}{\sqrt{8}}\begin{array}{cc} & \text{Zones}\\ \begin{bmatrix}1&1&1&1&1&1&1&1\\1&1&1&1&-1&-1&-1&-1\\\sqrt{2}&\sqrt{2}&-\sqrt{2}&-\sqrt{2}&0&0&0&0\\0&0&0&0&\sqrt{2}&\sqrt{2}&-\sqrt{2}&-\sqrt{2}\\2&-2&0&0&0&0&0&0\\0&0&2&-2&0&0&0&0\\0&0&0&0&2&-2&0&0\\0&0&0&0&0&0&2&-2\end{bmatrix} & \begin{array}{l}0\\1\\\left.\begin{array}{l}\\ \\\end{array}\right\}2\\\left.\begin{array}{l}\\ \\ \\ \\\end{array}\right\}3\end{array}\end{array}$$

Here we use the generative rules to define recursively the Haar matrices, and we have found two definitions:

1) The Haar matrix of order 2^n is obtained from the Haar matrix of order 2^{n-1} by simple Kronecker product with $[I_2]=\begin{bmatrix}1&0\\0&1\end{bmatrix}$ followed by rotation of the rows 0 and 2^{n-1} by $[I_2]$. This is the process $\mathscr{H}$ of [6], in terms of generative rules.

2) The Haar matrices are recursively defined by the relation:

$$[H_{2^n}\text{ nat}]=\{[F_2],[I_2],\cdots,[I_2]\}\otimes[H_{2^{n-1}}\text{ nat}] \tag{19}$$

The rows are obtained in "natural" order. To reorder them by their ranks, we need a "zonal bit reversal" ordering. A zone as defined in [6] is a set of coefficients with indexes between two successive powers of 2. A "zonal bit reversal" ordering is a bit reversal followed by a reordering in the original order inside each zone. For 8 coefficients the zonal bit reversal ordering gives:

Index	Binary representation	Bit reversal	Reordering inside zones	Final order (see Fig. 3(a))
0	000	000	000	0
1	001	100	100	4
2	010	010	010	2
3	011	110	110	3
4	100	001	001	1
5	101	101	011	6
6	110	011	101	5
7	111	111	111	7

With both definitions we obtain by recursive application of the diagram of Fig. 1 the algorithm of Fig. 3(a). This algorithm can be more conveniently organized as shown in Fig. 3(b) and give the rows directly ordered by their rank.

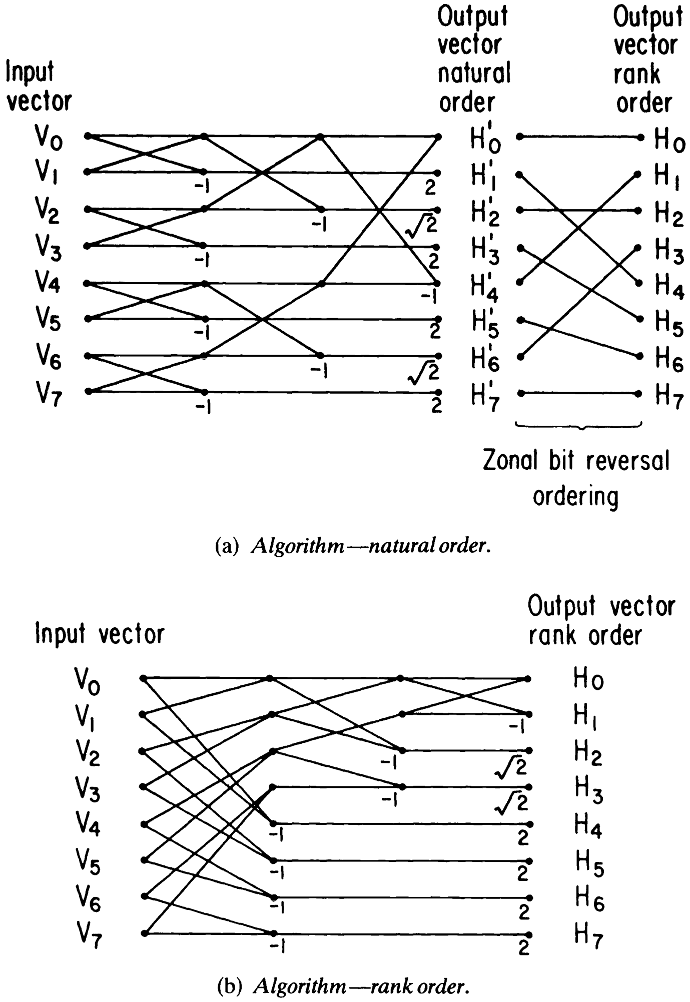
By application of (4) we obtain the following recursive formula for the number of additions:

$$\mathscr{A}_{2^n} = 2 \cdot \mathscr{A}_{2^{n-1}} + 2.$$

Hence

$$\mathscr{A}_{2^n} = 2(2^n - 1) \quad \text{with} \, \mathscr{A}_2 = 2.$$

2^{n-1} normalizations are also required. A modified Haar transform [18] obtained from the Haar transform by permutation of its columns is related to the Fourier



(a) *Algorithm—natural order.*

(b) *Algorithm—rank order.*

FIG. 3. *Haar transform (order 8).*

transform (see § 4.3): it can be defined recursively by:

$$[MH_{2^n}]=[Z][\{F_2\},[I_2],\cdots,[I_2]\}\otimes[MH_{2^{n-1}}]][P]^t \tag{20}$$

Globally the permutations $[Z]$ perform a bit reversal ordering inside each zone.

3.4. Slant transform. The Slant transform has been proposed by Enomoto et al. [28] for the order 8. Pratt et al. [29] have generalized this transform to any order 2^n and compared its performance with other transforms [30]. In this section we express the recursive generation of the Slant transform with our generative rules and compute the number of elementary operations required by its fast algorithm.

The Slant transforms of order 8, $[S_8]$, is as follows (in "natural" order).

$$[S_8]=\frac{1}{\sqrt{8}}\left[\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1\\
1 & -3 & 3 & -1 & 1 & -3 & 3 & -1\times 1/\sqrt{5}\\
7 & -1- & 9 & -17 & 17 & 9 & 1 & -7\times 1/\sqrt{5\times 21}\\
1 & -1- & 1 & 1 & 1 & -1 & -1 & 1\\
7 & 5 & 3 & 1 & -1 & -3 & -5 & -7\times 1/\sqrt{21}\\
1 & -3 & 3 & -1 & -1 & 3 & -3 & 1\times 1/\sqrt{5}\\
3 & 1- & 1 & -3 & -3 & -1 & 1 & 3\times 1/\sqrt{5}\\
1 & -1- & 1 & 1 & -1 & 1 & 1 & -1
\end{array}\right]
\begin{array}{c}
\text{Zequencies}\\
0\\ 7\\ 3\\ 4\\ 1\\ 6\\ 2\\ 5
\end{array}$$

The rows can be reordered by zequencies with the same permutation as the W–H transform in natural order.

The Slant transform of order 2^n in natural order is obtained from the Slant transform of order 2^{n-1} in natural order by simple Kronecker product with $[F_2]$ followed by rotation of the rows 2^{n-2} and 2^{n-1} (Rule 2) by the matrix

$$\begin{bmatrix}\sin\theta_n & \cos\theta_n\\ \cos\theta_n & -\sin\theta_n\end{bmatrix}\quad \text{with}\sin\theta_n=\sqrt{\frac{2^{2n-2}-1}{2^{2n}-1}},\quad 0<\theta<\frac{\pi}{2}.$$

This choice of θ_n introduces in the Slant matrix $[S_{2^n}]$ the Slant vector $\mathbf{S}$ with components linearly decreasing.

But some normalizations can be delayed to the last stage of computation and the rows 2^{n-2} and 2^{n-1} are rotated by the matrix

$$\begin{bmatrix}2^{n-1} & -\dfrac{(2^{2n-2}-1)}{3}\\ 1 & 2^{n-1}\end{bmatrix}$$

requiring 2 shifts, 2 additions, 1 multiplication. The corresponding algorithm is shown in Fig. 4.

Number of elementary operations. By making use of relations (1) and (5), we determine that the slant transform algorithm requires $(n+1)2^n-2$ additions, 2^n-2 shifts, $2^{n-2}-1$ multiplications and finally $2^n-2^{n-2}-1$ normalizations at the last stage of computation.

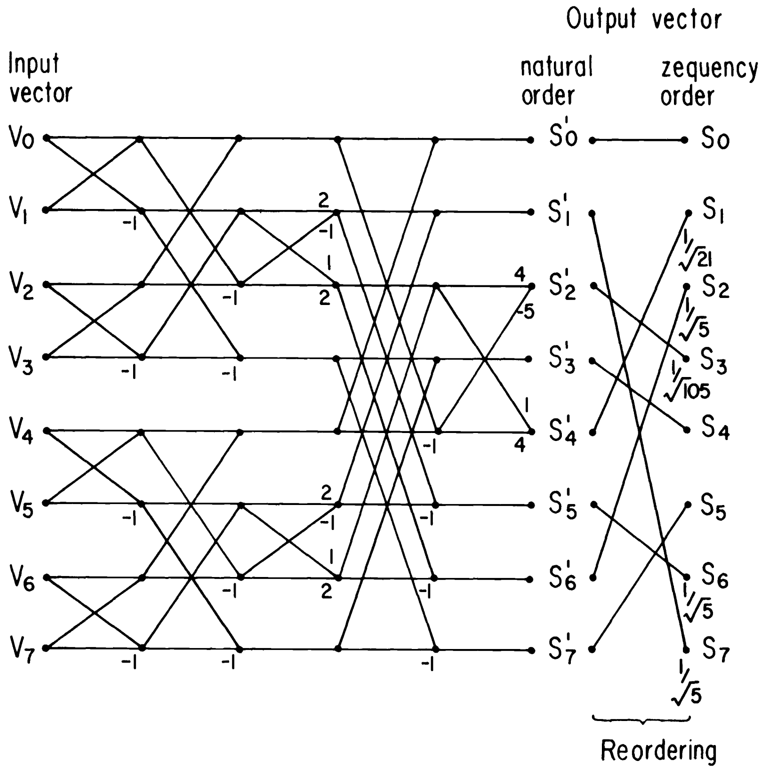


FIG. 4. *Slant transform of order* 8: *fast algorithm.*

However, the algorithm at the order 4 can be performed with 8 additions, 2 multiplications as shown in [29] instead of 10 additions and 2 shifts as above. This order 4 algorithm can be used in the recursive definition to trade 2^{n-1} additions and 2^{n-1} shifts for 2^{n-1} multiplication in the results above.

By the use of this structural decomposition the authors have defined new Slant transforms and in particular the Slant Haar Transform [31].

4. Additional properties and generalizations of unitary transforms. We discuss briefly the complex extension of a real transform, and the generalization to multidimensional FUT. We also point out some additional relations between transforms suggested by the unified framework presented.

4.1. Complex extension of a real transform. From a real unitary matrix $[RT]$ with rows $RT_0, \cdots, RT_{N-1}$, we construct a complex extension noted $[CT]$ with rows $CT_0, \cdots, CT_N$ by creating two complex rows CT_p and CT_q from two real rows RT_m RT_n as follows:

$$(21) \qquad CT_q = \frac{1}{\sqrt{2}}(RT_m - jRT_n), \qquad CT_q = \frac{1}{\sqrt{2}}(RT_m + jRT_n)$$

Then the complex transform $\mathcal{V} = \mathcal{R} + j\mathcal{I}$ of a complex input vector $V = R + jI$ is expressed uniquely from the real transforms of $\mathcal{R}$ and $\mathcal{I}$ denoted $\tilde{\mathcal{R}}$ and $\tilde{\mathcal{I}}$:

$$\mathcal{V}_p = CT_p \cdot V = \frac{1}{\sqrt{2}}(RT_m - RT_n)(R + jI)$$

or

$$\mathcal{V}_p = \frac{1}{\sqrt{2}}(\tilde{\mathcal{R}}_m + \tilde{\mathcal{I}}_n) + j(\tilde{\mathcal{I}}_m - \tilde{\mathcal{R}}_n)$$

and similarly

$$\mathcal{V}_q = \frac{1}{\sqrt{2}}(\tilde{\mathcal{R}}_m - \tilde{\mathcal{I}}_n) + j(\tilde{\mathcal{I}}_m + \tilde{\mathcal{R}}_n).$$

With these relations the properties of complex transforms can be deduced from those of the real transform. In the literature, besides the real and complex Fourier transforms, the complex W–H transform [31][32] (also called complex BIFORE transform), complex Haar transform (also called complex modified BIFORE transform) [33] have been defined. Note that the complex W–H transform obtained by relations (21) would have entries $(\pm 1 \pm j)/\sqrt{2}$; commonly the rows are then rotated by $(1 + j)/\sqrt{2}$ to give a transform with entries ±1 and $\pm j$. The rows of the complex W–H transform can be ordered according to a generalized frequency defined as the number of clockwise rotations around the origin when following cyclically the entries of a row.

Although this extension may seem trivial, we think it may help to "demystify" some complex transforms so constructed.

4.2. Multidimensional transforms. The techniques presented for the one dimensional transforms extend to multidimensional separable transforms. Let us denote an input array of p dimensions by $A_{i_q}, \cdots, i_p$ and the p dimensional separable transform by $T_{u_1, \cdots, u_p, i_1, \cdots, i_p} = T^1_{u_1 i_1}, T^2_{u_2 i_2}, \cdots, T^p_{u_p i_p}$. Then the transformed array

$$B_{u_1, \cdots, u_p} = \sum_{i_1} \sum_{i_2} \cdots \sum_{i_p} A_{i_1, \cdots, i_p} T_{u_1, \cdots, u_p, i_1, \cdots, i_p}$$

can be written

$$B_{u_1, \cdots, u_p} = \sum_{i_p} T^p_{u_p i_p} \sum_{i_{p-1}} \cdots \sum_{i_1} A_{i_1, \cdots, i_p} T^1_{u_1 i_1}.$$

If we express both arrays as one dimensional vectors A and B, for which indexes are obtained by lexicographic ordering of the indexes $(i_1, \cdots, i_p)$ and $(u_1, \cdots, u_p)$, the multidimensional transform can be expressed as a one dimensional transform:

$$A = [[T^1] \otimes [T^2] \cdots \otimes [T^p]]B, \qquad A = [T]B.$$

The multidimensional transform has been reduced to a one dimensional transform.[8] This expression now allows the evaluation of the number of elementary operations and other generalizations discussed previously.

4.3. Relations between transforms. Two transforms with similar structures will often be related by matrix relations or energy invariants between the two sets of transformed coefficients.

a) *Matrix relations between transforms of same order.* In [6], relations between the W–H and Haar transform were established. Similar relations hold for other transforms with "overlapping" structures. With the framework developed here these relations can be obtained immediately by examination of the recursive definitions of both transforms.

b) *Energy invariants.* By Parseval's theorem the total energy of the transform coefficients of a same vector with different transforms is preserved. However, it may happen that the energy of a subset of coefficients is the same for some transforms: we say then there is an energy invariant between these transforms. Energy invariants are most likely when the transforms have an identical structure with different factors. For example, by direct comparison of the algorithms for the Fourier, W–H and modified Haar, it is clear that the transformed coefficients before respective reorderings have identical energies in the "zones" of consecutive coefficients. This leads to the energy invariants for transforms of order 8 shown in Table 2.

TABLE 2

Zone	Fourier (frequencies)	W–H (zequencies)	Mod. Haar (rank)
0	0	0	0
1	4	7	1
2	2, −2	3, 4	2, 3
3	1, 3, −1, −3	1, 2, 5, 6	4, 5, 6, 7

Conclusions. In this work we have presented a unified treatment of unitary transforms having a fast algorithm. The use of recursive rules to describe unitary transforms allows a systematic way to view known transforms, to generate new transforms and provide a general approach to evaluate the number of elementary operations required by each transform algorithm. This framework has been used here to discuss common FUT, but we must add that all FUT known to the authors fit easily and that many new ones can be generated.

The framework provided can be used in several other studies and applications of unitary transforms. In particular, a faster computation of transform domain covariance matrices was found [35] and an error analysis of unitary transforms has been carried out, which provides new insight and results even for

[8] The use of separability in the generation of the transform T is similar to the general decomposition of a Kronecker product into block matrices, discussed in § 1, with block matrices applying to subvectors of the input vector.

the well known FFT algorithms. In each potential application of FUT, consideration must be given to the loss of performance of each FUT as compared to an optimal "slow" transformation and to the computational aspects of each FUT. A systematic study of applications with algorithm complexity as an explicit parameter reveals significant differences in the ranking of commonly used FUT and which points out where additional FUT would have definite merit is in preparation.

REFERENCES

[1] I. J. Good, *The interaction algorithm and practical Fourier analysis*, J. Roy. Statistical Soc. Ser. B., 20 (1958), pp. 361-372; *Addendum*, Ibid., 22 (1960), pp. 372-375.

[2] J. W. Cooley and J. W. Tukey, *An algorithm for the machine computation of complex Fourier series*, Math. Comput., 19 (1965), pp. 297-301.

[3] W. M. Gentleman and G. Sande, *Fast Fourier transform—for fun and profit*, AFIPS, 1966 Fall Joint Com. Conf., pp. 563-578.

[4] J. E. Whetchel and D. F. Guinn, *The fast Fourier-Hadamard transform and its use in signal representation and classification*, Eascon '68 Rec., IEEE Press, pp. 561-573.

[5] *Proceedings*, "*Application of Walsh Functions*", National Technical Information Service, U.S. Department of Commerce, Springfield, VA, 1970: AD-707 431, 1971: AD-727 000, 1972: AD-744 650.

[6] B. J. Fino, *Relations between Haar and Walsh/Hadamard transforms*, Proc. IEEE (Lett.), 60 (1972), pp. 647-648.

[7] J. E. Shore, *On the application of Haar functions*, IEEE Trans. Comm., COM-21 (1973), pp. 209-216.

[8] W. K. Pratt, *Generalized Wiener filtering computation techniques*, IEEE Trans. Computers, C-21 (1972), pp. 636-641.

[9] H. C. Andrews, *An Introduction to Mathematical Techniques in Pattern Recognition*, John Wiley, New York, 1972.

[10] S. J. Campanella and G. S. Robinson, *A comparison of orthogonal transformations for digital speech processing*, IEEE Trans. Comm., COM-19 (1971), pp. 1045-1050.

[11] H. C. Andrews, *Computer Techniques in Image Processing*, Academic Press, New York, 1970.

[12] T. S. Huang, W. F. Schreiber and O. J. Tretiak, *Image processing*, Proc. IEEE, 59 (1971), pp. 1586-1609.

[13] P. A. Wintz, *Transform picture coding*, Proc. IEEE, 60 (1972), pp. 809-820.

[14] H. C. Andrews and J. Kane, *Kronecker matrices, computer implementation and generalized spectra*, J. Assoc. Comput. Mach., 17 (1970), pp. 260-268.

[15] H. C. Andrews and K. L. Caspari, *A generalized technique for spectral analysis*, IEEE Trans. Computers, C-19 (1970), pp. 16-25.

[16] ———, *Degrees of freedom and modular structure in matrix multiplication*, Ibid., C-20 (1971), pp. 133-141.

[17] N. Ahmed, K. R. Rao and R. B. Schultz, *A generalized discrete transform*, Proc. IEEE, 59 (1971), pp. 1360-1362.

[18] ———, *A class of discrete orthogonal transforms*, to appear.

[19] H. F. Harmuth, *Transmission of Information by Orthogonal Functions*, 2nd ed., Springer-Verlag, New York, 1972.

[20] C-K. Yuen, *Remarks on the ordering of Walsh functions*, IEEE Trans. Computers (Corresp.), C-21 (1972), p. 1452.

[21] H. S. Stone, *Parallel processing with the perfect shuffle*, IEEE Trans. Computers, C-20 (1971), pp. 153-161.

[22] P. Y. Schwartz, J. Poncin and B. Fino, *Statistical properties of orthogonal transforms*, Proc. Conf. on Digital Processing of Signals in Communications, vol. 23, pp. 151-174, London, Apr. 1972.

[23] J. A. Glassman, *A generalization of the fast Fourier transform*, IEEE Trans. Computers, C-19 (1970), pp. 105-113.

[24] D. K. KAHANER, *Matrix description of the fast Fourier transform*, IEEE Trans. Audio Electroacoust., AU-18 (1970), pp. 442–450.

[25] M. DRUBIN, *Kronecker product factorization of the FFT matrix*, IEEE Trans. Computers, C-20 (1971), pp. 590–593.

[26] R. SINGLETON, *An algorithm for computing the mixed radix fast Fourier transform*, IEEE Trans. Audio Electroacoust., AU-17 (1969), pp. 93–103.

[27] B. J. FINO AND V. R. ALGAZI, *A unified matrix approach to Walsh Hadamard transforms*, IEEE Trans. Computers, C-25 (1976), pp. 1142–1145.

[28] H. ENOMOTO AND K. SHIBATA, *Orthogonal transform coding system for television signals*, Proc. 1971 Symp. on Appl. of the Walsh Functions, pp. 11–17. Nat. Tech. Inf. Serv.: AD-727 000.

[29] W. K. PRATT, L. R. WELCH AND W. H. CHEN, *Slant transform for image coding*, Proc. 1972 Symp. on Appl. of the Walsh Functions, pp. 229–234. Nat. Tech. Inf. Serv.: AD-744 650.

[30] W. K. PRATT, *Walsh functions in image processing and two dimensional filtering*. Proc. 1972 Symp. on Appl. of the Walsh Functions, pp. 14–22, Nat. Tech. Inf. Serv.: AD-744 650.

[31] B. J. FINO AND V. R. ALGAZI, *Slant Haar transform*, Proc. IEEE, 62 (1974), 5, pp. 653–654.

[32] N. AHMED AND K. R. RAO, *Complex BIFORE Transform*, Electron. Lett., 6 (1970), no. 8, pp. 256–258.

[33] F. R. OHNSORG, *Application of Walsh functions to complex signals*, Proc. 1970 Symp. on Applications of the Walsh Functions, pp. 123–127. Nat. Tech. Inf. Serv.: AD-707 431.

[34] K. R. RAO AND N. AHMED, *Modified complex BIFORE Transform*, Proc. IEEE, 60 (1972), pp. 1010–1012.

[35] B. J. FINO AND V. R. ALGAZI, *Computation of transform domain covariance matrices*, Proc. IEEE, 63 (1975), pp. 1628–1629.

[36] S. W. GOLOMB AND L. D. BAUMERT, *The search for Hadamard matrices*, Amer. Math. Monthly, 70 (1973), pp. 12–17.

[37] K. R. RAO AND N. AHMED, *Discrete orthogonal transforms and their applications*, Engineering Dept. Univ. of Texas, Austin, 1976.

[38] N. AHMED AND K. R. RAO, *Orthogonal Transforms for Digital Signal Processing*, Springer-Verlag, Berlin/New York, 1975.

SIAM J. COMPUT.
Vol. 6, No. 4, December 1977

PROSPECTS AND LIMITATIONS OF AUTOMATIC ASSERTION GENERATION FOR LOOP PROGRAMS*

JAYADEV MISRA†

Abstract. The problem of generation of loop invariants from the input, output assertions of a loop program (**while** B **do** S) is considered. The problem is theoretically unsolvable in general. As a special case we consider assertions of the form $x\,R\,y$, where R denotes a binary relation, x denotes the variables manipulated by the program and y denotes variables that are not modified by the program. We derive conditions for R such that if any loop program has $x\,R\,y$ as the input and output assertions, then $x\,R\,y$ is a loop invariant. These conditions for R are shown to be necessary and sufficient in that if some R' does not meet these conditions, then there are loop programs for which $x\,R'\,y$ holds at entrance and exit, though not following every iteration. In particular it is shown that if R is an equivalence relation, then under certain reasonable restrictions on the loop, $x\,R\,y$ holds at entrance and exit of the loop if and only if it holds after every iteration.

Key words. assertion, loop program, verification

1. Introduction. One of the major difficulties in mechanical program proving is to generate suitable assertions for a given program, given its input, output specifications. In theory, the problem is unsolvable. The most difficult aspect of assertion generation (by humans or algorithms) is in locating a "loop invariant" [4] for every loop. For a loop of the form {**while** B **do** S}, a loop invariant is a proposition P such that $P \wedge B\ \{S\}\ P$.[1] It then follows that if P is true on entrance to the loop, it can be asserted to be true on exit. In order to show that a proposition Q_2 is true at exit given that a proposition Q_1 is true on entry, it is sufficient to locate a proposition P such that (i) $Q_1 \Rightarrow P$ (ii) $P \wedge B\ \{S\}\ P$ (iii) $P \wedge \neg B \Rightarrow Q_2$.

An important problem in mechanical program verification is to obtain a loop invariant P as above, given Q_1, Q_2. Several heuristic techniques have been reported [5], [8], [10]. Recently an interesting scheme called "subgoal induction" [9] has been introduced which seems to be effective in a large number of cases.

Due to lack of suitable general techniques, it is important to characterize certain classes of input/output propositions for which the invariant may be obtained algorithmically. Such a characterization is interesting if it includes most of the commonly occurring forms of propositions which arise in actual programs. The present paper is a step in that direction.

We will consider loops of the form {**while** $B(x)$ **do** $S(x)$}: x denotes the set of variables on which the loop operates; x has an initial value on entry to the loop. The value of x is modified by the loop body. Let $x\,R\,y$ denote that x is related to y under R. A loop *preserves* a relation R if

$$x\,R\,y\ \{\textbf{while}\ B(x)\ \textbf{do}\ S(x)\}\ x\,R\,y.$$

* Received by the editors July 22, 1975, and in final revised form February 2, 1977. A preliminary version of some of these results appears in [6].

† Department of Computer Sciences, University of Texas at Austin, Austin, Texas 78712. This work was supported by the National Science Foundation under Grants GJ36424 and DCR75-09842.

[1] Using the notation introduced by Hoare [4].

y denotes some variables that are not modified by the loop. Verbally, the loop *preserves* the relation R if $x\,R\,y$ true on entrance to the loop implies it remains true on exit from the loop with the modified x, assuming termination. For instance, $x > y$ {**while** $B(x)$ **do** $x := x+1$} $x > y$, for any B. A loop *uniformly preserves* a relation R if $x\,R\,y$ is a loop invariant; i.e., if

$$x\,R\,y \wedge B(x)\,\{S(x)\}\,x\,R\,y.$$

Clearly, if R is uniformly preserved, then R is preserved by the loop. The converse however is not true.

In this paper, we characterize the class of relations R having the property that if R is preserved by *any* loop, then it is uniformly preserved. Clearly, the characterization conditions are trivial for any specific loop **while** $B(x)$ **do** $S(x)$, namely $x\,R\,y \wedge B(x)$ $\{S(x)\}$ $x\,R\,y$; the definition itself. Our interest in studying such characterization is to prove/disprove $x\,R\,y$ {**while** $B(x)$ **do** $S(x)$} $x\,R\,y$, when R meets the given characterization, by proving/disproving that $x\,R\,y$ is a loop invariant.

We will use the notion of *closure* introduced in [1]. We will define a class of relations called generalized equivalence relations (GE relation) and show that if R is a GE relation and is preserved by any loop having closure, then it is uniformly preserved. Conversely, if R is not a GE relation, then there exists a loop that preserves R, but does not preserve it uniformly. Any equivalence relation is shown to be a GE relation. Conditions for proving/disproving that a loop computes a certain function can be derived from this characterization.

This paper generalizes the results in [1]. However, a knowledge of that paper is not necessary to follow the results presented here. Implications of these results in automatic program verification are discussed.

2. Some preliminary notions. We will be working with loops of the form {**while** B **do** S}. We need to make explicit mention of variables on which the program operates. Consider the following schema, which we call $W(B, S)$.

```
begin
    declaration for variables $t$; {This is optional}
    while $B$ do $S$
end;
```

We adopt the following conventions about $W(B, S)$.

(i) $W(B, S)$ accepts input in certain global variables. The set of global variables will usually be denoted by x. Let x_0 denote the initial values of x before entry into the loop.

(ii) The variables t as defined above are called local variables of $W(B, S)$. Local variables initially have undefined values. A local variable gets a value when it is assigned one during computation. For the rest of the paper, global and local refer to global and local variables x, t of $W(B, S)$ respectively.

(iii) B is a predicate over some or all global variables. The rationale for such a requirement is that local variables have undefined values on entry and certain clauses in B may otherwise be undefined as a result.

(iv) The program $W(B, S)$ does not terminate if it ever accesses (examines/uses) a variable having an undefined value.

(v)The output of $W(B, S)$ appears in global variables x. Thus, the effect of execution of the loop is to modify the values of x.

Local variables t of $W(B, S)$ are indeed in a certain sense global to "**while** B **do** S". However, we believe that the distinction between local and global variables is important considering (i), (ii), (iv) and (v). Furthermore, local variables t of $W(B, S)$ are different from any local variable that S may have; during iterations t may retain values from iteration to iteration whereas local variables of S have undefined values at the beginning of every iteration.

Our treatment of variables is general. We are not specifically interested in the kind of data that a variable may represent: one variable may represent a tree, another may represent a file segment etc. We require that the variable values be drawn from a prespecified domain, but there is no restriction on the domain itself.

The next notion is fundamental; it was introduced in [1]. ("Domain" refers to the set of initial variable values of interest.)

DEFINITION 1. A domain D is *closed* with respect to $W(B, S)$ if and only if $x \in D$ is a loop invariant, i.e.

$$x \in D \wedge B(x)\,\{S(x)\}\, x \in D.$$

Observation. If D is closed, then starting with any initial value $x_0 \in D$, the variable values after every iteration must be from D. If the loop terminates, the final values are from D.

The importance of closure was demonstrated in [1], [7] where it was shown that a knowledge of closure is essential in locating a suitable loop invariant. We will assume closure of the input for the rest of the paper.

Example 1.

```
while $v \neq 0$ do
begin
    $u := u+1; v := v-1$
end;
```

Let

$$D = \{(u, v) | u, v \text{ integer}; v \geqq 0\},$$

$$D' = \{(u, v) | u, v \text{ integer}; v \geqq 30\},$$

$$D'' = \{(u, v) | u, v \text{ integer}; u \geqq 30\},$$

D, D'' are closed with respect to the given program. D' is not closed since with $(u, v) = (5, 30) \in D'$, we obtain $(6, 29)$ after one iteration, which is not in D'. □

Depending on the context $S(x)$ would either denote that S uses variables x (as in $x\,R\,y\ \{S(x)\}\ x\,R\,y$) or the value computed by S when x denotes the initial values of its variables (as in $S(x)\,R\,y$).

DEFINITION 2. A set D is *range inclusive* with respect to a function F if for every $a \in D$, $F(a)$ is defined and $F(a) \in D$.

DEFINITION 3. $W(B, S)$ *computes* a function F over a domain D if

(i) D is range inclusive with respect to F

and

(ii) for every input $x_0 \in D$, $W(B, S)$ halts and produces $F(x_0)$ as the output (in the global variables).

The following theorem is the basis of the results appearing in the next section. It is from [1], [7].

THEOREM 1′ (see [1], [7]). *Suppose D is closed with respect to $W(B, S)$. Let D be range inclusive with respect to a given function F. $W(B, S)$ computes F over D if and only if all of the following conditions hold.*

1) *$W(B, S)$ terminates for every input from D.*

2) $(x \in D \wedge \neg B(x)) \Rightarrow (F(x) = x)$.

3) $[F(x) = F(y)]$ *is a loop invariant for $W(B, S)$.*

Furthermore, conditions 1), 2) *and* 3) *are mutually independent.*

Condition 3) is the important invariant condition. It states that if F is the function computed by $W(B, S)$, then $F(x)$ remains identical during successive iterations with modified values of x in each iteration. Clearly, $F(x)$ must be defined for every such x generated during iterations: this is guaranteed by the requirement of closure on D.

The following are the significant aspects of Theorem 1′:

(i) the conditions in the theorem are necessary and sufficient. Thus proving/disproving these conditions proves/disproves the claim. This is in contrast to many assertion generation systems which provide only sufficient conditions.

(ii) The form of the invariant is independent of B and S.

3. Relations uniformly preserved by a loop. Conditions 1), 2) and 3) in the statement of Theorem 1′ may be labeled as termination, boundary and iteration conditions. The boundary condition is easy to derive (and usually simple to prove) by considering the exit conditions. The iteration condition is the one that leads to the loop invariant which captures the "dynamics" of the loop. In this paper, we are primarily interested in the generation of the iteration condition.

The major contribution of this paper is a generalization of Theorem 1′. We characterize the class of binary relations which are preserved by any loop if and only if they are uniformly preserved.

The motivation behind this extension is twofold. First, we often want to prove a certain relationship between input and output of a loop without knowing the exact functional relationship. For instance, we may want to show that the output value is larger in magnitude than the input or that the output array is a permutation of the input array, etc. Secondly, we hope to establish a theoretical limitation on what kinds of loop invariants can be generated without examining the loop body.

Clearly, if R is uniformly preserved then it is preserved. However, the converse is not true, even for transitive relations (such as $\leqq$ on integers), as shown in the following example.

Example 2. Let $W(B, S)$ be the following program.

```
while $v \neq 1$ do
    if odd($v$) then $v := v + 1$
              else $v := v/2$;
```

$D = \{v \mid v \geqq 1$ and v integer$\}$. Let the relation R be defined as follows:

$$v\,R\,u \Leftrightarrow v \leqq u.$$

Clearly,

$$v\,R\,u\,\{W(B, S)\}\,v\,R\,u.$$

However, $v\,R\,u$ is not a loop invariant (as can be seen with $v = 3$ and $u = 3$). □

We first derive the conditions on R, dependent on B and independent of S. Next, we remove the dependence on B. We are thus given

(i) a binary relation R on a domain D;

(ii) that D is closed with respect to $W(B, S)$ and $W(B, S)$ terminates for every input from D;

(iii) and that $W(B, S)$ preserves the relation R. We ask for the necessary and sufficient conditions, independent of S, under which $x\,R\,y$ is a loop invariant.

DEFINITION 4. Given a domain D and a binary relation R on D, $(a, b \in D)a \geqq b$ if and only if $(\forall\, c \in D)(b\,R\,c \Rightarrow a\,R\,c)$; $a \equiv b$ if and only if $a \geqq b$ and $b \geqq a$. Note that, if $(\nexists\, c \in D, b\,R\,c)$, $(\forall\, a \in D)(a \geqq b)$. $\geqq$ will be called the *derived relation* of R.

Observation. For any R, the derived relation $\geqq$ is reflexive and transitive and $\equiv$ is an equivalence relation.

Notation. $\neg(x\,R\,y)$ will denote that x is not related to y under R.

LEMMA 1. *Let F denote the function computed by $W(B, S)$ on the closed domain D. If R is preserved by $W(B, S)$, then $F(x) \geqq x$, $\forall\, x \in D$.*

Proof. If R is preserved then $x\,R\,y \Rightarrow F(x)\,R\,y$. Hence the lemma follows from definition. □

DEFINITION 5. R is a *GE relation (generalized equivalence relation) with respect to B* if and only if

$$(\forall\, a, b \in D)[B(a) \wedge B(b) \wedge \exists\, c \in D[c \geqq a \wedge c \geqq b] \Rightarrow a \equiv b].$$

Observation. If R is a GE relation with respect to B then

$$[B(a) \wedge B(b) \wedge a \geqq b] \Rightarrow [a \equiv b].$$

This follows by using the fact that $a \geqq a$.

Example 3. The following are examples of GE relations:

(i) Let $D = \{x \mid x \text{ integer}; x \geqq 0\}$. For some fixed k,

$$B(x)\colon x > k.$$

Define R to be

$$x\,R\,y \Leftrightarrow |x - y| \leqq k, \qquad x, y \in D.$$

Note that if $B(x)$ is true then there is no $z \neq x$ for which $z \geqq x$. Hence (trivially) R is a GE relation with respect to B.

(ii) $D = \{x \mid x$ is an undirected graph$\}$.

$$(x, y \in D)\; x\,R\,y \Leftrightarrow x, y \text{ are isomorphic.}$$

It can be shown (see next lemma) that R is a GE relation for any B. □

LEMMA 2. *If R is an equivalence relation, then it is a GE relation with respect to every B.*

Proof. We first show that

$$[x \geqq y] \Leftrightarrow [x\, R\, y]$$

(i) $x\, R\, y \Rightarrow x \geqq y$:

$$x\, R\, y \wedge y\, R\, z \Rightarrow x\, R\, z,$$

since R is an equivalence relation. Thus, $x\, R\, y \Rightarrow x \geqq y$.

(ii) $x \geqq y \Rightarrow x\, R\, y$:

$$\forall z\, [y\, R\, z \Rightarrow x\, R\, z].$$

Since R is an equivalence relation, $y\, R\, y$ holds. Hence, $x\, R\, y$.

It thus follows that $[x \geqq y] \Leftrightarrow [x\, R\, y]$.

i.e., $[x \geqq y] \Leftrightarrow [x\, R\, y] \Leftrightarrow [y\, R\, x] \Leftrightarrow [y \geqq x]$,

i.e., $[x\, R\, y] \Leftrightarrow [x \equiv y]$,

i.e., $\exists\, z \in D[z \geqq x \wedge z \geqq y] \Rightarrow \exists\, z \in D[z \equiv x \wedge z \equiv y] \Rightarrow [x \equiv y]$.

Thus R is a GE relation for any B. □

The following theorem is the central result. In the statement of the theorem, only those $W(B, S)$ are considered for which D is closed and $W(B, S)$ terminates for all inputs from D.

THEOREM 2. *Let R be a binary relation and B a predicate on a given domain D. If R is a GE relation with respect to B then R is uniformly preserved by any W(B, S) if it is preserved. Conversely, suppose R is not a GE relation with respect to B and for some S is preserved by W(B, S). Then there exists W(B, S′) for which R is preserved though not uniformly.*

Proof. Let R be a GE relation with respect to B. Let F be the function computed by any $W(B, S)$ on domain D. By assumption, D is closed with respect to $W(B, S)$ and $W(B, S)$ terminates for every input from D. First we will show that

$$x\, R\, y \wedge B(x)\, \{S(x)\}\, x\, R\, y$$

i.e.,

$$x\, R\, y \wedge B(x) \Rightarrow S(x)\, R\, y.$$

Case (i). $\neg B(S(x))$:

$$B(x) \Rightarrow F(x) = S(x)$$

$$x\, R\, y \Rightarrow F(x)\, R\, y,$$

since R is preserved. Hence,

$$x\, R\, y \wedge B(x) \Rightarrow S(x)\, R\, y.$$

Case (ii). $B(S(x))$: The proof is by contradiction. Suppose that

$$x\, R\, y \wedge B(x) \wedge \neg(S(x)\, R\, y).$$

Then

$$x \not\equiv S(x),$$

since $x\,R\,y$ and $\neg(S(x)\,R\,y$. Using Lemma 1, $F(x)\geqq x$ and $F(S(x))\geqq S(x)$. Using Theorem 1′, $F(x)=F(S(x))$. We thus have

$$B(x)\wedge B(S(x))\wedge F(x)\geqq x\wedge F(x)\geqq S(x)\wedge x\neq S(x).$$

Hence, R is not a GE relation with respect to B, contradiction.

Next we show that if R is not a GE relation with respect to some B and, for some S, R is preserved by $W(B,S)$, then there exists S' such that

(i) D is closed with respect to $W(B,S')$ and
(ii) $W(B,S')$ terminates for every input from D, and
(iii) R is preserved by $W(B,S')$, and
(iv) R is not uniformly preserved by $W(B,S')$.

We first state a claim whose proof is similar to that of Lemma 1.

Claim. If R is uniformly preserved by $W(B,S)$ then $B(x)\Rightarrow S(x)\geqq x$.

If R is not a GE relation with respect to B then there exist $x_1, x_2, x_3\in D$ such that

$$B(x_1)\wedge B(x_2)\wedge x_3\geqq x_1\wedge x_3\geqq x_2\wedge x_1\neq x_2.$$

Since $x_1\neq x_2$, either $x_1\ngeqq x_2$ or $x_2\ngeqq x_1$. Without loss in generality assume that $x_2\ngeqq x_1$.

The proof proceeds by constructing S'. Consider the following program.

```
while $B$ do
    if $x = x_1$ then $x := x_2$
        else if $x = x_2$ then $x := F(x_3)$
            else $S$;
```

It can be verified that conditions (i), (ii), (iii) are met by this program. Next we show that R is not uniformly preserved by this program. With input x_1, we obtain x_2 and then $F(x_3)$. However, $x_2\ngeqq x_1$. Hence, according to the previous claim R is not uniformly preserved. Note that the first "else" clause in S' ensures that the program would terminate when input with x_1. □

Theorem 2 says that given any $W(B,S)$ and R which is a GE relation with respect to B, in order to prove that R is preserved, it is necessary and sufficient to prove that R is uniformly preserved. Conversely if R is not a GE relation with respect to B, it is sufficient though not necessary to prove that R is uniformly preserved in order to show that R is preserved.

COROLLARY 1. *Let R be an equivalence relation. For any $W(B,S)$ (assuming termination and closure of domain) R is preserved, if and only if it is uniformly preserved.*

Proof. Use Lemma 2 and Theorem 2. □

Example 4. Consider a program $W(B,S)$ for sorting an array x of integers. It is required to prove at the output that the resulting array is sorted and is a permutation of the input array. To prove the latter, we need to prove

$$PERM\,(x,x_0)\,\{W(B,S)\}\,PERM\,(x,x_0),$$

where $PERM\,(x,x_0)$ stands for "x is a permutation of x_0". Clearly $PERM$ is an equivalence relation. It is then necessary and sufficient to prove that $PERM\,(x,x_0)$

is a loop invariant,

$$PERM\ (x, x_0) \wedge B(x) \Rightarrow PERM\ (S(x), x_0),$$

i.e.,

$$B(x) \Rightarrow PERM\ (x, S(x)). \quad \square$$

The conditions derived in Theorem 2 clearly apply to the iteration condition in Theorem 1′. This can be seen easily by defining an equivalence relation R on D such that $x\ R\ y \Leftrightarrow [F(x) = F(y)]$. Clearly if $W(B, S)$ computes F, R is preserved. Using Theorem 2, it follows that it must be uniformly preserved.

We next formulate the conditions on R independent of B.

DEFINITION 6. R is a *GE relation on a domain* D if and only if it is a GE relation with respect to every predicate B (binary valued total function on D).

THEOREM 3. *R is a GE relation on D if and only if its derived relation ($\geqq$) is an equivalence relation.*

Proof. If $\geqq$ is an equivalence relation then

$$[c \geqq a \wedge c \geqq b] \Rightarrow [a \geqq b \wedge b \geqq a] \Rightarrow [a \equiv b].$$

Hence R is a GE relation for any B. Conversely, if $\geqq$ is not an equivalence relation then there exist a, b such that $a \geqq b$ and $a \not\equiv b$. Consider some predicate B for which $B(a)$ and $B(b)$ are true. R is not a GE relation with respect to this B since

$$B(a) \wedge B(b) \wedge a \geqq a \wedge a \geqq b \wedge a \not\equiv b. \quad \square$$

For any relation R, if we define the successor set of a, $T(a) = \{b \mid a\ R\ b\}$, then R is a GE relation if and only if no successor set $T(a)$ strictly includes another successor set $T(b)$, since otherwise $a \geqq b$ and $b \not\geqq a$.

Observation. If R is a GE relation then either all successor sets are null or none is.

In program proving, it would be easier to consider general GE relations rather than GE relations with respect to a specific B. Theorem 3 provides a useful technique for proving that a certain R is a GE relation. The next theorem essentially provides the verification conditions that must be proved to ensure that a GE relation is preserved.

THEOREM 4. *Let R be a GE relation. Then (assuming termination and closure)*

$$x\ R\ y\ \{W(B, S)\}\ x\ R\ y,$$

if and only if

$$B(x) \Rightarrow [x \equiv S(x)].$$

Proof. We first show that if R is a GE relation,

$$x\ R\ y\ \{W(B, S)\}\ x\ R\ y$$

if and only if

$$x \equiv z\ \{W(B, S)\}\ x \equiv z.$$

Let F be the function computed by $W(B, S)$ (on the domain D). Then if R is preserved by $W(B, S)$, $F(x) \geqq x$ or $F(x) \equiv x$, since R is a GE relation. This says that the input x and output $F(x)$ belong to the same equivalence class under $\equiv$.

Conversely, let

$$x \equiv z \, \{W(B, S)\} \, x \equiv z.$$

Thus, $F(x) \equiv x$. Hence $F(x) \geqq x$ or $x \, R \, y \Rightarrow F(x) \, R \, y$. Hence, R is preserved by $W(B, S)$.

Using Corollary 3,

$$x \equiv z \, \{W(B, S)\} \, x \equiv z$$

if and only if $\equiv$ is uniformly preserved, i.e.,

$$x \equiv z \wedge B(x) \Rightarrow S(x) \equiv z,$$

i.e.,

$$B(x) \Rightarrow [x \equiv S(x)]. \quad \square$$

Example 5. The following program is claimed to compute the greatest common divisor of m, n using successive subtraction.

```
begin
    integer $t$;
    while $m \neq n$ do
    begin
      if $m < n$ then begin $t := m$; $m := n$; $n := t$ end;
      $m := m - n$
    end
end;
```

Let $D = \{(m, n) \mid m, n \text{ integer}; \, m, n > 0\}$. Let GCD be the function of two arguments that has the value of the greatest common divisor of the arguments. Let H be a function from domain D to range D, defined as follows:

$$H(m, n) = (\mathrm{GCD}(m, n), \mathrm{GCD}(m, n)).$$

We wish to show that

$$[H(m, n) = H(m_0, n_0)] \, \{W(B, S)\} \, [H(m, n) = H(m_0, n_0)],$$

where $W(B, S)$ represents the above program and $(m, n), (m_0, n_0) \in D$ at entrance to the loop.

The reason for using H instead of GCD is that the former is a function from D to D, as required by Theorem 4, whereas the latter is a function from D to a subset of positive integers.

We must first prove closure and termination.

(i) Closure:

$$m, n \text{ integer} \wedge m > 0 \wedge n > 0 \wedge m \neq n \, \{S\} \, m, n \text{ integer}, m > 0, n > 0.$$

Equivalently, we must show,

$$[m, n \text{ integer}, m > 0 \wedge n > 0 \wedge m < n] \Rightarrow [m, n - m \text{ integer} \wedge n - m > 0 \wedge m > 0]$$

and

$$[m, n \text{ integer} \wedge m>0 \wedge n>0 \wedge m>n] \Rightarrow [n, m-n \text{ integer} \wedge m-n>0 \wedge n>0].$$

(ii) Termination: It is then necessary and sufficient to show that $H(m, n) = H(m_0, n_0)$ is a loop invariant, i.e.

$$[H(m, n) = H(m_0, n_0)] \wedge m \neq n \; \{S\} \; [H(m, n) = H(m_0, n_0)],$$

i.e.

$$[m \neq n \wedge m < n] \Rightarrow [H(m, n) = H(n-m, m)]$$

and

$$[m \neq n \wedge m > n] \Rightarrow [H(m, n) = H(m-n, n)]$$

i.e.,

$$[m < n \Rightarrow \text{GCD}(m, n) = \text{GCD}(n-m, m)]$$

and

$$[m > n \Rightarrow \text{GCD}(m, n) = \text{GCD}(m-n, n)].$$

All that remains to be proved is the boundary condition given below, in order to show that m, n both have the value of $\text{GCD}(m_0, n_0)$ at the exit.

$$\{m = n \wedge [H(m, n) = H(m_0, n_0)]\} \Rightarrow \{m = \text{GCD}(m_0, n_0) \wedge n = \text{GCD}(m_0, n_0)\}. \quad \square$$

Finally, we show that a certain simple class of relations, as given in Example 3, can be shown to be preserved by extending the given relation to an equivalence relation and proving that the latter is preserved.

DEFINITION 7. Given any relation R on D, define the *reflexive*, *symmetric*, *transitive closure* R^* *of* R as follows.

$$a\,R^*\,b \Leftrightarrow (a = b) \vee a\,R\,b \vee b\,R\,a \vee [\exists\, c\; (a\,R^*\,c \wedge c\,R^*\,b)], \quad a, b, c \in D.$$

Thus, R^* is an equivalence relation. Under certain conditions, it is both necessary and sufficient to prove that R^* is uniformly preserved, in order to show that R is preserved.

THEOREM 5. *Let R be any relation on D and R^* be its reflexive, symmetric transitive closure. Suppose for some predicate B on D,*

$$x_1\,R^*\,x_2 \wedge \neg B(x_1) \Rightarrow x_1\,R\,x_2, \quad \forall\, x_1, x_2 \in D.$$

Then

$$x\,R\,y\;\{W(B, S)\}\;x\,R\,y$$

if and only if R^* is uniformly preserved, assuming closure and termination.

Proof. We show that

$$x\,R\,y\;\{W(B, S)\}\;x\,R\,y \quad \text{if and only if} \quad x\,R^*\,y\;\{W(B, S)\}\;x\,R^*\,y.$$

Since R^* is an equivalence relation, the statement in the theorem would follow. Let F be the function computed by $W(B, S)$. Suppose

$$x\,R\,y\;\{W(B, S)\}\;x\,R\,y.$$

Then

$$x\,R\,y \Rightarrow F(x)\,R\,y \Rightarrow F(x)\,R^*\,x.$$

Thus, x and $F(x)$ belong to the same equivalence class under R^*, or

$$x\,R^*\,y\;\{W(B,S)\}\;x\,R^*\,y.$$

Next, suppose, $x\,R^*\,y\;\{W(B,S)\}\;x\,R^*\,y$. Then on termination, $\neg B(x) \wedge x\,R^*\,y \Rightarrow x\,R\,y$, or $x\,R^*\,y\;\{W(B,S)\}\;x\,R\,y$.

Since $x\,R\,y \Rightarrow x\,R^*\,y$, it follows that $x\,R\,y\;\{W(B,S)\}\;x\,R\,y$. □

Example 6.

```
•
  begin
      while $v \neq 0$ and $v \neq 1$ do $v := v-2$
  end;
```

$D=\{v \mid v \geqq 0;\ v$ integer$\}$. $v, u \in D$, $v\,R\,u \Leftrightarrow (v \leqq u)$ and u, v have identical parity (both even or odd). $v\,R^*\,u \Leftrightarrow u, v$ have identical parity.

$$(v, u \in D)\;\{v\,R^*\,u \wedge [v=0 \vee v=1]\} \Rightarrow [v \leqq u],$$

and u, v have identical parity.

Hence, according to Theorem 5, it is necessary and sufficient to prove that $v\,R^*\,u$ is a loop invariant, in order to show that R is preserved; i.e.

$$v\,R^*\,u \wedge v \neq 0 \wedge v \neq 1 \Rightarrow (v-2)\,R^*\,u.$$

Termination and closure must be proven separately. □

4. Summary and conclusion. We have shown that any equivalence relation is uniformly preserved if it is preserved by a loop program. Theorem 5 extends the results somewhat for relations that are essentially equivalence relations except for certain boundary conditions. A practical outcome of this result is that loop invariants may be generated algorithmically for certain classes of input/output relations. There is no need to look through the body of the loop to generate the invariant, provided closure and termination have been proven separately.

Unfortunately, the results also establish that such conditions cannot be obtained for any other classes of relations. Thus, even a simple transitive relation such as "≦" on positive integers is not uniformly preserved even though it is preserved. We believe that one needs to look at the program body S, for generating the loop invariant, for all other classes of relations.

One promising direction of research is to consider other classes of relations and "reasonable" programs. The loop invariant could be generated if the program meets certain reasonable restrictions; for example, we may assume that a program operating on stacks may not process any other element, before processing the top element, (i.e. it should not be allowed to save the top element, process and remove the second element from top and then restore the top element). Some preliminary results appear in [3], [7].

Another problem we have not considered in this paper is the problem of nonclosed domains. Frequently, a loop is preceded by initializations which restrict the input domain. Most of the time, the domain will not be closed with respect to the program. It is often required to prove a certain relation (such as a functional equality) in the restricted domain. This problem has been considered for the case of functional equality in [2], [3], [7]. This seems to be the major problem in synthesis of loop invariants. It seems likely, however, that by suitably restricting the operations of the program, it may be possible to prove that the program computes a certain relation over a superset of the given domain (which is closed) from which the stated conjecture may be proven.

Acknowledgment. The author is indebted to the referees for their constructive comments.

REFERENCES

[1] S. Basu and J. Misra, *Proving loop programs*, IEEE Trans. on Software Engrg., 1 (1975), pp. 76–86.

[2] ———, *Deterministic generation of inductive assertions*, IEEE Workshop on Automated Theorem Proving, Argonne National Lab., Argonne, IL, 1975.

[3] ———, *Some classes of naturally provable programs*, Proc. Second International Symposium on Reliable Software, San Francisco, 1976.

[4] C. A. R. Hoare, *An axiomatic approach to computer programming*, Comm. ACM, 12 (1969), pp. 576–580, 583.

[5] S. Katz and Z. Manna, *A heuristic approach to program verification*, Proc. 3rd International Conference on Artificial Intelligence, Stanford Univ., Stanford, CA, 1973.

[6] J. Misra, *Relations uniformly conserved by a loop*, Proc. 9th International Symposium on Proving and Improving Programs, Arc et Senans, France, 1975, pp. 71–80.

[7] ———, *Some aspects of verification of loop computation*, unpublished manuscript.

[8] M. Moriconi, *Semiautomatic synthesis of inductive predicates*, ATP-16, Dept. of Mathematics, Univ. of Texas at Austin, 1974.

[9] J. H. Morris and B. Wegbreit, *Subgoal Induction*, Xerox Palo Alto Research Center, 1975.

[10] B. Wegbreit, *Heuristic methods for mechanically deriving inductive assertions*, Proc. 3rd International Conf. on Artificial Intelligence, Stanford Univ., Stanford, CA, 1973.

SIAM J. COMPUT.
Vol. 6, No. 4, December 1977

AN ANALYSIS OF A GOOD ALGORITHM FOR THE SUBTREE PROBLEM*

STEVEN W. REYNER†

Abstract. A good algorithm is analyzed for deciding if one tree is a subtree of another tree. If both trees are rooted, the smaller tree has n vertices and the larger has m vertices; then the total number of computations is $O\ (nm^{1.5})$ or better, depending on how good an algorithm one has for a maximal matching in a bipartite graph.

Key words. algorithm, algorithmic analysis, bipartite graphs, computational complexity, matching, subtree

1. Introduction. We present and analyze a good algorithm for deciding if one rooted tree S is a subtree of another rooted tree T. This algorithm is equivalent to the one sketched by Matula [3]. When one deletes the root of a rooted tree, one obtains subtrees, each of which is now rooted at that vertex adjacent to the original root. We use these rooted subtrees together with a good algorithm for a maximal matching in a bipartite graph to determine whether or not S is a subtree of T.

2. The algorithm.

ALGORITHM. 1) Delete the roots of S and T to obtain rooted subtrees $S_1, \cdots, S_p$ and $T_1, \cdots, T_q$.

2) (Recursively) Decide if S_i is a rooted subtree of T_j and form a $p \times q$ matrix A with $a_{ij} = 1$ (0) if S_i is (not) a rooted subtree of T_j.

3) Apply an algorithm for obtaining a maximal matching in a bipartite graph to A to decide if all the rooted subtrees of S can be matched with rooted subtrees of T.

3. Data storage. For computational purposes, we store a rooted tree as a vector as follows. (See Busacker and Saaty [1].) The vector for an isolated vertex is 1. For more complicated rooted trees, the first entry is the number of vertices of the tree and the remaining entries are formed by arranging the vectors of the rooted subtrees (obtained by deleting the original root), these vectors arranged in increasing (lexicographical) order.

Given a rooted tree, it is trivial to obtain its rooted subtrees. If the vector V describes the tree and I is the location of a root of a subtree, then the vector describing the subtree is $V(I), V(I+1), \cdots, V(I+V(I)-1)$. Since in deciding if a tree is a rooted subtree of another, one need not alter either for this algorithm, the computations involved in obtaining the subtrees are insignificant.

If one programs this algorithm using a language which is recursive and in which arrays are allocated dynamically (such as APL), one could use two subprograms BIPARTITE and SUBTREE, two global vectors VECTS and VECTT and several local matrices A. A global variable is one available to all subprograms. A local variable is one available only in a main program or

* Received by the editor March 20, 1976, and in revised form January 15, 1977.

† Department of Mathematics, State University of New York at Oswego, Oswego, New York 13126. This work was performed while author was on sabbatical at State University of Colorado, Fort Collins, Colorado.

subprogram where it is defined, plus any other routine to which it is specifically passed. The vectors VECTS and VECTT would store the two rooted trees S and T as previously explained. BIPARTITE uses as input a 0-1 matrix A and returns a 1 if a maximal matching of rows to columns using 1's uses all the rows (the number of independent 1's equals the number of rows) and returns 0 otherwise. SUBTREE would receive as input two local scalars I and J, the location of the roots (at the current stage) in VECTS and VECTT. We refer to these rooted subtrees as subtree I and subtree J. SUBTREE creates a matrix A which is VECTS(I)-1 by VECTT(J)-1. Set $A(K, L)$ to 1 if the Kth vertex adjacent to any beyond the vertex corresponding to I is terminal. Set $A(K, L)$ to 0 if this Kth vertex is not terminal but the Lth vertex adjacent to and beyond the vertex corresponding to J is terminal. If neither is terminal, then SUBTREE invokes SUBTREE (creating an additional, independent A) to decide if subtree K is a subtree of subtree L ($A(K, L)$ is 1 yes, 0 no). When A is completed, BIPARTITE is invoked to decide if subtree I is a subtree of subtree J. SUBTREE returns 1 if subtree I is a subtree of subtree J, and 0 otherwise.

By being recursive, several different A's are in use at any one particular moment which the computer keeps track of. The total significant variable storage in use at any time is the two vectors VECTS and VECTT plus several A's. The total storage required by the various A's is obviously bounded both by the order of S times the maximum degree minus 1 of T and by the order of T times the maximum degree minus 1 of S.

4. Computational bounds. The key to this being a good algorithm is using a good algorithm for obtaining a maximal matching in a bipartite graph rather than putting subtrees together in all possible combinations.

The algorithm for a maximal bipartite matching with best proven bound on number of computations currently appears to be given by Hopcroft and Karp [2]. In this algorithm, the number of computations is shown to be no worse than $O(rs^{1.5})$ where the bipartite graph matches r to s vertices with $r \leqq s$. Obviously, the efficiency of the matching algorithm dictates the efficiency of the subtree algorithm. Let n and m denote the number of vertices in S and T respectively. We now prove two theorems relating the efficiency of a matching algorithm to the efficiency of our subtree algorithm.

THEOREM 1. *Given an algorithm for bipartite matching which requires at most $O(rs^u)$ operations where $r \leqq s$ and $u > 1$, the subtree algorithm will require at most $O(nm^u)$ operations.*

Proof. Choose b large enough so that the matching algorithm requires at most brs^u operations for each r and s, and so that the subtree algorithm requires at most bnm^u operations whenever n and m satisfy $(n-1)((m-1)^u + m - 1) > nm^u$. Such a b exists since only finitely many n and m satisfy the above inequality and the marriage algorithm is $O(rs^u)$. Note this gives us a starting point for induction. We prove that at most bnm^u operations are required for any n and m, namely when $(n-1)((m-1)^u + m - 1) \leqq nm^u$. We now assume n and m satisfy this last inequality. If the valences of the roots of S and T are d and f respectively, and the number of vertices in S_i and T_j are n_i and m_j respectively, then the number of computations involved in the subtree algorithm is at most

$$bdf^u+\sum_{i=1}^{d}\sum_{j=1}^{f} bn_im_j^u=bdf^u+b(n-1)\sum_{j=1}^{f} m_j^u$$

$$\leqq b(n-1)\Big(f^u+\sum_{j=1}^{f} m_j^u\Big)\leqq b(n-1)(f^u+f-1+(m-f)^u)$$

computations. Since the second derivative of this last term is positive $1\leqq f\leqq m-1$, the maximum occurs at an end point and an upper bound is the larger of $b(n-1)(1+(m-1)^u)\leqq bnm^u$ and $b(n-1)((m-1)^u+m-1)\leqq bnm^u$ (by assumption); thus at most bnm^u operations are required.

THEOREM 2. *Given an algorithm for bipartite matching which requires at most brs operations, the subtree algorithm will require at most bmn* ln (n) *operations.*

Proof. We again proceed inductively. If $n=1$ this is obvious. Given the tree and subtree descriptions of the last proof, the number of computations involved is at most

$$\sum_{i=1}^{d}\sum_{j=1}^{f} bn_im_j \ln(n_i)+bdf=b(m-1)\Big(\sum_{i=1}^{d} n_i\ln(n_i)\Big)+bdf$$

$$\leqq b(m-1)((n-d)\ln(n-d)+d).$$

Since the second derivative is positive $1\leqq d\leqq n-1$, an upper bound is the larger of $b(m-1)((n-1)\ln(n-1)+1)\leqq bnm\ln(n)$ (for $n\geqq 2$) and $b(m-1)(0+m-1)\leqq bnm\ln(n)$, as required.

5. Conclusion. If one is interested in solving the corresponding problem for unrooted trees, one may root S at will and root T in all m possible ways. Thus one needs at most m times as many computations as indicated in Theorems 1 and 2.

The matching algorithm involves upwards of rs bits of data, thus it seems unlikely that a matching algorithm could be better than $O(rs)$. Consequently, Theorems 1 and 2 appear to cover all possible cases, and we have a good algorithm for deciding the subtree problem.

REFERENCES

[1] R. BUSACKER AND T. SAATY, *Finite Graphs and Networks*, McGraw-Hill, New York, 1965, pp. 196–199.

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

[3] DAVID W. MATULA, *An Algorithm for subtree identification*, SIAM Rev., 10 (1968), pp. 273–274 (Abstract).

SIAM J. COMPUT.
Vol. 6, No. 4, December 1977

ON RELATING TIME AND SPACE TO SIZE AND DEPTH*

ALLAN BORODIN†

Abstract. Turing machine space complexity is related to circuit depth complexity. The relationship complements the known connection between Turing machine time and circuit size, thus enabling us to expose the related nature of some important open problems concerning Turing machine and circuit complexity. We are also able to show some connection between Turing machine complexity and arithmetic complexity.

Key words. space, depth, time, size, computational complexity, parallel arithmetic complexity, parrallel time

1. Introduction. Fischer and Pippenger [7] have shown that a $T(n)$ time bounded Turing machine (TM) can be simulated on n bits by a combinational (Boolean) circuit with $O(T(n) \log T(n))$ gates (see also Schnorr [25]). In this paper, we observe that nondeterministic $S(n)$ tape bounded Turing machines can be simulated by circuits of depth $O(S(n)^2)$. In doing so, we relate the power of nondeterminism for space bounded computations to the depth required for the transitive closure problem. As a consequence of this development we show a relationship between the TIME-SPACE problem and the SIZE-DEPTH problem (equivalently the SIZE-FORMULA SIZE problem).

There has always been some ambiguity between the terminology of circuit complexity and TM complexity. In particular, depth is sometimes referred to as "time". But here "time" implicitly means *parallel time*, since several gates in a combinational circuit can operate in parallel. (In arithmetic complexity, depth is almost always referred to as parallel time.) In Pratt and Stockmeyer [22], we are introduced to vector machines, a general parallel machine model; general in the sense that inputs can be of arbitrary length (see also Hartmanis and Simon [9]). It is then shown that polynomial time (that is, parallel time) for these vector machine corresponds to TM polynomial space. Motivated by the simulations of Pratt and Stockmeyer [22] and Hartmanis and Simon [9], it is not hard to see that the crux of our space simulation should rely on the transitive closure problem. In the next section, we define the models more carefully and present the basic simulation. In § 3, we discuss relationships between TM and circuit complexity. In § 4, we conclude with some observations concerning arithmetic complexity.

2. Turing machines, circuits and the basic simulation. We assume that the reader is familiar with Chapters 6 and 10 of Hopcroft and Ullman [11]. Our TM model is an "off-line" machine with a two-way read only input tape. For time bounded computations we allow an arbitrary but finite number of work tapes. For tape bounded computations, it is sufficient to have just one work tape. The benefit of the read only tape is that it allows us to consider tape bounds less than the length of the input. Usually we consider Turing machines as acceptors or recognizers but

* Received by the editors March 30, 1976, and in revised form December 6, 1976.

† Department of Computer Science, University of Toronto, Toronto, Ontario, Canada M5S 1A7. This research was supported by the National Research Council of Canada.

sometimes we will need to consider transducers, in which case we append a write only output tape.

For a nondeterministic machine M, say that $L \subseteq \Sigma^*$ is *accepted* in time $T(n)$ (space $S(n)$) by M if $w \in L$ iff on input w there is a valid computation leading to an accepting state which uses $\leqq T(|w|)$ steps (respectively, $S(|w|)$ work tape cells). Here $|w|$ denotes the length of the string. Without loss of generality, we can restrict ourselves to $\Sigma = \{0, 1\}$.

A combinational (Boolean) circuit is a labeled acyclic, directed graph (a network). Nodes with in-degree $= 0$ are called *input nodes*, nodes with out-degree $= 0$ are called *output nodes*. Interior nodes (i.e. noninput nodes including output nodes) represent (i.e. are labeled with) logical gates f: $\{\text{true, false}\}^r \to \{\text{true, false}\}$. Since we shall only be concerned with asymptotic complexity bounds, without loss of generality we can use the complete basis AND (denoted $\wedge$), inclusive OR ($\vee$) and NOT ($\neg$). (See Savage [23, pp. 662, 663].) We can have arbitrary fan-out and allow the constants {true, false} as inputs. By associating true with '1' and false with '0', we think of every Boolean circuit as realizing a function f: $\{0, 1\}^n \to \{0, 1\}$. That is, let $A^n \subseteq \{0,1\}^n$; we say A^n is *realized by circuit* C if C has n nonconstant input nodes (labeled $x_1, \dots, x_n$) and C accepts (i.e. outputs $1 = \text{true}$) iff $x_1x_2 \dots x_n$ is in A^n. As a notational convenience, if $A \subseteq \{0, 1\}^*$, let $A^n = A \cap \{0, 1\}^n$.

The size of a circuit C is the number of interior nodes or gates, and the depth of C is the length of the longest path in C. We will also have need to encode a circuit C as a string $\bar{C}$ in $\{0,1\}^*$. This can be done in a straightforward way; i.e. topologically order the network, give addresses to each of the nodes, and then a circuit can be given as a sequence of instructions.

If the output of C depends on all n inputs $x_1, \dots, x_n$, the size of C must be $\geqq n - 1$; it follows that $|\bar{C}| \geqq d \cdot \text{size } C \cdot \log \text{size } C$ for some constant $d > 0$. (Note that the chosen basis implies fan-in $\leqq 2$.)

Finally, we let $\text{SIZE}_A(n)$ (respectively $\text{DEPTH}_A(n)$) be the minimum size (depth) required for a circuit to realize A^n. Using this notation, we recall the time-size simulation result.

THEOREM 1 (Fischer and Pippenger [7]). *Let A be recognized by a deterministic $T(n) \geqq n$ time bounded* TM. *Then there exists $d > 0$ such that* $\text{SIZE}_A(n) \leqq d \cdot T(n) \cdot \log T(n)$.

Analogously, we have the following:

THEOREM 2. *Let A be accepted by a* nondeterministic $S(n) \geqq \log_2 n$ *space bounded* TM. *Then there exists a $d > 0$ such that* $\text{DEPTH}_A(n) \leqq d \cdot S(n)^2$.

Proof. Let M be a $S(n)$ tape bounded nondeterministic TM with $S(n) \geqq \log_2 n$. Say M has q states and s symbols on its work tape and that the input $w \in \{0, 1\}^n$. Thinking of w as fixed, the computation sequence is "determined" by the state, the input tape head position, the work tape head position, and the contents of the work tape. Thus there are at most $N = q \cdot n \cdot S(n) \cdot s^{S(n)}$ *configurations*. We can think of the acceptance problem as the transitive closure problem for a graph with N nodes, whose edges correspond to the allowable moves determined by M and w.

Let $X = (x_{ij})$ where $x_{ij} = 1$ iff there is a move from configuration i to configuration j, and let $X^* = (x^*_{ij})$ be the transitive closure. That is, $x^*_{ij} = 1$ iff there is a path from configuration i to configuration j. (See Fig. 1.)

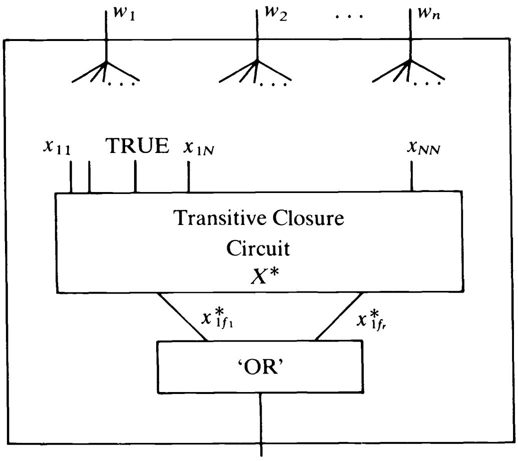


FIG. 1

Let i be the number of a configuration corresponding to the input head being on square k. Then

w_k is directly connected to x_{ij} iff there is a move from configuration i to j *only when* the kth input bit is 1.

w_k is negated and then connected to x_{ij} iff there is a move from configuration i to j *only when* the kth input is 0.

w_k is not connected to x_{ij} iff there is (is not) a move from configuration i to j independent of the value of the kth input bit. In this case x_{ij} is set to the appropriate constant.

Let 1 be the number of the starting configuration and let f_j $(1 \leqq j \leqq r)$ correspond to accepting configurations. By using $\log(N-1)$ levels of Boolean matrix multiplication, it is well known that an $N \times N$ transitive closure circuit requires only $\log^2 N$ depth. Also, an N-way 'OR' can obviously be realized with depth $\log N$. The theorem follows since $N = q \cdot n \cdot S(n) \cdot s^{S(n)}$ and therefore $\log N < d_1 \cdot S(n)$.

Remark. Let A be accepted by M as in Theorem 2. With a little care in the numbering of the configurations, the mapping $1^n \rightarrow \bar{C}_n$ (where C_n realizes A^n) is computable by a *deterministic* $S(n)$ space bounded transducer if $S(n)$ is tape constructible. This follows because of the "uniformly constructive" nature of the $\log^2 N$ depth transitive closure circuits (i.e. they can be generated in $\log N$ space). We shall have more to say about "uniformity" in § 3.

Open Problem 1. If M is *deterministic* $S(n)$ tape bounded, can we improve the simulation so that $\mathrm{DEPTH}_A(n) \leqq d \cdot S(n)$?

Since each output in the $N \times N$ transitive closure problem can be computed in nondeterministic $\log N$ space, it follows that improving the simulation for nondeterministic machines is equivalent to improving the depth required for the transitive closure problem. In particular, if the $N \times N$ transitive closure problem can be realized with depth $\leqq c \cdot \log^\alpha n$ (where $1 \leqq \alpha \leqq 2$) then $\mathrm{DEPTH}_A(n) \leqq d \cdot S(n)^\alpha$ in Theorem 2.

Open Problem 2. Let A be recognized by a deterministic $S(n)$ tape, $T(n)$ time bounded machine. Can we realize A^n by circuits with $\mathrm{SIZE}_A\ (n) \leqq c_i T(n)^{k_1}$ and (simultaneously) $\mathrm{DEPTH}_A\ (n) \leqq c_2 T(n)^{k_2}$ for some constants c_1, c_2, k_1, k_2?

Let $\leqq_{\log}$ represent log space reducibility (see Jones and Laaser [12]). We could allow nondeterministic transduction here but we might as well follow the standard meaning of deterministic log space many-one reducibility. By converting every log space transducer to one with a separate track keeping count (in binary) of the number of output bits thus far in the computation, we can generalize Theorem 2 as follows:

THEOREM 3. *Let $A \leqq_{\log} B$. Then*

$$\mathrm{DEPTH}_A\ (n) \leqq d \cdot \log^2 n + \max_{m \leqq cn^k} \mathrm{DEPTH}_B\ (m)$$

for some constants c, d and k.

Proof. Let m be the log space transducer which reduces A to B. Then on input w of length n, M can output at most $N = cn^k$ bits since M is $\log n$ space bounded. Let y_i^0 (respectively, y_i^1) be the 'OR' of those configurations where the ith bit being output is a '0' (respectively, '1'). If y_i^0 and y_i^1 are both false, then we know that M outputs less than i bits on input w. Knowing m, the exact number of bits output by M on w, we can "activate" the circuit C_m for B^m with inputs y_i^1 $(1 \leqq i \leqq m)$. (See Fig. 2.)

Following standard notation, let P be the class of languages recognizable in deterministic polynomial time. B is called *log space complete* for P if

(i) B is in P,

(ii) A in P implies $A \leqq_{\log} B$.

Cook [3], Jones and Laaser [12] and Ladner [29] exhibit a variety of natural sets which are log space complete for P. We can define an analogous concept for circuits. Namely, let us say that B is *depth complete for polynomial size circuits* if

(i) $\mathrm{SIZE}_B\ (n) \leqq p(n)$ for some polynomial p.

(ii) Let A be such that $\mathrm{SIZE}_A\ (n) \leqq p_1(n)$ for some polynomial p_1. Then there exist constants c and k, and a polynomial q such that for all n there is an n-input circuit T_n with the following properties:

(a) depth $T_n \leqq c \cdot \log^k n$

Note: of course, we would like $k = 1$.

(b) T_n outputs $\langle y_1, \cdots, y_N \rangle = f_1(\langle x_1, \cdots, x_n \rangle)$ and $\langle z_1, \cdots, z_N \rangle = f_2(\langle x_1, \cdots, x_n \rangle)$ for some fixed $N \leqq q(n)$.

(c) there is a unique output z_m with value 1 and for this m we have $x_1 \cdots x_m$ in A^n iff $y_1 \cdots y_m$ in B^m.

Our definition has been chosen so that the construction in Theorem 3 immediately yields:

COROLLARY 1. *If B is log space complete for P, then B is depth complete for polynomial size circuits.*

We think of depth (log space) complete sets as being "hardest polynomially computable sets" with respect to depth (space) requirements. Ladner defines the following "circuit value problem" and shows it to be log space complete for P:

$$V = \{x_1 \cdots x_n \bar{\#}\ \bar{C} \mid C \text{ outputs true on input } x_1 \cdots x_n\}.$$

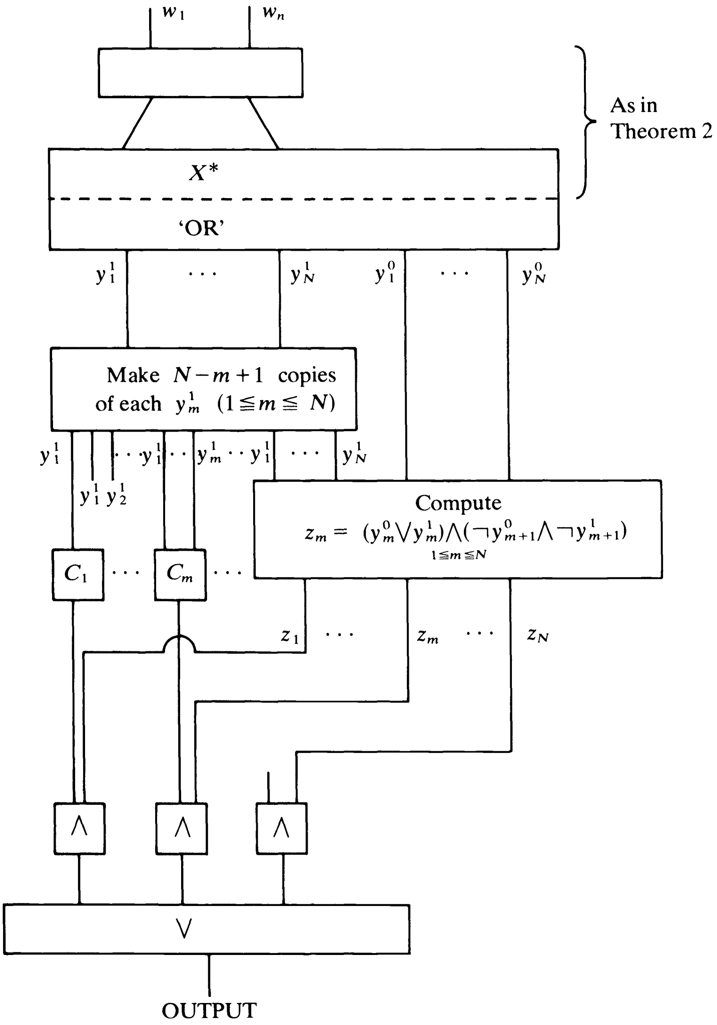

w_1
w_n
As in Theorem 2
X^*
'OR'
y_1^1 . . . y_N^1 y_1^0 . . . y_N^0
Make $N-m+1$ copies of each y_m^1 $(1\leqq m\leqq N)$
y_1^1 . . y_1^1 . . y_m^1 . . y_1^1 . . . y_N^1
$y_1^1\ y_2^1$
Compute $z_m=(y_m^0\vee y_m^1)\wedge(\neg y_{m+1}^0\wedge\neg y_{m+1}^1)$ $1\leqq m\leqq N$
C_1 . . . C_m . . .
z_1 . . . z_m . . . z_N
$\wedge$
$\wedge$
$\wedge$
$\vee$
OUTPUT

FIG. 2

Corollary 1 verifies the obvious fact that V is depth complete for polynomial size circuits.

3. Relating open problems in Turing machine and circuit complexity. As usual, we let DTIME $(T(n))$ (respectively, DSPACE $(S(n))$, NSPACE $(S(n))$ denote the class of languages accepted in deterministic Time $T(n)$ (respectively, deterministic and nondeterministic space $S(n)$). Analogously, define SIZE$(T(n))=\{B|\text{SIZE}_B(n)\leqq c\cdot T(n)$ for some constant $c\}$ and DEPTH$(S(n))$

$= \{B|\text{DEPTH}_B(n) \leqq c \cdot S(n)\}$. We first want to complete the relationship between TM space and circuit depth and thus we need a "converse" to Theorem 2.

LEMMA 1. *Let V be the circuit value problem. There is a deterministic* TM M *which recognizes V such that on input $x_1 \cdots x_n \# \bar{C}$, M only uses space bounded by* Depth C + log size C.

Proof. The idea is just to recursively evaluate (say, first the left and then the right) inputs to a given gate. A straightforward implementation using a pushdown store would require depth $C \cdot$ log size C storage (i.e. depth C levels in the store, log size C space for each gate address entry). We use a suggestion by S. Cook to improve this to the desired bound. In evaluating the output of C, we only need to store the full address of the gate currently being evaluated while the status (e.g. left input has value "false" and right input now being evaluated) of each gate on the pushdown store can be accommodated within constant space. We can recompute the address of any gate on the stack by working up from the bottom of the stack (i.e. the circuit output gate) via the status entries.

DEFINITION. We say that A is *uniformly* in DEPTH $(S(n))$ if there is a constant c such that for all n, there is a circuit $\bar{C}_n$ of depth $\leqq c \cdot S(n)$ realizing A^n and, moreover, $\bar{C}_n$ can be generated in deterministic space $S(n)$; i.e. the transformation $1^n \rightarrow \bar{C}_n$ is deterministic $S(n)$ space computable.

THEOREM 4. *Suppose A is uniformly in* DEPTH $(S(n))$, $S(n) \geqq \log n$. *Then A is in* DSPACE $(S(n))$.

Note. We view this as a "converse" to Theorem 2 for that result can be stated as: "If A is in NSPACE $(S(n))$, then A is uniformly in DEPTH $(S(n)^2)$".

Proof. Given an input $w = x_1 \cdots x_n$, we apply Lemma 1. Now whenever we need to know the ith bit of $\bar{C}_n$ (as in the lemma), we compute it (in the required space) by using the uniformity hypothesis.

It is well-known that one can define arbitrarily complex or nonrecursive sets A such that for all n, $A^n = \phi$ or $A^n = \Sigma^n$. Since a trivial circuit realizes A^n for each n, it is clear that A in DEPTH $(S(n))$ does not imply that A in DSPACE $(S'(n))$ for any $S'(n) \geqq S(n)$. In order to relate space and depth, we chose to assert a uniformity condition on the circuits. There is another choice. Following Schnorr [25], Meyer and Stockmeyer [17] suggest "making the Turing machines nonuniform" by giving them oracles. Then they observe that our Theorems 2 and 4 can be modified as follows:

(a) If A is recognized by a nondeterministic $S(n)$ space bounded TM M with a $\{0, 1\}^*$ oracle, then A is in DEPTH $(S(n)^2)$.

(b) If A is in DEPTH $(S(n))$, then A is recognized by a deterministic $S(n)$ space bounded TM M with a $\{0, 1\}^*$ oracle.

In (b), the oracle is used to encode the appropriate efficient circuit. In both (a) and (b) we count the space needed for the oracle tape questions. This formulation does have a very nice mathematical appeal. We have, however, chosen to assert the uniformity of circuits because from a "practical" point of view, experience tells us that if we can show A is in DEPTH $(S(n))$, then we usually can show A is uniformly in DEPTH $(S(n))$. The same choice also exists for the TIME-SIZE relationship. Here "uniformly in SIZE $(T(n))$" means that we can generate the appropriate circuit description in deterministic time $T(n)$. Then we can relate uniform size and time or (see Schnorr [25]) we can instead relate size and time of Turing machines with oracles. Using Schnorr's [25] model, nonuniformity takes form in a $\{0, 1\}^*$ oracle explicitly listed on a separate tape; Meyer [16] shows that with the more conventional model which uses a separate tape for inputs to an oracle, nonuniformity for time takes form in a $\{0\}^*$ oracle.

We can use Theorems 2 and 4 to make explicit the role of transitive closure in Savitch's [24] construction. Throughout the remainder of this paper we let α ($1 \leqq \alpha \leqq 2$) be such that the $N \times N$ transitive closure problem can be realized (uniformly) with depth $\leqq c \cdot \log^\alpha N$.

COROLLARY 2. *Let α be as above. Then* NSPACE $(S(n)) \subseteq$ DSPACE $(S(n)^\alpha)$ *for all tape constructible $S(n) \geqq \log n$.*

Proof. Let M be a nondeterministic $S(n)$ space bounded TM accepting A.

For every input $w = x_1 \cdots x_n$ we can generate a circuit C_n corresponding to M, w (as in Theorem 2). By hypothesis, this can be done in deterministic space $S(n)$, and depth $C_n \leqq c \cdot S(n)^\alpha$ for some constant c (see the remarks following Theorem 2).

That is, A is uniformly in DEPTH $(S(n)^\alpha)$ and hence A is in DSPACE $(S(n)^\alpha)$. More constructively, by using Theorem 2 and Lemma 1, we can produce a deterministic $S(n)^\alpha$ tape bounded machine M' recognizing A. □

One of the most important problems in computational complexity concerns efficient space simulations of time bounded computations. In particular, there is a conjecture that DTIME $(T(n)) \subseteq$ DSPACE $(\log^k T(n))$ for some constant k. (Indeed, $k = 1$ is still possible.) Cook [3], and Cook and Sethi [4] present important evidence that the conjecture is false, and this represents the concensus of opinion at this time. On the positive side, Hopcroft, Paul and Valiant [10] have shown that DTIME $(T(n)) \subseteq$ DSPACE $T(n)/\log T(n))$. Independent of this result (and independent of our observations), Paterson and Valiant [19] proved that SIZE $(T(n)) \subseteq$ DEPTH $(T(n)/\log T(n))$, noting that their result only had significance when $T(n) \approx n$. The known relationships between TIME-SIZE, and SPACE-DEPTH are not refined enough to show that either of these results follows from the other, but we can show that the problems are related.

COROLLARY 3. *Suppose* DTIME $(n) \subseteq$ NSPACE $(S(n))$. *Then* SIZE $(T(n)) \subseteq$ DEPTH $(S'(n))$ *uniformly for any time constructible $T(n)$ where $S'(n) = [S(T(n) \log^3 T(n))]^\alpha$. Here uniformly means that the required $cS'(n)$ depth circuit can be constructed in deterministic space $S'(n)$. Recall $\alpha \leqq 2$.*

Proof. Given a $T(n)$ size circuit C, we encode it as a word $\bar{C}$ of length $\leqq c \cdot T(n) \cdot \log T(n)$. A straightforward circuit simulation by a TM can be performed in deterministic time m^2, where m is the length of the circuit description. Recently, Pippenger [21] showed how to recognize the circuit value problem V in time $m \cdot \log^2 m$. By hypothesis, and using a standard translation argument, we have DTIME $(T(n)) \subseteq$ DSPACE $(S[T(n)])$. Hence there is a deterministic TM M recognizing V in space $S(m \cdot \log^2 m)$; in particular, $x_1 \cdots x_n \# \bar{C}$ will be accepted or rejected in space $S(m \cdot \log^2 m) = S(T(n) \cdot \log^3 T(n))$. Hence there is a circuit C' of depth $c \cdot S'(n)$ which realizes $x_1 \cdots x_n \# \bar{C}$. Finally we can fix the input gates for $\bar{C}$ and the resulting circuit has depth $\leqq c \cdot S'(n)$.

For example, we have "DTIME $(n) \subseteq$ NSPACE $(\log^k n)$ implies SIZE $(T(n)) \subseteq$ DEPTH $(\log^{k\alpha} T(n))$ uniformly".

COROLLARY 4. *Suppose* SIZE $(n) \subseteq$ DEPTH $(S(n))$ *uniformly. Then* DTIME $(T(n)) \subseteq$ DSPACE $(S[T(n) \cdot \log T(n)])$ *for all constructible* $T(n)$.

Proof. Let M be $T(n)$ time bounded. We construct an equivalent M'. M' on input $w = x_1 \cdots x_n$ constructs a circuit C_n of size $c \cdot T(n) \log T(n)$ according to the Fischer–Pippenger simulation. (We claim with their oblivious $T \cdot \log T$ machine that this can be done in space $\log T(n)$.) Then, by hypothesis, we construct an equivalent circuit C'_n of depth $d \cdot S[T(n) \cdot \log T(n)]$ and finally apply Lemma 1 or Theorem 4 to produce the desired M'.

Again, for example, "SIZE $(n) \subseteq$ DEPTH $(\log^k n)$ uniformly implies DTIME $(T(n)) \subseteq$ DSPACE $(\log^k T(n))$". Note that Corollary 4 is "almost good enough" to derive the Hopcroft, Paul and Valiant [10] result from the Paterson and Valiant [19] construction. (The latter construction can be realized in space $n/\log n$). Summarizing, we have shown that the TIME-SPACE problem for Turing machines is "roughly" equivalent to an "efficiently constructive" version of the SIZE-DEPTH problem for circuits.

Circuits with a fan-out = 1 restriction correspond to formulas. Spira [26] has shown that a formula of size $T(n) \geqq n$ can be transformed to an equivalent formula of depth $\leqq c \cdot \log T(n)$. (Consider also Brent's [30] analogous result for arithmetic expressions). Moreover, it should be clear that formula size $\leqq 2^{\text{depth}}$. Hence, if we are looking for an example where formula size is exponentially larger than (arbitrary circuit) size, we might as well look at any of the languages which are log space complete for P. It should be noted that Spira's construction is reasonably uniform; that is, a formula of size n can be transformed within $\log^2 n$ space to an equivalent formula of depth $c \cdot \log n$. (This transformation should be compared with the hypothesis of Corollary 4.) We do not see how to construct a depth $c \cdot \log n$ formula in space $\log n$. However, using Lynch's [15] $\log n$ formula evaluation in conjunction with Theorem 2, we can construct a depth $c \cdot \log^2 n$ formula in space $\log n$.

4. Some comments on arithmetic circuits. An arithmetic circuit is like a Boolean circuit except that now the inputs are indeterminates $x_1, \cdots, x_n$ (and possibly constants $c_i \in F$, F a field), the internal gates are $+$, $-$, $\times$, $\div$, and the outputs are considered to be elements of $F(x_1, \cdots, x_n)$ (see Borodin and Munro [1]). For definiteness, let's take $F = Q$, the rationals. Strictly speaking, the size-depth question for arithmetic circuits is not a problem. Kung [14] has shown that x^{2k} requires k depth (depth is called parallel time in the literature of arithmetic complexity) and x^{2k} can obviously be realized with size (or sequential time) k. However, if one restricts attention to functions of small degree, the size-depth question is meaningful.

Throughout the remainder of this discussion, let us restrict our attention to the computation of multivariate polynomials or rational functions $p(x_1, \cdots, x_n)$ of degree $\leqq n$. (Also, if we do not allow arbitrary constants in Q as inputs, then we should also restrict the coefficients occurring in p.)

To argue the case that arithmetic complexity and the more traditional studies of computational complexity are related, let us consider a current problem concerning parallel arithmetic computations. Csanky [5] has shown that if PWR $(A) = \{A^2, A^3, \cdots, A^n | A$ an $n \times n$ matrix$\}$ is computable in $L(n)$ depth (parallel steps) then A^{-1}, det A, coefficients of char (A) would all be computable in $O(L(n))$ depth. Schönhage has demonstrated that a converse also holds; specifically, PWR (A) can be obtained from B^{-1} where

$$B=\begin{pmatrix} I & A & & & \\ & I & A & & \\ & & \ddots & \ddots & \\ & & & \ddots & A \\ & & & & I \end{pmatrix}.$$

Now we know $\log n \leqq L(n) \leqq \log^2 n$ and the question arises as to whether or not $L(n)=O(\log n)$. To dramatize the consequences of Corollary 2, we can make the following observation.

COROLLARY 5. *We now consider only circuits with* $+$, $-$, $\times$ (*no* $\div$) *and constants in* Q.

Suppose there is a deterministic $L(n)$ *transformation* $1^n \to \bar{C}_n$ *which generates a depth* $L(n)$ *arithmetic circuit* C_n *realizing* $A^n_{n\times n}$ (*say* $L(n)=\log^\beta n$). *Then* NSPACE $(S(n)) \subseteq$ DSPACE $(S(n)^\beta \log S(n))$ *for all constructible* $S(n) \geqq \log n$.

Proof. Let $A^*=(a^*_{ij})$ be the transitive closure (considered as a set of Boolean functions). Let $A=(a_{ij})$ be a matrix with $\{0, 1\}$ integer entries, and assume $a_{ii}=1$, $1 \leqq i \leqq n$. Letting $\tilde{A}=(\tilde{a}_{ij})=A^n$ (with respect to arithmetic matrix multiplication), we then have $a^*_{ij}=\min(\tilde{a}_{ij}, 1)$. Starting with a 0-1 matrix A, we know $\tilde{a}_{ij} \leqq n^n$. We would like to simulate (integer) arithmetic as in Munro [18] and Fischer and Meyer [6] but "mod n^n" arithmetic is $n \cdot \log n$ bit arithmetic and costs depth $\log n$. Instead following another suggestion by S. Cook, we can simulate the arithmetic mod p_i $(1 \leqq i \leqq m)$ where $\{p_i, \cdots, p_m\}$ are the first m primes and $\prod^m p_i \geqq n^n$. Since by the prime number theorem the number of primes less than x is asymptotically equal to $x/\log x$, this can certainly be done with $p_m \leqq cn \cdot \log^2 n$. (In the case of rational constants $q=r/s$, we must make sure that s^{-1} mod p_i exists; that is, we choose our $\{p_i\}$ so that no such s is equal to 0 mod p_i. Since we are only considering circuits with $L(n) \leqq \log^2 n$, the size of the circuit is $\leqq n^{\log n}$ and so we need only avoid at most $n^{\log n}$ "bad primes".) The depth cost of the mod p_i arithmetic results in a $\log \log n$ factor. We do not need to reconstruct any $\tilde{a}_{ij}$ since $\tilde{a}_{ij}=0$ iff $\tilde{a}_{ij}$ mod $p_i=0$ for all p_i. Thus a transitive closure circuit of depth $L(n) \cdot \log \log n$ can be constructed (by a deterministic $L(n)$ tape TM). Corollary 2 completes the proof.

Some discussion on Corollary 5 is appropriate. The restriction that C_n be *uniformly* generated is certainly necessary (see also Corollary 2). Our viewpoint, however, is that the discovery of a depth efficient method for A^n would be sufficiently constructive to yield the uniformity hypothesis. The restriction that $\div$ is not allowed in Corollary 5 is both annoying and possibly unnecessary. It is annoying because division is obviously necessary for A^{-1} and thus any method for A^n which is derived from A^{-1} would use division. The problem with $\div$ is that during the computation we might be dividing by a very large y for which $y \equiv 0$ mod (p_i) for all small primes p_i. For the computation of a polynomial of degree n, one can use the method of Strassen [27] to eliminate $\div$, but this results in an $O(\log n)$ factor in the depth bound. It is not known whether this factor can be improved. Yet in spite of all our restrictions, one has "the feeling" that an

$O(\log n)$ depth method for any of the problems $\det A$, A^{-1}, A^n would lead to a positive solution to the LBA problem (i.e. NSPACE $(n)=$ DSPACE (n)? See Hartmanis and Hunt [8]). The present consensus is that this is very unlikely; that is, NSPACE $(S(n)) \neq$ DSPACE $(S(n))$ for constructible $S(n)$, and indeed any improvement to Savitch's NSPACE $(S(n)) \subseteq$ DSPACE $(S(n)^2)$ would be a significant result for "traditional computational complexity".

Looking at Schönhage's observation on how to use A^{-1} to compute A^n, one sees that a depth efficient circuit for A^n can be composed of a $c \cdot \log n$ depth transformation $\langle y_1, \cdots, y_m \rangle = f(\langle x_1, \cdots, x_n \rangle)$ followed by a circuit for A^{-1}. In other words, we can define a reducibility for arithmetic circuits (as we could for Boolean circuits but here m should only depend on n) in analogy to the log space TM reducibility. Motivated by Corollary 1, and Paterson and Valiant [19], we are led to ask the following questions.

Open Problem 3. Is there a "natural" class of polynomial or rational functions which are depth complete for polynomial size arithmetic circuits? Can every rational function computable in size $T(n)$ be computed in depth $T(n)/\log T(n)$? (Note: we are still only considering rational functions $f(x_1, \cdots, x_n)$ of degree $\leqq n$.) Is depth $\log^{\alpha} T(n)$ possible?

In general, one cannot expect that positive results for Boolean computations always have arithmetic analogues. For example, Pippenger [20] shows that every Boolean symmetric function on n variables has formula size $\leqq n^{3.6}$ (and hence depth $\leqq c \cdot \log n$) whereas the elementary symmetric function $\sum_{1 \leqq i_1 < i_2 \cdots \leqq n} x_{i_1} x_{i_2} \cdots x_{i_{n/2}}$ appears to need $O(\log^2 n)$ depth. Moreover, even positive results for arithmetic computations may not always have Boolean analogues. For example, we can simulate a Boolean circuit by an arithmetic circuit; i.e. $x \vee y$ is simulated by $x + y - x \times y$, $\neg x$ by $1 - x$. But even if every size $T(n)$ arithmetic circuit (computing functions of degree $\leqq n$) was transformable into an equivalent depth $\log^{\alpha} T(n)$ circuit, a corresponding result would not necessarily hold for Boolean circuits. Suppose we try the following: Given Boolean circuit C_1, first convert to the arithmetic circuit C_2 which "simulates" C_1, then transform to an equivalent C_3 and finally obtain a depth efficient Boolean circuit C_4 by simulating C_3 with mod 2 arithmetic. The problem is that circuit C_2 (which is only "equivalent" to circuit C_1 for $\{0, 1\}$ valued inputs) may be computing arithmetic functions whose degrees can be exponential in the size of C_2, and hence exponential in n. What is missing is the concept of the degree of a Boolean function. In any case, we consider it a major open problem to exhibit a "polynomial size" function, Boolean or arithmetic, which is provably not computable in $O(\log n)$ depth.

5. Conclusion. The main results of this paper establish a relation between TM space and circuit depth. This can be interpreted as another piece of evidence (see Pratt and Stockmeyer [22], Hartmanis and Simon [9] and more recently Chandra and Stockmeyer [2], Kozen [13], and Tourlakis [28]), that parallel time and space are roughly equivalent within a polynomial factor. The simplicity of the circuit model focuses our attention on the importance of the transitive closure problem. As a result, we have been able to unify a number of open problems in computational complexity. We also claim that questions in "traditional" computational complexity have relevance to arithmetic complexity and conversely.

Acknowledgments. The author is greatly indebted to S. Cook for a number of important suggestions, and to A. Meyer, L. Stockmeyer and the referees for their helpful comments.

REFERENCES

[1] A. Borodin and I. Munro, *The Computational Complexity of Algebraic and Numeric Problems*, American Elsevier, New York, 1975.

[2] A. K. Chandra and L. Stockmeyer, *Alternation*, Proc. of Seventeenth Annual Symp. on Foundations of Computer Science, Houston, Oct. 1976, pp. 98–108.

[3] S. Cook, *An observation on time-storage tradeoff*, Proc. of the Fifth Annual ACM Symp. on Theory of Computing, May 1973, pp. 29–33.

[4] S. Cook and R. Sethi, *Storage requirements for deterministic polynomial time recognizable languages*, Proc. of Sixth Annual ACM Symp. on Theory of Computing, May 1974, pp. 33–39.

[5] L. Csanky, *Fast Parallel Matrix Inversion Algorithms*, Proc. of Sixteenth Annual Symp. on Foundations of Computer Science, Oct. 1975, pp. 11–12.

[6] M. J. Fischer and A. Meyer, *Boolean matrix multiplication and transitive closure*, Proc. Twelfth Annual IEEE Symp. on Switching and Automata Theory, Oct. 1971, pp. 129–131.

[7] M. Fischer and N. Pippenger, *M. J. Fischer Lectures on Network Complexity*, Universität Frankfurt, preprint, 1974.

[8] J. Hartmanis and H. B. Hunt, *The LBA Problem and its Importance in the Theory of Computing*, SIAM-AMS Proc., vol. 7. American Mathematical Society, Providence, RI, 1974.

[9] J. Hartmanis and J. Simon, *On the power of multiplication in random access machines*, Proc. of Fifteenth Annual Symp. on Switching and Automata Theory, Oct. 1974, pp. 13–23.

[10] J. Hopcroft, W. Paul and L. Valiant, *On time versus space and related problems*, Proc. of Sixteenth Annual Symp. on Foundation of Computer Science, Oct. 1975, pp. 57–64.

[11] J. Hopcroft and J. Ullman, *Formal Languages and their Relation to Automata*, Addison-Wesley, Reading, MA, 1969.

[12] N. D. Jones and W. T. Laaser, *Complete problems for deterministic polynomial time*, Proc. Sixth Annual ACM Symp. on Theory of Computing, May 1974, pp. 40–46.

[13] D. Kozen, *On parallelism in Turing machines*, Proc. of Seventeenth Annual Symp. on Foundation of Computer Science, Houston, Oct. 1976, pp. 89–97.

[14] H. T. Kung, *Some complexity bounds for parallel computation*, Proc. of Sixth Annual ACM Symp. on Theory of Computing, May 1974, pp. 323–333.

[15] N. Lynch, *Log Space Recognition and Translation of Parenthesis Languages*, preprint.

[16] A. Meyer, Personal communication, 1975.

[17] A. Meyer, and L. Stockmeyer, Personal communication, 1975.

[18] I. Munro, *Efficient determination of the transitive closure of a graph*, Information Processing Lett., 1 (1971), no. 2.

[19] M. S. Paterson and L. G. Valiant, *Circuit size is nonlinear in depth*, Univ. of Warwick Theory of Computation Rep., vol. 8, Sept. 1975.

[20] N. Pippenger, *Short formulae for symmetric functions*, IBM Res. Rep. RC-5143, Yorktown Heights, NY, 1974.

[21] ———, *The Complexity of Monotone Boolean Functions*, Math Systems Theory, submitted.

[22] V. Pratt and L. Stockmeyer, *A characterization of the power of vector machines*, J. Comput. System Sci., 12 (1976), pp. 198–221.

[23] J. Savage, *Computational work and time on finite machines*, J. Assoc. Comput. Mach., 19 (1972), pp. 660–674.

[24] W. Savitch, *Relationships between nondeterministic and deterministic tape complexities*, J. Comput. System Sci., 4 (1970), pp. 177–192.

[25] C. P. Schnorr, *The network complexity and the Turing machine complexity of finite functions*, Acta Informat., 7 (1976), pp. 95–107.

[26] P. M. Spira, *On time hardware complexity tradeoffs for Boolean functions*, Fourth Hawaii International Symp. on Systems Science, 1971, pp. 525–527.

[27] V. STRASSEN, *Vermeidung von Divisionen*, J. Reine Angew. Math., 264 (1973), pp. 184–202.

[28] G. TOURLAKIS, *A universal parallel machine and the efficient simulation of storage bounded sequential computations*, Dept. of Computer Science and Mathematics Tech. Rep. 1, Atkinson College, York University, Toronto, Canada, Dec. 1976.

[29] R. LADNER, *The circuit value problem in Logspace complete for P*, SIGACT News, 7 (1975), pp. 18–20.

[30] R. P. BRENT, *The parallel evaluation of several arithmetic expressions*, J. Assoc. Comput. Mach., 21 (1974), pp. 201–206.

SIAM J. COMPUT.
Vol. 6, No. 4, December 1977

GENERATING BINARY TREES LEXICOGRAPHICALLY*

F. RUSKEY AND T. C. HU†

Abstract. We represent a binary tree by the level numbers of its leaves from left to right. Thus every binary tree of n leaves corresponds to a sequence of n numbers. We first give the necessary and sufficient conditions for a sequence to represent a binary tree; then we give an algorithm for generating all the feasible sequences lexicographically as a list. Also, algorithms are developed to determine the position of a given sequence, or to generate the sequence of a given position. Finally, it is shown that the average time per sequence generated is constant (independent of the length of the sequence).

Key words. binary tree, feasible sequence, ranking algorithm

1. Introduction. Binary trees play an important role in computer science. The number of binary trees with n leaves is well-known (see for example, Knuth [2, p. 389]); but no algorithm exists which explicitly lists all the binary trees of n leaves in some natural order. Here, we represent a binary tree by the level number of its leaves. Thus a binary tree of three leaves is either (1, 2, 2) or (2, 2, 1). In general, a binary tree of n leaves is represented by a sequence of n numbers $(a_1, a_2, \cdots, a_n)$. In this paper, we develop an algorithm for generating explicitly n-number sequences representing binary trees lexicographically as a list. Without explicitly generating the list, we also give a ranking algorithm for determining the position of a given sequence on the list, and an unranking algorithm for creating the sequence occupying a given position. Finally, the efficiency of the generating algorithm is analyzed.

2. Feasible sequences. Not every sequence of n positive integers represents the level numbers of a binary tree with n leaves. Those which do are called *feasible sequences*. We also call $a_1, a_2, \cdots, a_j$ a *feasible initial sequence for n* if there exist integers $a_{j+1}, \cdots, a_n$ such that $a_1, a_2, \cdots, a_n$ is a feasible sequence. Now consider a sequence $a_1, a_2, \cdots, a_{k-2}, a_{k-1}, a_k, a_{k+1}, \cdots, a_n$, with a leftmost pair $a_{k-1} = a_k = q$. The process of replacing the pair a_{k-1}, a_k by $q-1$ to get the new sequence $a_1, a_2, \cdots, a_{k-2}, q-1, a_{k+1}, \cdots, a_n$ is called a *reduction from the left.* Continuing this process until no further reductions from the left are possible we get a final sequence called the *left-reduced sequence* for $a_1, a_2, \cdots, a_n$. For example, the left reduced sequence for 3, 6, 6, 5, 4, 7, 7, 6, 5 is 2, 4. *Reduction from the right* and *right reduced sequence* are defined analogously, except that we take the rightmost equal pair. The following lemmas are well known.

LEMMA 0 [2, p. 404]. *A necessary condition for a sequence $a_1, a_2, \cdots, a_n$ to be feasible is that*

$$\sum_{j=1}^{n} 2^{-a_j} = 1.$$

LEMMA 1 [1]. *A sequence $a_1, a_2, \cdots, a_n$ is feasible if and only if a series of $n-1$ reductions from the right or left (in any order) reduce the original sequence to the single integer* 0.

* Received by the editors October 5, 1976, and in revised form February 15, 1977.

† Division of Computer Science, University of California, San Diego, La Jolla, California 92037. This work was supported by the U.S. Army Research Office under Grant DAAG29-76-C-0031 and the National Science Foundation under Grant DCR75-06270.

Thus if a sequence is feasible, then a reduction yields another feasible sequence and if the reduced sequence is feasible then the original sequence was also feasible. We next state and prove two easy lemmas which lead to a characterization of feasible initial sequences.

LEMMA 2. *Let $a_1, a_2, \cdots, a_n$ be a sequence of positive integers and suppose there is a k such that $a_1 < a_2 < \cdots < a_{k-1}$ and $a_k > a_{k+1} > \cdots > a_n$; then $a_1, a_2, \cdots, a_n$ is a feasible sequence if and only if*

(i) *$a_{k-1} = a_k = n-1$ and*

(ii) *$a_1, a_2, \cdots, a_{k-1}, a_{k+1}, \cdots, a_n$ are all distinct.*

Proof. Suppose that (i), (ii) are true. They tell us that $\{a_1, a_2, \cdots, a_n\}$ regarded as a set is equal to $\{1, 2, \cdots, n-1\}$. The sequence is obviously feasible if $n = 2$. For $n > 2$, we can do a reduction from the left, i.e. replace a_{k-1}, a_k by $n-2$. This new sequence satisfies the conditions of the lemma.

Suppose that $a_1, a_2, \cdots, a_n$ is a feasible sequence. Clearly $a_{k-1} = a_k$. Consider the binary decimal representation of $\sum_{j=1}^{k-1} 2^{-a_j}$. There is a one in the jth place to the right of the decimal point if and only if $a_i = j$ for some $1 \leqq i \leqq k-1$. Similar remarks apply to $\sum_{j=k}^{n} 2^{-a_j}$. Since these two sums must add together to one we see that there could not be a one (or a zero) in the same position in both binary decimals before the a_kth place. In other words, (ii) holds and $a_k = n-1$. Q.E.D.

Note that the tree represented by the feasible sequence of the above lemma is one of maximal height among binary trees of n leaves. In the next lemma we are replacing the rightmost node of a binary tree by the binary tree with level numbers $1, 2, \cdots, m-1, m, m$.

LEMMA 3. *If $a_1, a_2, \cdots, a_n$ is a feasible sequence then so is $a_1, a_2, \cdots, a_{n-1}, a_n+1, a_n+2, \cdots, a_n+m, a_n+m$ for any positive integer m.*

Proof. Reduce $a_1, a_2, \cdots, a_{n-1}, a_n+1, \cdots, a_n+m, a_n+m$ from the right m times. Q.E.D.

This last lemma tells us that a feasible initial sequence for n is also a feasible initial sequence for $n+m$, m a positive integer. The following theorem tells us how many nodes we need to complete a binary tree, given the level numbers of the first j nodes.

THEOREM 1. *A sequence $a_1, a_2, \cdots, a_j$ $(j < n)$ is a feasible initial sequence for n if and only if its left reduced sequence $r_1, r_2, \cdots, r_l$ satisfies the following conditions:*

(i) $1 \leqq r_1 < r_2 < \cdots < r_l$,

(ii) $r_l \leqq n - j + l - 1$.

Proof. (sufficient) Suppose (i) and (ii) hold. Let S_i be the sequence $r_{i+1}-1, r_{i+1}-2, \cdots, r_i+1$ where we define $r_0 = 0$. The sequence is empty if $r_{i+1} - r_i = 1$. Also let $S_l = r_l$. By (i) the sequence $r_1, r_2, \cdots, r_l, S_l, S_{l-1}, \cdots, S_1, S_0$ satisfies the conditions of Lemma 2 and is thus a feasible sequence. Since the sequence $S_l, S_{l-1}, \cdots, S_1, S_0$ has $r_l - l + 1$ elements and $a_1, a_2, \cdots a_j, S_l, S_{l-1}, \cdots, S_0$ is a feasible sequence, $a_1, a_2, \cdots, a_j$ is a feasible initial sequence for $r_l + j - l + 1$. But by (ii) and Lemma 3 it is also a feasible initial sequence for n.

(necessary) If (i) does not hold then there is a k such that $r_k > r_{k+1}$ and $r_k > r_{k-1}$ if $k \neq 1$. Now at each step of the reduction process the level numbers only decrease and thus r_k could never be reduced and so any sequence $r_1, \cdots, r_l, a_{j+1}, \cdots, a_n$ could not be left reduced to 0.

If (ii) does not hold then $r_l > n-j+l-1$, but no tree $r_1, \cdots, r_l, a_{j+1}, \cdots, a_n$ with $n-j+l$ leaves can have a leaf at level greater than $n-j+l-1$. Q.E.D.

COROLLARY. *Let $a_1, a_2, \cdots, a_j$ be a feasible initial sequence for n $(j<n-1)$ with left reduced sequence $r_1, r_2, \cdots, r_l$; then $a_1, a_2, \cdots, a_j, a_{j+1}$ is a feasible initial sequence for n if and only if*

Case 1. $r_l = l$. $\qquad l+1 \leqq a_{j+1} \leqq n-j+l-1,$

Case 2. $r_l > l$. $\qquad r_l \leqq a_{j+1} \leqq n-j+l-1.$

Proof. If the upper bound is not satisfied then condition (ii) of Theorem 1 is violated. If the lower bound is not satisfied then condition (i) of Theorem 1 is violated. If the a_{j+1} lies in the indicated ranges then both conditions of Theorem 1 are satisfied.

If $j=n-1$ in the above corollary then $a_n = r_l$. Q.E.D.

3. The algorithm. Given a feasible sequence $a_1, a_2, \cdots, a_n$ we give a procedure for determining the next feasible sequence in our lexicographic listing. The first sequence is $1, 2, \cdots, n-2$, $n-1$, $n-1$ and the last is $n-1$, $n-1$, $n-2, \cdots, 2, 1$. We can test for the last sequence by noting that it is the only feasible sequence such that $a_1 = n-1$.

Suppose that we had a binary tree and wanted to produce the next one in our lexicographic order. Intuitively, we would wish to leave as much of the left part of the tree as possible unchanged, increase the level number of some leaf by adding subtrees to that leaf, and finally to readjust the remaining nodes in the right part of the tree to make their sequence of level numbers as lexicographically small as possible while maintaining feasibility. What actually happens is described next.

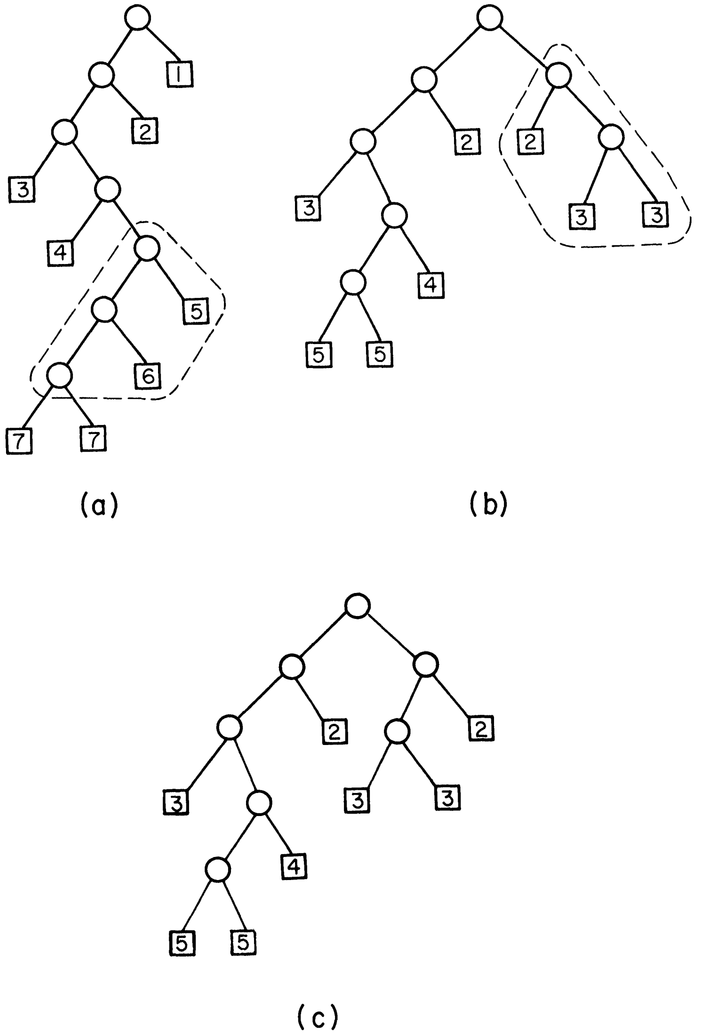
Let $a_{k-1} = a_k = q$ be the rightmost equal pair. Then the leaf with level number $r = a_{k-2}$ will become the father of two leaves at level $r+1$. Let t be the largest integer such that $a_{k-1}, a_k, a_{k+1}, \cdots, a_{k+t} = q, q, q-1, \cdots, q-t$ and $k+t<n$. Also let $p = a_n$. Readjustment of the right part of the tree is possible only if $t>0$. An example of $t>0$ is shown in Fig. 1(a) and the case $t=0$ is shown in Fig. 1(b) and Fig. 1(c). We can think of the two nodes with level numbers a_{k-1}, a_k as becoming the sons of the node with level number a_{k-2}. The nodes which can be readjusted are those enclosed by the dotted line in Fig. 1(a). They always form a lexicographically maximal tree of height t. They are changed into a lexicographically minimal tree of height t and then attached to the node with level number $p = a_n$ as shown in Fig. 1(b).

In terms of sequences, if $k=n$ then the next feasible sequence is $a_1, a_2, \cdots, a_{n-3}, r+1, r+1, q-1$ as shown in Fig. 1(b), (c). Otherwise we have a sequence

$$(1) \qquad \begin{aligned} &a_1, a_2, \cdots, a_{k-3}, a_{k-2}, a_{k-1}, a_k, a_{k+1}, \cdots, a_{k+t}, a_{k+t+1}, \cdots, a_{n-1}, a_n \\ &= a_1, a_2, \cdots, a_{k-3}, r, q, q, q-1, \cdots, q-t, a_{k+t+1}, \cdots, a_{n-1}, p. \end{aligned}$$

If $t \neq 0$ then the next feasible sequence is

$$(2) \qquad \begin{aligned} &a_1 a_2, \cdots, a_{k-3}, r+1, r+1, q-t-q, a_{k+t+1}, \cdots, \\ &a_{n-1}, p+1, p+2, \cdots, p+t, p+t, \end{aligned}$$



(a)

(b)

(c)

FIG. 1

and if $t = 0$ then the next feasible sequence is $a_1, a_2, \cdots, a_{k-3}, r+1, r+1, q-1, a_{k+1}, \cdots, a_{n-1}, p$.

Example. What feasible sequence follows 3, 4, 7, 7, 6, 5, 2, 1? Here $k = 4$, $t = 2$ so according to our algorithm the next sequence is 3, 5, 5, 4, 2, 2, 3, 3. The sequence following 3, 5, 5, 4, 2, 2, 3, 3 is 3, 5, 5, 4, 2, 3, 3, 2.

We now give a detailed PASCAL-like procedure for going from a feasible sequence $a_1, a_2, \cdots, a_n$ to the next feasible sequence in our lexicographic order. This procedure is easily modified to not only produce the sequences but also the binary trees in their more conventional form (i.e., as a set of nodes with right son and left son pointers). The modified procedure would be subject to the same analysis as presented later in this paper for the original procedure.

```
procedure NextTree ($a_1, a_2, \cdots, a_n$: sequence);
    begin
    $k \leftarrow n$; $t \leftarrow 0$;
    while $a_{k-1} \neq a_k$ do $k \leftarrow k - 1$;
    while $k + t < n$ and $a_{k+t} = a_{k+t+1} + 1$ do $t \leftarrow t + 1$;
    $a_{k-1} \leftarrow a_{k-2} + 1$;
    $a_{k-2} \leftarrow a_{k-2} + 1$;
    $a_k \leftarrow a_k - t - 1$;
    if $t \neq 0$ then
        begin
        for $j \leftarrow k + 1$ to $n - t - 1$ do $a_j \leftarrow a_{j+t}$;
        for $j \leftarrow n - t$ to $n - 1$ do $a_j \leftarrow a_n + j - n + t + 1$;
        $a_n \leftarrow a_n + t$;
        end;
    end of NextTree;
```

NextTree can be made slightly more efficient by incorporating the following observation. Let k, t be defined as above for the sequence $a_1, a_2, \cdots, a_n$. If we now apply NextTree to this sequence we get a new rightmost equal pair $a_{k'-1} = a_{k'}$. The observation is that $k' = k + 1$ if $t = 0$ and $k' = n$ if $t \neq 0$.

THEOREM 2. *Given a feasible sequence $a_1, a_2, \cdots, a_n$, the procedure NextTree will produce the next feasible sequence in our lexicographic ordering.*

Proof. The proof consists of two parts: first, showing that (2) is indeed feasible and second, that there is no feasible sequence lying lexicographically between (1) and (2). The other cases ($k = n$, $t = 0$) are easier and their verification is omitted.

Let the left reduced sequence for $a_1, a_2, \cdots, a_{k-2}$ be $r_1, r_2, \cdots, r_l$. After $k - l - 2$ reductions from the left (1) becomes $r_1, r_2, \cdots, r_l, q, q, \cdots, q - t, a_{k+t+1}, \cdots, a_{n-1}, p$. By Lemma 2, $r_l = q - t - 1$ and $q = n - k + l + 1$, and $t + 1$ further reductions from the left yield the sequence

$$(3) \qquad r_1, r_2, \cdots, r_l, q - t - 1, a_{k+t+1}, \cdots, a_{n-1}, p.$$

On the other hand, if we reduce (2) from the left $k - l - 1$ times and reduce it from the right t times we are left with the same sequence (3). Thus the feasibility of (1) implies the feasibility of (2).

We now claim that the largest integer j such that $a_1, a_2, \cdots, a_{j-1}, a_j + m$ is a feasible initial sequence for n (where m is some positive integer) is $k - 2$. If $j > k - 1$ let $a_1, a_2, \cdots, a_{j-1}$ have left reduced sequence $s_1, s_2, \cdots, s_h$. Now $s_1, s_2, \cdots, s_h, a_j, \cdots, a_n$ is a feasible sequence and by Lemma 2 $s_h = a_j = n - j + h$. Since the reduced sequence for $a_1, a_2, \cdots, a_{j-1}, a_j + m$ is $s_1, s_2, \cdots, s_h, a_j + m$, by Theorem 1 the sequence is not a feasible initial sequence for n. If $j = k - 1$ then the left reduced sequence for $a_1, \cdots, a_{k-2}, a_{k-1} + m$ is $r_1, r_2, \cdots, r_l, a_{k-1} + m$ and again we are led to the conclusion that the sequence is not a feasible initial sequence for n. Thus $j = k - 2$. Also, $m = 1$ since (2) is feasible.

Thus the next sequence in our list starts with $a_1, a_2, \cdots, a_{k-3}, r + 1$. According to the corollary to Theorem 1 the next level number is $r + 1$. As noted before $a_1, a_2, \cdots, r + 1, r + 1$ left reduces to $r_1, r_2, \cdots, r_l$ where $r_l = q - l - 1$ and so the next level number is at least $q-l-1$. The next level numbers are seen to be $a_{k+t+1}, \cdots, a_{n-1}$ for if $s_1, s_2, \cdots, s_h$ is the reduced sequence for $a_1, a_2, \cdots, a_j$ where $k+t \leqq j \leqq n-2$ then by Lemma 2, $s_h = a_{j+1}$. Now the reduced sequence for $a_1, a_2, \cdots, a_{n-1}$ is $1, 2, \cdots, p$ and so the next level numbers are $p+1$, $p+2, \cdots, p+t$. The final level number $p+t$ is determined by feasibility. Q.E.D.

If we implement the algorithm in the straightforward manner then the running time of each step is bounded by a constant times $n-k$, the number of positions from the right until an equal adjacent pair is encountered. In the worst case this is $O(n)$ but on the average it is on the order of

$$\frac{1}{b_n} \sum_{j=2}^{n} (j-1) I_{nj}, \tag{4}$$

where b_n is the number of binary trees with n leaves, and I_{nj} is the number of binary trees with n leaves whose first $j-1$ level numbers are strictly increasing and such that the $(j-1)$st and jth level numbers are equal. We defer evaluating (4) until the following section where the necessary counting machinery is set up. There we find that (4) is less than 3 and thus that the average running time of each step is constant. Note that we do not count the time needed to print out the sequences. There is a way of implementing the algorithm such that the worst case running time for a single step of the algorithm is constant. However, it is more complicated and the overall running time increases so it will not be presented.

4. The ranking function. In this section we shall determine the rank of every feasible sequence; and we shall give an unranking algorithm for generating the sequence of a given position. Using the unranking algorithm we have a nice way of producing a random binary tree, drawn from the uniform distribution.

In general a ranking function r for an algorithm generating the elements of some set S is a bijection $r: S \to \{0, 1, \cdots, |S|-1\}$ such that $r(s) = i$ if and only if the ith element (counting from 0) generated by the algorithm is s. We then refer to s as the *rank i* element. If S is a subset of the set of n-tuples of positive integers and the n-tuples are generated in lexicographic order then one method of determining the ranking of an element $a_1, a_2, \cdots, a_n$ is as follows. Determine for each $k = 1, 2, \cdots, n$ the number A_k of elements of S whose first $k-1$ components are $a_1, a_2, \cdots, a_{k-1}$ and whose kth component is one of $1, 2, \cdots, a_k - 1$. The rank of $a_1, a_2, \cdots, a_n$ is then $A_1 + A_2 + \cdots + A_n$. This is the strategy employed below.

The above considerations lead us to ask: How many binary trees are there on n leaves whose first j level numbers are $a_1, a_2, \cdots, a_j$? Let $r_1, r_2, \cdots, r_l$ be the left reduced sequence for $a_1, a_2, \cdots, a_j$. The number of binary trees on n leaves whose first j level numbers are $a_1, a_2, \cdots, a_j$ is equal to the number of binary trees on $n-j+l$ leaves whose first l level numbers are $r_1, r_2, \cdots, r_l$. This is true since if $a_1, a_2, \cdots, a_j, a_{j+1}, \cdots, a_n$ is a feasible sequence then $r_1, r_2, \cdots, r_l$, $a_{j+1}, \cdots, a_n$ is also a feasible sequence. Now consider the path in the tree with level numbers $r_1, r_2, \cdots, r_l, a_{j+1}, \cdots, a_n$ from the root to the leaf with level r_l. There are r_l internal nodes along this path (see Fig. 2). At each of these internal nodes one of two cases occurs: (i) the left son is in the path or (ii) the right son is in the path. In case (i) the right son is the root of some subtree. In case (ii) the left son is a leaf since otherwise the level numbers could not be strictly increasing.

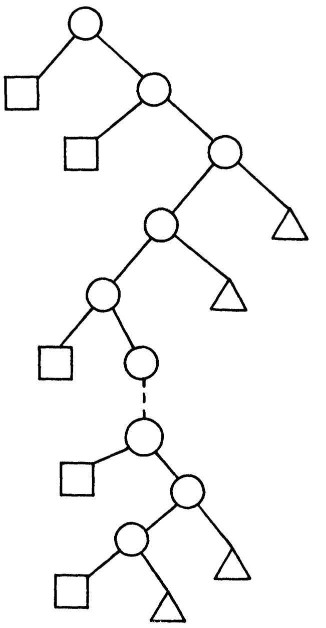


FIG. 2

In the figure each triangle represents a subtree and each square a leaf. The path is the circular nodes. Since $l-1$ of the path nodes have leaves as left sons the number of subtrees is $r_l - l + 1$. Among these $r_l - l + 1$ subtrees we have to distribute the remaining $n-j$ leaves. Thus the number of binary trees on n leaves whose first j level numbers are $a_1, a_2, \cdots, a_j$ is

$$(5) \qquad \sum_{\substack{\nu_1+\cdots+\nu_{r_l-l+1}=n-j\\ \nu_i\geqq 1}} b_{\nu_1}b_{\nu_2}\cdots b_{\nu_{r_l-l+1}}$$

where $b_\nu = \binom{2\nu-2}{\nu-1}\Big/\nu$ is the number of binary trees with ν leaves [2], and ν_i represents the number of leaves in the ith subtree. Note that (5) depends only on $r_l - l$ and j, and not otherwise on the sequence $a_1, a_2, \cdots, a_j$. Introduce the notation

$$(6) \qquad T(n,k) = \sum_{\substack{\nu_1+\cdots+\nu_k=n\\ \nu_i\geqq 1}} b_{\nu_1}b_{\nu_2}\cdots b_{\nu_k};$$

our earlier sum (5) becomes $T(n-j, r_l - l + 1)$. Note that in view of (5) $T(n, k)$ is the number of binary trees on $n+1$ *leaves* whose first level number is k. Using this interpretation we see immediately the initial conditions:

$$(7) \qquad T(n,n)=1 \quad \text{and} \quad T(n,0)=0.$$

We also have an elementary recurrence:

$$(8) \qquad T(n,k)=T(n-1,k-1)+T(n,k+1)$$

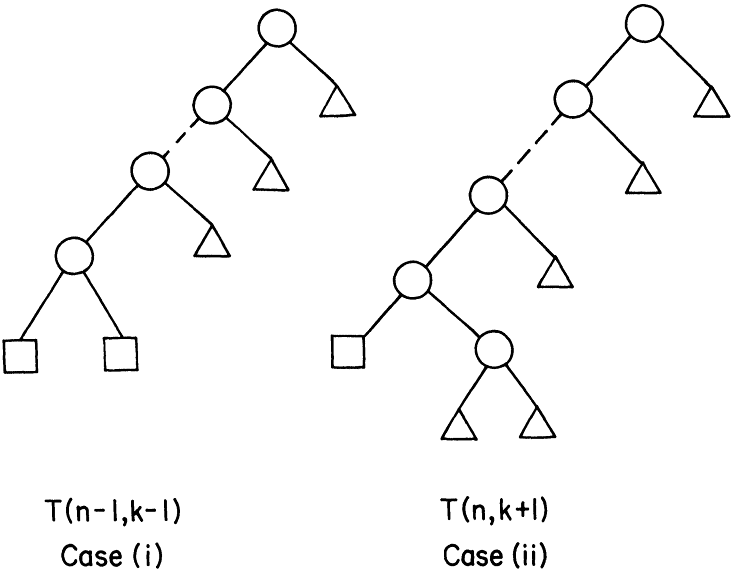


FIG. 3

which is proved by (see Fig. 3) classifying the right brother of the leaf with level k as (i) either being a leaf or (ii) as having a left and a right subtree.

There are $T(n-1, k-1)$ trees falling under the first classification and $T(n, k+1)$ falling under the second classification. Using (7) and (8) we can easily construct a table of the $T(n, k)$; see Table 1. Note that this gives us another way of determining b_n since $T(n, 1)=b_n$. We also have a simple expression for the $T(n, k)$.

THEOREM 3.

$$T(n, k)=\frac{k}{2n-k}\binom{2n-k}{n-k}=\binom{2n-k}{n-k}-2\binom{2n-k-1}{n-k-1}.$$

Proof. The second equality is a simple verification. Since the relations (7) and (8) uniquely define the $T(n, k)$ we need only verify that the

TABLE 1

n \ k	1	2	3	4	5	6	7	8
1	1							
2	1	1						
3	2	2	1					
4	5	5	3	1				
5	14	14	9	4	1			
6	42	42	28	14	5	1		
7	132	132	90	48	20	6	1	
8	429	429	297	165	75	27	7	1

$\binom{2n-k}{n-k}-2\binom{2n-k-1}{n-k-1}$ satisfy them also. Equation (7) is immediate and (8) follows at once from the basic binomial identity:

$$\binom{2n-k-1}{n-k}-2\binom{2n-k-2}{n-k-1}+\binom{2n-k-1}{n-k-1}-2\binom{2n-k-2}{n-k-2}$$
$$=\binom{2n-k}{n-k}-2\binom{2n-k-1}{n-k-1}. \quad \text{Q.E.D.}$$

Now let us determine the A_j for $j=1, 2, \cdots, n$. Note that $A_n=0$ since a_n is uniquely determined from the $a_1, \cdots, a_{n-1}$ by the relation $\sum_{i=1}^{n} 2^{-a_i}=1$, and so we can assume $j<n$. Let m be the minimal integer such that $a_1, a_2, \cdots, a_{j-1}, m$ is a feasible initial sequence for n and suppose $r_1, r_2, \cdots, r_l$ is the left reduced sequence for $a_1, a_2, \cdots, a_{j-1}$. Then by the corollary to Theorem 1 A_j is $\sum_{i=0}^{a_j-m-1}$ (number of binary trees with n leaves whose first j level numbers are $a_1, a_2, \cdots, a_{j-1}, m+i)=\sum_{i=0}^{a_j-m-1} T(n-j, m-l+i)$. Note that even if $r_1, \cdots, r_l$, $m+i$ left reduces to $s_1, \cdots, s_h$ then s_h-h+1 is still $m+i-l$. The following example illustrates how these calculations should be arranged.

Example. What is the ranking of the binary tree with level numbers 3, 5, 5, 4, 2, 3, 3, 2? We first calculate the left reduced sequences for 3; 3, 5; 3, 5, 5 etc. At each step we use the reduced sequence from the previous step to calculate the present reduced sequence. In this way at most $n-1$ reductions are performed. See Table 2. Thus the rank is $T(7,1) = T(7,2) + T(6,2) + T(6,3) + T(2,1) = 132+132+42+28+1=335$.

A detailed procedure which returns the rank of a feasible sequence $a_1, a_2, \cdots, a_n$ is given below.

```
procedure Rank ($a_1, a_2, \cdots, a_n$: sequence);
    begin
    $rank \leftarrow 0$; $m \leftarrow 1$;
    $l \leftarrow 0$; $r_0 \leftarrow 0$;
    for $j \leftarrow 1$ to $n-1$ do
        begin
        for $i \leftarrow 0$ to $a_j-m-1$ do $rank \leftarrow rank + T(n-j, m-l+i)$;
        $l \leftarrow l+1$; $r_l \leftarrow a_j$;
        while $r_{l-1}=r_l$ do
            begin
            $l \leftarrow l-1$
            $r_l \leftarrow r_l-1$;
            end;
        $m \leftarrow$ if $r_l=l$ then $l+1$ else $r_l$;
        end
end of Rank.
```

We can reverse this process to find the rank i binary tree on n leaves for any i.

Example. What is the 300th binary tree with 8 leaves? Since $T(7,1)+T(7,2)<300 \leqq T(7,1)+T(7,2)+T(7,3)$ the first level number is $1+2=3$,

TABLE 2

j	$a_1, \cdots, a_{j-1}$	$r_1, \cdots, r_l$	m	$a_j - m - 1$
1			1	1
2	3	3	3	1
3	3, 5	3, 5	5	−1
4	3, 5, 5	3, 4	4	−1
5	3, 5, 5, 4	2	2	−1
6	3, 5, 5, 4, 2	1	2	0
7	3, 5, 5, 4, 2, 3	1, 3	3	−1

where 1 is the minimal possible first level number and 2 is the number of terms whose sum is less than 300. Now $300-(132+132)=36$ and the minimal level number following 3 is 3. Since $36 < T(6, 2) = 42$ the first two level numbers are 3, 3 which left reduces to 2. The minimal level number following 2 is 2 and since $T(5, 1)+T(5, 2)<36 \leqq T(5, 1)+T(5, 2)+T(5, 3)$ the next level number is $2+2=4$. Continuing in this fashion we find the sequence of level numbers 3, 3, 4, 6, 6, 5, 3, 1.

We now present the general algorithm. Given the number of leaves n and rank of a binary tree the procedure Unrank will generate the sequence of n level numbers having that rank.

```
procedure Unrank ($n$, $rank$: integer);
    begin
    $r_0 \leftarrow 0$; $m \leftarrow 1$; $l \leftarrow 0$;
    for $j \leftarrow 1$ to $n-1$ do
        begin
        $i \leftarrow 0$; $tsum \leftarrow 0$;
        repeat
            $tsum \leftarrow tsum + T(n-j, m-l+i)$;
            $i \leftarrow i+1$;
        until $rank < tsum$;
        $rank \leftarrow rank - tsum + T(n-j, m-l+i-1)$;
        $a_j \leftarrow m+i-1$;
        $l \leftarrow l+1$; $r_l \leftarrow a_j$;
        while $r_{l-1} = r_l$ do
            begin
            $l \leftarrow l-1$;
            $r_l \leftarrow r_l - 1$;
            end;
        $m \leftarrow$ if $r_l = l$ then $l+1$ else $r_l$;
        end;
    $a_n \leftarrow r_l$;
    end of Unrank.
```

We now present linear algorithms for converting a feasible sequence into a binary tree and vice versa. Let us label the nodes of the binary tree with the integers 1 through $2n-1$. Nodes 1 through n will be the leaves, $n+1$ through $2n-1$ the internal nodes, and $2n-1$ the root. Left $[i]$ and Right $[i]$ will denote the left and right sons of node i. Algorithm ComputeDepths below takes a tree in such a representation and returns the level sequence $a_1, a_2, \cdots, a_n$.

ComputeDepths:

```
begin integer CurrentDepth, Root;
procedure CDFS ($j$: integer);
    begin
    if $j \leqq n$ then $a_j \leftarrow$ CurrentDepth
    else begin
        CurrentDepth $\leftarrow$ CurrentDepth $+ 1$;
        CDFS (Left $[j]$);
        CDFS (Right $[j]$);
        CurrentDepth $\leftarrow$ CurrentDepth $- 1$;
        end;
    end of CDFS;
CurrentDepth $\leftarrow 0$;
Root $\leftarrow 2n-1$;
CDFS (Root);
end of ComputeDepths.
```

Algorithm MakeTree below produces a binary tree given the feasible level sequence $a_1, a_2, \cdots, a_n$.

MakeTree:

```
begin integer CurrentDepth, Root, $j$, NextInternalNode;
procedure MDFS ($v$: integer);
    begin
    CurrentDepth $\leftarrow$ CurrentDepth $+ 1$;
    if CurrentDepth $= a_j$
    then begin
        Left $[v] \leftarrow j$;
        $j \leftarrow j+1$;
        end
    else begin
        NextInternalNode $\leftarrow$ NextInternalNode $- 1$;
        Left $[v] \leftarrow$ NextInternalNode;
        MDFS (NextInternalNode);
        end;
    if CurrentDepth $= a_j$
    then begin
        Right $[v] \leftarrow j$;
        $j \leftarrow j+1$;
        end
    else begin
        NextInternalNode $\leftarrow$ NextInternalNode $- 1$;
        Right $[v] \leftarrow$ NextInternalNode;
```

```
            MDFS (NextInternalNode);
            end;
        CurrentDepth ← CurrentDepth − 1;
        end of MDFS;
    for $j \leftarrow n$ down to 1 do Left [$j$] ← Right [$j$] ← 0;
    Root ← NextInternalNode ← $2n-1$;
    CurrentDepth ← 0;
    MDFS (Root);
    end of MakeTree;
```

Now, let us prove the remarks at the end of § 3. We first prove a lemma.

LEMMA 4. *The number of binary trees on n leaves whose first j level numbers are strictly increasing is* $T(n, j+1)$.

Proof. The jth level number a_j must lie in the range $j \leqq a_j \leqq n-1$. Once we know what a_j is we can consider the path from the root to a_j as in Fig. 2. This time however we are free to choose which $j-1$ of the a_j-1 internal nodes have leaves as left sons. Thus we wish to evaluate

$$\sum_{k=j}^{n-1} \binom{k-1}{j-1} T(n-j, k-j+1).$$

Recall

$$\begin{aligned} T(n-j, k-j+1) &= \frac{k-j+1}{2n-j-k-1}\binom{2n-j-k-1}{n-k-1} \\ &= \binom{2n-j-k-1}{n-k-1} - 2\binom{2n-j-k-2}{n-k-2}. \end{aligned}$$

Now

$$\begin{aligned} \sum_{k=j}^{n-1} \binom{k-1}{j-1}\binom{2n-j-k-1}{n-k-1} &= \sum_{k\geqq 0} \binom{n-k-2}{j-1}\binom{n+k-j}{k} \qquad (k \leftarrow n-k-1) \\ &= \sum_{k\geqq 0} \binom{n-k-2}{n-p}\binom{p+k-1}{k} \qquad (p \leftarrow n-j+1) \\ &= \sum_{k\geqq 0} \binom{(n-2)-k}{(n-2)-(p-2)}\binom{p+k-1}{k}. \end{aligned}$$

Which is of the form of a Vandermonde convolution (Riordan [3, p. 8, (3b)]) namely:

$$\sum_k \binom{n-k}{n-m}\binom{p+k-1}{k} = \binom{n+p}{m}.$$

So our sum is

$$\binom{n-2+p}{p-2} = \binom{2n-j-1}{n-j-1}.$$

Also

$$\sum_{k=j}^{n-1}\binom{k-1}{j-1}\binom{2n-j-k-2}{n-k-2}=\binom{2n-j-2}{n-j-2}.$$

Thus

$$\begin{aligned}\sum_{k=j}^{n-1}\binom{k-1}{j-1}T(n-j,k-j+1)&=\binom{2n-j-1}{n-j-1}-2\binom{2n-j-2}{n-j-2}\\&=T(n,j+1).\quad \text{Q.E.D.}\end{aligned}$$

COROLLARY. *The number of binary trees on n leaves whose first $j-1$ level numbers are strictly increasing and such that the $j-1$-st and j-th level numbers are equal is $T(n-1,j-1)$.*

Proof. By Lemma 4 the required answer is

$$T(n,j)-T(n,j+1)=T(n-1,j-1).\quad \text{Q.E.D.}$$

Thus $I_{nj}=T(n-1,j-1)$ and to estimate (4) we first evaluate $\sum_{j=2}^{n}(j-1)I_{nj}$.

LEMMA 5.

$$\sum_{j=2}^{n}(j-1)I_{nj}=T(n+1,3).$$

Proof. We first note that

$$\sum_{k=0}^{n}\binom{r+k}{k}=\binom{r+n+1}{n}$$

(Knuth [2, p. 54, eq. (10)]. Now

$$\begin{aligned}\sum_{j=2}^{n}(j-1)T(n-1,j-1)&=\sum_{k=1}^{n-1}kT(n-1,k)\\&=\sum_{k=1}^{n-1}k\binom{2n-k-2}{n-k-1}-2\sum_{k=1}^{n-1}k\binom{2n-k-3}{n-k-2}\\&=\sum_{k=1}^{n-1}k\binom{2n-k-2}{n-k-1}-2\sum_{k=2}^{n}(k-1)\binom{2n-k-2}{n-k-1}\\&=\sum_{k=1}^{n-1}(2-k)\binom{2n-k-2}{n-k-1}\qquad\qquad (k\leftarrow n-k-1)\\&=\sum_{k=0}^{n-2}(3-n+k)\binom{n+k-1}{k}\\&=\sum_{k=0}^{n-2}k\binom{n+k-1}{k}-(n-3)\binom{2n-2}{n-2}\\&=n\sum_{k=0}^{n-2}\binom{n+k-1}{k-1}-(n-3)\binom{2n-2}{n-2}\end{aligned}$$

$$= n\binom{2n-2}{n-3} - (n-3)\binom{2n-2}{n-2}$$

$$= \frac{3}{n+1}\binom{2n-2}{n-2} = \frac{3}{2n-1}\binom{2n-1}{n-2}$$

$$= T(n+1, 3) \quad \text{Q.E.D.}$$

Thus

$$\frac{1}{b_n}\sum_{j=2}^{n}(j-1)I_{nj} = \frac{\dfrac{3}{n+1}\dbinom{2n-2}{n-2}}{\dfrac{1}{n}\dbinom{2n-2}{n-1}}$$

$$= 3\frac{n-1}{n+1} < 3.$$

Thus the average amount of time for each step is indeed constant.

Acknowledgment. The authors wish to express their gratitude to both referees for their constructive comments and suggestions. The algorithms ComputeDepths and MakeTree are due to one of the referees. The authors have learned that Tony Trojanowski has an unpublished paper on ranking binary trees. His ranking scheme is different from ours, and provides an alternative approach to the problem.

REFERENCES

[1] T. C. Hu and A. C. Tucker, *Optimal computer search trees and variable-length alphabetical codes*, SIAM J. Appl. Math., 21 (1971), pp. 514–532.

[2] D. E. Knuth, *Fundamental Algorithms*, vol. 1, second ed., Addison-Wesley, Reading, MA, 1973.

[3] J. Riordan, *Combinatorial Identities*, John Wiley, New York, 1968.