SIAM J. COMPUT.
Vol. 3, No. 1, March 1974

THE COMPUTING TIME OF THE EUCLIDEAN ALGORITHM*

GEORGE E. COLLINS†

Abstract. The minimum, maximum and average computing times of the classical Euclidean algorithm are derived. With positive integer inputs of lengths m and n, and with output (greatest common divisor) of length k, $m \geqq n \geqq k$, the minimum is shown to be codominant with $n(m - n + 1) + k(n - k + 1)$, while both the maximum and the average are shown to be codominant with $n(m - k + 1)$.

Key words. Euclidean algorithm, greatest common divisor, arithmetic algorithms, algorithm analysis

1. Introduction. Knuth [11], Dixon [6], [7] and Heilbronn [8] have recently investigated in considerable depth the average number of divisions performed in the Euclidean algorithm for integers. Although many interesting questions remain unanswered, the relatively elementary result of Dixon in [7] already suffices to completely determine the average computing time of the Euclidean algorithm to within a constant factor, which is in any case dependent on the particular computer used and inessential details of the implementation. Such a determination of the average computing time of the Euclidean algorithm is the main result of the present paper. The maximum and minimum computing times of the Euclidean algorithm for integers will also be derived since, although their determination is quite elementary, they have apparently not previously been published. These computing times are all derived as functions of three variables, namely the lengths of the two inputs and the length of the resulting g.c.d. (greatest common divisor). Previous results on the computing time of the Euclidean algorithm ([2] and [11, § 4.5.2, Exercise 30]) have been limited to upper bounds on the maximum computing time.

2. Dominance and codominance. The relations of dominance and codominance between real-valued functions were introduced in [3], where they were used in the analysis of the computing time of an algorithm for polynomial resultant calculation. The related concepts and notation have subsequently been adopted by several authors, for example, Brown [1], Heindel [9] and Musser [12]. The definitions and some fundamental properties will be repeated here since they will not yet be familiar to many readers.

If f and g are real-valued functions defined on a common domain S, we say that f *is dominated by* g, and write $f \preceq g$, in case there is a positive real number c such that $f(x) \leqq c \cdot g(x)$ for all $x \in S$. We also say that g *dominates* f, and write $g \succeq f$. Dominance is clearly a reflexive and transitive relation. It is important to note that the definition is not restricted to functions of one variable since the elements of S may be n-tuples.

* Received by the editors February 12, 1973, and in revised form September 17, 1973. This work was supported by the National Science Foundation under Grant GJ-30125X, by the Wisconsin Alumni Research Foundation, and in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under Grant SD-183.

† Computer Sciences Department, University of Wisconsin, Madison, Wisconsin 53706.

Knuth [10, pp. 104–108] defines $f(x) = O(g(x))$ in case there is a positive constant c such that $|f(x)| \leqq c \cdot |g(x)|$. As long as one is dealing only with non-negative-valued functions, this formally coincides with the above definition of $f \preceq g$. Although Knuth implies that this definition is applicable only when f and g are functions of one variable, he in fact uses it for functions of more than one variable (e.g., [11, p. 388]) in a manner which is consistent with our definition. Thus dominance is apparently a new notation and terminology but not a new concept. Although Knuth discussed at length the logical weaknesses of the O-notation, he chose not to abandon it in favor of the more natural notation of an order relation.

If $f \preceq g$ and $g \preceq f$, then we say that f and g are *codominant*, and write $f \sim g$. Codominance is clearly an equivalence relation. If $f \preceq g$ but not $g \preceq f$, then we say that f *is strictly dominated by* g, and write $f \prec g$. We may also say that g *strictly dominates* f, and write $g \succ f$. Strict dominance is clearly irreflexive and transitive. Whereas the O-notation has no counterparts for the codominance and strict dominance relations, it will become apparent that these are important concepts in algorithm computing time analyses. Furthermore, the O-notation has a somewhat different meaning in asymptotic analysis than the one used by Knuth (see, e.g., [5]).

If f and g are functions defined on S and S_1 is a subset of S, it will often be convenient to write $f \preceq g$ *on* S_1 in case $f_1 \preceq g_1$, where f_1 and g_1 are the functions f and g restricted to S_1. Also, if $S \subseteq S_1 \times \cdots \times S_n$, a Cartesian product, we will denote by f_a the function f restricted to $(\{a\} \cap S_2 \times \cdots \times S_n) \cap S$; that is, $f_a(x_2, \cdots, x_n) = f(a, x_2, \cdots, x_n)$ for $(a, x_2, \cdots, x_n) \in S$. Similarly we may *fix* any other of the n variables of f.

Dominance and codominance have the following fundamental properties, most of which were listed by Musser in [12].

THEOREM 1. *Let f, f_1, f_2, g, g_1 and g_2 be nonnegative real-valued functions on S, and let c be a positive real number. Then*

(a) $f \sim cf$;
(b) *if* $f_1 \preceq g_1$ *and* $f_2 \preceq g_2$, *then* $f_1 + f_2 \preceq g_1 + g_2$ *and* $f_1 f_2 \preceq g_1 g_2$;
(c) *if* $f_1 \preceq g$ *and* $f_2 \preceq g$, *then* $f_1 + f_2 \preceq g$;
(d) $\max(f, g) \sim f + g$;
(e) *if* $1 \preceq f$ *and* $1 \preceq g$, *then* $f + g \preceq fg$;
(f) *if* $1 \preceq f$, *then* $f \sim f + c$;
(g) *if* $S \subseteq S_1 \times \cdots \times S_n$ *and* $a \in S_1$, *then* $f \preceq g$ *implies* $f_a \preceq g_a$;
(h) *if* $S = S_1 \cup S_2$, *then* $f \preceq g$ *on* S_1 *and* $f \preceq g$ *on* S_2 *implies* $f \preceq g$ *on* S.

Proof. These properties follow immediately from the definition, except for (e). To prove (e), apply (b) to $f \preceq f$ and $1 \preceq g$, obtaining $f \preceq fg$. Similarly $g \preceq fg$, so $f + g \preceq fg$ by (c). ∎

3. Computing time functions. Let A be any algorithm, and let S be the set of all valid inputs to A. In general, S will be denumerable, and its elements may be n-tuples. We associate with A a computing time function t_A defined on S, the positive integer $t_A(x)$ being the number of basic operations performed by the algorithm A when presented with the input x. This assumes that the algorithm

is unambiguously specified in terms of some finite set of basic operations. Changing the set of basic operations (as in reprogramming the algorithm for a different computer) will result in changing the computing time function t_A. Alternatively, we could take the view that this represents a change in the algorithm. However, if B_1 and B_2 are two sets of basic operations such that each operation in B_1 can be performed by a fixed sequence of operations in B_2, and vice versa, then the computing time functions associated with B_1 and B_2 for any algorithm A are codominant, and we will concern ourselves only with the codominance equivalence class of t_A. Thus the choice of basic operations is somewhat arbitrary. We assume a choice which is consistent with any of the existing, or conceivable, random access digital computers but, in order to avoid the triviality of finiteness, with a memory which is indefinitely expandable.

The function t_A is frequently too complex to be of interest for direct study. Instead, we ordinarily decompose S into a disjoint union $S = \bigcup_{n=1}^{\infty} S_n$, where each S_n is a nonempty finite set, S being a denumerable set. The choice of decomposition is made on the basis of some prior knowledge or some conjecture about the general behavior of t_A. Relative to a decomposition $\mathscr{S} = \{S_1, S_2, S_3, \cdots\}$ of S we define maximum, minimum and average computing time functions, t_A^+, t_A^- and t_A^*, on $\mathscr{S}$ as follows, where $|S_n|$ denotes the number of elements of S_n:

$$t_A^+(S_n) = \max_{x \in S_n} t_A(x), \tag{1}$$

$$t_A^-(S_n) = \min_{x \in S_n} t_A(x), \tag{2}$$

$$t_A^*(S_n) = \left\{ \sum_{x \in S_n} t_A(s) \right\} / |S_n|. \tag{3}$$

As an illustration, and in preparation for our analysis of the Euclidean algorithm, let us consider the computing times of the classical algorithms for arithmetic operations, that is, addition, subtraction, multiplication and division, of arbitrarily large integers. We assume that all integers are represented in radix form relative to an integral base $\beta \geqq 2$, as discussed by Knuth in [11, § 4.3]. We know that the computing times of these algorithms depend on the lengths of the inputs.

Following Musser [12] we denote by $L_\beta(a)$ the *β-length* of the integer a, that is, the number of digits in the radix form of a relative to the base β. If $\lceil x \rceil$ is the ceiling function of x, the least integer greater than or equal to x, and $\lfloor x \rfloor$ is the floor function of x, the greatest integer less than or equal to x, we have

$$L_\beta(a) = \lceil \log_\beta (|a| + 1) \rceil = \lfloor \log_\beta |a| \rfloor + 1 \tag{4}$$

for $a \neq 0$, and we define $L_\beta(0) = 1$.

In most contexts the base β is fixed, and we write simply $L(a)$ for the *length* of a. The omission of the subscript is further justified by the observation that, γ being any other base, we have

$$L_\beta \sim L_\gamma, \tag{5}$$

where L_β and L_γ are functions defined on the set I of all integers. In fact, we can use the definition (4) when a is any real number, and we then have

$$L_\beta(a) \sim \ln(|a| + 2) \quad \text{on } R, \tag{6}$$

where ln is the natural logarithm and R is the set of all real numbers, and (6) clearly implies (5). The length function also has the following easily verified fundamental properties (here I is the set of integers):

$$L(a \pm b) \preceq L(a) + L(b) \quad \text{for } a, b \in I, \tag{7}$$

$$L(ab) \sim L(a) + L(b) \quad \text{for } a, b \in I - \{0\}, \tag{8}$$

$$L([a/b]) \sim L(a) - L(b) + 1 \quad \text{for } a, b \in I \quad \text{and} \quad |a| \geqq |b| > 0, \tag{9}$$

where $[x] = \lfloor x \rfloor$ for $x \geqq 0$ and $[x] = \lceil x \rceil$ for $x < 0$.

THEOREM 2. (a) *Let* $S = \{(a_1, \cdots, a_n) : n \geqq 1$ *and* $a_1, \cdots, a_n \in I\}$. *Then* $L(\prod_{i=1}^n a_i) \leqq \sum_{i=1}^n L(a_i)$ *on* S.

(b) *Let* $\bar{S} = \{(a_1, \cdots, a_n) : n \geqq 1$ *and* $a_1, \cdots, a_n \in I - \{-1, 0, 1\}\}$. *Then* $L(\prod_{i=1}^n a_i) \sim \sum_{i=1}^n L(a_i)$ *on* $\bar{S}$.

Proof. $L(ab) \leqq L(a) + L(b)$ for $a, b \in I$, so $L(\prod_{i=1}^n a_i) \leqq \sum_{i=1}^n L(a_i)$ by induction on n, proving (a). To prove (b), assume first that $2 \leqq |a_i| < \beta$ for $1 \leqq i \leqq n$. Then

$$L\left(\prod_{i=1}^n a_i\right) \geqq \log_\beta \prod_{i=1}^n |a_1| = (\log_\beta 2) \log_2 \prod_{i=1}^n |a_i|$$

$$\geqq (\log_\beta 2) \log_2 2^n = (\log_\beta 2) n = (\log_\beta 2) \sum_{i=1}^n L(a_i),$$

so $\sum_{i=1}^n L(a_i) \leqq (\log_2 \beta) L(\prod_{i=1}^n a_i)$. Next, assume $L_\beta(a_i) \geqq 2$ for $1 \leqq i \leqq n$, and let $l_i = L_\beta(a_i)$. Then

$$L\left(\prod_{i=1}^n a_i\right) \geqq \log_\beta \left(\prod_{i=1}^n |a_i|\right) \geqq \log_\beta \left(\prod_{i=1}^n \beta^{l_i - 1}\right)$$

$$= \sum_{i=1}^n (l_i - 1) \geqq \left(\sum_{i=1}^n l_i\right) \Big/ 2,$$

so $\sum_{i=1}^n L(a_i) \leqq 2L(\prod_{i=1}^n a_i)$.

Combining these two cases, we assume $L(a_i) = 1$ for $1 \leqq i \leqq m$ and $L(a_i) \geqq 2$ for $m + 1 \leqq i \leqq n$. Then

$$\sum_{i=1}^n L(a_i) \leqq (\log_2 \beta) L\left(\prod_{i=1}^m a_i\right) + 2L\left(\prod_{i=m+1}^n a_i\right)$$

$$\leqq 2(\log_2 \beta)\left\{L\left(\prod_{i=1}^m a_i\right) + L\left(\prod_{i=m+1}^n a_i\right)\right\} \leqq 4(\log_2 \beta) L\left(\prod_{i=1}^n a_i\right)$$

since $L(a) + L(b) \leqq 2L(ab)$ for $a, b \in I - \{0\}$. ∎

As an immediate corollary of Theorem 2, we have

$$L(a^b) \sim bL(a) \quad \text{for } a, b \in I, \quad |a| \geqq 2 \quad \text{and} \quad b > 0. \tag{10}$$

If A, M and D are the classical algorithms for addition (or subtraction), multiplication and division, respectively, as described in [11, § 4.3], then we clearly have

$$t_A(a,b) \sim L(a) + L(b) \quad \text{for } a,b \in I - \{0\}, \tag{11}$$

$$t_M(a,b) \sim L(a)\cdot L(b) \quad \text{for } a,b \in I - \{0\}, \tag{12}$$

$$t_D(a,b) \sim L(b)\cdot L([a/b]) \quad \text{for } a,b \in I \text{ and } |a| > |b| > 0. \tag{13}$$

Thus, for these algorithms, the natural decomposition of the set $S = \{(a,b): a,b \in I\}$ consists of the sets $S_{m,n} = \{(a,b): L(a) = m \text{ and } L(b) = n\}$. If we write $t^+(m,n)$ in place of $t^+(S_{m,n})$, and similarly for t^- and t^*, then from (11), (12) and (13), and using (9), we have

$$t_A^+(m,n) \sim t_A^-(m,n) \sim t_A^*(m,n) \sim m + n, \tag{14}$$

$$t_M^+(m,n) \sim t_M^-(m,n) \sim t_A^*(m,n) \sim mn, \tag{15}$$

$$t_D^+(m,n) \sim t_D^-(m,n) \sim t_D^*(m,n) \sim n(m - n + 1) \quad \text{for } m \geqq n. \tag{16}$$

Thus for these algorithms the maximum, minimum and average computing times all coincide. This will not be the case for the Euclidean algorithm, to which we now turn.

4. The maximum and minimum computing times. For simplicity, and without loss of generality, we will consider the following version of the Euclidean algorithm, for which the permissible inputs are the pairs (a,b) of positive integers with $a \geqq b$. The output of the algorithm is the positive integer $c = \text{g.c.d.}(a,b)$.

ALGORITHM E.

Step 1. [Initialize.] $c \leftarrow a$; $d \leftarrow b$.

Step 2. [Divide.] Compute the quotient q and remainder r such that $c = dq + r$ and $0 \leqq r < d$, using Algorithm D (classical algorithm for division).

Step 3. [Test for end.] $c \leftarrow d$; $d \leftarrow r$; if $d \neq 0$, go to Step 2.

Step 4. Return.

This algorithm computes two sequences, $(a_1, a_2, \cdots, a_{l+2})$ and $(q_1, q_2, \cdots, q_l)$, such that $a_1 = a$, $a_2 = b$, $a_i = q_i a_{i+1} + a_{i+2}$ with $0 \leqq a_{i+2} < a_{i+1}$ for $1 \leqq i \leqq l$, and $a_{l+2} = 0$. $a_1, \cdots, a_{l+1}$ are the successive values assumed by the variable c, and $q_1, \cdots, q_l$ are the successive values assumed by the variable q. $(a_1, \cdots, a_{l+2})$ is called the *remainder sequence* of (a,b) and $(q_1, \cdots, q_l)$ is called the *quotient sequence* of (a,b). Steps 2 and 3 are each executed l times; this is the number of divisions performed, which we denote by $D(a,b)$.

By (13), the computing time for the ith execution of Step 2 is $\sim L(q_i)L(a_{i+1})$. The computing time for the ith execution of Step 3 is certainly dominated by $L(a_{i+1})$ since it at most requires copying the digits of a_{i+1} and a_{i+2}. In an implementation of the algorithm in which a large integer is represented by the list of its digits (e.g., [4]), such copying is unnecessary, and the computing time for

each execution of Step 3 is ~ 1. For the same reason, we will assume that the single executions of Steps 1 and 4 have computing times ~ 1. We then have

$$t_E(a,b) \sim \sum_{i=1}^{l} L(q_i) \cdot L(a_{i+1}). \tag{17}$$

If instead we were to assume that copying is required in Steps 1 and 3, (17) would still hold after adding $L(a_1)$ to the right-hand side. But $L(a_1) \sim L(q_1) + L(a_2) \preceq L(q_1)L(a_2)$, so (17) holds in any case.

From (17) we will derive the maximum, minimum and average computing times of Algorithm E, by analyzing the possible distributions of values of the a_i and q_i, obtaining the codominance equivalence classes of these computing times as functions of $L(a)$, $L(b)$ and $L(c)$. Thus we consider the decomposition of S into the sets

$$S_{m,n,k} = \{(a,b) : L(a) = m \text{ and } L(b) = n \text{ and } L(\text{g.c.d.}(a,b)) = k\}, \tag{18}$$

with $m \geqq n \geqq k \geqq 1$. We may verify that each set $S_{m,n,k}$ is nonempty as follows. If $m = k$, then $(\beta^{m-1}, \beta^{m-1}) \in S_{m,n,k}$. If $m > k$, let $a = \beta^{m-1} + \beta^{k-1}$ and $b = \beta^{n-1}$. Then $c = \text{g.c.d.}(a,b) = \beta^{k-1}$, $L(a) = m$, $L(b) = n$ and $L(c) = k$, so $(a,b) \in S_{m,n,k}$. As above, we will write $t_E^+(m,n,k)$ in place of $t_E^+(S_{m,n,k})$, and similarly for t_E^- and t_E^*.

THEOREM 3. $t_E^+(m,n,k) \preceq n(m-k+1)$.

Proof. Since $b = a_2 > a_3 > \cdots > a_{l+1}$, we have by (17) that

$$t_E(a,b) \preceq L(b) \sum_{i=1}^{l} L(q_i). \tag{19}$$

Since $L(q_i) \leqq L(q_i + 1)$ and $q_l \geqq 2$ we obtain, by Theorem 2,

$$\sum_{i=1}^{l} L(q_i) \preceq L\left(q_l \prod_{i=1}^{l-1} (q_i + 1)\right). \tag{20}$$

Since $a_i = q_i a_{i+1} + a_{i+2} > q_i a_{i+2} + a_{i+2}$, we have $q_i + 1 < a_i/a_{i+2}$ for $i < l$, and hence $\prod_{i=1}^{l-1}(q_i + 1) < a_1 a_2 / a_l a_{l+1}$. Combining this with $q_l = a_l/a_{l+1}$ yields

$$q_l \prod_{i=1}^{l-1} (q_i + 1) \leqq ab/c^2. \tag{21}$$

Since $L(ab/c^2) \leqq L(a^2/c^2) \sim L(a/c) \sim L(a) - L(c) + 1$, (19), (20) and (21) yield

$$t_E(a,b) \preceq L(b)\{L(a) - L(c) + 1\}, \tag{22}$$

from which Theorem 3 is immediate. ∎

We now proceed to prove that $t_E^+(m,n,k) \sim n(m-k+1)$, for which purpose we need the following two theorems.

THEOREM 4. $t_E(a,b) \succeq D(a,b)\{D(a,b) + L(\text{g.c.d.}(a,b))\}$.

Proof. Let $(q_1, \cdots, q_l)$ and $(a_1, \cdots, a_{l+2})$ be the quotient and remainder sequences of (a,b), $c = \text{g.c.d.}(a,b)$ and $k = L(c)$. By (17),

$$t_E(a,b) \succeq \sum_{i=1}^{l} L(a_i). \tag{23}$$

Since $a_{l+2} = 0$, $a_{l+1} = c$ and $a_i = q_i a_{i+1} + a_{i+2} \geqq a_{i+1} + a_{i+2}$, a simple induction shows that $a_{l+2-i} \geqq cF_i$, where F_i is the ith term of the Fibonacci sequence defined by $F_0 = 0$, $F_1 = 1$ and $F_{i+2} = F_i + F_{i+1}$. But [10, p. 82] $F_{i+1} \geqq \phi^i/\sqrt{5}$, where $\phi = (1 + \sqrt{5})/2$, and $\phi^2 > \sqrt{5}$ so $F_{i+3} > \phi^i$. Hence

$$\sum_{i=1}^{l} L(a_i) \geqq \sum_{i=2}^{l+1} \log_\beta (cF_i) \geqq l(\log_\beta c) + \sum_{i=1}^{l-2} \log_\beta \phi^i \geqq l(\log_\beta c) + \binom{l-2}{2}(\log_\beta \phi).$$

So for $k \geqq 2$ and $l \geqq 4$,

$$\sum_{i=1}^{l} L(a_i) \geqq \frac{1}{2}kl + \frac{1}{16}(\log_\beta \phi)l^2 \succeq kl + l^2,$$

while for $k = 1$ and $l \geqq 4$,

$$\sum_{i=1}^{l} L(a_i) \geqq \frac{1}{16}(\log_\beta \phi)l^2 \succeq l^2 \sim kl + l^2.$$

For $l \leqq 3$, $\sum_{i=1}^{l} L(a_i) \geqq L(c) = k \sim kl + l^2$. So by Theorem 1, part (h), $\sum_{i=1}^{l} L(a_i) \geqq kl + l^2$ for all k and l, proving the theorem, since $l = D(a, b)$. ∎

By an application of Theorem 4, together with an elementary construction utilizing the generalized Fibonacci sequences $F^{(h)}$ defined by $F_0^{(h)} = 1$, $F_1^{(h)} = h$ and $F_{i+2}^{(h)} = F_i^{(h)} + F_{i+1}^{(h)}$, one can obtain a proof that $t_E^+(m, m, k) \succeq n(m - k + 1)$. However, we will abstain from this construction, obtaining the result instead as a corollary of our analysis, in § 5, of the average computing time. Hence we proceed next to derive the minimum computing time of the Euclidean algorithm.

THEOREM 5. $t_E^-(m, n, k) \sim n(m - n + 1) + k(n - k + 1)$.

Proof. By (17), $t_E^-(m, n, k) \succeq L(q_1)L(a_2) \sim n(m - n + 1)$. Since $q_i = \lfloor a_i/a_{i+1} \rfloor$, we have $q_{i+1} > a_i/a_{i+1}$ and so $\prod_{i=1}^{l}(q_i + 1) > \prod_{i=1}^{l}(a_i/a_{i+1}) = a/c$. By (17),

$$t_E(a, b) \sim \sum_{i=1}^{l} L(q_i)L(a_{i+1}) \geqq L(c) \sum_{i=1}^{l} L(q_i) \sim L(c) \sum_{i=1}^{l} L(q_i + 1)$$

$$\geqq L(c)L(a/c) \geqq L(c)L(b/c) \sim L(c)\{L(b) - L(c) + 1\}.$$

Hence $t_E^-(m, n, k) \succeq k(n - k + 1)$ and by Theorem 1, part (c), $t_E^-(m, n, k) \succeq n(m - n + 1) + k(n - k + 1)$.

If $n = k$, let $a = \beta^{m-1}$ and $b = \beta^{n-1}$ so that $c = \beta^{n-1}$ and $D(a, b) = 1$. By (17), this shows that $t_E^-(m, n, k) \preceq n(m - n + 1) \preceq n(m - n + 1) + k(n - k + 1)$.

If $n > k$, let $a = \beta^{m-1} + \beta^{k-1}$ and $b = \beta^{n-1}$, so that $c = \beta^{k-1}$, $L(a) = m$ and $D(a, b) = 2$. Then by (17), $t_E^-(m, n, k) \preceq n(m - n + 1) + k(n - k + 1)$ for $n > k$. Application of Theorem 1, part (h), concludes the proof. ∎

5. The average computing time. As observed in the proof of Theorem 4, if $a \geqq b$ and $(a_1, a_2, \cdots, a_{l+1}, a_{l+2})$ is the remainder sequence of (a, b), then $a \geqq F_{l+1} \geqq \phi^l/\sqrt{5}$. Since $e > \sqrt{5}$, we have $l \ln \phi \geqq (\ln a) + 1$. That is,

$$D(a, b) \leqq (\ln \phi)^{-1}((\ln a) + 1), \tag{24}$$

with $(\ln \phi)^{-1} = 2.078 \cdots$. Dixon established in [6] that for every $\varepsilon > 0$

$$|D(a, b) - \tau \ln a| < (\ln a)^{1/2+\varepsilon} \tag{25}$$

for almost all pairs (a, b) with $u \geqq a \geqq b \geqq 1$, as $u \to \infty$, where

$$\tau = 12\pi^{-2} \ln 2, \tag{26}$$

and we have $\tau = 0.84276 \cdots$. By more elementary means, Dixon proved in [7] the weaker result that

$$D(a, b) \geqq \tfrac{1}{2} \ln a \tag{27}$$

for almost all pairs (a, b) with $u \geqq a \geqq b \geqq 1$ as $u \to \infty$. In the following, we will show how Dixon's weaker result can be used to prove that the average computing time of the Euclidean algorithm is codominant with its maximum computing time of $n(m - k + 1)$. Before proceeding to the detailed proof, however, we shall present an intuitive sketch.

It is a well-known result (see [11, § 4.5.2, Exercise 10]) that the proportion of pairs (a, b) with $u > a \geqq b \geqq 1$ for which g.c.d.$(a, b) = 1$ approaches $6\pi^{-2}$ as $u \to \infty$. We will first generalize this result to the pairs (a, b) with $u > a \geqq b \geqq v$ as $u - v \to \infty$. Next we set $u = \beta^{n-k+1/2}$ and $v = \beta^{n-k}$ and conclude, combining this result with Dixon's, that, for $n - k$ large, at least half of the pairs (a, b) for which $u > a \geqq b \geqq v$ satisfy both g.c.d.$(a, b) = 1$ and $D(a, b) \geqq \frac{1}{2} \ln a$. For each pair satisfying these conditions and each c with $\beta^{k-1} \leqq c < \beta^{k-1/2}$, we obtain a pair $(\bar{a}, \bar{b}) = (ac, bc)$ with g.c.d.$(\bar{a}, \bar{b}) = c$, $L(\bar{a}) = L(\bar{b}) = n$ and $L(c) = k$. If $m > n$, then from each pair $(\bar{a}, \bar{b})$ we obtain at least $\frac{1}{2}\beta^{m-n}$ pairs $(\bar{\bar{a}}, \bar{\bar{b}})$ of the form $(\bar{a}q + \bar{b}, \bar{b})$ for which $L(\bar{\bar{a}}) = m$ and these also satisfy $L(\bar{\bar{b}}) = n$, $L(\text{g.c.d.}(\bar{\bar{a}}, \bar{\bar{b}})) = k$ and $D(\bar{\bar{a}}, \bar{\bar{b}}) \geqq \frac{1}{2} \ln \beta^{n-k}$. The pairs $(\bar{\bar{a}}, \bar{\bar{b}})$ so obtained constitute at least $0.004\beta^{-2}$ of all pairs in $S_{m,n,k}$ and $t_E(\bar{\bar{a}}, \bar{\bar{b}}) \geq n(m - k + 1)$ for all $(\bar{\bar{a}}, \bar{\bar{b}})$, so $t_E^*(m, n, k) \geq n(m - k + 1)$ for $n - k > h$, say. But it is trivial that $t_E^*(m, n, k) \geq n(m - k + 1)$ for $n - k \leqq h$ for any constant h, and so $t_E^*(m, n, k) \sim n(m - k + 1)$.

THEOREM 6. *Let u and v be positive integers with $u > v$, let $w = u - v$, and let q be the number of pairs of integers (a, b) such that $u > a, b \geqq v$ and* g.c.d.$(a, b) = 1$. *Then* $|q/w^2 - 6/\pi^2| \leqq 2((\ln u) + 1)/w + u/w^2 + 1/u$.

Proof. Let v_k be the number of integers a such that $k|a$ and $u > a \geqq v$. Then

$$|v_k - w/k| < 1, \tag{28}$$

and v_k^2 is the number of pairs (a, b) for which $k|$g.c.d.(a, b) and $u > a, b \geqq v$. By the principle of inclusion and exclusion,

$$q = \sum_{k=1}^{u} \mu(k) v_k^2, \tag{29}$$

where μ is the Möbius function. By (28),

$$|v_k^2 - w^2/k^2| < 2w/k + 1. \tag{30}$$

Multiplying (30) by $\mu(k)/w^2$ and summing, we have, by (29),

$$|q/w^2 - \sum_{k=1}^{u} \mu(k)/k^2| < 2H_u/w + u/w^2, \tag{31}$$

where H_u is the harmonic sum $\sum_{k=1}^{u} 1/k$. Using

$$\sum_{k=1}^{\infty} \mu(k)/k^2 = \pi^2/6 \tag{32}$$

together with (31) yields

$$|q/w^2 - \pi^2/6| < 2H_u/w + u/w^2 + \sum_{k=u+1}^{\infty} 1/k^2. \tag{33}$$

But $\sum_{k=u+1}^{\infty} 1/k^2 < \int_u^{\infty} x^{-2}\,dx = 1/u$, and $H_u \leqq (\ln u) + 1$, which establishes the theorem after substitution in (33). ∎

THEOREM 7. *There is a positive integer h such that for $n - k > h$, there are at least $0.02\beta^{2n-2k+1}$ pairs (a, b) for which $\beta^{n-k+1/2} > a \geqq b \geqq \beta^{n-k}$, g.c.d.$(a, b) = 1$ and $D(a, b) \geqq \frac{1}{2} \ln a$.*

Proof. Set $u = \lceil \beta^{n-k+1/2} \rceil$, $v = \beta^{n-k}$, $w = u - v$. Since $\lim_{n-k\to\infty} (u/w) = (1 - \beta^{-1/2})^{-1} \leqq (1 - 1/\sqrt{2})^{-1} < 4$, it follows from Theorem 6 that

$$\lim_{n-k\to\infty} (q/w^2) = 6/\pi^2.$$

Since $6/\pi^2 > 0.6$ and g.c.d.(a, b) = g.c.d.(b, a) there exists an h_1 such that for $n - k > h_1$, there are at least $0.3w^2$ pairs (a, b) for which $u > a \geqq b \geqq v$ and g.c.d.$(a, b) = 1$. By Dixon's theorem there is an h_2 such that if $n - k > h_2$, then $D(a, b) < \frac{1}{2} \ln a$ for at most 0.05 pairs (a, b) with $u \geqq a$, $b \geqq 1$. Hence if $h = \max(h_1, h_2)$ and $n - k > h$, there are at most $\frac{1}{4}w^2$ pairs (a, b) for which $u > a \geqq b \geqq v$, g.c.d.$(a, b) = 1$ and $D(a, b) \geqq \frac{1}{2} \ln a$. The theorem follows since $w \geqq (\sqrt{\beta} - 1)\beta^{n-k}$ and $(\sqrt{\beta} - 1)^2/\beta \geqq (\sqrt{2} - 1)^2/2 \geqq 0.08$. ∎

THEOREM 8. *There is a positive integer h such that for $n - k > h$, there are at least $0.004\beta^{m+n-k}$ pairs (a, b) such that $a \geqq b$, $L(a) = m$, $L(b) = n$, L(g.c.d.$(a, b)) = k$ and $D(a, b) \geqq \frac{1}{2} \ln \beta^{n-k}$.*

Proof. Choose an h for which Theorem 7 holds. For every pair (a, b) satisfying Theorem 7 and every integer satisfying $\beta^{k-1} \leqq c < \beta^{k-1/2}$, we obtain a pair (ac, bc) with $ac \geqq bc$, $L(ac) = L(bc) = n$, L(g.c.d.$(ac, bc)) = L(c) = k$, and $D(ac, bc) = D(a, b) \geqq \frac{1}{2} \ln a \geqq \frac{1}{2} \ln \beta^{n-k}$. The mapping $f((a, b), c) = (ac, bc)$ thus defined is one-to-one so there are at least

$$(0.02\beta^{2n-2k+1})(\sqrt{\beta} - 1)\beta^{k-1} \geqq 0.008\beta^{2n-k+1}$$

pairs (a, b) with $a \geqq b$, $L(a) = L(b) = n$, L(g.c.d.$(a, b)) = k$ and $D(a, b) \geqq \frac{1}{2} \ln \beta^{n-k}$. If $m = n$, this completes the proof; so assume $m > n$. For each pair (a, b) with $L(a) = L(b) = n$, there are at least

$$\lfloor (\beta^m - \beta^{m-1})/a \rfloor \geqq (\beta^m - \beta^{m-1})/\beta^n \geqq (1 - \beta^{-1})\beta^{m-n-1} \geqq \tfrac{1}{2}\beta^{m-n}$$

pairs $(aq + b, a)$ with $L(aq + b) = m$. Since g.c.d.$(aq + b, a)$ = g.c.d.(a, b) and $D(aq + b, a) = D(a, b) + 1$, we obtain at least $(0.008\beta^{2n-k})(\frac{1}{2}\beta^{m-n}) = 0.004\beta^{m+n-k}$ pairs $(aq + b, a)$ for which $aq + b \geqq a$, $L(aq + b) = m$, $L(a) = n$, L(g.c.d.$(qa + b, a)) = k$ and $D(aq + b, a) \geqq \frac{1}{2} \ln \beta^{n-k}$. ∎

THEOREM 9. $t_E^*(m, n, k) \sim t_E^+(m, n, k) \sim n(m - k + 1)$.

Proof. Let $c_1 = \min(1, \frac{1}{2}\ln\beta)$. By Theorems 4 and 8, there exist h and $c_2 > 0$ such that

$$t_E(a, b) \geqq c_2 D(a, b)\{D(a, b) + L(\text{g.c.d.}(a, b))\} \geqq c_2 c_1 (n - k)\{c_1(n - k) + k\}$$
$$\geqq c_1^2 c_2 n(n - k)$$

for $n - k > h$ and for at least $0.004\beta^{m+n-k}$ elements of $S_{m,n,k}$. Every element of $S_{m,n,k}$ is of the form (ac, bc) with $a < \beta^{m-k+1}$, $b < \beta^{n-k+1}$ and $c < \beta^k$, so $S_{m,n,k}$ has at most $\beta^{m+n-k+2}$ elements. Hence, $t_E^*(m, n, k) \geqq 0.004 c_1^2 c_2 \beta^{-2} n(n - k) \sim n(n - k)$ for $n - k > h$. By Theorem 5, $t_E^*(m, n, k) \geq n(m - n + 1) \geqq n \geq n(n - k)$ for $n - k \leqq h$. Hence by Theorem 1, part (h), $t_E^*(m, n, k) \geq n(n - k)$. By Theorem 5, $t_E^*(m, n, k) \geq n(m - n + 1)$ so by Theorem 1, part (c),

$$t_E^*(m, n, k) \geq n(n - k) + n(m - n + 1) = n(m - k + 1). \tag{34}$$

The conclusion of the theorem is now immediate from Theorem 3, (34) and the obvious inequality $t^*(m, n, k) \leqq t_E^*(m, n, k)$.

REFERENCES

[1] W. S. BROWN, *On Euclid's algorithm and the computation of polynomial greatest common divisors*, J. Assoc. Comput. Mach., 18 (1971), pp. 478–504.

[2] G. E. COLLINS, *Computing time analyses for some arithmetic and algebraic algorithms*, Proc. 1968 Summer Institute on Symbolic Mathematical Computation, IBM Corp., Cambridge, Mass., 1969, pp. 197–231.

[3] ———, *The calculation of multivariate polynomial resultants*, J. Assoc. Comput. Mach., 18 (1971), pp. 515–532.

[4] ———, *The SAC-1 integer arithmetic system—Version III*, Tech. Rep. 156, Computer Sciences Dept., Univ. of Wisconsin, Madison, 1973.

[5] N. G. DEBRUIJN, *Asymptotic Methods in Analysis*, North-Holland, Amsterdam, 1961.

[6] J. D. DIXON, *The number of steps in the Euclidean algorithm*, J. Number Theory, 2 (1970), pp. 414–422.

[7] ———, *A simple estimate for the number of steps in the Euclidean algorithm*, Amer. Math. Monthly, 78 (1971), pp. 374–376.

[8] H. HEILBRONN, *On the average length of a class of continued fractions*, Abhandlungen aus Zahlentheorie und Analysis, VEB Deutscher Verlag, Berlin, 1968.

[9] L. E. HEINDEL, *Integer arithmetic algorithms for polynomial real zero determination*, J. Assoc. Comput. Mach., 18 (1971), pp. 533–548.

[10] D. E. KNUTH, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Addison-Wesley, Reading, Mass., 1968.

[11] ———, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, Addison-Wesley, Reading, Mass., 1969.

[12] D. R. MUSSER, *Algorithms for Polynomial Factorization*, Ph.D. thesis, Tech. Rep. 134, Computer Sciences Dept., Univ. of Wisconsin, Madison, 1971.

SIAM J. COMPUT.
Vol. 3, No. 1, March 1974

STACK REPLACEMENT ALGORITHMS FOR TWO-LEVEL DIRECTLY ADDRESSABLE PAGED MEMORIES*

R. R. MUNTZ AND H. OPDERBECK†

Abstract. In this paper we consider the application of the stack algorithm concept to a two-level paged storage hierarchy in which both levels are directly addressable by the central processor. Since both levels are directly addressable, pages need not reside in the first level of memory for a reference to be completed. The effectiveness of a page replacement algorithm in such a storage hierarchy is measured by the frequency of references to the first level of memory and the number of page promotions. It is shown that the stack algorithm concept can easily be extended to apply to two-level directly addressable memory hierarchies. A class of page replacement algorithms called stack replacement algorithms is defined. An efficient procedure exists for collecting data on the performance of stack replacement algorithms.

Key words. memory management, memory hierarchies, paging, stack algorithms

1. Introduction. The performance of a virtual memory system is to a large extent determined by the efficiency of moving information between the different levels of the physical memory hierarchy. In the case of two-level paging systems, the page traffic between the two levels is one of the most important design factors. Most of these systems use the page as the unit of information transfer, and referenced information must be in the first level of the storage hierarchy for a reference to be completed. This implies that whenever an information item is referenced which is not in the first level, the entire page on which this information item resides is transferred. Because of the locality of page references, this is in many cases an efficient policy. However, there are usually some pages which are referenced rather infrequently. For these pages it would be more efficient to transfer the referenced information item directly to the CPU and leave the corresponding page in the second-level memory. Memory hierarchies that use this strategy are called *two-level directly addressable paged memories.* The IBM 360/67 installation at Carnegie–Mellon University is an example of a computer system with this sort of a memory hierarchy [2], [3], [5]. Developments in storage technology which have made bulk core storage more economical make systems with two-level directly addressable memories more and more likely.

In this paper the first level of the two-level directly addressable memory hierarchy is assumed to be faster, smaller and more expensive than the second level. The second level is assumed to be large enough to contain all of the program and data.

The movement of a page from the second to the first level of memory will be called a *page promotion.* The first level of memory is assumed to consist of a fixed number, m, of page frames. Therefore page promotions and page replacements always occur at the same time if the first level of memory is full. For two-level directly addressable memories, the page promotion and page replacement decisions are usually not chosen independently but are part of a single policy. In

* Received by the editors May 25, 1973, and in revised form September 7, 1973.

† Computer Science Department, University of California at Los Angeles, Los Angeles, California 90024. This research was supported by the National Science Foundation under Grant GJ 809 and the U.S. Office of Naval Research, Mathematical and Information Sciences Division, under Contract N00014-69-0200-4027, NR 048-129.

referring to a policy we will use the term *replacement policy* (or *algorithm*) rather than the more exact but clumsier term, "promotion/replacement policy". Page promotions may be done only "on demand", i.e., a page in the second level of memory is promoted only at a time when it is referenced, or in a nondemand paging manner, i.e., pages may be promoted at any time.

Let $N = \{0, 1, 2, \cdots, n\}$ denote the set of pages for a given program. A program's dynamic behavior is, for our purposes, given by its reference string $\omega = r_1 r_2 \cdots r_t \cdots$, where $r_t \in N$ and r_t is the tth page referenced for $t \geqq 1$.

Let

δ_1 = the time to reference the first level of memory,

δ_2 = the time to reference the second level of memory,

Δ = the time to move a page from the second level of memory to the first level of memory.

Let ω be a given reference string and let $|\omega|$ denote its length. Let n_1 denote the number of references to a page which resides in the first level of memory, n_2 denote the number of references to a page which resides in the second level of memory, and n_3 denote the number of page promotions. Clearly, $n_1 + n_2 = |\omega|$. For a first level memory with given size, the average access time is given by

$$\text{average access time} = \frac{n_1 \cdot \delta_1 + n_2 \cdot \delta_2 + n_3 \cdot \Delta}{n_1 + n_2}.$$

We assume that $m < n$ and that $\delta_1 < \delta_2 < \Delta$. The size of the first level of memory and the policies that are used in managing the memory hierarchy will determine n_1, n_2 and n_3 for any given reference string.

The average access time is a common performance measure also for storage hierarchies in which only the first level is directly addressable. However, there is a much different set of policies from which to choose when both levels of memory are directly addressable. There are two fundamentally distinct approaches that can be taken in this case. First, a decision can be made a priori as to which pages should reside in the first level. For example, an analysis of system code or production programs could be used to determine which pages are most frequently referenced and a decision made as to the level on which they should reside. The second approach is to dynamically determine which pages should reside in the first level based on some measure of the current frequency of reference to the pages. This latter approach has been considered by some authors and seems to hold some promise [1], [6]. Much more experimental work is necessary to evaluate the efficacy of such dynamic management policies especially since they often require additional hardware.

The purpose of this paper is to show how the stack algorithm concept [4] is applicable to measurement studies of page replacement policies for a two-level directly addressable paged memory hierarchy. As in the more usual case in which only the first level is directly addressable, the stack algorithm concept can be utilized to efficiently collect experimental data on the performance of a given policy. A class of replacement policies called *stack replacement algorithms* is

defined. In one pass through a reference string, one can efficiently collect the data (n_1, n_2 and n_3) necessary to evaluate the performance of the stack replacement algorithm on this reference string for all sizes (number of page frames) of the first level of memory. The procedure for processing a reference string and collecting this data will be called a *stack processing procedure.*

In §§ 2 and 3 we consider two general classes of replacement rules for two-level directly addressable memories. In § 2 the replacement rules are such that a promotion of some page x may only take place as a result of a reference to page x ("demand paging"). In § 3 we consider replacement rules in which promotions and replacements take place at fixed intervals. Examples of both types of stack replacement algorithms are given, and the corresponding stack processing procedures are described. It is also shown that these stack replacement algorithms have the property that for any reference string, the number of references to the first level (n_1) is nondecreasing with the size of the first-level memory. The number of page promotions (n_3), however, may increase with the size of the first-level memory. This implies that the average access time may increase with an increase in the first-level memory size.

2. Stack replacement algorithms using continuous updating. For any replacement policy A, let $S_t(A, m)$ be the set of those pages that would be in the first level of memory at time t (just after processing r_t) if there were m page frames in the first level of memory. For the present we assume that pages can be promoted to the first level of memory only when they are referenced (this is similar to demand paging). Page replacements from the first level of memory are assumed to take place only when the first level of memory is full and an additional page is promoted to the first level. Thus, $S_t(A, m)$ may only contain pages that have been referenced up to time t.

A replacement policy A is a stack replacement policy iff

$$S_t(A, m) \subseteq S_t(A, m + 1) \forall t, \forall m, \forall \omega,$$

i.e., iff the inclusion property is satisfied. If the inclusion property holds, then we can define an ordered list of the l pages thus far referenced, $\underline{S}_t(A)$, such that the first m pages in this list are the pages that comprise $S_t(A, m)$. Of course, if $l < m$ pages have been referenced up to time t, then $S_t(A, m)$ is comprised of only those l pages. $\underline{S}_t(A)$ is called the *stack*. If the stack algorithm A is understood, we simply write $S_t(m)$ and $\underline{S}_t$ for $S_t(A, m)$ and $\underline{S}_t(A)$, respectively.

If a page x has been referenced at least once up to time t, then it appears at some position in the stack. Let $D_t(x)$ denote the position of page x in the stack just after processing reference r_t. Following the usual notation, we let $D_t(x) = \infty$ if x has not yet been referenced.

As stated above, we assume that if a page is promoted to the first level of memory at time t, then the page promoted is r_t. If the first level of memory was full, then some page must be replaced.

Let $y_t(m)$ be the page replaced from the first level of memory of size m at time t (i.e., due to reference r_t) if a replacement is made, or $\varnothing$ if no replacement is made. Clearly $S_t(m) = S_{t-1}(m) + r_t - y_t(m)$.

It is not difficult to show that the inclusion property holds iff

$$y_t(m+1) = \begin{cases} y_t(m) \quad \text{or} \quad \{S_{t-1}(m+1) - S_{t-1}(m)\} & \text{if } r_t \notin S_{t-1}(m+1), \\ \varnothing & \text{if } r_t \in S_{t-1}(m+1). \end{cases}$$

In words, the page replaced from an $(m+1)$-page first-level memory is either the page replaced from an m-page first-level memory or the additional page that is in the $(m+1)$-page memory but not in the m-page memory.

We now briefly review the stack processing procedure for a storage hierarchy in which only the first level is directly addressable. For a demand paging algorithm when only the first level of memory is directly addressable, we have

$$S_t(m) = \begin{cases} S_{t-1}(m) & \text{if } r_t \in S_{t-1}(m), \\ S_{t-1}(m) + r_t & \text{if } r_t \notin S_{t-1}(m) \quad \text{and} \quad |S_{t-1}(m)| < m, \\ S_{t-1}(m) + r_t - y_t(m) & \text{if } r_t \notin S_{t-1}(m) \quad \text{and} \quad |S_{t-1}(m)| = m, \end{cases}$$

where, for any set S, $|S|$ denotes the cardinality of S. Note that if $y_t(m+1) \neq \varnothing$, then $y_t(m) \neq \varnothing$. This follows since if $y_t(m+1) \neq \varnothing$, then $|S_{t-1}(m+1)| = m+1$, which implies that at least $m+1$ distinct pages have been referenced and therefore $|S_{t-1}(m)| = m$. Also $r_t \notin S_{t-1}(m)$ due to the inclusion property. This shows the well-known property that the page fault rate for stack replacement algorithms is nonincreasing with memory size if only the first level of memory is directly addressable. Figure 1 shows the stack processing procedure for this case.

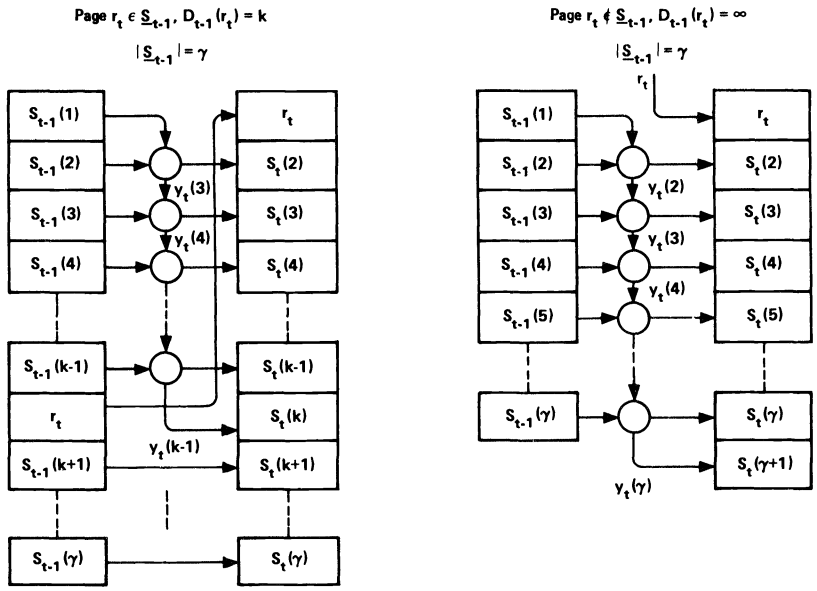


FIG. 1. *Stack processing procedure with one level of directly addressable memory*

Now consider the case in which both levels of memory are directly addressable. There is no longer the condition that a referenced page which resides in the second level must be promoted to the first level of memory. Now the decision as to whether or not to promote the referenced page to the first level may be a

function of the current priority of the referenced page and the priorities of the pages currently residing in the first level. We assume that the current priority of a page is equal to some measure of its frequency of use. The priority of a page x at time t will be denoted by $p_t(x)$.

The following general class of replacement policies is considered. We assume there is a priority threshold PT_p such that a referenced page is not a candidate for promotion unless its current priority is at least PT_p. Similarly, a page is not a candidate for replacement unless its current priority is less than some threshold PT_r. If the referenced page is a candidate for promotion and several pages are candidates for replacement, then the page with minimum current priority is replaced. PT_p and PT_r are not necessarily constant thresholds but may depend on the reference string. Some examples of such algorithms are in order.

Example 1. The referenced page always replaces the page with the lowest frequency of reference.

$$PT_p \equiv 0$$
$$PT_r \equiv \infty$$

Example 2. The referenced page replaces any page with a lower priority (or frequency of reference).

$$PT_p \equiv 0$$
$$PT_r \equiv p_t(r_t)$$

Example 3. The referenced page is a candidate for promotion if it has a current frequency of reference greater than or equal to K_1 (but can only replace a page with a lower frequency of reference).

$$PT_p \equiv K_1 > 0$$
$$PT_r \equiv p_t(r_t)$$

Example 4. The referenced page may be promoted only if its current frequency of reference is at least K_1 but can only replace a page with a frequency of reference less than K_2.

$$PT_p \equiv K_1 > 0$$
$$PT_r \equiv K_2 < \infty$$

Example 5. The referenced page r_t is always a candidate for promotion, but it can only replace a page y if $p_t(y) < p_t(r_t) - H$. H is a constant which is chosen to prevent rapid cycles of page x replacing page y, page y replacing page x, etc.

$$PT_p \equiv 0$$
$$PT_r \equiv p_t(r_t) - H.$$

It is clear that for this class of algorithms,

$$y_t(m) = \begin{cases} \varnothing & \text{if } p_t(r_t) < PT_p \quad \text{or} \quad |S_{t-1}(m)| < m \\ & \quad \text{or } p_t(\min [S_{t-1}(m)]) \geqq PT_r, \\ \min [S_{t-1}(m)] & \text{otherwise.} \end{cases}$$

Here $\min [S_{t-1}(m)]$ denotes the lowest priority page in $S_{t-1}(m)$. It is easy to verify that $y_t(m+1) = y_t(m)$ or $S_{t-1}(m+1) - S_{t-1}(m)$ or $\varnothing$. However, note that it is possible that $y_t(m+1) \neq \varnothing$ while $y_t = \varnothing$. This can occur if $p_t(\min [S_{t-1}(m)]) \geqq PT_r$ but $p_t(\min [S_{t-1}(m+1)]) < PT_r$. Thus the additional page in the $(m+1)$-page first-level memory is the only replaceable page in this case.

The stack processing procedure for this class of algorithms is very similar to the stack processing when only the first level of memory is directly addressable. If $p_t(r_t) < PT_p$ or if all pages with stack distance less than $D_{t-1}(r_t)$ have priorities greater than or equal to PT_r, then there is no change in the stack. If r_t does replace a page higher in the stack, say page x, then x is the page with the smallest stack distance whose priority is less than PT_r. Let page x be at stack distance l, i.e., $D_{t-1}(x) = l$. The stack processing procedure is illustrated in Fig. 2.

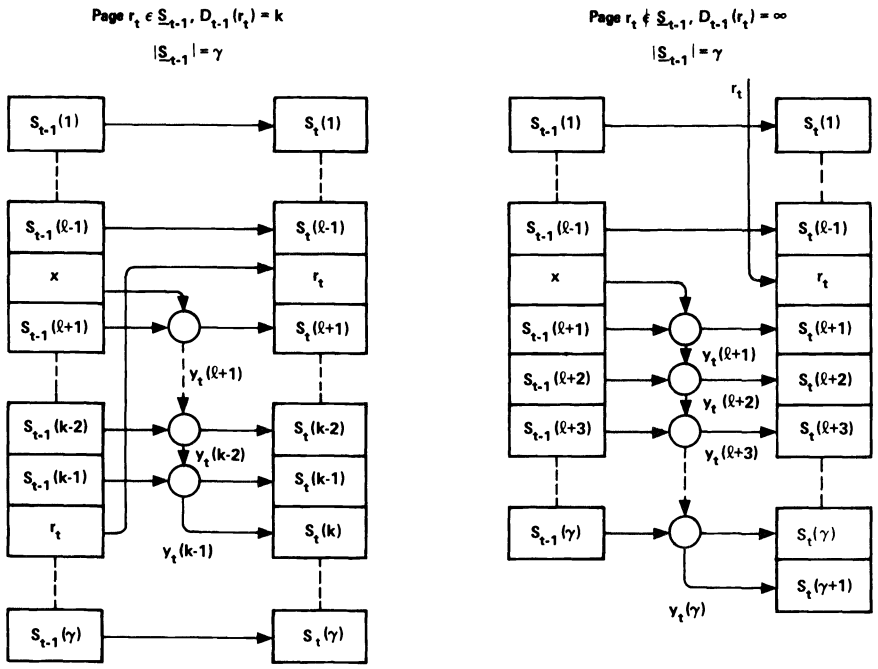


FIG. 2. *Continuous stack processing with two levels of directly addressable memory*

3. Stack replacement algorithms using batch updating. If only the first-level memory is directly addressable, the processing is interrupted and the page replacement algorithm is applied whenever a page in the second-level memory is referenced. In case of a two-level directly addressable memory, on the other hand, the processing need not be interrupted when a page in the second-level memory is referenced. Processing can continue by directly referencing the required information item in the second-level memory. Therefore we do not have to apply the page replacement algorithm at each reference to the second-level memory. The two levels of memory can be managed in a truly nondemand paging manner.

As an example, let us assume the page replacement algorithm is applied in intervals of some constant number of references. After each interval, a replacement of zero, one or more pages takes place. These replacements could be based on statistics about reference frequencies gathered during the last one or more intervals. Since each application of the page replacement algorithm may result in the replacement of several pages, such an algorithm will be called a *batch updating page replacement algorithm.*

Such a batch updating replacement algorithm has the following advantages. The processing is interrupted less frequently, and therefore the overhead associated with these interruptions is reduced. If the page transfer is controlled by a separate storage channel, the processor can be switched to another process during the page transfer. The frequency of this processor switching can be reduced if a batch updating replacement algorithm is used. In addition, the implementation of such a scheme appears to be simpler.

The conditions necessary for a batch updating page replacement algorithm to be a stack replacement algorithm are different from those for continuous updating. Let the batch replacement be initiated immediately after reference r_t, i.e., $S_{t-1}(m)$ and $S_t(m)$ are, respectively, the sets of pages in an m-page memory immediately before and after the batch replacement. Further, let $X_t(m)$ be the set of pages which are promoted to the first-level memory and $Y_t(m)$ be the set of pages that are replaced.

A batch updating page replacement algorithm is a stack replacement algorithm iff

(1) $$X_t(m) \subseteq X_t(m+1) + z,$$

(2) $$Y_t(m+1) \subseteq Y_t(m) + z,$$

(3) $$\text{if } z \in X_t(m), \text{ then } z \notin Y_t(m+1),$$

where z is the additional page in the $(m+1)$-page memory, i.e., $S_{t-1}(m+1) = S_{t-1}(m) + z$. (The proof is given in Appendix A.)

Now consider a batch updating page replacement algorithm with two fixed thresholds PT_r and PT_p, i.e., only pages with a current priority less than PT_r are candidates for replacement and only pages with a current priority greater than or equal to PT_p are candidates for promotion. Also let $PT_p > PT_r$, i.e., every promoted page has a higher priority than any of the replaced pages. Let us assume that for a given size of the first-level memory, there are r candidates for replacement and s candidates for promotion. Consider the following batch replacement policy:

1. If there are s candidates for promotion but only $r < s$ candidates for replacement, then the r candidates for replacement are replaced by the pages with the highest current priority.
2. If there are r candidates for replacement but only $s \leqq r$ candidates for promotion, then the s pages with the lowest current priority are replaced.

The proof that such a batch updating page replacement algorithm does not violate any of the conditions (1)–(3), and that it therefore is a stack replacement algorithm is given in Appendix B.

The stack processing procedure for the case of fixed thresholds and batch updating is a generalization of the case of continuous updating. It consists of a sequence of stack updating operations as described in the previous section. This shows informally that this batch updating page replacement algorithm is a stack replacement algorithm since each step preserves the inclusion property. The sequence of the individual stack updating operations is important since the pages with the highest priorities are promoted if there are more candidates for promotion than candidates for replacement.

Let us assume the stack contains s pages with a priority greater than or equal to PT_p. Further, let these s pages be denoted by $x_1, x_2, \cdots, x_{s-1}, x_s$, in decreasing priority order. The stack processing procedure consists of s "continuous-type" updating operations. The first of these s operations is done as if page x_1 had been referenced in the case of continuous updating with fixed thresholds PT_r and PT_p. This transforms stack $\underline{S}_{t'}$ into stack $\underline{S}_t^{(1)}$, where t' is the time of the most recent batch replacement operation. Thus $t - t'$ is the time interval between successive stack updating operations. Similarly, $\underline{S}_t^{(1)}$ is transformed into $\underline{S}_t^{(2)}$ by updating $\underline{S}_t^{(1)}$ as if page x_2 had been referenced, etc. This procedure results in s intermediary stacks $\underline{S}_{t'} = \underline{S}_t^{(1)}, \underline{S}_t^{(2)}, \cdots, \underline{S}_t^{(s)} = \underline{S}_t$. These intermediary stacks need not be distinct since any of the s individual updating operations may leave the stack unchanged. After these s continuous-type stack updating operations, the new stack, $\underline{S}_t$, is created. This stack then remains unchanged during the next $t - t'$ references.

To see that this stack processing procedure is correct, we give the following informal arguments. Consider the set of pages in the first m stack positions (for convenience we assume the stack contains at least m pages) and how this set of pages changes as a result of the stack updating. As each of the pages in the sequence $x_1, x_2, \cdots, x_s$ is processed in turn, the following cases can occur:

1. If $D_{t'}(x_i) \leqq m$,	then no change is made to the set of pages in the first m stack positions.
2. If $D_{t'}(x_i) > m$ and there is still a replaceable page with stack distance less than or equal to m,	then page x_i replaces the minimum priority page in the set of pages in the first m stack positions.
3. If $D_{t'}(x_i) > m$ and there are no longer any replaceable pages with stack distance less than or equal to m,	then no change is made to the set of pages in the first m stack positions.

We simply note that as long as there are replaceable pages in the first m stack positions, then processing a page x_i with stack distance greater than m will cause x_i to replace the lowest priority page. Note that if we ever have case 3 (i.e., we have exhausted the replaceable pages) then the highest priority promotable pages with stack distance greater than m have replaced all the replaceable pages. This follows since the sequence $x_1, x_2, \cdots, x_s$ is in decreasing order by priority.

Consider now the case where a page y is replaced by a page x iff $p_t(x) > p_t(y) + H$. If this rule is applied to a batch replacement procedure, the total number of

replacements is clearly dependent on the sequence of individual replacements. To preserve the inclusion property, the following stronger condition is needed for the batch replacement algorithm:

$$p_t(\min [X_t(m)]) > p_t(\max [Y_t(m)]) + H.$$

In words, the priority of the page with the lowest priority among the promoted pages must be greater than the constant H plus the priority of the page with the highest priority among the replaced pages. The proof that this replacement policy represents a stack algorithm is similar to the proof given in Appendix B.

The stack processing procedure in this case is similar to the stack updating in the case of fixed thresholds, i.e., it consists of a sequence of continuous-type updating operations. It is again important that the individual stack updating operations are done in order of decreasing priority of the pages in the stack. This guarantees that the stronger condition for the batch updating is satisfied and that the pages with the highest priorities are promoted. The continuous-type stack updating operation for some page x is done as if page x had been referenced in the case of continuous updating with the thresholds $PT_p = 0$ and $PT_r = p_t(x) - H$.

4. Data collection. As in the more usual case in which only the first level of memory is directly addressable, there is a general method of data collection which can be applied to any stack replacement algorithm. We first consider determining the number of references to the first level of memory (n_1). This performance measure corresponds to the success function as defined in [4]. It can be obtained in the same way. That is, we keep a counter D_i for each stack position ($i = 1, 2, \cdots, n, \infty$). Initially $D_i = 0$ for all i. For every page reference the stack distance of the referenced page is determined and the corresponding counter incremented by 1. When the stack processing of the reference string is finished, the sum $\sum_{i=1}^{j} D_i$ is equal to the number of references to the j-page first-level memory. We obtain the fraction of references to the first level of memory by dividing this sum by the total number of references. Since $D_i \geqq 0$ for all i, the percentage of references to the first-level memory is a nondecreasing function of the first-level memory size.

If only the first level of memory is directly addressable, every referenced page which does not reside in the first level must be transferred to the first level. The number of references to the second level of memory is therefore equal to the number of page faults, i.e., $\sum_{i=j+1}^{n} D_i + D_\infty$. In the case of two-level directly addressable memories, a reference to a page in the second level does not necessarily cause a page promotion. The number of page promotions to a j-page first-level memory can therefore not be derived from the final values of the counters D_i.

The number of page promotions (n_3) can be obtained for all sizes of the first-level memory in one pass through the reference string. For this purpose we keep a second set of counters E_i ($i = 1, 2, \cdots, n, \infty$). Initially $E_i = 0$ for all i. These counters will be updated in such a way that $\sum_{i=j+1}^{n} E_i + E_\infty$ is equal to the number of page promotions to a j-page first-level memory after the stack processing of the reference string. Let us assume that a page at stack position k is moved up in the stack to position l as shown in Fig. 2. This stack updating results in a page promotion for the first-level memories of the size $l, l+1, \cdots, k-1$.

We therefore have to increment the counter E_k by 1. However, for the first-level memories of the size $1, 2, \cdots, l-1$, this stack updating does not cause a page promotion. Therefore, the incrementing of counter E_k must be offset by decrementing the counter E_l by 1. The general rule for updating the counters is: whenever a page is moved up in the stack, the counter for the stack position where the page came from is incremented by 1 and the counter for the stack position where the page is inserted is decremented by 1.

Since the counters E_i are incremented as well as decremented, we can no longer assume that $E_i \geqq 0$ for all i. As a result, the sum $\sum_{i=j+1}^{n} E_i + E_\infty$ is in general not a strictly nonincreasing function of j. Therefore an increase in the size of the first level of memory may increase the number of page promotions.

Table 1 shows an example of the stack processing for a two-level directly addressable memory. We consider the case of continuous updating with fixed thresholds. The priority of a page x is given by the number of references to page x among the last seven references divided by seven. As an initial condition it is assumed that there were no references to any of the pages in the past. The initial priorities are therefore all equal to zero. In this example, the fixed thresholds are chosen arbitrarily to be $PT_p = \frac{1}{7}$ and $PT_r = \frac{3}{7}$.

The last column of Table 1 shows the final value of the counters D_i and E_i. For a 1-page first-level memory, the fraction of references to the first level is $\frac{1}{3}$ and the number of page promotions is 5. For a 2-page first-level memory, the

TABLE 1

Example of a continuous stack processing procedure

										$PT_p = 1/7$			$PT_r = 3/7$		
Reference String		1	2	3	1	1	1	1	2	2	3	3	1	1	3	2
Stack	1	1	2	3	1	1	1	1	1	1	1	1	1	1	3	3
	2	-	1	2	3	3	3	3	2	2	3	3	3	3	1	2
	3	-	-	1	2	2	2	2	3	3	2	2	2	2	2	1
Priority of Pages	1	1/7	1/7	1/7	2/7	3/7	4/7	5/7	4/7	4/7	4/7	3/7	3/7	3/7	2/7	2/7
	2	0	1/7	1/7	1/7	1/7	1/7	1/7	2/7	2/7	2/7	2/7	2/7	2/7	2/7	2/7
	3	0	0	1/7	1/7	1/7	1/7	1/7	1/7	1/7	1/7	2/7	2/7	2/7	3/7	3/7
D_1		0	0	0	0	1	2	3	3	3	3	3	4	5	5	5
D_2		0	0	0	0	0	0	0	0	1	1	2	2	2	3	3
D_3		0	0	0	1	1	1	1	2	2	3	3	3	3	3	4
D_∞		1	2	3	3	3	3	3	3	3	3	3	3	3	3	3
E_1		-1	-2	-3	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-5	-5
E_2		0	0	0	0	0	0	0	-1	-1	-2	-2	-2	-2	-1	-2
E_3		0	0	0	1	1	1	1	2	2	3	3	3	3	3	4
E_∞		1	2	3	3	3	3	3	3	3	3	3	3	3	3	3

fraction of references to the first level increases to $\frac{8}{15}$. However, the number of page promotions also increases to 7. This shows that the allocation of an additional page frame may actually increase the number of page promotions.

5. Conclusions. The stack algorithm concept can be extended to the evaluation of two-level directly addressable memory hierarchies. Measurement data on the percentage of references to the first-level memory and the number of page promotions can be collected for all sizes of the first-level memory in one pass through the reference string. The stack replacement algorithms for two-level directly addressable memory hierarchies can be used in a continuous updating or a batch updating mode, i.e., the replacement can be made after each reference or the updating can be deferred for some time, thus reducing overhead costs. These evaluation techniques should be very helpful in the design of two-level directly addressable memory hierarchies.

Appendix A. Proof of conditions for batch updating. The proof is given in two parts.

1. Every stack replacement algorithm satisfies (1)–(3).

(i) Let $v \in X_t(m)$ and $v \neq z$; therefore $v \notin S_{t-1}(m+1)$ because of the inclusion property. This implies $v \in X_t(m+1)$ since $S_t(m) \subseteq S_t(m+1)$. The case $z \in X_t(m)$ works trivially.

(ii) Let $v \in Y_t(m+1)$ and $v \neq z$; therefore $v \in S_{t-1}(m)$ because of the inclusion property. This implies $v \in Y_t(m)$ since $S_t(m) \subseteq S_t(m+1)$ and $v \notin S_t(m+1)$. The case $z \in Y_t(m+1)$ works trivially.

(iii) If $z \in X_t(m)$, then $z \in S_t(m+1)$ and therefore $z \notin Y_t(m+1)$.

2. Every page replacement algorithm that satisfies conditions (1)–(3) is a stack replacement algorithm. Since the first-level memory is empty initially, the inclusion property holds at time $t = 0$. Therefore, we have to show that the inclusion property is preserved if the batch updating is done according to the conditions (1)–(3). Let $v \in S_t(m)$. We have to show that $v \in S_t(m+1)$.

(i) Let $v \in S_{t-1}(m)$; therefore $v \notin Y_t(m)$ and $v \neq z$. Because of condition (2) $v \notin Y_t(m+1)$. But $S_{t-1}(m) \subseteq S_{t-1}(m+1)$ now implies $v \in S_t(m+1)$.

(ii) Let $v \in X_t(m)$ and $v \neq z$; then $v \in X_t(m+1)$ because of condition (1), and therefore $v \in S_t(m+1)$.

(iii) Let $v \in X_t(m)$ and $v = z$; therefore $v \in S_{t-1}(m+1)$. Because of condition (3), $v \notin Y_t(m+1)$, and therefore $v \in S_t(m+1)$.

Appendix B. Proof for batch updating with fixed thresholds. Let $\min^r[S_t(m)]$ denote that subset of $S_t(m)$ which is comprised of the r pages with lowest priority. Let $\max^r[S_t(m)]$ be similarly defined for the highest priority pages in $S_t(m)$. The sets $X_t(m)$ and $Y_t(m)$ can now be defined as follows:

$$X_t(m) = \max^r[N - S_{t-1}(m)],$$
$$Y_t(m) = \min^r[S_{t-1}(m)],$$

where r is such that if x_i is the ith page in $N - S_{t-1}(m)$ in increasing priority

order and y_i is the ith page in $S_{t-1}(m)$ in decreasing priority order, then

(4) $$p_t(x_i) \geqq PT_p, \qquad 1 \leqq i \leqq r,$$

(5) $$p_t(y_i) < PT_r, \qquad 1 \leqq i \leqq r,$$

(6) at least one of the conditions (4) and (5) is not satisfied for $i = r + 1$.

Since the inclusion property holds initially, we have to show that conditions (1)–(3) are satisfied under the assumption that $S_{t-1}(m) \subseteq S_{t-1}(m+1)$.

Assume $S_{t-1}(m+1) = S_{t-1}(m) + z$.

(1) $$X_t(m) \subseteq X_t(m+1) + z.$$

Let $x_i \in X_t(m)$ be the ith page in $N - S_{t-1}(m)$ in increasing priority order. Clearly, if $x_i = z$, then condition (1) holds. If $x_i \neq z$, $\exists$ at least i pages in $S_{t-1}(m)$ with priority less than $\min(PT_r, p_t(x_i))$. Since $S_{t-1}(m) \subseteq S_{t-1}(m+1)$, the same is true for $S_{t-1}(m+1)$. Since there are less than i pages in $N - S_{t-1}(m)$ with priority greater than $p_t(x_i)$, it follows that $x_i \in X_t(m+1)$.

(2) $$Y_t(m+1) \subseteq Y_t(m) + z.$$

Let $y_i \in Y_t(m+1)$ be the ith page in $S_{t-1}(m+1)$ in decreasing priority order. If $y_i = z$, condition (2) obviously holds. If $y_i \neq z$, $\exists$ at least i pages in $N - S_{t-1}(m+1)$ with priority greater than $p_t(y_i)$ and greater than or equal to PT_p. The same is true for $N - S_{t-1}(m)$ since $N - S_{t-1}(m+1) \subseteq N - S_{t-1}(m)$. But there are less than i pages in $S_{t-1}(m)$ with priority less than $p_t(y_i)$; therefore $y_i \in Y_t(m)$.

(3) $$\text{if } z \in X_t(m), \text{ then } z \notin Y_t(m+1).$$

Let $z = x_i$. Then $\exists$ at least i pages in $S_{t-1}(m)$ with priority less than $\min(PT_r, p_t(z))$. The same is true for $S_{t-1}(m+1)$ since $S_{t-1}(m) \subseteq S_{t-1}(m+1)$. However, there are exactly $i - 1$ pages in $N - S_{t-1}(m)$ with priority greater than $p_t(z)$. Since $N - S_{t-1}(m+1) \subseteq N - S_{t-1}(m)$, there are at most $i - 1$ pages in $N - S_{t-1}(m+1)$ with priority greater than $p_t(z)$. Thus $z \notin Y_t(m+1)$.

REFERENCES

[1] J. L. Baer, *On program placement in a directly executable hierarchy of memories*, to appear.

[2] R. E. Fikes, H. C. Lauer and A. L. Vareha, *Steps towards a general purpose time-sharing system using large-capacity core storage and TSS/360*, Proc. 23rd National ACM Conf. ACM publication, 1968, pp. 7–18.

[3] H. C. Lauer, *Bulk-core in a 360/67 time-sharing system*, Proc. AFIPS 1967 Fall Joint Computer Conf., pp. 601–609.

[4] R. L. Mattson, J. Gecsei, D. R. Slutz and I. L. Traiger, *Evaluation techniques for storage hierarchies*, IBM Systems J., 9 (1970), pp. 78–117.

[5] A. L. Vareha, R. M. Rutledge and M. M. Gold, *Strategies for structuring two-level memories in a paging environment*, Proc. 2nd ACM Symp. on Operating Systems Principles, ACM publication, 1969, pp. 54–59.

[6] J. C. Williams, *Experiments in page activity determination*, Proc. AFIPS 1972 Spring Joint Computer Conf., AFIPS Press, Montvale, N.J., pp. 739–748.

SIAM J. COMPUT.
Vol. 3, No. 1, March 1974

PICTURE SKELETONS BASED ON ECCENTRICITIES OF POINTS OF MINIMUM SPANNING TREES*

R. E. OSTEEN† AND P. P. LIN‡

Abstract. Special properties of the eccentricities of points of trees are developed. An algorithm based on those properties is presented for the generation of all diametral paths of a given nontrivial tree.

The algorithm is adapted for a certain picture processing application. Given a discrete spatially quantized picture, a grey-distance is defined for neighboring picture cells to produce a graph with weighted lines. A minimum spanning tree of a connected component of the graph is submitted to the modified algorithm, which produces a skeleton of the picture object. The skeleton is further reduced to a smaller tree with weighted lines, viz., the unique tree homeomorphic to the skeleton having no point of degree two. This reduced skeleton with weighted lines constitutes a very compact representation of the picture object, facilitating object classification.

Key words. eccentricity, diametral path, tree, graph theory, minimum spanning tree, picture skeleton, feature extraction, picture processing

1. Introduction. A discrete spatially quantized picture is a matrix over a finite set of grey levels. This paper is concerned with a more compact representation of the objects of such a picture for purposes of pattern classification.

A grey-distance is defined for neighboring picture cells to produce a graph with weighted lines. A minimum spanning tree (MST) of a component of the weighted graph is then found. A pruning process then is performed on the MST to produce the object skeleton. Further substantial reduction in object representation is achieved by replacing branches of the skeleton by suitably weighted lines. The reduced skeleton—which is the unique tree homeomorphic to the skeleton having no point of degree two—constitutes the input to an object recognition or pattern classification process.

The remainder of the paper is organized as follows. First, theorems are presented concerning the special properties of the eccentricities of points of trees. These properties provide the basis for an efficient algorithm for the identification of all the diametral paths of a given nontrivial tree. This algorithm is then modified for the present application, resulting in the tree pruning algorithm for use on an MST of an object from a picture. The method is illustrated by means of a chromosome picture.

2. Definitions and theorems. The following definitions and terminology are as in Harary [2]. A *graph* $G = (\mathscr{V}(G), \mathscr{E}(G))$ consists of a nonempty finite set $\mathscr{V}(G)$ of *points* and a set of *lines* $\mathscr{E}(G) \subset \{\mathscr{V}' : \mathscr{V}' \subset \mathscr{V}(G) \text{ and } |\mathscr{V}'| = 2\}$. (A line $\{x, y\}$ is commonly rendered more briefly as xy.) A *walk* in G of *length* n is a sequence of points of $G, v_0, v_1, \cdots, v_n$, such that $v_{i-1}v_i \in \mathscr{E}(G)$ for each $i = 1, 2, \cdots, n$. A *path* in G is a walk in which no point has more than one occurrence, i.e., if $i \neq j$, then $v_i \neq v_j$ for $i, j = 0, 1, 2, \cdots, n$; a *cycle* in G is a walk of length

* Received by the editors January 11, 1973, and in revised form June 22, 1973. This work was done at the University of Florida Center for Informatics Research. It was supported by the Office of Naval Research through the Information Systems Program under Contract no. N00014-68-A-0173-0001, NR 049-172, and by the Army Research Office, Durham, under Grant no. DA-ARO-D-31-124-70-G92.

† Electrical Engineering Department, Florida Institute of Technology, Melbourne, Florida 32901.
‡ Bell Telephone Laboratories, New Brunswick, New Jersey 08903.

greater than two in which $v_0 = v_n$ but no other point has a multiple occurrence, i.e., if $0 < i < n$ and $i \neq j$, then $v_i \neq v_j$ for $j = 0, 1, \cdots, n$. G is a *connected* graph in case each pair of its points are joined by a path. That is, if $u \in \mathscr{V}(G)$, $v \in \mathscr{V}(G)$, and $u \neq v$, then there exists a path in $G, v_0, v_1, \cdots, v_n$, with $v_0 = u$ and $v_n = v$.

In a connected graph $G = (\mathscr{V}(G), \mathscr{E}(G))$, the *distance* $d(u, v)$ between points u and v, is defined to be zero if $u = v$ and to be the minimum of the lengths of paths joining u and v otherwise. The *eccentricity* $e(v)$ of a point v is the maximum over points u of $d(u, v)$. The *center* $\mathscr{C}(G)$ of G is the subset of $\mathscr{V}$ consisting of those points (the *central points*) of least eccentricity. The *radius* $R(G)$ is the least eccentricity and the *diameter* $D(G)$ is the greatest eccentricity of the points of G. A *radial path* in G is a path of length $R(G)$ from a central point to a point whose distance from the central point is $R(G)$; a *diametral path* in G is a path of length $D(G)$ joining a pair of points whose distance is $D(G)$.

A *tree* $T = (\mathscr{V}(T), \mathscr{E}(T))$ is a connected graph with no cycles. If $|\mathscr{V}(T)| = 1$, T is the *trivial tree*; otherwise, T is a *nontrivial* tree. A *leaf* or *endpoint* of a nontrivial tree is a point of degree one, i.e., a point adjacent to just one other point. A nontrivial tree has at least two endpoints. $\mathscr{L}(T)$ denotes the set of all endpoints of T. The *periphery* $\mathscr{P}(T)$ of the nontrivial tree T consists of the points of T (the *peripheral points*) having maximum eccentricity. Thus, $e(v) = D(T)$ if and only if $v \in \mathscr{P}(T)$.

Each pair of points of a nontrivial tree T are joined by a *unique* path. Consequently, if $x, y, \cdots, z$ is a path in T, if $w \neq y$ and w is adjacent to x, then $w, x, y, \cdots, z$ is the path in T from w to z. From this follows that if $d(u, v) = e(u)$, then $v \in \mathscr{L}(T)$.

If T has more than 2 points, then $\mathscr{L}(T) \neq \mathscr{V}(T)$, i.e., some points of T are not endpoints. If $x \in \mathscr{L}(T)$ and $xy \in \mathscr{E}(T)$, then $e(y) = e(x) - 1$. The removal from T of the points of any subset of $\mathscr{L}(T)$ produces a subtree of T. For each point x of T', the subtree of T obtained by removing all the endpoints of T, $e'(x) = e(x) - 1$, where e' denotes eccentricity in T'. In particular, $D(T') = D(T) - 2$ and $R(T') = R(T) - 1$.

Some other rather obvious consequences of the definitions are as follows, where T is a tree of more than two points. Every peripheral point is an endpoint. Each diametral path in T joins a pair of peripheral points. If u is a peripheral point, there is a peripheral point v to which u is joined by a diametral path. T has at least two peripheral points. The set of diametral paths of T coincides with the set of paths of T of greatest length.

More pertinent properties are stated in the following theorems, whose proofs are given in the Appendix.

THEOREM 1. *Let T be a tree of diameter $D(T)$, radius $R(T)$ and center $\mathscr{C}(T)$. Then*

(i) *if $D(T)$ is even, then $\mathscr{C}(T)$ is a singleton; if $D(T)$ is odd, then $\mathscr{C}(T)$ consists of a pair of adjacent points;*

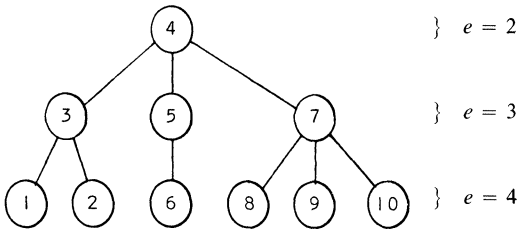
(ii) $e(x) = R(T) + \min\{d(x, c) : c \in \mathscr{C}(T)\}$, *for any point x of T; and*

(iii) *if $x_0, x_1, \cdots, x_{D(T)}$ is a diametral path in T, then for each $i = 0, 1, 2, \cdots, D(T)$, $e(x_i) = \max\{i, D(T) - i\}$.*

THEOREM 2. *If $u \in \mathscr{V}(T) - \mathscr{C}(T)$, then there is exactly one point v such that $uv \in \mathscr{E}(T)$ and $e(v) = e(u) - 1$.*

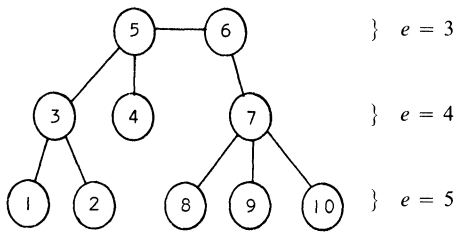
Because of Theorem 2, one can define on $\mathscr{V}(T) - \mathscr{C}(T)$ the *predecessor function*: $pred(x) = y \in \mathscr{V}(T)$ such that $e(y) = e(x) - 1$ and $xy \in \mathscr{E}(T)$. It is not uncommon to represent a graph by the sequence of (deleted) neighborhoods of the points, where the *neighborhood* $\mathscr{N}(x)$ of a point x consists of all those points y such that $xy \in \mathscr{E}(G)$. The predecessor function may then be specified as follows: if $x \in V(T) - C(T)$, then $pred(x)$ is the unique point $y \in \mathscr{N}(x)$ such that $e(x) = e(y) + 1$.

Indeed, the predecessor function permits a very compact representation of a tree. Let $V(T) = \{1, 2, \cdots, p\}$. Then the domain of the predecessor function can be extended to $V(T)$ as follows: if $x \in \mathscr{C}(T)$, then $pred(x) = 0$. This function, which amounts only to an array of p numbers, uniquely specifies the tree, i.e., provides a complete representation of the tree. (The possibility of such a compact representation is not astonishing, since $\mathscr{E}(T)$ consists of $p - 1$ lines.) The predecessor function is illustrated below; the first tree has a center of one point, the second has a center of two points.



x	$pred(x)$
1	3
2	3
3	4
4	0
5	4
6	5
7	4
8	7
9	7
10	7

(*a*) *A tree of one central point*



x	$pred(x)$
1	3
2	3
3	5
4	5
5	0
6	0
7	6
8	7
9	7
10	7

(*b*) *A tree of two central points*

FIG. 1. *Illustrations of the predecessor function*

With the aid of these theorems, a number of other propositions concerning a tree T of three or more points may be readily proved. Each radial path in T contains $\mathscr{C}(T)$. If x and y are adjacent points not both in $\mathscr{C}(T)$, then $|e(x) - e(y)| = 1$. Thus if $x \notin \mathscr{C}(T)$, then there is just one point $y \in \mathscr{N}(x)$ such that $e(x) = e(y) + 1$; for each other point $z \in \mathscr{N}(x)$, $e(z) = e(x) + 1$. A path from a point u of length

$e(u)$ contains $\mathscr{C}(T)$; in particular, $\mathscr{C}(T)$ is contained in any diametral path. A radial path joins a central point and a peripheral point. If $d(u, v) = e(u)$, then $v \in \mathscr{P}(T)$. Suppose that $\mathscr{P}(T) \subset \mathscr{A} \subset \mathscr{L}(T)$ and let $T' = T - \mathscr{A}$, the subtree of T obtained by removing from T each point of $\mathscr{A}$; then for each point x in T', $e'(x) = e(x) - 1$; in particular, $R(T') = R(T) - 1$ and $D(T') = D(T) - 2$. Suppose $3 \leqq |\mathscr{P}(T)|$; there is a point $x \in \mathscr{P}(T)$ such that $D(T - x) = D(T)$, $\mathscr{P}(T - x)$ $\mathscr{P}(T) - x$, and $e'(z) = e(z)$ for each point z of $T' = T - x$. Suppose $x \in \mathscr{L}(T)$, $xy \in \mathscr{E}(T)$, and $2 < \deg(y)$; then $D(T - x) = D(T)$, and $e'(z) = e(z)$ for each point z of $T' = T - x$.

Furthermore, if $D(T)$ is even, then $R(T) = D(T)/2$; if $D(T)$ is odd, then $R(T) = (1 + D(T))/2$ (a proof may be found in Ore [4]).

By means of the preceding, one may verify easily the following additional properties. A path containing the center and joining a central and a peripheral point is a radial path. A path containing the center and joining a pair of peripheral points is a diametral path.

Finally, one has the following proposition (again, the proof is given in the Appendix).

THEOREM 3. *If x, y and z are distinct peripheral points of T and $d(x, y) = D(T)$, then $d(x, z) = D(T)$ or $d(y, z) = D(T)$.*

Now, let T be a nontrivial tree of diameter $D(T)$ with periphery $\mathscr{P}(T)$. Define the binary relation Q on $\mathscr{P}(T)$ as follows: if $x, y \in \mathscr{P}(T)$, xQy if and only if $d(x, y) < D(T)$. Since $d(x, x) = 0 < D(T)$, Q is reflexive. Since $d(x, y) = d(y, x)$, Q is symmetric. Because of the preceding theorem, Q is transitive. Hence, Q is an equivalence relation.

Suppose now that $\mathscr{C}(T)$ consists of two points, a and b. Since $ab \in \mathscr{E}(T)$ and T has no cycles, for no point x is $d(x, a) = d(x, b)$. Define

$$\mathscr{R}_a = \{x : x \in \mathscr{P}(T),\ d(x, a) < d(x, b)\},$$

and

$$\mathscr{R}_b = \{x : x \in \mathscr{P}(T),\ d(x, b) < d(x, a)\};$$

clearly, $\mathscr{P}(T)$ is the disjoint union of $\mathscr{R}_a$ and $\mathscr{R}_b$. Suppose $u, v \in \mathscr{R}_a$. Then $d(u, a) = d(v, a) = d(u, b) - 1 = R(T) - 1 = (D(T) - 1)/2$. Since d is in fact a metric,

$$d(u, v) \leqq ((D(T) - 1)/2) + ((D(T) - 1)/2) = D(T) - 1 < D(T),$$

so that uQv. Similarly, if $u, v \in \mathscr{R}_b$, uQv. On the other hand, suppose $u \in \mathscr{R}_a$ and $v \in \mathscr{R}_b$. Then $d(u, a) = (D(T) - 1)/2$, $d(v, b) = (D(T) - 1)/2$, $d(a, b) = 1$; and since u and v are joined by just one path, $d(u, v) = ((D(T) - 1)/2) + ((D(T) - 1)/2) + 1 = D(T)$. Thus $(u, v) \notin Q$ if $u \in \mathscr{R}_a$ and $v \in \mathscr{R}_b$. Evidently, the equivalence classes of the relation Q are the sets $\mathscr{R}_a$ and $\mathscr{R}_b$.

Now suppose that the nontrivial tree T has a single central point, c. For each $x \in \mathscr{N}(c)$, the neighborhood of c, define

$$\mathscr{R}_x = \{y : y \in \mathscr{P}(T),\ d(y, x) < d(y, z) \text{ if } x \neq z \in \mathscr{N}(c)\}.$$

That $\mathscr{R}_x$ is well-defined follows from the fact that if $y \neq c$ and x and z are distinct points adjacent to c, then $d(y, x) \neq d(y, z)$. Indeed, if $y \in \mathscr{R}_x$ and $x \neq z \in \mathscr{N}(c)$, then $d(y, x) = d(y, z) - 2$. Let $W = x_0, x_1, \cdots, x_{n-1}, x_n, x_{n+1}, \cdots, x_{2n}$ be a

diametral path in T, where $D(T) = 2n$. Then $x_n = c$, and x_{n+1} and x_{n-1} are distinct points of $\mathcal{N}(c)$. Evidently, $x_0 \in \mathcal{R}_{x_{n-1}}$ and $x_{2n} \in \mathcal{R}_{x_{n+1}}$. Thus, there are at least two points x, y in $\mathcal{N}(c)$ such that $\mathcal{R}_x \neq \varnothing \neq \mathcal{R}_y$. As in the case $|\mathcal{C}(T)| = 2$, one may prove that the collection $\{\mathcal{R}_x : x \in \mathcal{N}(c), \mathcal{R}_x \neq \varnothing\}$ partitions $\mathcal{P}(T)$ into the equivalence classes of the relation Q.

In view of the foregoing, it is convenient to define the *hub* $\mathcal{H}(T)$ of a nontrivial tree T, as follows:

$$\mathcal{H}(T) = \begin{cases} \mathcal{C}(T) & \text{if } |\mathcal{C}(T)| = 2, \\ \mathcal{N}(c) & \text{if } \mathcal{C}(T) = \{c\}. \end{cases}$$

The collection of equivalence classes of Q is then given by $\Pi = \{\mathcal{R}_x : x \in \mathcal{H}(T), \mathcal{R}_x \neq \varnothing\}$, where $\mathcal{R}_x$ is defined as above. If $|\mathcal{C}(T)| = 2$, $|\Pi| = 2$. If $\mathcal{C}(T) = \{c\}$, then $2 \leqq |\Pi| \leqq \deg(c) = |\mathcal{N}(c)|$.

Knowing $\mathcal{H}(T)$, $\mathcal{P}(T)$, and the predecessor function, one may determine Π with very little effort; in particular, the distance function is not needed.

From Π, $\mathcal{H}(T)$ and the predecessor function, one may readily generate all diametral paths: if and only if $x \in \mathcal{R}_i$, $y \in \mathcal{R}_j$, for some $\mathcal{R}_i, \mathcal{R}_j \in \Pi$ with $\mathcal{R}_i \neq \mathcal{R}_j$, then x and y are joined by a diametral path. The particular diametral path is easily generated via the predecessor function: beginning with $z = x$, one replaces z by $pred(z)$ until $z \in \mathcal{H}(T)$; the same is done beginning with $z = y$; these two shortest paths from the peripheral points x and y to points in the hub, together with the unique central point c if $|\mathcal{C}(T)| = 1$, form the diametral path joining x and y. Thus we obtain the following definitions. Let $x \in \mathcal{P}(T)$ and $y \in \mathcal{H}(T)$, with $x \in \mathcal{R}_y \in \Pi$; the path from x to y is the *spoke from* x, and the path from y to x is the *spoke to* x. If $\mathcal{R}_y \in \Pi$ then the *spoke set* $\Sigma(\mathcal{R}_y)$ consists of the spokes from x to y for each $x \in \mathcal{R}_y$. It is evident that all spokes are the same length, $R(T) - 1$, and y. Thus we obtain the following definitions. Let $x \in \mathcal{P}(T)$ and $y \in \mathcal{H}(T)$.

The hub $\mathcal{H}(T)$ is immediate, given the center $\mathcal{C}(T)$. To find the center and simultaneously determine the predecessor function, one may proceed as suggested in the proof of Theorem 1. A tower of subtrees, $S_0, S_1, \cdots, S_n$, is constructed, with $S_0 = T$ and $S_n = K_1$ or $S_n = K_2$, by removing the endpoints of S_i to obtain S_{i+1}. The point or pair of points of S_n constitute the center of T. For central points of T, $pred(x) = 0$; otherwise, if and only if $x \in \mathcal{L}(S_i)$ and $xy \in \mathcal{E}(S_i)$ for some i, $y = pred(x)$. Clearly, the number n of leaf-strippings is the sum of the spoke length and $2 - |\mathcal{C}(T)|$. That is, $R(T) = n + |\mathcal{C}(T)| - 1$ and $D(T) = n + R(T)$.

The preceding definitions and propositions may now be integrated into an algorithm for the generation of all the diametral paths of a given nontrivial tree T. The algorithm may be summarized as follows, in six steps.

Step 1. Initialize the subtree S to be the tree T.

Step 2. Find $\mathcal{C}(T)$, the predecessor function, $R(T)$, and $D(T)$.

Step 3. Find $\mathcal{P}(T)$ and the eccentricity of each point of T.

Step 4. Find the hub $\mathcal{H}(T)$.

Step 5. Find the spoke sets.

Step 6. Combine the spokes into the diametral paths.

3. Tree diametral paths algorithm. The algorithm is given below in detail. The nontrivial tree T of PT points, $1, 2, \cdots, PT$, is assumed to be represented by

the neighborhoods $NT(I)$, for $I = 1, 2 \cdots, PT$. $DT(I)$ denotes the degree of point I in T. The subtree S of T (Step 2) consists of PS points of T; $NS(I)$ and $DS(I)$ denote the neighborhood and the degree of point I of S.

ALGORITHM 1 (Tree diametral paths algorithm).

Step 1. Initialize.

1.1. Set $RP = 0$. (After Step 2, RP = number of leaf strippings.)
1.2. Set $PS = PT$.
1.3. For each $I = 1, 2, \cdots, PT$:
1.3.1. Set $PRED(I) = 0$. (After Step 2, only central points have $PRED = 0$.)
1.3.2. Set $DS(I) = DT(I)$.
1.3.3. Set $NS(I) = NT(I)$.
1.4. END OF STEP.

Step 2. Find the center, diameter and radius, and determine the predecessor function.

2.1. If $PS \leqq 2$, then go to Step 2.2.
2.1.1. Add 1 to RP.
2.1.2. Initialize X to be the set of endpoints of S.
2.1.2.1. Set $X = \varnothing$.
2.1.2.2. For $I = 1, 2, \cdots, PT$:
2.1.2.2.1. If $DS(I) = 1$, then set $X = X \cup \{I\}$.
2.1.3. For each I in X:
2.1.3.1. Subtract 1 from PS.
2.1.3.2. Set $PRED(I) = J$, the single member of $NS(I)$.
2.1.3.3. Set $DS(I) = 0$.
2.1.3.4. Subtract 1 from $DS(J)$.
2.1.3.5. Remove point I from $NS(J)$.
2.1.4. Go to Step 2.1.
2.2. Set $NC = 0$.
2.3. Set $C = \varnothing$.
2.4. For $I = 1, 2, \cdots, PT$:
2.4.1. If $PRED(I) = 0$:
2.4.1.1. Add 1 to NC.
2.4.1.2. Set $C = C \cup \{I\}$.
2.5. Set $RAD = RP + NC - 1$.
2.6. Set $DIAM = RAD + RP$.
2.7. END OF STEP.

Step 3. Find the periphery and the eccentricity.

3.1. Set $ECX = RAD$ (eccentricity of central points is $R(T)$).
3.2. Set $DONE = \varnothing$.
3.3. Set $SUCC = C$ (the successors are initially the central points).
3.4. Set $X = SUCC$.
3.5. For each I in $SUCC$:
3.5.1. Set $ECC(I) = ECX$.
3.5.2. Remove point I from $SUCC$.
3.6. Set $DONE = DONE \cup X$.
3.7. Set $PER = X$ (finally, the periphery).
3.8. For each I in X:

3.8.1. Set $SUCC = SUCC \cup NT(I)$.
3.9. Set $SUCC = SUCC - DONE$. (Points of eccentricity $ECX + 1$.)
3.10. If $SUCC \neq \varnothing$, then go to Step 3.4.
3.11. END OF STEP.
Step 4. Find the hub.
4.1. If $NC = 1$, then go to Step 4.2.
4.1.1. Set $HUB = C$.
4.1.2. Set $NH = 2$ (hub size).
4.1.3. For $I = 1, 2$:
4.1.3.1. Set $SP(I) = \varnothing$. (Initially, each set of spokes is null.)
4.1.4. END OF STEP.
4.2. The center consists of one point.
4.2.1. Set $HUB = NT(I)$, where I is the single element of C.
4.2.2. Set $NH = DT(I)$, where I is the one central point.
4.2.3. For $I = 1, 2, \cdots, NH$:
4.2.3.1. Set $SP(I) = \varnothing$.
4.2.4. END OF STEP.
Step 5. Find the spoke sets.
5.1. For each I in PER: (Trace via $PRED$ the spoke.)
5.1.1. For each $J = 1, 2, \cdots, RAD$:
5.1.1.1. Set the Jth point of the spoke to I.
5.1.1.2. Set $I = PRED(I)$.
5.1.2. Set I to be the last point of the spoke.
5.1.3. For each $J = 1, 2, \cdots, NH$:
5.1.3.1. If $I = HUB(J)$, then set $K = J$.
5.1.4. Include the spoke in $SP(K)$.
5.2. END OF STEP.
Step 6. Combine the spokes into diametral paths.
6.1. Set $DP = \varnothing$ (The set of diametral paths).
6.2. For $I = 1$ to $NH = 1$:
6.2.1. For each spoke α in $SP(I)$:
6.2.1.1. For $J = J + 1$ to NH:
6.2.1.1.1. For each spoke β in $SP(J)$:
6.2.1.1.1.1. If $NC = 1$, then set the $(RAD + 1)$st point of the point sequence of the diametral path to the unique central point.
6.2.1.1.1.2. For $K = 1$ to RAD:
6.2.1.1.1.2.1. Set the Kth point of the diametral path point sequence to the Kth point of spoke α.
6.2.1.1.1.2.2. Set the $(DIAM + 2 - K)$th point of the diametral path point sequence to the Kth point of spoke β.
6.2.1.1.2. Include the new diametral path in DP, the collection of all diametral paths.
6.3. END OF ALGORITHM.

4. Semidiametral paths. For those applications which require it, one may readily generalize the concepts of diametral paths, radial paths, and spokes in trees.

An *eccentric path* from a point u is a path of length $e(u)$ joining points u and v, when $d(u, v) = e(u)$. An eccentric path from an endpoint u is termed a *semidiametral path.* A *ray* is a path joining an endpoint with the central point at the greatest distance from the endpoint. Clearly, $\mathscr{C}(T)$ is contained in each ray. It is apparent that semidiametral paths and rays generalize diametral paths and radial paths, respectively. As previously remarked, an eccentric path from a point u joins u with a peripheral point and contains $\mathscr{C}(T)$. Analogously to diametral paths, a path containing $\mathscr{C}(T)$ and joining an endpoint and a peripheral point is a semidiametral path.

The equivalence relation Q on $\mathscr{P}(T)$ may be easily extended to $\mathscr{L}(T)$ as follows. Let u be an endpoint; then there exists a *unique* point $x \in \mathscr{H}(T)$ such that $d(u, x) \leqq d(u, y)$ for all $y \in \mathscr{H}(T)$, because of the acyclicity of T. Define, for $x \in \mathscr{H}(T)$, $\mathscr{R}'_x = \{y : y \in \mathscr{L}(T)$, and $d(y, x) < d(y, z)$ if $x \neq z \in \mathscr{H}(T)\}$; then, since $\mathscr{R}'_x \neq \varnothing$ for any $x \in \mathscr{H}(T)$, $\Pi' = \{\mathscr{R}'_x : x \in \mathscr{H}(T)\}$ partitions $\mathscr{L}(T)$. The extended relation Q' on $\mathscr{L}(T)$ of Q on $\mathscr{P}(T)$ is the equivalence relation induced by the partitioning Π' of $\mathscr{L}(T)$. (Although the present application does not require it, one could similarly extend Q' to the set of all points except the central point, if the tree has a unique central point.) A *semispoke* from an endpoint u is the path joining u and x, where $x \in \mathscr{H}(T)$ and $u \in \mathscr{R}'_x$, i.e., the shortest path from u to a point of the hub. As before, a semispoke is generated without difficulty by means of the predecessor function and the knowledge of the hub. For $x \in \mathscr{H}(T)$, the semispoke set $\Sigma'(\mathscr{R}'_x)$ consists of the semispokes from y to x for each $y \in \mathscr{R}'_x$. A semidiametral path is then formed from a spoke and a semispoke (possibly a spoke), from different semispoke sets $\Sigma'(\mathscr{R}'_x)$ and $\Sigma'(\mathscr{R}'_y)$.

5. Picture skeletons. An n-level digitized picture can formally be represented by a matrix g whose elements take the values $0, 1, \cdots, n-1$. That is, a photograph is subjected to spatial sampling, and a quantization of the intensity. Figure 2 is an 8-level picture of a human chromosome—from Levi and Montanari [3].

As in Zahn [5], a weighted graph is formed from the picture array. The points of the weighted graph are the elements of the picture of nonzero grey-level or with a "neighbor" of nonzero grey-level. An element (r, s) of a picture array is a neighbor of element (i, j) in case element (r, s) is one of the eight elements nearest to element (i, j), i.e.,

$$0 < [(i - r)^2 + (j - s)^2]^{1/2} < 2.$$

Two points in the graph are joined by a line if the corresponding elements in the picture array are neighbors and at least one element has a nonzero value. The line weight of a pair of adjacent points in the graph is equal to the "grey-distance" between the corresponding elements in the picture array. A grey-distance appropriate to the example of Fig. 2 is the Euclidean distance divided by the average intensity. That is, if (i, j) and (r, s) are neighbors not both of zero intensity, then

$$d((i, j), (r, s)) = \frac{[(i - r)^2 + (j - s)^2]^{1/2}}{\frac{1}{2}(g(i, j) + g(r, s))}.$$

A spanning forest of a graph is a forest having the same point set as the graph, no line not in the graph and preserving the connectivity of components

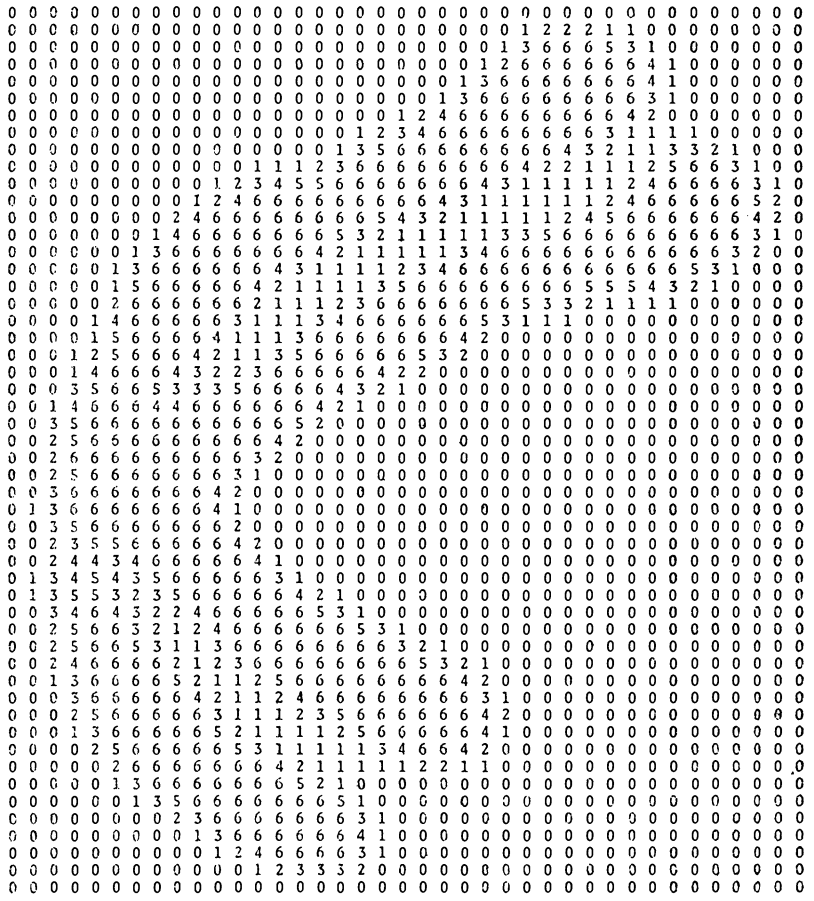


FIG. 2. *An 8-level picture of a human chromosome*

in the graph. A minimum spanning forest (MSF) of a graph with weighted lines is a spanning forest of minimum weight, the weight of a forest being the sum of the weights of its lines. A component of an MSF of a weighted graph is a minimum spanning tree (MST) of a component of the graph. An efficient algorithm is given in Gower and Ross [1] for the identification of an MST of a connected weighted graph. This algorithm is easily extended to produce an MSF of the weighted graph formed from a picture array. An MST of reasonable size in such an MSF can be considered as a tree structure representation of an object in the picture. For example, an MST representing the chromosome of Fig. 2 is given in Fig. 3.

An additional level of feature extraction condenses further the MST representation of a picture object as in Fig. 3 to an MST "skeleton". A *skeleton* T' of a tree T is a subtree of T whose points have the same eccentricities in T' as they do in T, i.e., a subtree of T which includes at least one diametral path of T; moreover, each endpoint of T' is an endpoint of T. A skeleton T' of T may be specified by a subset $\mathscr{L}(T')$ of the set $\mathscr{L}(T)$ of all endpoints of T. That is, T' is the subtree of T generated by $\mathscr{L}(T') \subset \mathscr{L}(T)$, by which is meant the minimal subtree of T

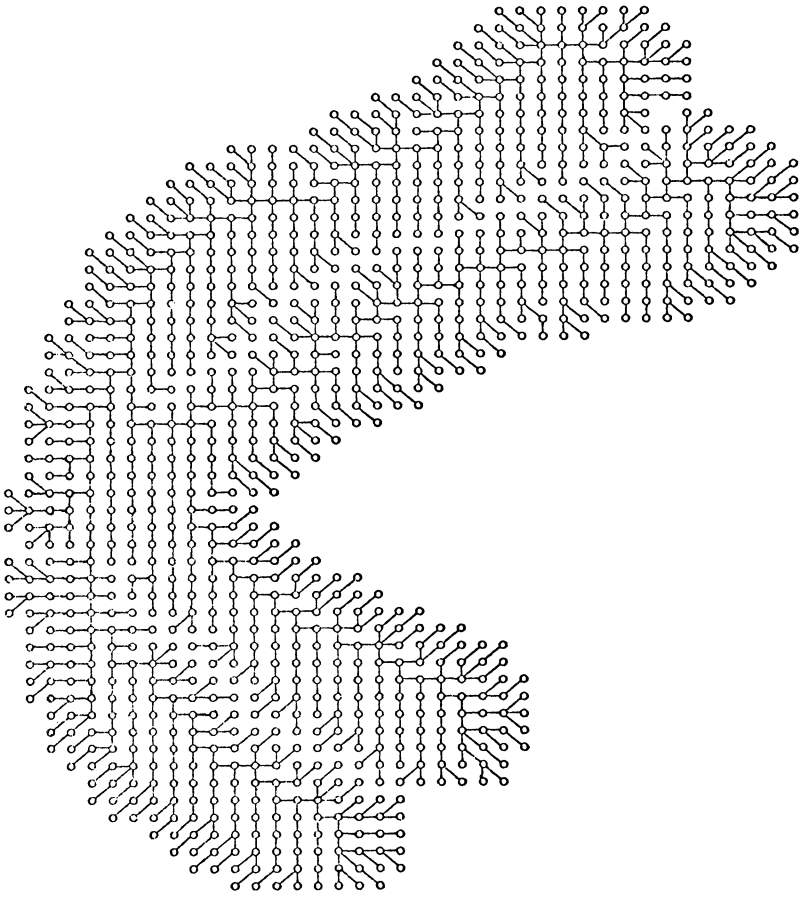


FIG. 3. *An MST obtained from Fig.* 2

which includes all the points in $\mathscr{L}(T')$, viz., the subtree generated by the set of all points of T lying on paths joining points of $\mathscr{L}(T')$. For example, if $\mathscr{L}(T') = \mathscr{L}(T)$, then $T' = T$; if $\mathscr{L}(T') = \mathscr{P}(T)$, then T' consists of the union of all the diametral paths of T.

It remains to consider the method of determining the subset $\mathscr{L}(T')$ of $\mathscr{L}(T)$ for the specification of a skeleton of an MST of a picture object. First, the requirement that $\mathscr{L}(T') \subset \mathscr{P}(T)$ is overly constraining, since not all the "legs" of the picture will be of the same "length". That is, an MST with only two or three peripheral points may have a few other endpoints of large eccentricity which reflect indispensable features of the object. On the other hand, the condition $\mathscr{P}(T) \subset \mathscr{L}(T')$ is also unjustified. For example, if two peripheral points are adjacent to the same third point, not both are required to reflect the feature of the one leg of which the two peripheral points are close boundary points. Thus, the endpoints of a skeleton of a tree T need not include all the peripheral points of T, and may include nonperipheral endpoints of T. Rather than imposing such a relationship on $\mathscr{L}(T')$ and $\mathscr{P}(T)$, a more pertinent property of T' is that it have no "short branches".

A *branch* is a path joining nodes or endpoints and having no intermediate nodes, a node being a point of degree greater than two. An *interior* branch joins two nodes; an *exterior* branch has an endpoint as one extreme point (an exterior branch joins two endpoints only in case the tree has just two endpoints). A *short branch* is an exterior branch of length less than or equal to $\theta = \lfloor \alpha R(T) \rfloor$, where $0 < \alpha < 1$.

The pruning algorithm below produces a skeleton T' from a tree T by iteratively deleting short branches. If the current subtree has an exterior branch of length one, it is trimmed; this continues until there is no exterior branch of length one. Then if there is an exterior branch of length two, it is trimmed; this continues until there is no exterior branch of length less than or equal to two. This process continues through the lengths $k = 1, 2, \cdots, \theta$, to produce a skeleton having no exterior branch of length less than or equal to $\alpha R(T)$. The pruning algorithm was applied to the MST of Fig. 3 with $\alpha = \frac{1}{4}$; the resulting skeleton, shown in Fig. 4, has only 4 endpoints and 2 nodes.

The pruning algorithm is similar to Step 5 of the tree diametral paths algorithm (§ 3) which finds the spoke sets of a tree: both proceed from endpoints toward the center by means of the predecessor function. The pruning algorithm

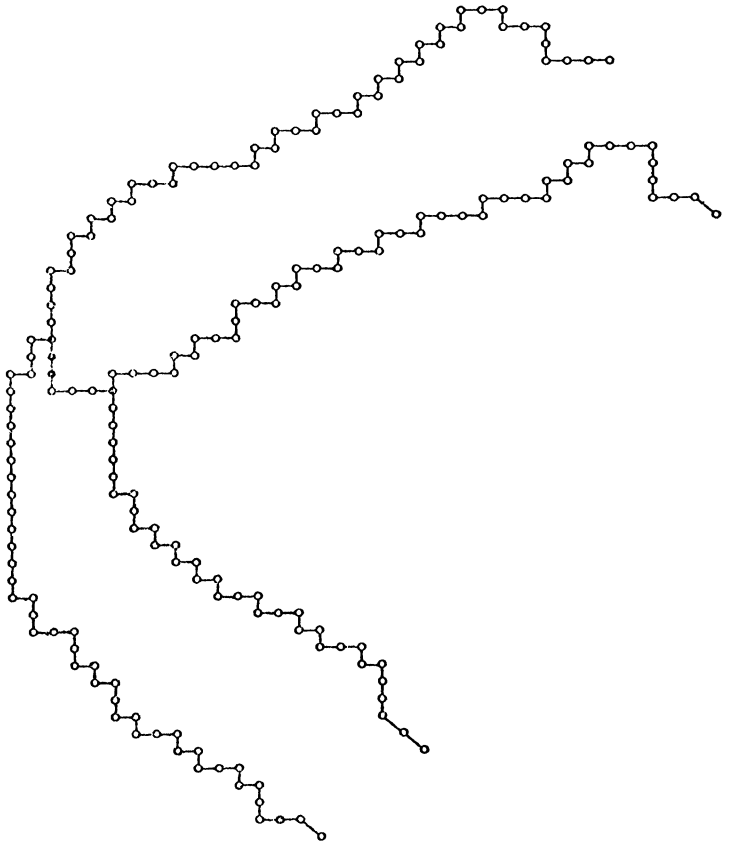


FIG. 4. *The skeleton of the picture shown in Fig.* 2

starts a path from each endpoint of T and traces a path toward the center, either until a node is encountered or until the path length is $\lfloor \alpha R(T) \rfloor$. Those endpoints of T which survive to fulfill the latter condition constitute the endpoints of the skeleton T' of T.

ALGORITHM 2 (Tree pruning algorithm). The algorithm terminates with the skeleton T' of the given tree T specified as follows. The M endpoints of T' are named in $VO(I)$, for $I = 1, 2, \cdots, M = |\mathscr{L}(T')|$; $\mathscr{L}(T')$ together with T completely defines the skeleton T'. If v_i is a point of T', then the eccentricity of v_i in T' is the same as its eccentricity in T; in particular, T' has the same radius, diameter and center as T. Moreover, the degree of v_i in T' is given by $DS(I)$; and the predecessor of v_i in T' is its predecessor in T.

Step 1. Initialize VO and VN to the sequence of all endpoints of T; $DS = DT$ (ultimately, DS gives the degrees of the points of the skeleton); and $M = |\mathscr{L}(T)|$.

Step 2. For each $k = 1, 2, \cdots, \lfloor \alpha R(T) \rfloor$:

2.1. Set $N = M$; $M = 0$.

2.2. For each $J = 1, 2, \cdots, N$:

2.2.1. If $2 < DS(PRED(VN(J)))$ then execute 2.2.1.1; otherwise, execute 2.2.1.2.

2.2.1.1. Decrement $DS(PRED(VN(J)))$.

2.2.1.2. (Continue this path another step toward the center.)

2.2.1.2.1. Increment M.

2.2.1.2.2. Set $VN(M) = PRED(VN(J))$.

2.2.1.2.3. Set $VO(M) = VO(J)$.

Step 3. END OF ALGORITHM.

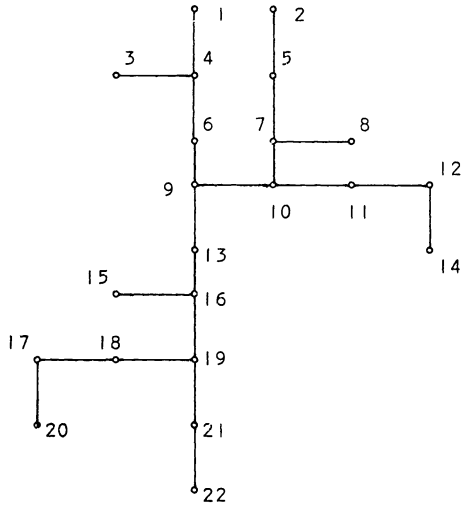
The action of the pruning algorithm is illustrated in Fig. 5, using a tree of radius 5 and $\alpha = \frac{1}{2}$.

The skeleton may be more conveniently represented by the specification of its branches. For example, the branches of the skeleton of Fig. 5 join the pairs of points (3, 9), (2, 10), (14, 10), (20, 9) and (9, 10). The identification of the branches and their lengths is easily and efficiently achieved by means of the predecessor function, the degree function and the set of skeleton endpoints.

The skeleton T' may be further usefully condensed to a tree T'' with weighted lines as follows: the points of T'' are adjacent in case they are the extreme points of a branch of T', i.e., lines of T'' correspond to branches of T'. T'' may be termed the *form* of T', for the following reasons: T' and every tree homeomorphic to T' may be obtained from T'' by a sequence of subdivisions of lines of T'', and T'' is the unique tree homeomorphic to T' having no point of degree 2.

The weights of the lines of the form T'' of the skeleton T' of the MST T of a picture object may be the lengths of the corresponding branches in T', i.e., the numbers of lines of the respective branches of T'. To eliminate the dependency of the weights upon the fineness of the picture grid and upon the size of the object, these line weights may be normalized with respect to the radius of T'.

For example, Fig. 4 gives a skeleton T' of the MST T (Fig. 3) of a picture (Fig. 2) of a human chromosome. $R(T') = 51$; the exterior branch lengths are 49, 46, 34 and 45; and the interior branch length is 6. The form T'' of T' is given in Fig. 6 with the normalized branch lengths in T' for the line weights.



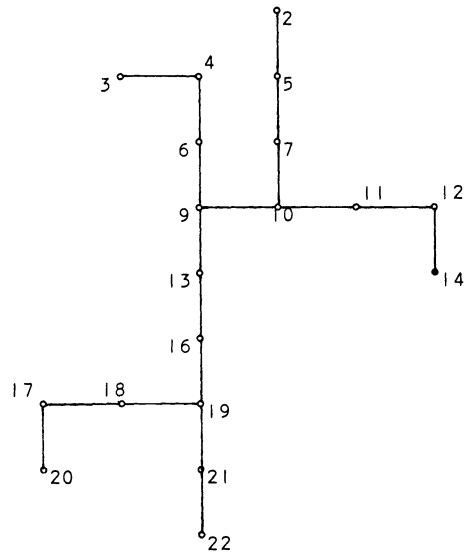
(a) *Tree T*

I	DT(I)	*PRED(I)*
1	1	4
2	1	5
3	1	4
4	3	6
5	2	7
6	2	9
7	3	10
8	1	7
9	3	13
10	3	9
11	2	10
12	2	11
13	2	0
14	1	12
15	1	16
16	3	13
17	2	18
18	2	19
19	3	16
20	1	17
21	2	19
22	1	21

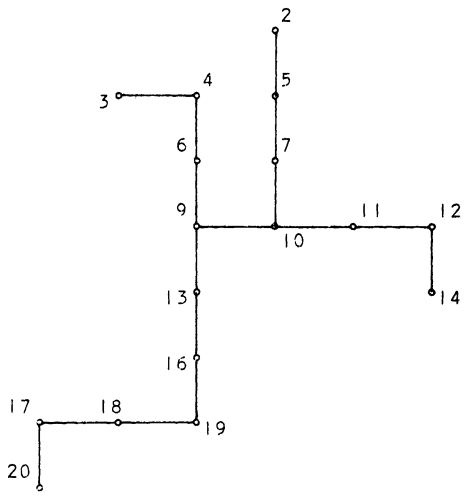
(b) *Degrees and predecessors*

	VN	VO
INITIAL	1, 2, 3, 8, 14, 15, 20, 22	1, 2, 3, 8, 14, 15, 20, 22
K = 1	5, 4, 12, 17, 21	2, 3, 14, 20, 22
K = 2	7, 6, 11, 18	2, 3, 14, 20

(c) *Formation of L(T') by the pruning algorithm*



(d) *The tree after the first pruning iteration*



(e) *The tree after the second (final) pruning iteration*

FIG. 5. *An illustration of the pruning algorithm*

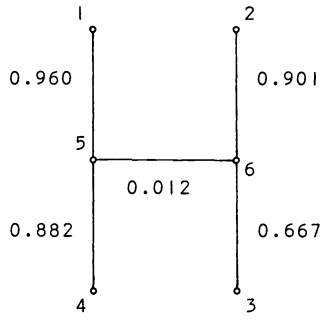


FIG. 6. *The weighted form of the skeleton of Fig.* 4

The chromosome example illustrates the tremendous degree of condensation of representation—from the picture object array (Fig. 2), to the form (Fig. 6) of the skeleton of the MST (Fig. 4). Furthermore, the skeleton—and so, the form—is apparently not very sensitive to changes in the threshold α, $0 < \alpha < 1$: the skeleton of Fig. 4 is the output of the pruning algorithm for all values of α no greater than 0.64 and no less than 0.22 (i.e., for any number of iterations of the algorithm not less than 11 and not greater than 33).

The line weight suggested above—the graph theoretical distance between the terminal points of a branch of the skeleton—could easily be supplemented by the Euclidean distance between the terminal points of a branch. Indeed, the points of the form could be labeled with their readily available picture array coordinates, thereby permitting the subsequent calculation of Euclidean distances and angular displacements among the lines, if the specific picture problem can profitably utilize such data for the pattern classification task.

Although feature extraction is the main concern here, a few remarks concerning pattern classification are in order. Suppose that classification is performed by reference to the forms, $S_1'', S_2'', \cdots, S_n''$, of typical patterns from the respective pattern classes. To classify a pattern, an MST T is first found. Then rather than use a threshold to produce a single skeleton T' from the pruning algorithm, a sequence of skeletons $T_1', T_2', \cdots$, is formed by iterating the pruning algorithm $R(T) - 1$ times or until the resulting skeleton has fewer endpoints than any of the typical forms S_i''.

Beginning with the last member of the sequence T_j' having more endpoints than any of the typical forms and proceeding to the end of the sequence, a form T_j'' is formed for each distinct element of the sequence of skeletons. After deleting repetitions from the sequence of forms, one has a tower of forms $T_1'', T_2'', \cdots, T_k''$: for each $k = 1, 2, \cdots, k - 1$, T_{k+1}'' is a subtree of T_k'' having strictly fewer points than does T_k''.

Finally, classification proceeds as follows. For each class i, one finds the last member of the tower T_k'' having more endpoints than does S_i'' and the first having fewer; the resulting subsequence of the tower has two or three members. Each of them is matched against S_i''; the score of the closest match is then taken to be the score $s(T, S_i'')$ of the MST T with respect to the ith pattern class. This method eliminates the necessity to choose a threshold, and considers for each class the most appropriate form of T for the match.

6. Conclusions. Certain very useful special properties of eccentricities of points in trees have been identified. As a consequence of these properties, a compact and productive representation of trees—the predecessor function—has been provided; and an efficient algorithm has been presented for the determination of the center, radius, diameter, and periphery of a tree, as well as its predecessor function and the set of all its diametral paths.

The theory has been applied to a feature extraction problem in picture processing. A tree pruning algorithm iteratively trims short branches joining endpoints and nodes from a minimum spanning tree of a picture object, producing a "skeleton" of the picture object. This MST skeleton differs from the skeleton produced by the method of Levi and Montanari [3] in several respects. The Levi–Montanari skeleton includes object width information, from which the picture can be approximately reconstructed. On the other hand the MST skeleton for a connected picture object is itself connected and is everywhere one spot wide. Indeed, the fact that the MST skeleton is a tree permits even further contraction of the object representation preparatory to pattern classification.

Specifically, a method was described to condense an MST skeleton for matching purposes into a small weighted tree—the "form" of the skeleton, representing the topographical essence of the picture object. An example—a picture of a human chromosome—was provided to illustrate the method of abstracting a skeleton from an MST, and the form from the skeleton. The degree of condensation of object representation was seen to be very large—from the many hundreds of points of the MST, to the couple of hundred points of the skeleton, to the six points of the form of the skeleton. Still the form was seen to preserve the essentials of the shape of the object. Indeed, the (weighted) form of the skeleton of the MST of the picture object resembles a "stick drawing" of the object—an eminently appropriate form of feature extraction output for certain picture processing problems.

Applicability of the scheme requires that the object pictures be amenable to an MST representation for purposes of pattern classification. That is, the grey-level representation of objects must permit the application of a suitably chosen grey-distance function to produce weighted graphs whose minimum spanning trees adequately reflect the key features of the objects.

In the chromosome example of Fig. 2, the general character of the picture is that the grey-level increases from the boundary toward the interior. Defining the grey-distance to vary inversely with the average grey-level of a pair of neighboring points assures the existence of a long path in any MST from the central region along each of the four "arms" of the chromosome (see Figs. 3 and 4). (On the other hand, this particular grey-distance is altogether unsuitable for use with objects having high grey-levels near the boundary and low levels in the interior.)

The fairly large region in the interior of the chromosome having uniform high grey-level illustrates a limitation on the general strategy of deriving skeletons from minimum spanning trees in grey-distance weighted graphs for picture objects: the nonuniqueness of the MST. (Indeed, the number of distinct MST's for the chromosome example is indisputably unmanageably large.)

The processing order of picture points in our programmed implementation of the MST algorithm produces an MST with the long paths through the regions

of maximum grey-level hugging the upper boundaries of those regions, leading to a skeleton which is maximally upward displaced relative to the picture. Reversing the scanning order (or rotating the picture through 180°) would lead to a skeleton which is maximally downward displaced relative to the picture. Similarly, scanning columnwise from left to right or from right to left would produce skeletons maximally left or right displaced.

The unfortunate significance of this is that the skeleton is not fixed relative to the picture, independently of the angular orientation of the picture in the plane. It is not clear whether or not it would be feasible—and if so, whether or not it would be profitable—to modify the MST skeleton process so as to obtain a skeleton well centered within the object. One way in which this might be done is (i) to generate the four MST's maximally displaced to the left, right, top and bottom by varying the scanning order of the MST algorithm; (ii) to produce a skeleton from each of the four trees; and (iii) to combine the four skeletons into a single well centered skeleton.

However, in spite of the dependence of the skeleton on orientation due to the multiplicity of MST's, the choice of any MST whatsoever for the chromosome picture results in the production by the pruning algorithm—for a wide range of "short branch" thresholds—of a skeleton whose further reduction to a "form" is either (i) an "H" as in Fig. 6, with four legs of comparable length and a relatively short "bar"; or (ii) an "X" with four legs of comparable length.

Appendix. Proofs of theorems.

THEOREM 1. *Let T be a tree of diameter $D(T)$, radius $R(T)$ and center $\mathscr{C}(T)$. Then*

(i) *if $D(T)$ is even, then $\mathscr{C}(T)$ is a singleton; if $D(T)$ is odd, then $\mathscr{C}(T)$ consists of a pair of adjacent points;*

(ii) $e(x) = R(T) + \min\{d(x, c): c \in \mathscr{C}(T)\}$, *for any point x of T; and*

(iii) *if $x_0, x_1, \cdots, x_{D(T)}$ is a diametral path in T, then for each $i = 0, 1, \cdots, D(T)$, $e(x_i) = \max\{i, D(T) - i\}$.*

Proof. The proof is by induction on the diameter.

First, the only trees of diameter less than 2 are K_1 and K_2, the complete graphs on 1 and on 2 points. Obviously, all three statements hold for these two trees.

As induction hypothesis, assume that the statements hold for all trees of diameter less than $2k$, where k is some particular natural number. Let T be a tree of diameter $2k$ or $2k + 1$, and T' be the subtree of T obtained by removing from T all its endpoints. Since $D(T') = D(T) - 2$, $D(T')$ is less than $2k$. Consequently, by the induction hypothesis, the statements hold for T'.

Since $D(T') = D(T) - 2$, $D(T)$ is even if and only if $D(T')$ is even. $\mathscr{C}(T') = \mathscr{C}(T)$, because $e'(x) = e(x) - 1$ for each point x of T'. By the induction hypothesis, $\mathscr{C}(T) = \mathscr{C}(T')$ is a singleton if $D(T')$ is even, and therefore if $D(T)$ is even; and $\mathscr{C}(T) = \mathscr{C}(T')$ is a pair of adjacent points if $D(T')$ is odd, and therefore if $D(T)$ is odd. Thus, the first statement holds for trees of diameter less than $2(k + 1)$; this completes the proof of the first assertion.

By the induction hypothesis, $e'(x) = R(T') + \min\{d(x, c): c \in \mathscr{C}(T')\}$, where x is any point of T', i.e., any nonendpoint of T. But since $e'(x) = e(x) - 1$, $R(T')$

$= R(T) - 1$ and $\mathscr{C}(T') = \mathscr{C}(T)$, this becomes $e(x) = R(T) + \min\{d(x, c): c \in \mathscr{C}(T)\}$. Suppose now that $x \in \mathscr{L}(T)$, and let y be the point adjacent to x. Since y is a point of T', $e(y) = R(T) + \min\{d(y, c): c \in \mathscr{C}(T)\}$, as above. Since $x \notin \mathscr{C}(T)$ and y is the only point adjacent to x, it is clear that $\min\{d(x, c): c \in \mathscr{C}(T)\} = 1 + \min\{d(y, c): c \in \mathscr{C}(T)\}$. But since $e(x) = e(y) + 1$, this implies that $e(x) = R(T) + \min\{d(x, c): c \in \mathscr{C}(T)\}$. Therefore for any point x of T,

$$e(x) = R(T) + \min\{d(x, c): c \in \mathscr{C}(T)\}.$$

This completes the induction step of the proof of the second assertion of the theorem.

Let $W = x_0, x_1, \cdots, x_n$ be a diametral path in T, with $n = D(T)$. Since x_0 and x_n are peripheral points, they are endpoints; therefore $e(x_1) = e(x_{n-1}) = n - 1$. Let $W' = x_1, x_2, \cdots, x_{n-1}$. Then W' is a path in T' of length $n - 2$ and $D(T') = D(T) - 2$, W' is a diametral path in T', and x_1 and x_{n-1} are peripheral points of T'. By the induction hypothesis, $e'(x_i) = \max\{(i - 1), (n - 2) - (i - 1)\} = \max\{i - 1, n - i - 1\}$ for each $i = 1, 2, \cdots, n - 1$. Since $e(x) = e'(x) + 1$ for points x of T', $e(x_i) = 1 + e'(x_i) = \max\{i, n - i\}$ for $i = 1, 2, \cdots, n - 1$. Since also $e(x_0) = e(x_n) = \max\{i, n - i\}$, it follows that $e(x_i) = \max\{i, n - i\}$ for each x_i on W. This completes the inductive step of the proof of the third statement of the theorem.

THEOREM 2. *If $u \in \mathscr{V}(T) - \mathscr{C}(T)$, then there is exactly one point v such that $uv \in \mathscr{E}(T)$ and $e(v) = e(u) - 1$.*

Proof. Let $b \in \mathscr{C}(T)$ be such that $e(u) = R(T) + d(u, b)$, and let $W = u, v, x_2, \cdots, x_{n-1}, b$ be the unique path from u to b; since $u \notin \mathscr{C}(T)$, $u \neq b$, so that the unique path from v to b is $v, x_2, \cdots, x_{n-1}, b$ and $d(v, b) = d(u, b) - 1$. Consequently, $e(v) = e(u) - 1$ with $uv \in \mathscr{E}(T)$.

Suppose v' is a point such that $uv' \in \mathscr{E}(T)$ and $e(u) = e(v') + 1$. Let $b' \in \mathscr{C}(T)$ be such that $e(v') = R(T) + d(v', b')$, and $W' = v', y_1, y_2, \cdots, y_{m-1}, b'$ the path from v' to b'. Since $e(v') = e(u) - 1$, $m = n - 1$. Point u does not occur on W', for otherwise, $e(u) < e(v')$, contrary to the assumption that $e(v') = e(u) - 1$. Since also $uv' \in \mathscr{E}(T)$, the path from u to b' is $u, v', y_1, \cdots, y_{n-1}, b'$, the length of which is $m + 1 = n$. Thus, $d(u, b) = d(u, b')$, with $b, b' \in \mathscr{C}(T)$. If $b \neq b'$, then $bb' \in \mathscr{E}(T)$, so that $|d(u, b) - d(u, b')| = 1$; so since $d(u, b) = d(u, b')$, $b = b'$. Therefore, if $v \neq v'$, then W and W' are two distinct paths joining u and b, an impossibility. Hence, $v = v'$, i.e., the point adjacent to u, of eccentricity one less, is unique.

THEOREM 3. *If x, y and z are distinct peripheral points of T and $d(x, y) = D(T)$, then $d(x, z) = D(T)$ or $d(y, z) = D(T)$.*

Proof. Let $W = x_0, x_1, x_2, \cdots, x_{n-1}, x_n$ be the path from x to y, where $n = D(T)$, $x_0 = x$, and $x_n = y$. Since z is neither x nor y, z is not an endpoint of W; since $z \in \mathscr{P}(T) \subset \mathscr{L}(T)$, z is not an intermediate point of W. Thus, z does not occur on W. $1 \leqq \min\{d(z, x_i): i = 0, 1, \cdots, n\}$. Let x_i be a point of W such that $d(z, x_i) \leqq d(z, x_j)$ for all x_j on W, and $W' = y_0, y_1, \cdots, y_k$ be the path from z to x_i, where $y_0 = z$, $y_k = x_i$, and $k = d(z, x_i)$. If $r = 0, 1, \cdots, k - 1$, y_r does not lie on W, since otherwise $d(z, y_r) < d(z, x_i)$ with y_r on W, contrary to the hypothesis on x_i. Therefore, $W_1 = y_0, y_1, \cdots, y_k, x_{i-1}, x_{i-2}, \cdots, x_1, x_0$ is the path from z to x and $W_2 = y_0, y_1, \cdots, y_k, x_{i+1}, x_{i+2}, \cdots, x_n$ is the path from z to y.

Since W is a diametral path, W contains $\mathscr{C}(T)$. If $\mathscr{C}(T)$ consists of two points, they are adjacent, and so occur consecutively on W. Consequently, one of W_1 and W_2 contains $\mathscr{C}(T)$, and since it joins peripheral points, is therefore a diametral path.

Acknowledgments. The authors are grateful to Drs. C. K. Chen and Harry Blum for their careful consideration of the first draft of this paper, and for their helpful comments and suggestions.

We are also indebted to Dr. R. H. Cofer, who suggested a tree representation of picture objects and defined the problem with which this paper has been concerned.

REFERENCES

[1] J. C. GOWER AND G. J. S. ROSS, *Minimum spanning trees and single linkage cluster analysis*, Applied Statistics, 18 (1969), pp. 54–64.

[2] F. HARARY, *Graph Theory*, Addison-Wesley, Reading, Mass., 1969.

[3] G. LEVI AND U. MONTANARI, *A grey-weighted skeleton*, Information and Control, 17 (1970), pp. 62–91.

[4] O. ORE, *Theory of Graphs*, American Mathematical Society, Providence, Rhode Island, 1962.

[5] C. T. ZAHN, *Graph-theoretical methods for detecting and describing gestalt clusters*, IEEE Trans. Computers, C-20 (1971), pp. 68–86.

SIAM J. COMPUT.
Vol. 3, No. 1, March 1974

ON BACKTRACKING: A COMBINATORIAL DESCRIPTION OF THE ALGORITHM*

JAY P. FILLMORE AND S. G. WILLIAMSON†

Abstract. A basic algorithm for solving many discrete problems is the so-called "backtracking" algorithm. The basic problem is that of generating the elements of a subset S_0 of a finite set in an efficient manner. If a group G acts on S_0, then one might wish to obtain only nonisomorphic elements of S_0. In this paper the basic backtracking algorithm is described in terms of chains of partitions on the set S. The corresponding isomorph rejection problem is described in terms of G-invariant chains of partitions on S. Examples and flow charts are given.

Key words. backtracking, isomorph rejection, constructive combinatorics, parallel search, eight queens problem.

1. Introduction. Backtracking, as it is usually formulated [3], is a search algorithm to determine all elements $(x_1, \cdots, x_n)$ of a Cartesian product $X_1 \times \cdots \times X_n$ which satisfy a given true-false valued "criterion" function: $\varphi(x_1, \cdots, x_n) = \text{true}$. An element satisfying this condition is built up coordinate by coordinate: x_1 in X_1 is chosen first, then x_2 in X_2, and so on. If, after the choice of the first k coordinates, $\varphi(x_1, \cdots, x_k, -, \cdots, -)$ is never true, no matter what the choice of the remaining $n - k$ coordinates, the kth coordinate x_k is changed; this step is the "backtrack" which gives the algorithm its name. The efficiency of the algorithm is due to the fact that at the point where the above backtrack occurs, $M_{k+1} \times \cdots \times M_n$ possibilities are ruled out, M_i being the number of elements in the set X_i. This formulation is tied to the fact that the set being searched is a Cartesian product and the algorithm rapidly finds large numbers of nonsolutions.

The fact that the solutions of $\varphi(x_1, \cdots, x_n) = \text{true}$ are a subset of a Cartesian product does not mean that this backtracking procedure will yield an efficient algorithm; indeed, the fixed structure of the Cartesian product may introduce unnecessary complications. Moreover, the ability to accept large numbers of solutions, as well as reject large numbers of nonsolutions, is not incorporated. Finally, the Cartesian product description makes awkward the detection of isomorphs, that is, solutions carried into one another under the action of a group.

A simpler, more versatile, procedure is described in this paper. Briefly: given a set S and a T-0-F valued criterion function φ on S, we partition S into three sets, the sets where φ takes the values T, 0 and F; these sets are built from families of subsets of S which are prescribed in advance. A brief discussion of the problem of isomorph rejection is given in a similar framework. The concluding section of this paper describes the notion of "parallel search", a method which can significantly reduce the number of steps needed to carry out a backtrack search by performing more than one related search simultaneously.

2. Backtracking. Let S be a finite set. The backtrack algorithm will be described in terms of partitions π of S into disjoint blocks B. The partition π' *refines*

* Received by the editors November 16, 1972. Research of the second author was sponsored by the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under AFOSR Contract/Grant no. 71-2089.

† Department of Mathematics, University of California at San Diego, La Jolla, California 92037.

π, written $\pi > \pi'$, if for any block B' in π' there is a block B in π with B containing B'. In a chain of partitions of S each one refines the previous: $\pi_1 > \pi_2 > \cdots > \pi_r$.

Let φ be a T-0-F valued "criterion" function on S. We wish to partition S into the three sets where φ is T, 0 and F. We denote by φ_π a T-0-F valued "test" function defined on π, i.e., the arguments of φ_π are the blocks of π, related to φ as follows:

$$\text{if } \varphi_\pi(B) = \begin{cases} T, \\ 0, \\ F, \end{cases} \quad \text{then} \quad \begin{cases} \varphi(s) = T & \text{for all } s \text{ in } B, \\ \text{no conclusion}, \\ \varphi(s) = F & \text{for all } s \text{ in } B, \end{cases}$$

for B in π.

A *backtrack search* on the set S is defined by the following data: a chain of partitions $\pi_1 > \pi_2 > \cdots > \pi_r$ of S and a numbering (linear ordering) $B(i_1, \cdots, i_k)$ of the blocks of π_k such that $B(i_1, \cdots, i_{k-1})$ contains $B(i_i, \cdots, i_k)$ for $1 \leqq i_k \leqq M(i_1, \cdots, i_{k-1})$, the number of blocks of π_k contained in $B(i_1, \cdots, i_{k-1})$. The backtrack search is carried out according to the flow chart in Fig. 1.

Note that the numbering tells us that $B(i_1, \cdots, i_k)$ is a subblock of $B(i_1)$ in π_1, $B(i_1, i_2)$ in $\pi_2, \cdots, B(i_1, \cdots, i_{k-1})$ in π_{k-1}.

At the conclusion of the algorithm, the set S has been partitioned into three subsets S_T, S_0, and S_F. S_T and S_F are the disjoint union of blocks of various π_k; $\varphi(s) = T$ for every s in S_T, $\varphi(s) = F$ for every s in S_F; if a block appears in S_T or S_F, no subblock needed to be tested. S_0 is the disjoint union of blocks of π_r.

If π_r is the discrete partition of S, and φ_{π_r} is defined by $\varphi_{\pi_r}(\{s\}) = \varphi(s)$, then this disjoint union of three sets is the one originally sought.

Let us note the following important observation: backtrack searching is recursive; it may be applied again to the set S_0 after choosing a chain of partitions and test functions for this set. This allows us to define the above backtrack process recursively: choose a partition π of S and a test function φ_π related to the criterion function φ as before. On the basis of φ_π, classify the blocks of π as subsets of S_T, S_0, or S_F. Repeat this procedure for each block B of π which is contained in S_0, using the partition $\pi'|B$, where π' denotes the next partition in the chain of partitions, and its restriction $\pi'|B$ consists of all blocks of π' contained in B. The use of a partition of S_0 other than the partition $\pi'|B$ yields a more general backtrack algorithm in which choices of partitions can be made to depend on information obtained during the search.

Example. The usual search of a Cartesian product [3] is formulated in the present theory as follows: let S be $X_1 \times X_2 \times \cdots \times X_n$ and let φ be the true-false valued criterion function on S. We wish to determine those elements $(x_1, \cdots, x_n)$ of S for which $\varphi(x_1, \cdots, x_n) =$ true. For $1 \leqq k \leqq n$, define partitions:

$$\pi_k = \{B(i_1, \cdots, i_k) | 1 \leqq i_1 \leqq M_1, \cdots, 1 \leqq i_k \leqq M_k\},$$

$$B(i_1, \cdots, i_k) = \{a_{i_1}\} \times \cdots \times \{a_{i_k}\} \times X_{k+1} \times \cdots \times X_n,$$

where a_{i_x} is the i_xth element of the set X_x. Define test functions: for $1 \leqq k < n$,

$$\varphi_{\pi_k}(B) = \begin{cases} F & \text{if } \varphi(s) = \text{false for all } s \text{ in } B, \\ 0 & \text{otherwise}; \end{cases} \qquad B \text{ in } \pi_k;$$

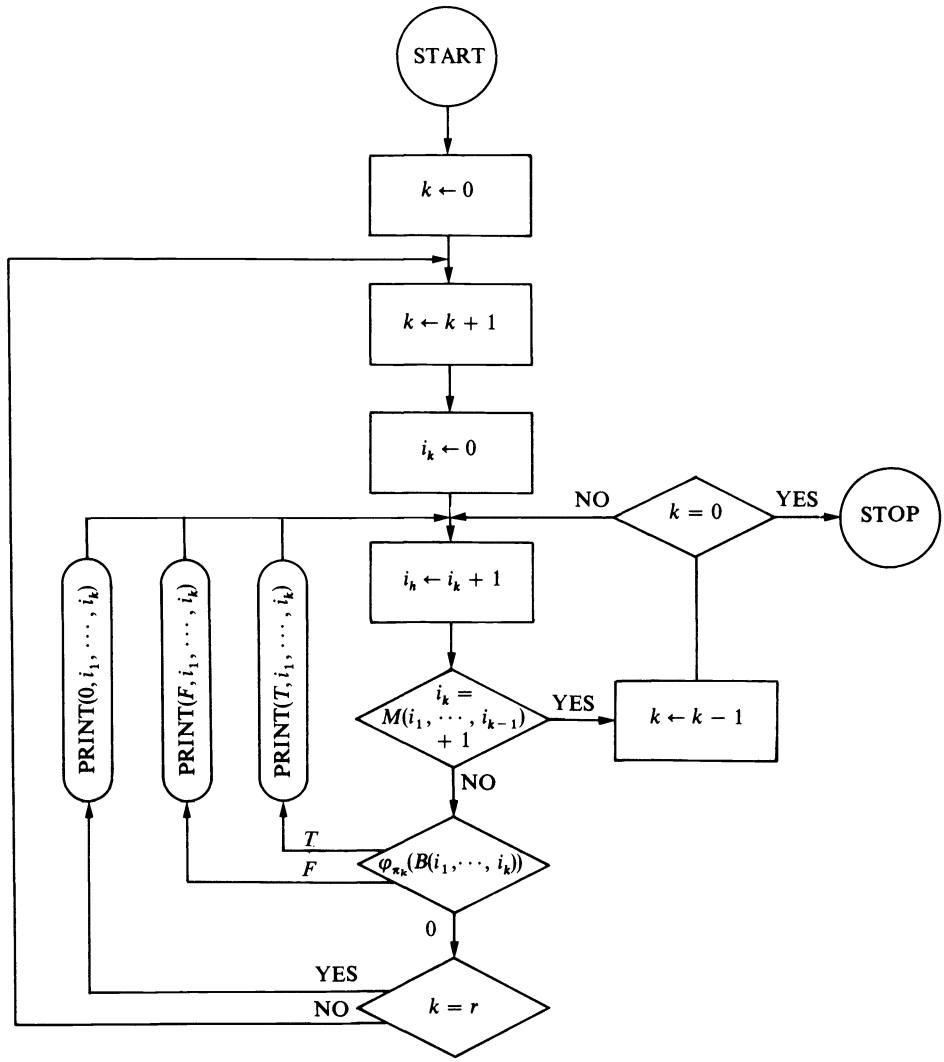


FIG. 1. *Flow chart of the backtrack algorithm*

for $k = n$,

$$\varphi_{\pi_n}(B) = \begin{cases} F & \text{if } \varphi(s) = \text{false}, \\ T & \text{if } \varphi(s) = \text{true}. \end{cases} \qquad B = \{s\} \quad \text{in } \pi_n,$$

This construction may be generalized slightly by defining π_A for a subset A of $\{1, \cdots, n\}$ to be the partition consisting of blocks $X'_1 \times \cdots \times X'_n$, where $X'_k = X_k$ if k is not in A and X'_k is a single element subset of X_k if k is in A. The test functions are defined as before. A chain of partitions results by taking a chain of subsets of $\{1, \cdots, n\}$.

Every backtrack search on a set S' can be formulated as a backtrack search on a Cartesian product which is not necessarily the set S. Indeed, given $\pi_1 > \pi_2 > \cdots > \pi_r$, a chain of partitions of the set S, take X_k to be the set of blocks of π_k.

The test functions must be suitably defined. It will be found that the Cartesian product formulation, in general, unnecessarily expands the amount of searching required to arrive at the "solutions".

Example. Let $f(x)$ be a continuously differentiable function on $0 \leqq x \leqq 1$. Let S be a finite subset of this interval. We wish to classify the points s of S as to whether $f(s) > 0$ or $f(s) \leqq 0$. That is, the criterion function on S will be

$$\varphi(s) = \begin{cases} T & \text{if } f(s) > 0, \\ F & \text{if } f(s) \leqq 0. \end{cases}$$

Suppose that $f(x)$ satisfies $|f'(x)| \leqq M$ on $0 \leqq x \leqq 1$. Choose partitions $\pi_1 > \pi_2 > \cdots > \pi_r$ of S with the restriction that the blocks of π_k are intervals of S, that is, intervals of $0 \leqq x \leqq 1$ intersected with S. Let a_k be the maximum length of an interval in π_k. Define test functions φ_{π_k} by

$$\varphi_{\pi_k}(B) = \begin{cases} T & \text{if } f(b) > Ma_k, \\ F & \text{if } f(b) \leqq -Ma_k, \\ 0 & \text{otherwise,} \end{cases}$$

where, for each B in π_k, a fixed b in B is chosen. At the completion of the search, S will be the disjoint union of three sets: the first is the disjoint union of blocks of varying sizes on which $f(s) > 0$, the second is the disjoint union of blocks of varying sizes on which $f(s) \leqq 0$, and the third is the disjoint union of blocks of length $\leqq Ma_r$ where no conclusion is made. This latter set, usually relatively small, may be searched point by point. Clearly, this generalizes to several variables. Note that, in this example, S does not have a Cartesian product structure which can be utilized in the search.

3. Isomorph rejection. Let G be a finite group which acts on the set S. Two elements s and s' of S are called *isomorphs* if $s' = gs$ for some g in G. One wants to partition S according to the values T, 0 or F taken on by the criterion function φ; a reasonable assumption is that $\varphi(s') = \varphi(s)$ when s and s' are isomorphs.

Specifically, one wants to find one representative from each equivalence class of isomorphs and classify it according to the value of φ. Moreover, one wants to do this efficiently, so that no comparisons need to be made during or after the search. Frequently, the search can be structured in such a way that the group action is methodically taken into account.

The building block of the algorithm is the following, which we shall call an *elementary search on the set S in the presence of the group G*. This consists merely of listing the orbits of G as it acts on the set S and selecting one element from each orbit.

The algorithm now proceeds as follows: beginning with the set S and the group G which acts on it, construct a partition π of S which is stable under G, that is, if B is a block of π, so also is gB, the set of all gs with s in B. The group G acts on the set π. Perform an elementary search on the set π in the presence of the group G. For each block B which results from this search, compute the subgroup G_B of G consisting of all g in G for which $gB = B$. G_B acts on the set B, and for each of these blocks, B, one repeats the process, with S replaced by B and G replaced by G_B.

The rejection of isomorphs terminates when either G_B consists of the identity only, in which case the block B may be searched directly without the possibility of isomorphs, or B consists of a single element, in which case G_B is the symmetry group of this element.

Before describing a general method for solving the isomorph rejection problem, we illustrate with an example.

Example. The problem of eight queens. The original problem, due to M. Bezzel [2], is to place eight nonattacking queens on an eight by eight chess board. There are known to be ninety-two solutions, but many of these are isomorphs under the obvious action of the eight-element dihedral group; it is known that there are twelve nonisomorph solutions. We will illustrate a backtracking algorithm to solve the problem of placing five or fewer nonattacking queens on a five by five chess board; this will permit us to list every step. The case of an eight by eight chess board is no more difficult, only longer.

We begin by numbering the squares of the five by five chess board in any way we please; we number them from 1 to 25 in the order that characters are printed on a page.

Take the empty square bearing the smallest number, in this case 1, and compute its orbit under the full symmetry group of the square—in this case, squares 1, 5, 21 and 25. The largest number of queens which may be placed in these four squares is one; the other possibility is zero. We begin by placing one queen on square 1. By the use of the symmetry group, we need not consider placing one queen on square 5, square 21, or square 25; this amounts to picking a representative of an orbit for the symmetry group acting on the configuration of queens on these four squares. Later, when we have backtracked to these four squares, we will pick another representative of an orbit under this action. We will keep track of these choices by always placing the "largest" first. That is, if the presence or absence of a queen on a square is interpreted as a binary digit, square 1 is the most significant bit, square 25 the least significant bit. A record must be kept of the orbits as they are used.

From the recursive viewpoint, the problem is now the following: place four or fewer nonattacking queens on the 21 squares which are exclusive of squares 1, 5, 21 and 25 so that they are not attacked by the queen in square 1. Note there is to be a queen in square 1, and no queen in squares 5, 21 and 25. The symmetry group of this problem is the group of order two consisting of the identity and reflection in the diagonal through squares 1 and 25. The orbit in the chess board containing the least numbered unspecified square consists of squares 2 and 6. The "largest" placement of queens on this orbit is to place no queen on either square 2 or square 6.

The algorithm now continues in this recursive fashion. As soon as the identity group is reached, isomorphs are rejected, and the search may proceed in any manner. But in fact, the above prescription will reduce to the usual backtrack scheme of placing a queen on the "first" possible square it can occupy.

When orbits of various groups acting on the chess board fill up all 25 squares, a solution has been reached. When one backtracks to a certain orbit on the chess board, one advances the configuration of queens to occupy that orbit to be the next "smaller" orbit representative for the symmetry group of that orbit acting

on the configuration of queens in that orbit.

An initial and a terminal segment of this algorithm is executed step by step in Figs. 2 and 3. Orbits are numbered as they are set down; queens are denoted by circles. For the sake of brevity, only those steps in the execution of the algorithm that change the configuration of queens, are a backtrack, or are terminal are recorded. The terminal steps, indicated by the whole board being filled by labeled orbits, are solutions.

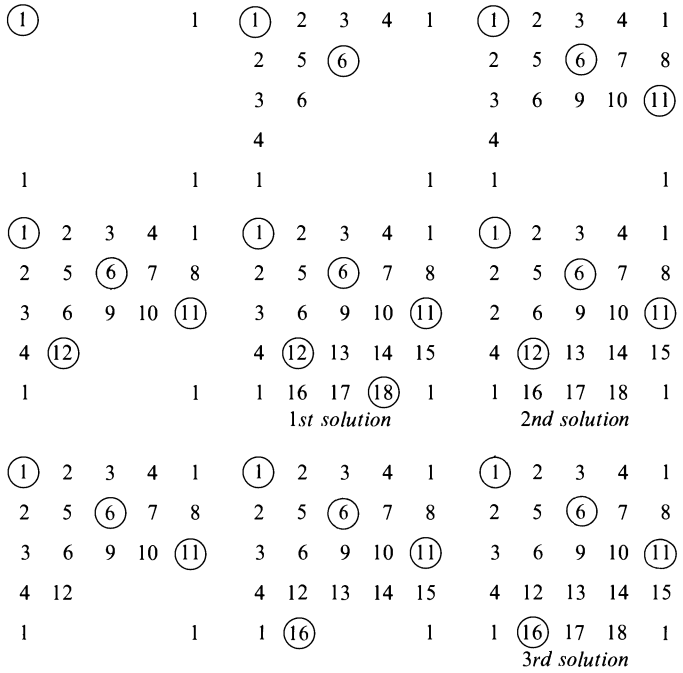
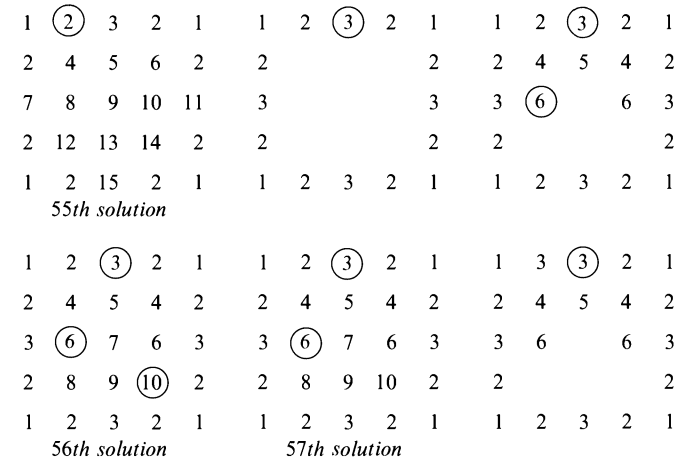


FIG. 2. *An initial segment of the algorithm*



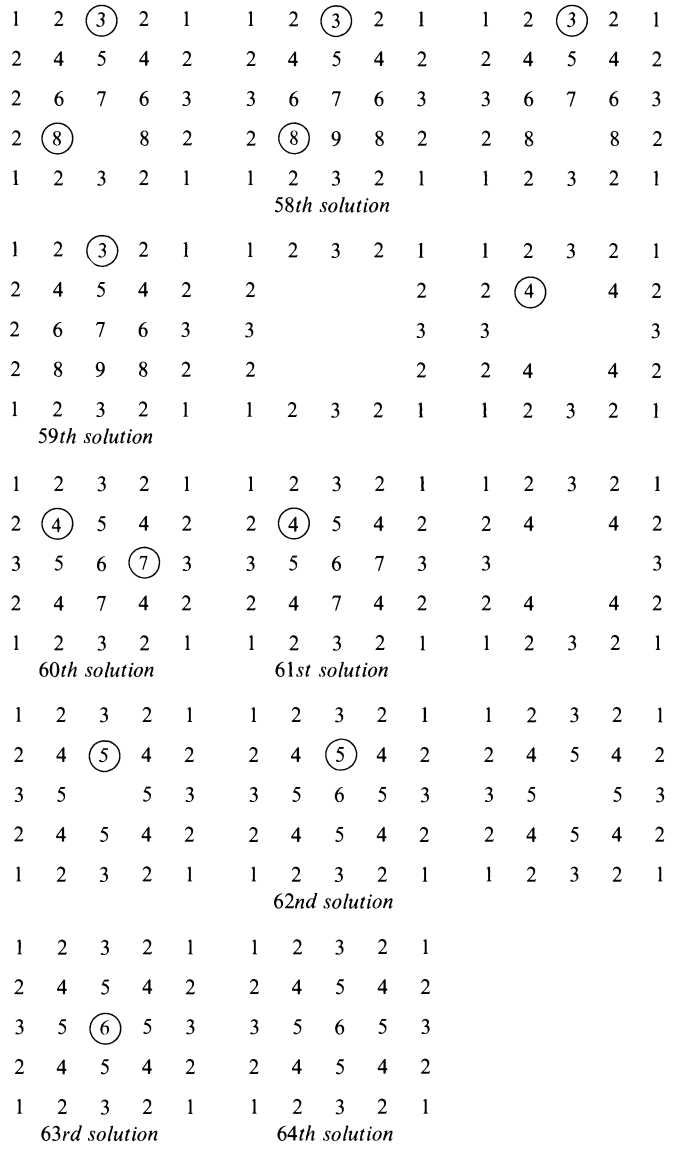


FIG. 3. *Terminal segment of the algorithm*

To connect this example with the general description, we take S to be the set of all ways to place five or fewer nonattacking queens on the five by five chess board. Partitions π of S which are stable under G, or the subgroup of G at a particular stage of the algorithm, are obtained by specifying configurations on an orbit of the group as it acts on the board.

Before leaving this example, we wish to indicate one possible machine implementation, which is much like the one used in the example of the next section.

A certain amount of preliminary analysis is done to create the following lists: first, the subgroups of the group of all symmetries of the square are numbered in

any order from 1 to 10 (not all will actually be needed). For each group, all possible orbits on the five by five chess board are described. Any configuration on the chess board can be represented by a 25 bit word; we will number the bits from 1 to 25 so as to correspond to the numbering of the squares used above on the board. Given a subgroup, its orbits on the board are listed and numbered by the first occupied square, that is, by the first nonzero bit of the corresponding word. Thus, we have a doubly indexed list, the first index for the group number, the second index for the orbit number, and the elements of the list are orbits on the chess board. The values of the second index which are used depend on the first index. Second: for each of the orbits listed above, one lists the possible configurations of queens which may occupy the orbit, choosing one representative configuration under the action of the subgroup in question as it acts on the configurations in the orbit. These representative configurations are labeled consecutively. We now make up three triply indexed lists, the indices being the group number, the orbit number, as above, and the configuration number. The first list consists of the configuration of queens in the orbit: it is a 25 bit word. The second list consists of the squares on the chess board which are attacked by the queens of the configuration in the corresponding entry of the first list. The third list gives, for each subgroup, orbit and configuration of queens in the orbit, the number of the subgroup of the given subgroup which is the stability group of the given configuration of queens.

The following ALGOL code requires several machine-oriented procedures which we describe first. They are similar to those available on the Burroughs B6700 machine.

We regard Boolean variables as 25 bit words; the Boolean operations treat the Boolean words bitwise. There is an integer procedure FIRSTONE which gives the number of the first nonzero bit of a Boolean word. There is a Boolean procedure FILLED which tells if data has been filled into a Boolean word. There is a Boolean procedure EMPTY which is true if all bits of Boolean word are zero. Let FF denote the Boolean word consisting of all bits zero; let TT denote the Boolean word with all bits one. The Boolean relation IS compares two words and is true if they are bitwise identical.

The section of the program denoted READ DATA reads the prepared lists into the declared arrays. Note that for this problem there will be at most six configurations of queens in any one orbit, an additional value for the index must be provided for, so that one can detect when the list of configurations is exhausted. Note also, that array elements corresponding to no data must be filled with symbols which will cause FILLED to produce false.

```
begin
Boolean array ORB[1:10, 1:25], QNS, ATK[1:10, 1:25, 1:7];
integer array SGN[1:10, 1:25, 1:7], G, O, Q[1:26];
Boolean B, X;                 B is the union of orbits on the board;
                              X is the configuration of queens.
integer K;                    K is the same as the integer numbering the orbits
                                in the hand execution above.
label AK, BK, AQ, FINIS;
```

```
READ DATA
    B: = FF;
    X: = FF;
    K: = 0;                 The number of the full symmetry group of the
    G[1]: = 10;             square is 10.
AK: K: = K + 1;
    O[K]: = FIRSTONE(NOT B);
    B: = B OR ORB[G[K], O[K]];
    Q[K]: = 0;
AQ: Q[K]: = Q[K] + 1;
    if NOT FILLED(QNS[G[K], O[K], Q[K]]) then goto BK;
    if NOT EMPTY(X AND ATK[G[K], Q[K]]) then goto AQ;
    X: = X OR QNS[G[K], O[K], Q[K]];
    G[K + 1]: = SGN[G[K], O[K], Q[K]];
    if NOT (B IS TT) then goto AK;
    print (X);
    X: = X AND(NOT ORB[G[K], O[K]]);
    goto AQ;
BK: X: = X AND(NOT ORB[G[K], O[K]]);
    B: = B AND (NOT ORB [G[K], O[K]]);
    K: = K - 1;
    if K EQL 0 then goto FINIS;
    goto AQ;
FINIS:  end.
```

A general setting for backtracking, which incorporates the above recursive procedure in a "dynamic" setting would contain the following ingredients.

Let S be the set which we wish to search. For certain subsets A of S, we have given a rule which assigns a partition π_A of A; certain of these subsets are designated by this rule as terminal. Each partition π_A is assumed to linearly ordered; we assume given a rule which to each element B of π_A associates a larger element B' in the given linear ordering of π_A. For convenience, we let 0 and ∞, respectively, denote unique elements less than and greater than every element of π_A. If π is any partition, $\cup\pi$ denotes the union of its blocks; π^- denotes the partition from which $\cup\pi$ was chosen in the execution of the algorithm. φ_π is a T, 0, F-valued test function on the blocks of π, related to φ as previously.

The algorithm is now defined by the flow chart in Fig. 4. Note that a record must be kept during the execution of the algorithm in order to determine π^-.

At the conclusion of the algorithm, three collections of blocks will have been determined: a collection of blocks which test T, a collection of blocks which test F, and a collection of blocks from terminal partitions which test 0. Moreover, only blocks which are permitted to appear by the succession rule B to B' are obtained. This last feature may be used to incorporate what is usually known as "preclusion", and plays a central role in isomorph rejection.

4. Parallel search. Let φ be a criterion function on the set S. A backtrack search will be said to be a *parallel search* if there are given two test functions φ'_π and φ''_π which are related to the criterion function as in Table 1. The headings of

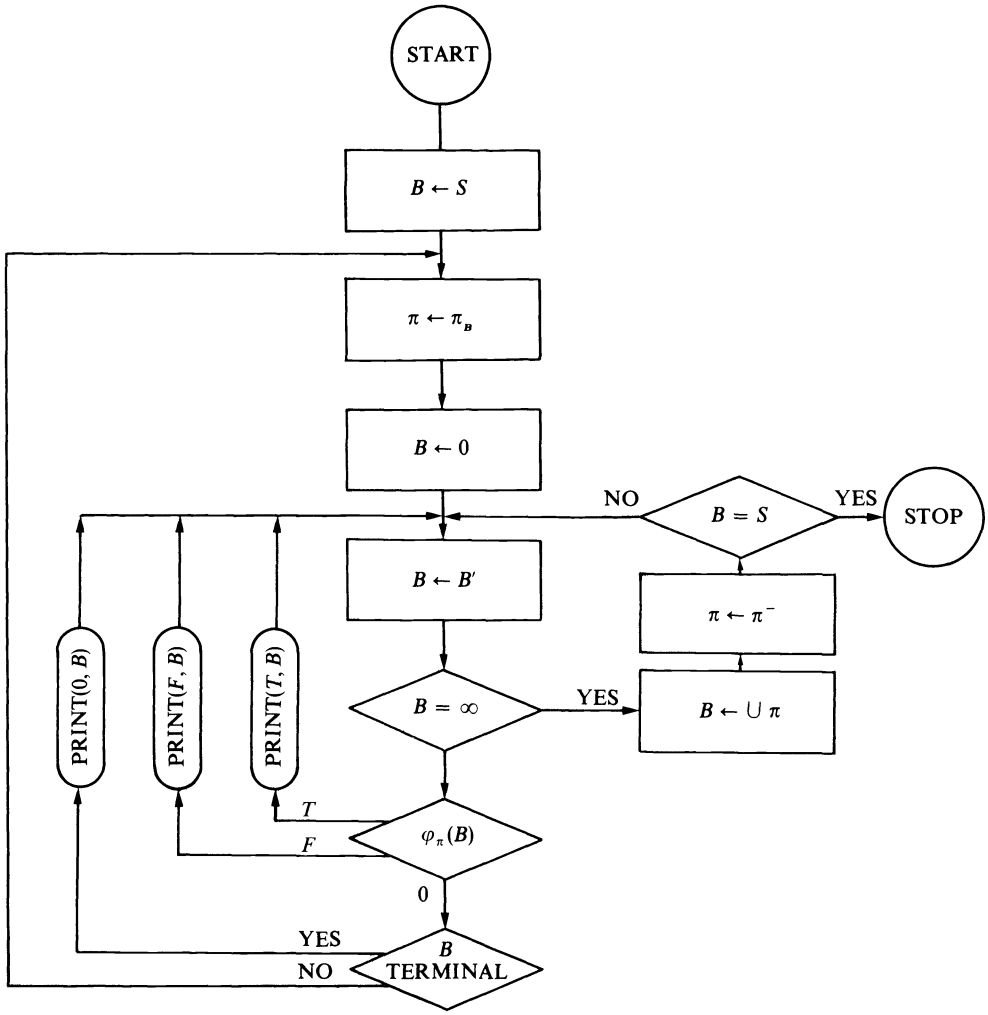


FIG. 4. *Flow chart of the backtrack algorithm; dynamic viewpoint*

the table are the values of $\varphi'_\pi(B)$ and $\varphi''_\pi(B)$; the entries of the table are the values of $\varphi(s)$ for every s in B, B is a block of π. Essentially, this says that as the backtrack is being executed, a decision is to be made on the basis of φ'_π or of φ''_π when possible, and conflicting decisions need not be made.

TABLE 1

$\varphi'_\pi(B)$ \ $\varphi''_\pi(B)$	T	0	F
T	T	T	cannot occur
0	T	0	F
F	cannot occur	F	F

The effect of having two tests will be to come to decisions that permit backtracking more often than would occur with either test alone. This will speed up the execution of the algorithm.

If the algorithm is executed on a multiple processor machine, the two tests can be genuinely performed in parallel. Parallel search, of course, extends to several tests.

Example. The SOMA *cube.* For a detailed description of this "puzzle", see [1] or the booklet published by Parker Brothers, Inc., Salem, Mass., 1969, which accompanies the puzzle.

We will describe the three by three by three SOMA cube as follows: number the cells in the cube from 1 to 27:

back	3	6	9	center	12	15	18	front	21	24	27
plane:	2	5	8;	plane:	11	14	17;	plane:	20	23	26.
	1	4	7		10	13	16		19	22	25

A 1 or 0 is recorded in a 27 bit word, reading left to right, according to whether a cell is occupied or not. For example, 600400400 in octal represents the L-shaped piece placed in the leftmost plane, long side down, short side to the rear. The seven pieces, together with an identifying letter, are the following:

A:	600400400	D:	600300000
B:	640400000	E:	620020000
C:	400600400	F:	600220000
		G:	640000000

In the Parker Brothers booklet, these are, respectively, pieces 2, 7, 3, 4, 5, 6 and 1. Numerous procedures for listing the solutions to SOMA, either by hand or machine computation, have been described (for references, see [1]). In order to illustrate parallel search as formulated above, it suffices to consider the most straightforward backtracking scheme, which we describe as follows.

Seven lists are constructed. The A list consists of the 144 possible positions and orientations of the A piece. These are represented octally as above. Similarly the other six lists are constructed.

List	A	B	C	D	E	F	G
Length	144	64	72	72	96	96	144

These lists are constructed prior to the search for all ways of assembling the cube.

Let X_1 be the set of integers from 1 to 144, let X_2 be the set of integers from 1 to 64, and so on to X_7. The set to be searched is the Cartesian product $S = X_1 \times X_2 \times \cdots \times X_7$. The criterion function for the problem is defined by:

$$\varphi(i_1, i_2, \cdots, i_7) = \begin{cases} T & \text{if } (i_1)_1 \text{ OR } \cdots \text{ OR } (i_7)_7 = 777777777, \\ F & \text{otherwise,} \end{cases}$$

where $(i)_k$ denotes the 27 bit word appearing in the ith entry of X_k, and OR denotes bit by bit Boolean "or". The criterion function specifies that the 27 cells of the cube should be filled using the available pieces.

The search on a Cartesian product was described in § 2. We describe briefly the blocks in the partitions of the chain $\pi_1 > \pi_2 > \cdots > \pi_7$. Note that π_7 is the discrete partition.

π_1 consists of 144 blocks B(1), $\cdots$, B(144), where

$$\begin{aligned} \mathrm{B}(i) &= \{(i_1, x_2, \cdots, x_7) | x_2 \in X_2, \cdots, x_7 \in X_7\} \\ &= \{i_1\} \times X_2 \times \cdots \times X_7. \end{aligned}$$

π_2 consists of $144 \cdot 64 = 9216$ blocks $B(i_1, i_2)$, $i_1 = 1, \cdots, 144$, $i_2 = 1, \cdots, 64$, where

$$\mathrm{B}(i_1, i_2) = \{i_1\} \times \{i_2\} \times X_3 \times \cdots \times X_7.$$

Note that $\mathrm{B}(i_1, i_2)$ is a subblock of $\mathrm{B}(i_1)$.

$\vdots$

π_7 consists of $144 \cdot 64 \cdot 72 \cdot 72 \cdot 96 \cdot 96 \cdot 144 = 6.340 \times 10^{13}$ (approx.) blocks $\mathrm{B}(i_1, \cdots, i_7)$, each consisting of a single element of S.

Tests φ_π on the blocks are defined as follows:

φ_{π_1}: $\varphi_{\pi_1}(\mathrm{B}(i_1)) = 0$ always.

$$\varphi_{\pi_2}\colon \varphi_{\pi_2}(\mathrm{B}(i_1, i_2)) = \begin{cases} 0 & \text{if } |(i_1)_1 \text{ OR } (i_2)_2| = 8, \\ F & \text{if } |(i_1)_1 \text{ OR } (i_2)_2| < 8. \end{cases}$$

$$\varphi_{\pi_3}\colon \varphi_{\pi_3}(\mathrm{B}(i_1, i_2, i_3)) = \begin{cases} 0 & \text{if } |(i_1)_1 \text{ OR } (i_2)_2 \text{ OR } (i_3)_3| = 12, \\ F & \text{if } |(i_1)_1 \text{ OR } (i_2)_2 \text{ OR } (i_3)_3| < 12. \end{cases}$$

φ_{π_4}:
φ_{π_5}: } similarly defined using 16, 20, 24 on the right.
φ_{π_6}:

$$\varphi_{\pi_7}\colon \varphi_{\pi_7}(\mathrm{B}(i_1, \cdots, i_7)) = \begin{cases} T & \text{if } |(i_1)_1 \text{ OR } \cdots \text{ OR } (i_7)_7| = 27, \\ F & \text{if } |(i_1)_1 \text{ OR } \cdots \text{ OR } (i_7)_7| < 27. \end{cases}$$

Here $|x|$ denotes the number of nonzero bits in the word x. Note that $\varphi_{\pi_7}(\mathrm{B}(i_1, \cdots, i_7)) = \varphi(i_1, \cdots, i_7)$, $\mathrm{B}(i_1, \cdots, i_7)$ being the discrete block $\{(i_1, \cdots, i_7)\}$. These tests simply determine whether or not the partially assembled pieces are nonoverlapping or not.

The SOMA cube admits the group of order 48 of all symmetries of a cube. Isomorph rejection is carried out in the following way: clearly the partitions $\pi_1, \cdots, \pi_7$ are stable under the action of the group, i.e., the blocks of any one are permuted among themselves. The orbits of the group acting on π_1 are represented on List A by:

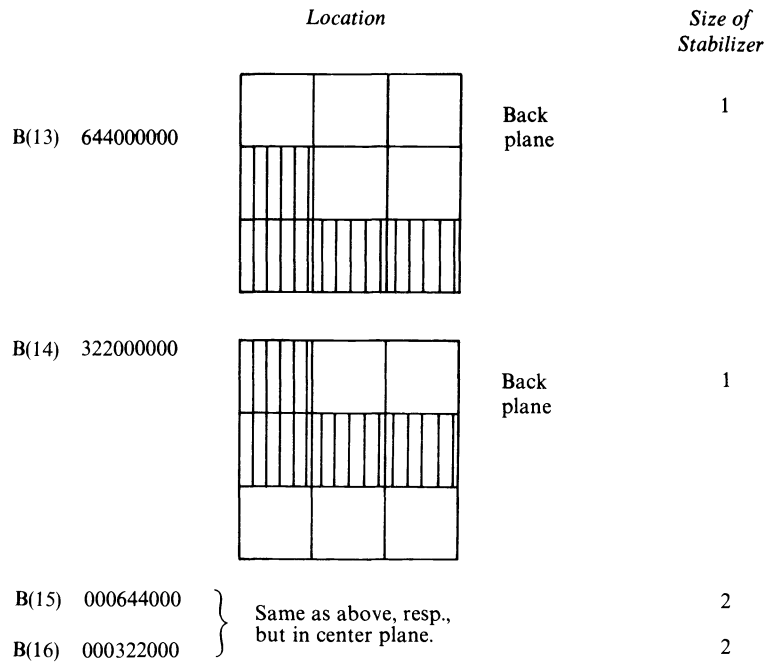


FIG. 5

The B list is so arranged that no cell of the front plane is filled for $i_2 = 1, 2, \cdots, 32$, and no cell of the back plane is filled for $i_2 = 33, 34, \cdots, 64$. Thus the first 32 words of this list represent the orbits of the group of order 2 when acting on B(15) and B(16). The stabilizers of blocks B(15, 1) to B(15, 32) and blocks B(16, 1) to B(16, 32) are the group consisting of the identity alone.

To summarize the search: search the Cartesian product $S = X_1 \times \cdots \times X_7$ using $i_1 = 13, 14, 15, 16$ only; use $i_2 = 1, \cdots, 64$ if $i_1 = 13$ or 14, $i_2 = 1, \cdots, 32$ if $i_1 = 15$ or 16. All elements of the lists $X_3, \cdots, X_7$ are used.

This search was carried out on a Burroughs B6700 machine. A slight modification of the backtracking algorithm was used: when $i_1, \cdots, i_6$ are chosen, i_7 is unique, so when backtracking from $k = 7$, we would go to $k = 6$ rather than trying to advance with $k = 7$. The search yielded 240 inequivalent solutions. Counting each one 48 times yields the $48 \cdot 240 = 11{,}520$ solutions stated in the literature.

The search for the solutions to the SOMA cube may be conducted as a parallel search. The two test functions will be: the test function φ_π used above, which we will now denote by φ'_π, and a second test function φ''_π which we now describe.

Two cells of the cube will be called *adjacent* if they share a common square; a cell will be called *isolated* if it is not filled, but all cells adjacent to it are filled. If a piece is removed from an assembled or partially assembled cube, none of the resulting unfilled cells can be isolated. Hence, at any point in the assembly where an isolated cell is produced, the algorithm should backtrack.

In the notation used above, let

$$\begin{array}{ll} b_1 = 400000000 & c_1 = 240400000 \\ b_2 = 040000000 & c_2 = 424040000 \\ b_3 = 004000000 & c_3 = 042004000 \\ \vdots & \vdots \\ b_{27} = 000000001 & c_{27} = 000001012 \end{array}$$

be the list of cells b_j in the cube and the cells c_j adjacent to each one of them. Define φ''_π as follows:

$\underline{\varphi''_{\pi_1}}$: $\varphi''_{\pi_1}(\mathrm{B}(i_1)) = 0$ always.

$$\underline{\varphi''_{\pi_2}}\colon \varphi''_{\pi_2}(\mathrm{B}(i_1, i_2)) = \begin{array}{ll} 0 & \text{if } ((i_1)_1 \text{ OR } (i_2)_2 \text{ AND } b_j = 000000000 \text{ implies } ((i_1)_1 \\ & \text{OR } (i_2)_2) \text{ AND } c_j \neq c_j \text{ for } j = 1, 2, \cdots, 27. \\ F & \text{otherwise.} \end{array}$$

AND denotes bit by bit Boolean "and".

$\underline{\varphi''_{\pi_3}}$: $\left.\begin{array}{l} \\ \vdots \\ \\ \underline{\varphi''_{\pi_6}}: \end{array}\right\}$ defined similarly to φ''_{π_2} using $((i_1)_1 \text{ OR } \cdots \text{ OR } (i_k)_k)$ for $k = 3, 4, 5, 6$.

$\underline{\varphi''_{\pi_7}}$: $\varphi''_{\pi_7}(\mathrm{B}(i_1, \cdots, i_7)) = 0$ always.

The above is a search using two test functions; with this problem we could also view it in terms of 28 test functions.

This parallel search was carried out, again on the Burroughs B6700 machine. The parallel tests were necessarily performed sequentially and, because it appeared to be the faster test, φ'_π was performed first.

The results are in Table 2.

TABLE 2

Search	Test	Program length (machine code)	Execution time
nonparallel	$\varphi_\pi\,(=\varphi'_\pi)$	367 words	337.1 seconds
parallel	φ'_π and φ''_π	1,223 words	72.4 seconds

The parallel search for this implementation is faster by a factor of 4.66.

To obtain a slightly more detailed picture of the search, we introduce crude tree "profiles". Let C_k, $k = 1, \cdots, 7$, denote the number of times testing was done on some block of π_k. $C_1 + \cdots + C_7$ reflects the total "cost" for all testing. Let N_k denote the number of blocks of π_k which test 0 or T; in the SOMA cube example, this represents the number of assemblies or partial assemblies of the cube. For the flow chart, see Fig. 6.

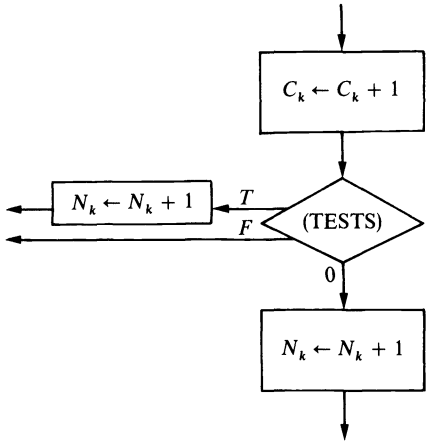


FIG. 6

The profiles for the two searches on SOMA are shown in Table 3. The ratio of "costs" is 4.83, which closely reflects that of the execution times.

TABLE 3

Nonparallel			Parallel		
k	N_k	C_k	k	N_k	C_k
1	4	4	1	4	4
2	108	192	2	101	192
3	1,993	7,776	3	1,316	7,272
4	14,576	143,496	4	4,506	94,752
5	25,004	1,399,296	5	3,717	432,576
6	3,909	2,400,384	6	378	356,832
7	240	545,764	7	240	37,300
total		4,496,912	total		928,928

Acknowledgment. The authors wish to thank Mr. Dennis White for several helpful suggestions in the preparation of this manuscript.

REFERENCES

[1] M. GARDNER, *Mathematical games*, Sci. Amer., 227 (1972), pp. 176–182.
[2] J. GINSBURG, *Gauss's arithmetization of the problem of 8 queens*, Scripta Math., 5 (1938), pp. 63–66.
[3] S. W. GOLOMB AND L. BAUMERT, *Backtrack programming*, J. Assoc. Comput. Mach., 12 (1965), pp. 516–524.

SIAM J. COMPUT.
Vol. 3, No. 1, March 1974

COMPUTING THE WEAK COMPONENTS OF A DIRECTED GRAPH*

JEAN FRANCOIS PACAULT†

Abstract. The weak components of a directed graph G are defined as follows: two vertices u and v of G belong to the same weak component if there is a directed path from u to v and from v to u (then u and v belong to the same strong component) or if one can go from u to v and back through a sequence of "nonpath" steps. The weak components can be determined with an algorithm involving O(max (number of vertices, number of edges)) computation time. The algorithm first determines a partition $C_1, \cdots, C_p$ of the set V of vertices of G such that the weak components are the unions of C_i's of consecutive subscripts, and then groups consecutive C_i's together to form the weak components.

Key words. algorithm, connectivity, directed path, directed graph, weak component

1. Introduction. The concept of weak components of a directed graph was first introduced by R. L. Graham, D. E. Knuth and T. S. Motzkin [2]: two vertices u and v belong to the same *weak component*[1] if they belong to the same strong component—that is, if there is a directed path from u to v and from v to u—or if one can go from u to v and back through a sequence of "nonpath" steps. (There is a "nonpath" from u to v if and only if there is no directed path from u to v.) It has been shown that the weak components constitute the finest partition of the set of vertices of the graph which is totally ordered by the relation that the graph represents.

Knuth [1, Prob. 34] stated as an open problem the task of finding all weak components of a given directed graph as efficiently as possible.

Following is an algorithm involving an O(max (number of edges, number of vertices)) number of steps, if the algorithm is implemented on a random access computer, to determine the weak components of an acyclic graph. This algorithm first partitions the set of vertices V into subsets $C_1, \cdots, C_p$ such that the weak components are the union of C_i's of consecutive subscripts, and then groups the C_i's together to form the weak components (the C_i's are such that any path of the graph leads through a sequence of C_i's, with i strictly decreasing, and, for all i, for any vertex v belonging to C_i, for any j smaller than i, there is a path from v to some vertex in C_j).

For simplicity, we shall assume that we are dealing with an acyclic graph. However, both the algorithm and the mathematical results it relies on can be extended to determine the weak components of any directed graph, acyclic or not, by first using Tarjan's algorithm [3] to find the strong components.

2. Definitions. Let G be the directed graph of a binary relation R over a finite set V. We assume G is acyclic; see § 5. Let R^+ denote the transitive closure of R, and let R^- be the complementary relation.

---

* Received by the editors March 14, 1973. This research was sponsored by the United States Air Force, Air Force Office of Scientific Research, under Grant AFOSR-71-2076.

† Department of Electrical Engineering and Computer Sciences and the Electronics Research Laboratory, University of California at Berkeley, Berkeley, California 94720.

[1] The concept of weak component should be distinguished from the concept of ordinary connected "components", which are defined for the undirected graph formed by disregarding the orientations. Some authors have used the term "weak component" for the latter concept.

DEFINITION 1. The binary relation R_1 over V defined by

$$a R_1 b \Leftrightarrow (a = b \text{ or } (a R^+ b \text{ and } b R^+ a))$$

is an equivalence relation. The equivalence classes it determines are the *strong components* of G. As G is here assumed to be acyclic, the strong components of G are the nodes of G.

DEFINITION 2. The binary relation R_2 over V defined by

$$a R_2 b \Leftrightarrow (a R_1 b \text{ or } (a R^{+\,-\,+} b \text{ and } b R^{+\,-\,+} a))$$

is an equivalence relation [2]. The equivalence classes it determines are the *weak components* of G.

From now on, unless otherwise specified, G is assumed to be an acyclic graph. The strong components of G are then the vertices of G.

We define the partition $C_1, \cdots, C_p$ as follows: vertex u belongs to C_{i+1} if and only if the longest path from u to any terminal vertex is of length i.

DEFINITION 3. Vertex u belongs to C_1 if and only if there is no arc coming out of u.

DEFINITION 4. Vertex u belongs to C_{i+1} if and only if all the arcs coming out of u go to nodes in some C_j's, for values of j smaller than $i + 1$, and there is a node v in C_i such that (u, v) is an arc of G.

These two definitions can be rewritten as follows.

DEFINITION 3'. $u \in C_1 \Leftrightarrow (\forall\, v \in V, u R^- v)$.

DEFINITION 4'. $u \in C_{i+1} \Leftrightarrow ((\forall\, v \in V, u R v \Rightarrow v \in C_j, j \leqq i)$ and $(\exists\, v \in C_i$ such that $u R v))$.

3. Theorems.

LEMMA 1. *If u belongs to C_i, v belongs to C_j and i is smaller than or equal to j, then there is no path from u to v. (That is, $u R^{+\,-} v$).*

Proof. If not, then there exist $u_0 = u, u_1, \cdots, u_n = v$, for some n greater than zero, such that $u = u_0 R u_1, \cdots, u_{n-1} R u_n = v$.

It follows that u_g belongs to C_k, for some k smaller than i, for all $g = 1, \cdots, n$, contradicting the fact that i is smaller than j. □

COROLLARY. *If u and v belong to the same C_i, then they belong to the same weak component.*

LEMMA 2. *If C_i and C_k, for k greater than j, are contained in the same weak component W, then, for all j, $i \leqq j \leqq k$, C_j is contained in W.*

Proof. Let C_i and C_k be part of a weak component W, let j be such that $i \leqq j \leqq k$, and let u belong to C_i, v belong to C_j, w belong to C_k.

Since w and u are both in W, we have $w R^{+\,-\,+} u$. Since $i \leqq j \leqq k$, Lemma 1 proves that $u R^{+\,-} v$ and $v R^{+-} w$. Therefore, $w R^{+-+} v$ and $v R^{+-+} w$, that is C_j is included in W. □

Lemmas 1 and 2 prove that the weak components are the unions of consecutive blocks of C_i's. All we must do to complete the determination of weak components is to characterize the boundaries between these blocks.

THEOREM 1. *C_i and C_{i+1} are both contained in the same weak component if and only if there exists a vertex u belonging to C_{i+1}, and a vertex v belonging to C_j, for some j smaller than $i + 1$, such that $u R^{+\,-} v$.*

Proof. (i) If such u, v exist, we have $u\, R^{+-} v$ and, by Lemma 1, for any w in C_i, $v\, R^{+-} w$ and $w\, R^{+-} u$. Therefore, we have $u\, R^{+-+} w$ and $w\, R^{+-} u$. Hence C_i and C_{i+1} are both contained in the same weak component.

(ii) Conversely, assume that C_i and C_{i+1} are contained in the same weak component W.

Then for any x in C_{i+1} and any w in C_i, there exist $u_0 = x, u_1, \cdots, u_n = w$ such that $x = u_0\, R^{+-} u_1, \cdots, R^{+-} u_n = w$. Therefore, there exists k such that u_k belongs to C_l, for some l greater than i, and u_{k+1} belongs to C_j for some j smaller than $i + 1$.

If l is greater than $i + 1$, there exists u belonging to C_{i+1} such that $u_k\, R^{+} u$. Otherwise, let u be u_k. Let v be u_{k+1}. It follows that $u\, R^{+-} v$. □

4. Description of an algorithm to find the weak components. An algorithm to find the weak components can be based on the preceding theory as follows.

Determine the partition $C_1, \cdots, C_p$ of V by performing a depth-first search [3] of the graph, and set label $(v) := i$ if v belongs to C_i. Determine for each node v the function lowest $(v) = \min(\text{label}(w) \mid w\, R\, v)$.

Then, for each node, increase lowest (v) by one if there exists w in $C_{\text{lowest}(v)}$ such that $w\, R^{-} v$. (Therefore, it is intuitively seen that if v belongs to C_i, then $C_{\text{lowest}(v)-1}, \cdots, C_i$ will be contained in the same weak component. This is proved by Lemma 3 and its corollaries.)

Then group consecutive C_i's together, according to the criterion of Theorem 1, by grouping together subsets $C_i, \cdots, C_{i+k}$ if there is a node v in C_i such that lowest (v) equals $i + k + 1$.

A possible implementation is as follows. (G is represented by an adjacency structure, such that w is in the adjacency list of v if and only if $v\, R\, w$.)

```
begin
  integer p, i;
  procedure partition (v)
  comment this procedure determines label (v), which is the subscript of the C_i
    that v belongs to. Number (i) is the number of nodes belonging to C_i, and
    p the number of subsets C_i in the partition. n(w) is the number of nodes v
    in C_lowest(w) such that v R w;
  begin
    for w in the adjacenty list of v and w is not yet labeled do
    partition (w);
    label (v) := 1;
    for w in the adjacency list of v do
    label (v) := max (label (v), label (w) + 1);
    p := max (p, label (v));
    number (label (v)) := number (label (v)) + 1;
    for w in the adjacenty list of v do
    begin if lowest (w) = label (v) then n(w) := n(w) + 1
      else if lowest (w) > label (v) then
        begin n(w) := 1;
          lowest (w) := label (v);
        end
```

```
    end
  end partition;
  procedure weakcomp
  comment this procedure groups consecutive C_i's together, once lowest (v) has
  been determined for each node v;
  begin integer i, k;
    for v a vertex do
    begin if n(v) < number (lowest (v))
      then lowest (v) := lowest (v) + 1;
    end;
    for v a vertex do
      high (label (v)) := max (high (label (v)), lowest (v));
    i := 0;
    k := 0;
L2  start new component
L1  i := i + 1.
    if i ≦ p then
    begin
      put C_i in current component;
      k := max (k, high (i));
      if k > i + 1 go to L1 else go to L2;
    end;
  end weakcomp;
  p := 0;
  for i := 1 step 1 until V do number (i) := high (i) := 0;
  comment V is the total number of nodes;
  for v a vertex do
    begin label (v) := 0; lowest (v) := V + 1; end;
  for v a vertex and v is not yet labeled do partition (v);
  weakcomp;
end;
```

THEOREM 2. *This algorithm actually determines the partition* $C_1, \cdots, C_p$.

Proof. The proof of this fact is straightforward from the definitions.

LEMMA 3. *Let* v *belong to* C_i, *and let* k *be equal to lowest* (v); *then*

(i) *either, for all* u*'s in* $C_{i+1} \cup \cdots \cup C_{k-1}$, *we have* $u R^{+-} v$ *and for all* w*'s in* C_k, *we have* $w R^{+} v$, *or*

(ii) *for all* u*'s in* $C_{i+1} \cup \cdots \cup C_{k-2}$, *we have* $u R^{+-} v$, *and there exist* w_1 *and* w_2 *in* C_{k-1} *such that* $w_1 R^{+} v$ *and* $w_2 R^{+-} v$.

Proof. The definition of the function lowest (v), as it is computed in procedure weakcomp, clearly implies either that for all u's in $C_{i+1} \cup \cdots \cup C_{k-1}$, we have $u R^{-} v$ and for all w's in C_k, we have $w R^{+} v$, or that, for all u's in $C_{i+1} \cup \cdots \cup C_{k-2}$, we have $u R^{-} v$, and there exist w_1 and w_2 in C_{k-1} such that $w_1 R v$ and $W_2 R^{-} v$.

Moreover, if u belongs to C_j with j smaller than $k = \text{lowest } (v)$, and if $u R^{-} v$ holds, we have $u R^{+-} v$, for otherwise there would exist some node x, with label (x) greater than i and smaller than j, such that $u R^{+} x R v$. Hence we

would have lowest (v) smaller than or equal to label (x) + 1, which is smaller than k. □

COROLLARY 1. *Let v belong to* C_i, *and* lowest (v) *be equal to* $i + 1$. *Then, for all u's in* C_{i+1}, *we have* $u R^+ v$.

Proof. Case (ii) of Lemma 3 cannot occur here, for there is no node w in $C_{\text{lowest}(v)-1} = C_i$ such that $w R^+ v$. Therefore, we are in case (i), and all the nodes u in C_{i+1} are such that $u R^+ v$. □

COROLLARY 2. *Let k be the maximum, over all nodes v belonging to a given* C_i, *of lowest* (v) (*i.e., k is equal to* high (i), *as it is computed in procedure* weakcomp). *Then* $C_i, C_{i+1}, \cdots, C_{k-1}$ *are contained in the same weak component.*

Proof. The proof is straightforward after Theorem 1 and Lemma 3.

THEOREM 3. *This algorithm actually determines the weak components.*

Proof. The result of the test "$k > i + 1$" performed in procedure weakcomp determines whether C_i and C_{i+1} are contained in the same weak component. We want to show that C_i and C_{i+1} are parts of the same weak component if and only if the condition "$k > i + 1$" holds in procedure weakcomp.

Clearly, when the test is performed in procedure weakcomp, we have $k = \max(\text{high}(j) \mid j \leqq i)$.

(a) If k is bigger than $i + 1$, then there exists j smaller than or equal to i such that $k = \text{high}(j)$. Hence, after Corollary 2 of Lemma 3, C_j and C_{k-1} are contained in the same weak component. Therefore, after Lemma 2, C_i and C_{i+1} are contained in the same weak component.

(b) If k is equal to $i + 1$, we prove by induction on $i - j$ that, for all u's in C_{i+1} and for all v's in C_j, we have $u R^+ v$ (property P), for j smaller than $i + 1$.

Property P holds for $j = i$, after Corollary 1 of Lemma 3, since high (i) is equal to $i + 1$.

If property P holds for $h = i, i - 1, \cdots, j + 1$, then, for v belonging to C_j, either

—lowest (v) is smaller than $i + 1$. Then after Lemma 3, there exists w in $C_{\text{lowest}(v)}$ or $C_{\text{lowest}(v)-1}$, such that $w R v$. (w can belong to $C_{\text{lowest}(v)-1}$ only if lowest (v) is greater than $j + 1$). Therefore, as property P holds for h greater than j, for all u's in C_{i+1}, we have $u R^+ w R v$, or

—lowest (v) is equal to $i + 1$. Then, after Lemma 3, either

  —for all u's in C_{i+1}, we have $u R^+ v$, or

  —there exists w in C_i such that $w R v$. Therefore, as property P holds for h equals i, for all u's in C_{i+1}, we have $u R^+ w R v$.

Thus, if k is equal to $i + 1$, for all u's in C_{i+1} and all v's in $C_i \cup \cdots C_1$, we have $u R^+ v$. Hence, by Theorem 1, C_{i+1} and C_i are not contained in the same weak component.

(c) k cannot be smaller than $i + 1$, as high (i) is always bigger than i. □

THEOREM 4. *This algorithm requires* O(max (number of vertices, number of edges)) *computation time, if it is implemented on a random access computer.*

Proof. The proof is straightforward, since procedure partition is called exactly once for each vertex.

5. Possible extension. This algorithm can be extended to be used for any graph, acyclic or not, as follows. First, use Tarjan's algorithm [3] to determine

the strong components by performing a depth first search. For each strong component SC so determined, compute label (SC) and lowest (SC) according to procedure partition. Then shrink the strong components SC to get an acyclic graph, by discarding the edges interior to the strong components, and creating a new vertex v such that label (v) = label (SC), lowest (v) = lowest (SC), where the set of edges coming into or going out of v is the same as the set of edges coming into or going out of SC. Then apply the procedure weakcomp to the acyclic graph.

Acknowledgment. Sincere thanks are due to Professor E. L. Lawler for his encouragement and his help in preparing the manuscript, and to the two anonymous referees for their valuable criticisms.

REFERENCES

[1] V. CHVATAL, D. A. KLARNER AND D. E. KNUTH, *Selected combinatorial research problems*, Rep. STAN-CS-72-292, Computer Science Department, Stanford Univ., Stanford, Calif., 1972.

[2] R. L. GRAHAM, D. E. KNUTH AND T. S. MOTZKIN, *Complements and transitive closures*, Discrete Math., 2 (1972), pp. 17–30.

[3] R. TARJAN, *Depth-first search and linear graph algorithms*, this Journal, 1 (1972), pp. 146–160.

SIAM J. COMPUT.
Vol. 3, No. 1, March 1974

FINDING DOMINATORS IN DIRECTED GRAPHS*

ROBERT TARJAN†

Abstract. This paper describes an algorithm for finding dominators in an arbitrary directed graph. The algorithm uses depth-first search and efficient algorithms for computing disjoint set unions and manipulating priority queues to achieve a time bound of $O(V \log V + E)$ if V is the number of vertices and E is the number of edges in the graph. This bound compares favorably with the $O(V(V + E))$ time bound of previously known algorithms for finding dominators in arbitrary directed graphs, and with the $O(V + E \log E)$ time bound of a known algorithm for finding dominators in reducible graphs. If $E \geqq V \log V$, the new algorithm requires $O(E)$ time and is optimal to within a constant factor.

Key words. algorithm, binary tree, complexity, connectivity, depth-first search, directed graph, dominator, equivalence algorithm, graph, immediate dominator, priority queue, search, set union, stack, topological sorting, tree

1. Introduction. The following graph-theoretic[1] problem arises when one is attempting to optimize computer code [1], [2]: suppose G is a directed graph with a start vertex s. (G might represent the flow between blocks of a computer program; vertex s then represents the initial block of the program.) If vertex d lies on every path from vertex s to vertex i, then d is called a *dominator* of i. If d is a dominator of i and every other dominator d' of i also dominates d, then d is called an *immediate dominator* of i. It is easy to prove that each vertex has a unique immediate dominator if it has any dominators [1], [2]. We wish to find the immediate dominator of each vertex in the graph.

The dominators problem is relatively new and has not been studied extensively. Aho and Ullman's algorithm [1] for finding dominators deletes each vertex v in turn from G and tests which vertices are reachable from s. Any reachable vertex is not dominated by v. This algorithm requires $O(V(V + E))$ time if the problem graph has V vertices and E edges. Purdom and Moore's algorithm [3] has the same time bound; no previously published algorithm is faster in general. See [2], [4], [5] for other algorithms. Aho, Hopcroft and Ullman [6] have constructed an $O(V + E \log E)$ algorithm for finding dominators in a restricted class of graphs called reducible graphs [7], [8], [9]. Their algorithm is based on an efficient method for finding least common ancestors in trees.

This paper describes the use of depth-first search [10] to reveal the structure of directed graphs. Using efficient algorithms for computing disjoint set unions [11], [12], [13] and for manipulating priority queues [14], [15], we may calculate dominators from the search information. The resultant dominators algorithm

* Received by the editors April 26, 1973. This research was supported in part by the National Science Foundation under Grant GJ-33170X while the author was at Stanford University.

† Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, California 94720.

[1] The graph-theoretic definitions used in this paper are more or less standard. For those unfamiliar with graph theory, these definitions appear in Appendix A, along with a definition of the big "O" notation.

has an $O(V + E)$ space bound and an $O(V \log V + E)$ time bound. The method is optimal to within a constant factor if $E \geqq V \log V$.

The paper is divided into several sections. Section 2 describes depth-first search and its application to directed graphs. Section 3 describes four dominator-preserving graph transformations which use search information and which form the heart of the dominators algorithm. Section 4 outlines the algorithm. Sections 5 and 6 give the details of some of the necessary calculations, and § 7 presents the complete algorithm. Section 8 gives an even faster algorithm for finding dominators in certain special graphs, suggesting that a faster algorithm may exist in general. Section 9 gives conclusions.

2. Depth-first search. We wish to calculate IDOM(v), the immediate dominator of v, for each vertex v in a directed graph G with V vertices and E edges. Figure 1 shows a graph for which we might wish to solve this problem. We begin by exploring G starting at vertex s and marking all vertices reached. Vertex s and vertices remaining unmarked have no dominators, while all other vertices have dominators. The problem is then reduced to finding dominators in the subgraph G_1 whose vertices are all those reachable from s. In G_1 each vertex has a dominator. Furthermore, we have the following lemma.

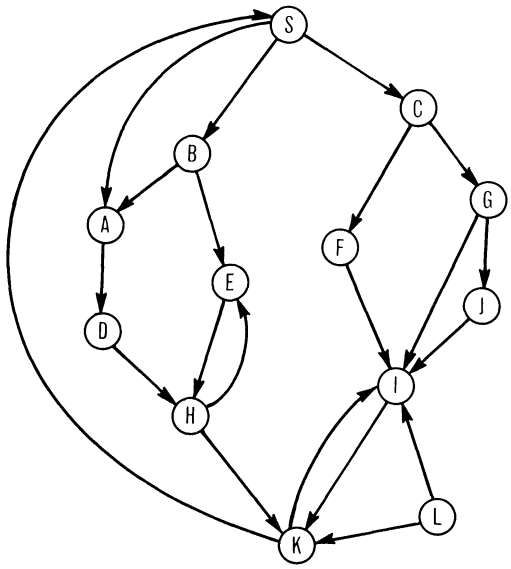


FIG. 1. *A directed graph in which we wish to find dominators*

LEMMA 1 [1], [2]. *We may construct a tree (called the dominator tree of G_1) whose vertices are those of G_1 and such that w is a son of v in the tree if and only if v is the immediate dominator of w. The ancestors of w in the tree are precisely the dominators of w.* (Figure 2 shows the dominator tree of the graph in Fig. 1.)

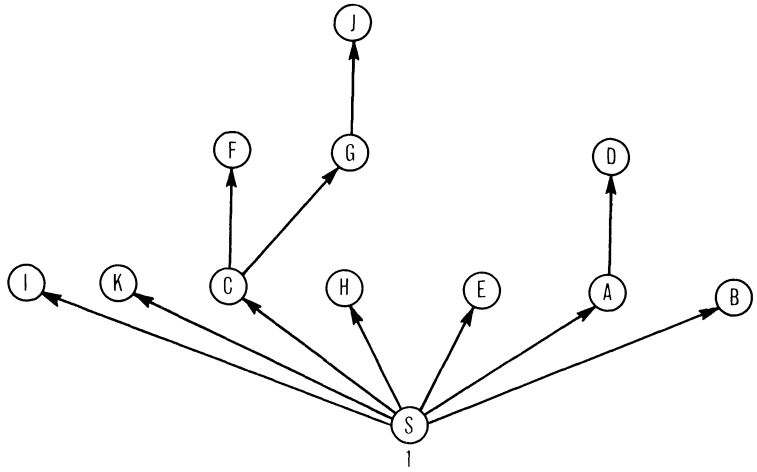


FIG. 2. *The dominator tree of the graph in Fig.* 1. *Vertex L in Fig.* 1 *has no dominators.*

To mark the vertices reachable from s, we carry out a depth-first search of G [10]. That is, we start at vertex s and choose an edge leading from s to explore. Traversing the edge leads to a vertex, either new or already reached. In general, we continue the search by selecting and traversing an unexplored edge from the most recently reached vertex which still has unexplored edges. Eventually each edge will be traversed exactly once. To implement such a search, we use a set of adjacency lists $A(v)$, one for each vertex v in the graph; if (v, w) is an edge of G, then w appears in the adjacency list $A(v)$. Each edge is represented exactly once. Here is a recursive procedure for carrying out a depth-first search:

```
procedure DFS(v);
  begin comment v is the most recently reached vertex;
    MARK(v) := true;
    for w ∈ A(v) do
      if ¬MARK(v) then DFS(w);
  end;
```

The following statements will then mark every vertex reachable from s, by applying DFS:

```
comment mark all vertices reachable from s;
for each vertex v do MARK(v) := false;
DFS(s);
```

A depth-first search yields much more information than just which vertices are reachable from the start vertex of the search. In particular, it gives enough information about the connectivity structure of the graph to efficiently determine dominators. Let us add a few more calculations to the search. (Henceforth for convenience we shall assume that *all* vertices in G are reachable from s.)

DFS is a recursive procedure; the successively reached new vertices are input parameters to DFS and thus are stored on a stack (in any implementation of DFS).

This stack contains all vertices reached which may still have unexplored edges, and the vertices as they appear in order on the stack determine a path in G from s to the current vertex being examined during the search. Suppose we keep track of which vertices are stacked at any given time, and that we number the vertices from 1 to V in the order they are reached during the search. Then a depth-first search of a directed graph partitions the edges traversed into four classes:

(1) Edges (v, w) with w unmarked when (v, w) is explored, called *tree arcs.*
(2) Edges (v, w) with w stacked when (v, w) is explored, called *fronds.*
(3) Edges (v, w) with NUMBER(v) < NUMBER(w) and w unstacked when (v, w) is explored, called *reverse fronds.*
(4) Edges (v, w) with NUMBER(v) > NUMBER(w) and w unstacked when (v, w) is explored, called *cross-links.*

Lemma 2 below gives the properties of edges in these four classes. In particular, the tree arcs determine a spanning tree T of G which has root s. One more numbering scheme based on depth-first search is useful. Let the vertices of G be numbered from V to 1 as they are unstacked during the search. We shall denote this numbering by SNUMBER(v). Lemma 3 below gives properties of SNUMBER's. Here is an elaborated version of the depth-first search procedure which computes both types of number for each vertex, divides edges into their classes, and also counts the number of descendants ND(v) of each vertex v in the spanning tree T.

```
procedure CLASSIFY(G, s);
  begin comment the edge-classifying procedure uses the following elaborated
    version of DFS. Variable m denotes the last NUMBER assigned to any
    vertex. Variable n denotes the last SNUMBER assigned to any vertex. The
    procedure assumes that G is represented as a set of adjacency lists A(v).
    NUMBER(v) = 0 if and only if v has not been reached. SNUMBER(v) = 0
    if and only if v has not yet been unstacked;
  procedure DFSEARCH(v);
    begin
      m := NUMBER(v) := m + 1;
      ND(v) := 1;
      for w ∈ A(v) do
        if NUMBER(w) = 0 then
          begin
            label(v, w) a tree arc;
            DFSEARCH(w);
            ND(v) := ND(v) + ND(w);
          end
          else if SNUMBER(w) = 0 then
            begin comment vertex w is stacked;
              label(v, w) a frond;
            end
          else if NUMBER(v) < NUMBER(w) then
              label(v, w) a reverse frond
          else label(v, w) a cross-link;
```

```
        n := SNUMBER(v) := n − 1;
        comment v is now unstacked;
      end;
    comment to classify the edges we initialize and call DFSEARCH;
      m := 0;
      n := V + 1;
      for each vertex v do NUMBER(v) := SNUMBER(v) := 0;
    DFSEARCH(s);
  end;
```

Figure 3 illustrates what CLASSIFY does to the graph in Fig. 1. It should be clear that this elaborated version of depth-first search correctly numbers the vertices and classifies the edges. Reference [10] contains a proof that these calculations require $O(V + E)$ time and space. Lemmas 2–5, given without proof, state basic properties of the numbers calculated by CLASSIFY.

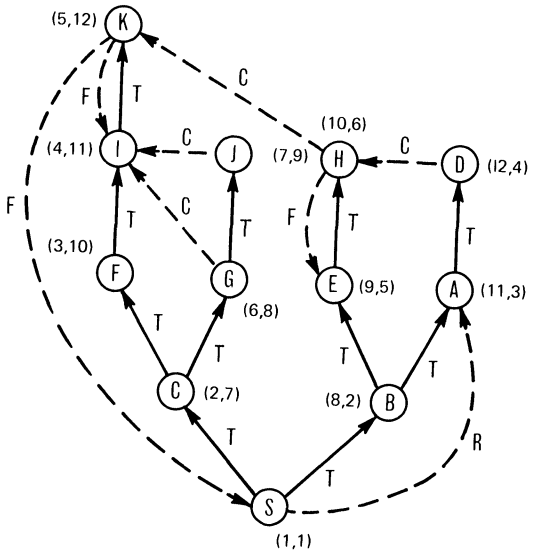


FIG. 3. *Depth-first search applied to graph in Fig.* 1. *Tree arcs are labeled* T, *fronds* F, *reverse fronds* R, *and cross-links* C. *Numbering vertices from* 1 *to* V *as vertices are reached during the search gives the first number at each vertex. Numbering the vertices from* V *to* 1 *as vertices are unstacked during the search gives the second number at each vertex.*

LEMMA 2. *Suppose that all vertices of a directed graph* G *are reachable from vertex* s, *and that the edges of* G *are divided into classes using* CLASSIFY(G, s). *Then*:

(i) *The tree arcs form a directed tree* T *with root* s *which contains all vertices in* G. *We shall denote the existence of a tree arc* (v, w) *by* $v \to w$, *and the existence of a path from* v *to* w *in* T *by* $v \overset{*}{\to} w$.

(ii) *If* (v, w) *is a frond, then* NUMBER(w) < NUMBER(v), *and* $w \overset{*}{\to} v$ *in* T.

(iii) *If* (v, w) *is a reverse frond, then* $v \overset{*}{\to} w$ *in* T.

(iv) *If* (v, w) *is a cross-link, then neither* $v \overset{*}{\to} w$ *in* T *nor* $w \overset{*}{\to} v$ *in* T.

LEMMA 3. *If* (v, w) *is a tree arc, a reverse frond, or a cross-link,* $\text{SNUMBER}(v) < \text{SNUMBER}(w)$. *If* (v, w) *is a frond,* $\text{SNUMBER}(v) > \text{SNUMBER}(w)$.

LEMMA 4. *Let* v *be a vertex in* G. *Then the number of descendants of* v *in the spanning tree* T *is given by:*

$$\text{ND}(v) = 1 + \sum_{v \to w} \text{ND}(w).$$

LEMMA 5. *Statements* (i), (ii), (iii) *and* (iv) *below are equivalent.*

(i) $v \overset{*}{\to} w$ *in* T.

(ii) $\text{NUMBER}(v) \leqq \text{NUMBER}(w) < \text{NUMBER}(v) + \text{ND}(v)$.

(iii) $\text{SNUMBER}(v) \leqq \text{SNUMBER}(w) < \text{SNUMBER}(v) + \text{ND}(v)$.

(iv) $\text{NUMBER}(v) \leqq \text{NUMBER}(w)$ *and* $\text{SNUMBER}(v) \leqq \text{SNUMBER}(w)$.

Lemma 5 gives us three methods for identifying the descendants of a vertex and allows us to dynamically identify fronds, reverse fronds, and cross-links if we so desire.

LEMMA 6. *G is acyclic if and only if G has no fronds.*

Proof. If G has a frond (v, w), then the frond and the set of tree arcs joining v and w form a cycle. If G has no fronds, all edges (v, w) satisfy $\text{SNUMBER}(v) < \text{SNUMBER}(w)$. Since any cycle in G must have at least one arc (v, w) with $\text{SNUMBER}(v) > \text{SNUMBER}(w)$, G has no cycles.

COROLLARY 7. *If G is an acyclic directed graph,* CLASSIFY *assigns* SNUMBER*'s to G so that if* (v, w) *is an edge,* $\text{SNUMBER}(v) < \text{SNUMBER}(w)$. *Thus* CLASSIFY *gives an* $O(V + E)$ *algorithm for "topologically sorting" the edges of G.* (See Knuth [16] for further discussion of this problem.)

LEMMA 8. *Let p be a path from v to w in G. Let vertices be identified by their number. Suppose* $v < w$. *Then p contains some common ancestor of v and w in T.*

Proof. Let T_u with root u be the smallest subtree of T containing all vertices on the path p. We prove that p passes through u. If $u = v$ or $u = w$, the result is immediate. Otherwise, let $u_1 < u_2 < \cdots < u_n$ be the sons of u such that for each u_i, some descendant of u_i is on p. For any i, let T_{u_i} be the subtree of T with root u_i. If $n = 1$, p must pass through u since p is minimal. If $n > 1$, there must be some u_i, u_j, $i < j$, such that p leads from T_{u_i} to T_{u_j}. This is true since $v < w$ and all the vertices in T_{u_i} are numbered smaller than all the vertices in T_{u_j} if $i < j$. But p can only get from T_{u_i} to T_{u_j} by passing through u, since the only edges leading from lower numbered vertices to higher numbered ones are tree arcs and reverse fronds. The lemma follows.

The properties of depth-first search presented above may be used to construct a good algorithm to solve the dominators problem. One way to find dominators is to convert G into an equivalent acyclic graph, by deleting each frond and replacing it by an equivalent set of reverse fronds and cross-links to preserve dominators. In the resultant acyclic graph, the dominators may be found for the vertices in SNUMBER order from 1 to V, since any path leads through vertices with increasing SNUMBER. This algorithm has an $O(V^2)$ time bound [17]; the time bound is not linear in the number of edges because the number of added reverse fronds and cross-links may be large. To get a faster algorithm, we must be a little more clever.

The idea we use is to convert G into a graph with no fronds and no cross-links by adding suitable reverse fronds. We use four simplifying transformations which preserve dominators to accomplish this. First we delete all fronds and replace them with a simpler set of fronds, at most one leaving each vertex. Then we convert cross-links to reverse fronds, we combine fronds and reverse fronds to give new reverse fronds, and we delete all but one reverse frond entering each vertex. The last three transformations must be carried out simultaneously with the dominator calculations. The computation proceeds through the vertices in NUMBER order, from V to 1. Section 3 describes the four transformations and proves that they preserve dominators.

3. Dominator-preserving transformations. Suppose that a depth-first search of a directed graph G is carried out using CLASSIFY(G, s), and that all vertices of G are reachable from the start vertex s. For any vertex v, let $F(v) = \{w | w \neq v$ and $\exists u(w \overset{*}{\to} v \to u$ in T and (u, w) is a frond of $G)\}$. Let HIGHPT(v) be the highest numbered vertex in $F(v)$ if $F(v)$ is nonempty. (Here and henceforth we shall only use one numbering of vertices, NUMBER as calculated by CLASSIFY.) Since each element in $F(v)$ is an ancestor of v in T, it is clear that $w \in F(v)$ implies $w \overset{*}{\to}$ HIGHPT(v). Let G' be the graph formed from G by deleting all fronds and adding a new frond (v, HIGHPT(v)) for each vertex v for which HIGHPT(v) is defined. This is our first transformation, called *frond replacement*. Figure 4 shows the graph in Fig. 3 transformed in this way.

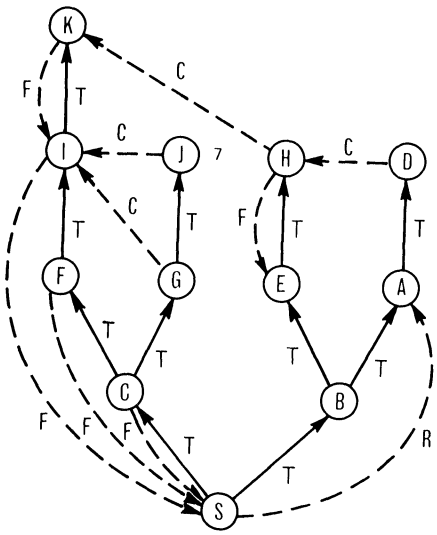


FIG. 4. *Frond replacement applied to the graph in Fig. 3. Vertices are numbered in search order. Vertex* K *loses a frond; vertices* I, F *and* C *gain a frond.*

We have the following results.

LEMMA 9. *Let G' be formed from G by frond replacement. If $w \overset{*}{\to} v \overset{*}{\to} u$ in T (the spanning tree of G and G'), and if (u, w) is a frond in G, then there is a path from v to w in G' which consists only of fronds.*

Proof. The proof is by induction on the length of the tree path from w to v. If $w = v$, then the lemma is true since there is a path containing no edges from any vertex to itself. Let the lemma be true whenever the tree path from w to v has length less than k, and suppose the tree path from w to v has length k. Since G contains a frond (u, w) with $w \overset{*}{\to} v \overset{*}{\to} u$, $w \in F(v)$ and HIGHPT(v) is defined. Furthermore HIGHPT(v) $\neq v$ and $w \overset{*}{\to}$ HIGHPT(v) $\overset{*}{\to} v$. By the induction hypothesis, there is a path of fronds from HIGHPT(v) to w in G'. Adding (v, HIGHPT(v)) to the front of this path gives a path in G' from v to w which consists only of fronds. By induction the lemma is true.

LEMMA 10. *Vertex d dominates vertex v in G if and only if d dominates v in G'.*

Proof. Suppose w does not dominate v in G. Then in G there is a path from s to v which does not contain w. Suppose this path contains a frond (u, u') with $u' \overset{*}{\to} w$. Then we may replace the part of p up to and including the last such frond by a path of tree arcs. This gives us a path in G from s to v which contains neither w nor any frond (u, u') such that $u' \overset{*}{\to} w$. If we now replace each frond in the path by the corresponding path of fronds in G' guaranteed by Lemma 9, we get a path p' in G' from s to v which doesn't contain w, and w does not dominate v in G'.

Conversely, suppose w does not dominate v in G'. Then there is a path p' in G' from s to v which doesn't contain w. Suppose this path contains a frond $(u,$ HIGHPT(u)) with HIGHPT(u) $\overset{*}{\to} w$. Then we may replace the part of p' up to and including the last such frond by a path of tree arcs. This gives us a path p'' in G' from s to v which contains neither w nor any frond $(u,$ HIGHPT(u)) with HIGHPT(u) $\overset{*}{\to} w$. Corresponding to any remaining frond $(u,$ HIGHPT(u)) on path p'' there is a frond $(u',$ HIGHPT(u)) in G with $u \overset{*}{\to} u'$. If we replace each frond $(u,$ HIGHPT(u)) in p'' by a path of tree arcs from u to u' followed by the frond $(u',$ HIGHPT(u)), we get a path p in G from s to v which doesn't contain w. It follows that w does not dominate v in G, and the lemma is true.

To calculate dominators in G, we apply frond replacement and calculate dominators in the transformed graph G'. Observe that if frond replacement is applied to G', the result is G' itself. Henceforth we shall assume that G is a graph which has been explored using CLASSIFY and whose fronds have been replaced as specified above. We shall identify vertices using the NUMBER assigned to them by CLASSIFY.

LEMMA 11. *Let (u, v) and (u_1, v) be two reverse fronds in G, with $u_1 > u$. Let G' be the graph formed from G by deleting edge (u_1, v). We call this transformation "reverse frond deletion". Then d dominates v in G if and only if d dominates v in G'.*

Proof. Since G' is a subgraph of G, every path in G' is a path in G. Thus if d dominates v in G, d dominates v in G'. Conversely, suppose w does not dominate v in G. Then there is a path p in G from s to v which doesn't contain w. If p doesn't contain (u_1, v), then p is a path in G' and w doesn't dominate v in G'. Suppose p contains (u_1, v). If w is a descendant of u_1, we may replace the part of p up to and including edge (u_1, v) by the path of tree arcs from s to u followed by the frond (u, v) to get a path p' in G' from s to v which doesn't contain w. If w is not a descendant of u_1, we may replace (u_1, v) in p' by the path of tree arcs from u_1 to v and get a path p' in G' from s to v which doesn't contain w. In no case can w dominate v in G', and the lemma is true.

LEMMA 12. *Let v be a vertex in G such that one frond* $(v, \text{HIGHPT}(v))$ *leaves* v, *at most one reverse frond (say* (u, v)*) enters* v, *and no cross-links or fronds enter* v. *Let G' be the graph formed from G by deleting frond* $(v, \text{HIGHPT}(v))$ *and adding* $(u, \text{HIGHPT}(v))$ *if* (u, v) *is defined and* $(u, \text{HIGHPT}(v))$ *is a reverse frond (i.e.,* $u < \text{HIGHPT}(v)$*). We call this transformation "frond deletion". Then d dominates w in G if and only if d dominates w in G', assuming no frond deletions have been applied to vertices $x < v$.*

Proof. Suppose x does not dominate w in G. Then there is a path p in G from s to w which doesn't contain x. If p doesn't contain $(v, \text{HIGHPT}(v))$, then p is a path in G'. If p contains $(v, \text{HIGHPT}(v))$, then p must contain either (u, v) or the tree arc entering v. If x is not a proper ancestor of $\text{HIGHPT}(v)$, then we can replace the part of p up to and including $(v, \text{HIGHPT}(v))$ by the path of tree arcs from s to $\text{HIGHPT}(v)$ and have a path in G' which doesn't contain x. Suppose x is a proper ancestor of $\text{HIGHPT}(v)$. If the edge before $(v, \text{HIGHPT}(v))$ in p is (u, v) and $u < \text{HIGHPT}(v)$, then we may replace (u, v) and $(v, \text{HIGHPT}(v))$ by $(u, \text{HIGHPT}(v))$ and have a path in G' which doesn't contain x. If the edge before $(v, \text{HIGHPT}(v))$ is a tree arc, or if it is (u, v) and $u \geqq \text{HIGHPT}(v)$, then by Lemma 9 we may replace $(v, \text{HIGHPT}(v))$ and the edge before it by a path of fronds and have a path in G' which doesn't contain x. In any case x doesn't dominate w in G'.

Conversely, suppose x does not dominate w in G'. Then there is a path p' in G' from s to w which doesn't contain x. If p' doesn't contain $(u, \text{HIGHPT}(v))$, then p' is a path in G. Suppose p' contains $(u, \text{HIGHPT}(v))$. If $x \neq v$, we may replace $(u, \text{HIGHPT}(v))$ in p' by (u, v) and $(v, \text{HIGHPT}(v))$ to give a path in G which doesn't contain x. If $x = v$, we may replace the part of p' up to and including $(u, \text{HIGHPT}(v))$ by the path of tree arcs from s to $\text{HIGHPT}(v)$ and have a path in G which doesn't contain w. In no case does x dominate w in G, and the lemma is true.

The dominators algorithm works in the following way: first we apply frond replacement to G. Next, we process the vertices from V to 1. To process a vertex v, we convert all incoming cross-links to reverse fronds (by a transformation yet to be described), we eliminate all but one reverse frond entering v by applying reverse frond deletion, and we eliminate the frond (if any) leaving v by applying frond deletion. We are left with at most one tree arc and one reverse frond entering v. We then update the partially calculated dominators and proceed to the next vertex.

In order to understand the transformation of cross-links into reverse fronds, we must examine in detail the way dominators are calculated. Let G be a graph which has been explored using CLASSIFY and whose fronds have been replaced. Let $G(i)$ be the subgraph of G which contains all the tree arcs in G plus all edges leading to vertices v such that $\text{NUMBER}(v) > i$. Let the *i-th semidominator* of vertex v (abbreviated $\text{SDOM}(i, v)$) be the immediate dominator of v in $G(i)$. In all that follows we shall assume that $v \neq 1$ (that is, v is not the start vertex) so all semidominators are defined. The dominators algorithm calculates $\text{SDOM}(i - 1, v)$ for all vertices when vertex i is processed. The SDOM values tell us how to convert cross-links into reverse fronds as well as giving dominators ($\text{SDOM}(0, v) = \text{IDOM}(v)$). The lemmas below describe semidominators. By combining these results with the lemmas above, we can build a dominators algorithm.

LEMMA 13. *If* $v \neq 1$ *and* $i \geqq v$, SDOM(i, v) *is the father of vertex* v *in the tree generated by* CLASSIFY.

Proof. $G(i)$ contains only one edge leading to vertex v; namely, a tree arc. Every path from s to v must pass through this edge. The lemma follows.

LEMMA 14. *For all* i, *if* $v \neq 1$ *then* SDOM$(i-1, v) \overset{*}{\rightarrow}$ SDOM(i, v).

Proof. For all i, if $v \neq 1$, the only dominators of v in $G(i)$ lie on the tree path from s to v. Furthermore, $G(i)$ is a subgraph of $G(i-1)$, so if w dominates v in $G(i-1)$, w must dominate v in $G(i)$. The lemma follows.

LEMMA 15. *If* $v \neq 1$ *and* SDOM$(i, v) = i$, *then* IDOM$(v) = i$.

Proof. By Lemma 14, IDOM(v) = SDOM$(0, v) \overset{*}{\rightarrow}$ SDOM(i, v). Thus if SDOM(i, v) dominates v in G, SDOM(i, v) = IDOM(v). Now we show by contradiction that SDOM(i, v) dominates v in G if SDOM$(i, v) = i$.

Suppose to the contrary that there is a path p in G from s to v which does not contain i = SDOM(i, v). We must have $i \overset{*}{\rightarrow} v$. Some edge in p must begin at a nondescendant of vertex i and lead to a descendant of i. Let (u, w) be the last such edge in p. Then $w > i$, and all edges following (u, w) in p have both endpoints among the descendants of i. Since all the descendants of i have numbers larger than i, and since u is not a descendant of i, we may replace the part of p up to edge (u, w) by a path of tree arcs from s to u and have a path in $G(i)$ which doesn't contain i. But this is a contradiction by the definition of SDOM(i, v), and SDOM(i, v) must dominate v in G. The lemma follows.

LEMMA 16. *If* $v \neq 1$ *and* $i \leqq v$, *then either* SDOM(i, v) = IDOM(v) *or* SDOM$(i, v) \overset{*}{\rightarrow} i$.

Proof. The proof is by induction on i. If $i = v$, SDOM$(i, v) \overset{*}{\rightarrow} i$ by Lemma 13. Let the lemma be true if $i_0 < i \leqq v$. Suppose $i = i_0$. Vertex i is a descendant of the father of vertex $i + 1$. By the induction hypothesis, either SDOM$(i+1, v)$ = IDOM(v) or SDOM$(i+1, v) \overset{*}{\rightarrow} i + 1$. If SDOM$(i+1, v)$ = IDOM(v), then SDOM(i, v) = IDOM(v) by Lemma 14. Otherwise SDOM$(i+1, v) \overset{*}{\rightarrow} i + 1$ and SDOM$(i+1, v) \neq i + 1$ by Lemma 15. Then by Lemma 14 and the comment above, SDOM$(i, v) \overset{*}{\rightarrow} i$, and the lemma follows by induction on i.

LEMMA 17. *If* u *does not dominate* v *in* G, *and edge* (u, v) *in* G *is replaced by edge* (IDOM$(u), v)$ *to form graph* G', *then* d *dominates* w *in* G *if and only if* d *dominates* w *in* G'.

Proof. Suppose x does not dominate w in G. Then there is a path p in G from s to w which does not contain x. If p does not contain edge (u, v) then p is a path in G'. Suppose p does contain (u, v). Then p contains IDOM(u), and we may replace the part of p leading from IDOM(u) to v by the edge (IDOM$(u), v)$ and have a path in G' from s to w which doesn't contain x. Thus x does not dominate w in G'.

Conversely, suppose x does not dominate w in G'. Then there is a path p' in G' from s to w which doesn't contain x. If (IDOM$(u), v)$ is not on p', then p' is a path in G. Suppose (IDOM$(u), v)$ is on p'. If $x = u$, then since u doesn't dominate v, there is a path q in G from s to v which doesn't contain x. By substituting q for the part of p' up to and including edge (IDOM$(u), v)$, we get a path p in G from s to w which doesn't contain x. Suppose $x \neq u$. If every path from IDOM(u) to u in G contains x, then x dominates u. Also, every path from s to x must contain IDOM(u), since IDOM(u) dominates u. Then IDOM(u) dominates x or

IDOM(u) = x, and either is a contradiction. It follows that there is some path r in G from IDOM(u) to u which doesn't contain x. Substituting r and the edge (u, v) for the edge (IDOM(u), v) in p', we get a path in G from s to w which doesn't contain x. In no case can x dominate w in G, and the lemma is true.

LEMMA 18. *Let (u, v) be a cross-link in G. Suppose G is transformed into a new graph G' by deleting (u, v) and adding edge* (SDOM(v, u), v). *We call this transformation "cross-link replacement". Then d dominates w in G if and only if d dominates w in G'.*

Proof. Since (u, v) is a cross-link, $v < u$. If SDOM(v, u) = IDOM(u), then the lemma is true by Lemma 17, since u cannot dominate v unless $u \overset{*}{\rightarrow} v$, and $u \overset{+}{\rightarrow} v$ is impossible by Lemma 2. Suppose SDOM(v, u) $\neq$ IDOM(u). Then SDOM(v, u) $\overset{+}{\rightarrow} v$ by Lemma 16. Now suppose x does not dominate w in G. Then there is a path p in G from s to w which doesn't contain x. If (u, v) is not an edge of p, then p is a path in G'. Suppose p contains (u, v). If x is not an ancestor of SDOM(v, u), we may replace the part of p up to and including edge (u, v) by the path of tree arcs from s to SDOM(v, u) and the edge (SDOM(v, u), v) to get a path in G' from s to w which doesn't contain x.

On the other hand, suppose x is an ancestor of SDOM(v, u). Consider the part of p from s to u. Let (y, z) be the last edge on this part of p with $y \leqq v$. Then y must be an ancestor of u, since by Lemma 8 any path from y to u must pass through a common ancestor a of y and u, and this a satisfies $a \leqq y \leqq v$. Vertex y must also be an ancestor of v, since $y \leqq v \leqq u$, and any ancestor of u which is not an ancestor of v will have a number greater than the number of v. The part of p from y to u lies in $G(v)$, so SDOM(v, u) $\overset{+}{\rightarrow} y$. Otherwise there would be a path in $G(v)$ from s to u which didn't contain SDOM(v, u), an impossibility. Thus we have a vertex y on p such that $x \overset{*}{\rightarrow}$ SDOM(v, u) $\overset{+}{\rightarrow} y \overset{+}{\rightarrow} v$. We may replace the part of p from y to v (including edge (u, v)) by the path of tree arcs from y to v. This gives a path in G' from s to w which doesn't contain x, and x does not dominate w in G'.

Conversely, suppose x doesn't dominate w in G'. Let p' be a path from s to w in G' which doesn't contain x. If p' doesn't contain (SDOM(v, u), v), then p' is a path in G. Suppose p' does contain (SDOM(v, u), v). If $x = u$, we may replace (SDOM(v, u), v) in p' by the path of tree arcs from SDOM(v, u) to v and get a path from s to w in G which doesn't contain x. If $x \neq u$, then there must be a path q in $G(v)$ (hence in G) from SDOM(v, u) to u which doesn't contain x. Otherwise SDOM(v, u) dominates x and x dominates u, which is impossible. Replacing (SDOM(v, u), v) in p' by q plus edge (u, v) gives a path from s to w in G which doesn't contain x. In no case can x dominate w in G, and the lemma is true.

Lemma 18 tells us how to transform cross-links into reverse fronds. Suppose that for a fixed v, we know SDOM(v, w) for all vertices w. To convert cross-link (u, v) into a reverse frond, we apply cross-link replacement. If the resultant edge is still a cross-link, we apply cross-link replacement to it. We continue until we get an edge which is a reverse frond.

Now all we need is a method for calculating semidominators. Lemma 13 tells us how to initialize the calculation. Lemma 15 tells us when we can stop calculating semidominators for a particular vertex. The next lemma indicates how to update the semidominator values.

LEMMA 19. *Let i be a vertex in G such that the only two edges entering i are a tree arc (u, i) and possibly a reverse frond (w, i). If no reverse frond enters i, then* $\mathrm{SOMD}(i-1, v) = \mathrm{SDOM}(i, v)$ *for all v. If a reverse frond (w, i) enters i, then* $\mathrm{SDOM}(i-1, v) = w$ *for all v such that $i \overset{*}{\rightarrow} v$ and $w \overset{*}{\rightarrow} \mathrm{SDOM}(i, v) \overset{*}{\rightarrow} u$, and* $\mathrm{SDOM}(i-1, v) = \mathrm{SDOM}(i, v)$ *for all other vertices v.*

Proof. If i has no frond, cross-link or reverse frond entering it, then $G(i) = G(i-1)$; thus the first part of the lemma is true. Suppose i has a reverse frond (w, i) entering it. Since $G(i-1)$ has no edges but tree arcs which lead to vertices numbered less than i, any path in $G(i-1)$ which contains (w, i) must terminate at a descendant of i. Thus the paths from s to nondescendants of i are the same in both $G(i)$ and $G(i-1)$. This means that if v is not a descendant of i, then $\mathrm{SDOM}(i, v) = \mathrm{SDOM}(i-1, v)$.

Suppose that v is a descendant of i. If $\mathrm{SDOM}(i, v) \neq \mathrm{SDOM}(i-1, v)$, then there must be a path p in $G(i-1)$ from s to v which doesn't contain $\mathrm{SDOM}(i, v)$, but no such path in $G(i)$. Path p must contain reverse frond (w, i). But if it is not true that $w \overset{*}{\rightarrow} \mathrm{SDOM}(i, v) \overset{*}{\rightarrow} u$, we may replace (w, i) in p by the path of tree arcs from w to i and get a path in $G(i)$ from s to v which doesn't contain $\mathrm{SDOM}(i, v)$. This contradiction implies that if $\mathrm{SDOM}(i-1, v) \neq \mathrm{SDOM}(i, v)$, then $w \overset{*}{\rightarrow} \mathrm{SDOM}(i, v) \overset{*}{\rightarrow} u$.

Now suppose it is true that $w \overset{*}{\rightarrow} \mathrm{SDOM}(i, v) \overset{*}{\rightarrow} u$. If $\mathrm{SDOM}(i, v) \overset{*}{\rightarrow} u$, then if $d \overset{*}{\rightarrow} \mathrm{SDOM}(i, v)$, d dominates $\mathrm{SDOM}(i, v)$ in $G(i)$ and d dominates v in $G(i)$. (The only arcs entering any ancestor of $\mathrm{SDOM}(i, v)$ in $G(i)$ are tree arcs since $\mathrm{SDOM}(i, v) \leqq u < i$.) On the other hand, any vertex which is not an ancestor of $\mathrm{SDOM}(i, v)$ cannot dominate v in $G(i)$. Thus in $G(i)$ the dominators of v are exactly the ancestors of $\mathrm{SDOM}(i, v)$. Now in $G(i-1)$, if $d \overset{*}{\rightarrow} w$, then d dominates w and d dominates v. If $d \overset{*}{\rightarrow} \mathrm{SDOM}(i, v)$ but d is not an ancestor of w, then d does not dominate v, since the path of tree arcs from s to w followed by the reverse frond (w, i) followed by the path of tree arcs from i to v is a path in $G(i-1)$ which doesn't contain d. Thus the dominators of v in $G(i-1)$ are exactly the ancestors of w, and $\mathrm{SDOM}(i-1, v) = w$. The lemma follows.

Now we have a method for calculating semidominators. We must not overlook one subtle point. To get dominators, we will calculate semidominators, applying the various dominator-preserving transformations to simplify the calculations. We must make sure that these transformations preserve not only dominators but also semidominators; otherwise the intermediate calculations may go haywire. The next lemma takes care of this worry.

LEMMA 20. *Let G' be formed from G by applying either reverse frond deletion, frond deletion, or cross-link replacement. Then all semidominator values of G and G' agree.*

Proof. We must compare dominators in $G(i)$ and $G'(i)$ to verify that the ith semidominators in G and G' are the same. Suppose G' is formed from G by reverse frond deletion. Let (u_1, v) be the deleted reverse frond, where (u, v) is another reverse frond in G and $u_1 > u$. If $i \geqq v$, then $G(i) = G'(i)$, and the lemma is true. If $i < v$, then both (u, v) and (u_1, v) appear in $G(i)$, $G'(i)$ is formed from $G(i)$ by reverse frond deletion, and the lemma is true by Lemma 11.

Suppose G' is formed from G by frond deletion. Let $(v, \mathrm{HIGHPT}(v))$ be the deleted frond and $(u, \mathrm{HIGHPT}(v))$ the added reverse frond, where only a tree arc

and a reverse frond (u, v) enter v, and only one frond leaves v. If $i \geqq \text{HIGHPT}(v)$, then $G(i) = G'(i)$, and the lemma is true. If $i < \text{HIGHPT}(v)$, then $(v, \text{HIGHPT}(v))$ and (u, v) are in $G(i)$, $(u, \text{HIGHPT}(v))$ is in $G'(i)$, and $G'(i)$ is formed from $G(i)$ by frond deletion. In this case the lemma follows by Lemma 12.

Suppose G' is formed from G by cross-link replacement. Let (u, v) be the cross-link in G which is replaced by $(\text{SDOM}(v, u), v)$ to form G'. If $i \geqq v$, then $G'(i) = G(i)$, and the lemma is true. If $i < v$, then $G(v)$ is a subgraph of $G(i)$, and the vth semidominator of u is the same in both $G(i)$ and in G. Edge (u, v) is in $G(i)$ and edge $(\text{SDOM}(v, u), v)$ is in $G'(i)$. It follows that $G'(i)$ is formed from $G(i)$ by cross-link replacement, and the lemma follows by Lemma 18.

4. An outline of the dominators algorithm. Now we have all the results needed to build a dominators algorithm. Below is an outline of the algorithm in ALGOL-like notation.

```
procedure DOMINATORS(G, s);
  begin
    a: apply CLASSIFY(G, s) to classify the edges and number the vertices of G
       reachable from s;
       let G1 be the subgraph of G containing all vertices reachable from s;
       let G1 have V1 vertices;
       for each vertex v of G not in G1 do IDOM(v) = 0;
    b: for each vertex v of G1 do calculate HIGHPT(v); apply frond replace-
       ment to G1 to form graph G2;
    c: for i := V1 step − 1 until 0 do
         for v := 1 until V1 do
           calculate SDOM (i, v);
       for i := 1 until V1 do IDOM(i) := SDOM(0, i);
  end;
```

This algorithm is straightforward and works correctly by the results in § 3. (Frond replacement preserves dominators by Lemma 12.) Step a requires $O(V + E)$ time by the discussion in § 2, and the total time required by all steps except b and c is $O(V + E)$. Using results in § 3 we can give some details of Step c:

```
c: comment calculate semidominators;
     begin
       for v := 1 until V1 do
       d: begin
            using reverse frond deletion, delete all reverse fronds but one
              entering vertex v;
            let this reverse frond be (LOWPT(v), v);
          end;
       e: for i := 1 until V1 do calculate SDOM(V1, i) using Lemma 13;
          for i := V1 step − 1 until 1 do
            f: for each cross-link (u, i) do
                 begin
                   g: convert (u, i) into a reverse frond (w, i) by repeated
```

```
                    cross-link replacement;
                  if two reverse fronds now enter i then delete one by
                    reverse frond deletion;
              end;
                if a frond (i, x) leaves i then delete it by frond deletion;
                if two reverse fronds now enter x then delete one by reverse
                  frond deletion;
                h: for v := 1 until V1 do apply Lemma 19 to calculate
                  SDOM(i − 1, v);
              end;
  end;
```

Consider this implementation of the semidominators calculation. For a fixed value of i, Step f deletes any frond leaving i. Thus before Step f is executed for any fixed j, all fronds entering j have been deleted. It follows that the transformations in Step c preserve semidominators, by Lemma 20. Thus Step c as implemented above works correctly. The total time required by Step c, not including Steps g and h, is obviously $O(V + E)$. The dominators algorithm thus has a linear time bound not including Steps b, g and h. These steps require some good data structures, which are presented in the next two sections.

5. Calculating HIGHPT(v). In this section we implement Step b, the calculation of HIGHPT values. A straightforward algorithm for calculating HIGHPT values requires $O(V^2)$ time. With a good scheme for implementing priority queues, such as Crane's using binary trees [14] or Hopcroft's using 3-2 trees [15], we may achieve an $O(E \log E)$ time bound. However, if we are a little more clever, then we can use a good algorithm for computing disjoint set unions and construct an almost-linear algorithm. First we sort the fronds (u, w) of G by the NUMBER of w. Then we calculate HIGHPT's by processing the fronds (u, w) in order from largest w to smallest w. We will label each vertex exactly once with a HIGHPT value. If (u, w) is the next frond to be processed, then each currently unlabeled vertex except w on the tree path from w to u has HIGHPT $= w$ and may be so labeled. Step b is:

```
b: comment calculate HIGHPT(v) for every vertex v in G1;
   begin
     for i := 1 until V1 do
       begin
         HIGHPT(i) := 0;
         set BUCKET(i) equal to the empty list;
       end;
l:   for each frond (u, w) in G1 do add (u, w) to BUCKET(w);
m:   for w := V1 step − 1 until 1 do
       while BUCKET(w) is not empty do
         begin
           let (u, w) ∈ BUCKET(W);
           delete (u, w) from BUCKET(W);
           n: for each vertex v ≠ w on the tree path from w to u satisfying
                HIGHPT(v) = 0 do HIGHPT(v) = w;
         end;
```

comment: after completion of Step m, all HIGHPT values will be correctly defined;
end;

Consider this calculation. Step l is a radix sort which orders the fronds on the NUMBER of their second vertex. For any vertex v, if there is a frond (u, w) with $w \neq v$ and $w \overset{*}{\to} v \overset{*}{\to} u$, then HIGHPT($v$) $\neq 0$ when Step m is finished; otherwise HIGHPT(v) $= 0$ when Step m is finished. If HIGHPT(v) $\neq 0$ when Step m is finished, then HIGHPT(v) is equal to the highest numbered vertex $w \neq v$ such that there is a frond (u, w) with $w \overset{*}{\to} v \overset{*}{\to} u$. It follows that Step m calculates HIGHPT values correctly.

For the algorithm to work efficiently, Step m must not reexamine vertices whose HIGHPT values have already been calculated. To take care of this problem, we use a fast method for computing unions of disjoint sets [11], [12], [13]. We shall have sets numbered 1 to V_1. If $v \neq 1$ is a vertex, then v will appear in the set whose number is the highest numbered unlabeled proper ancestor of v. Since vertex 1 never gets labeled, each vertex except 1 always appears in a set. Initially, if (v, w) is a tree arc, then w appears in the set named v.

To process frond (u, w), we find the set u_1 containing u, the set u_2 containing u_1, and so on, until we reach a set u_n such that $u_n \overset{*}{\to} w$. The vertices $u_1, u_2, \cdots, u_{n-1}$, and possibly u, are the unlabeled vertices on the tree path from w to u. We label them with HIGHPT value w, and then we compute the union of sets u_1, $u_2, \cdots, u_{n-1}, u_n$ (and possibly u) and number the union u_n. Step m becomes:

```
m: begin
     for i := 1 until V1 do SET(i) := the empty set;
     for each tree arc (v, w) do SET(v) := SET(v) ∪ {v}
     for w := V1 step − 1 until 1 do
       while BUCKET(w) is not empty do
         begin
           let (u, w) ∈ BUCKET(w);
           delete (u, w) from BUCKET(w);
           n: while ¬(u →* w) do begin
             x := FIND(u);
             if HIGHPT(u) = 0 then
               begin
                 SET(x) := SET(x) ∪ SET(u);
                 HIGHPT(u) := w;
               end;
             u := x;
           end;
         end;
end;
```

All the set unions in Step m are unions of disjoint sets. The operation FIND(x) computes the number of the set containing x as an element. Implementation and timing of the union and find operations are discussed in Appendix B. It is not hard to prove by induction on the number of vertices labeled that at all times

during execution of Step m, vertex w appears in set v if and only if v is the highest numbered unlabeled proper ancestor of w. It follows that Step m calculates HIGHPT values correctly.

LEMMA 21. *Step* b (*calculating* HIGHPT *values*) requires $O(V \log V + E)$ *time.*

Proof. Initialization requires $O(V)$ time. Step l requires $O(E)$ time. Step m requires $O(V + E)$ time for removing fronds from buckets. Step n requires $O(V + E)$ time exclusive of set unions and finds. There is one find for each frond plus at most one find for each vertex, giving $O(V + E)$ finds. There is one set union for each labeled vertex and one set union for each tree arc when the sets are initialized, giving in all $O(V)$ set unions. Using the implementation for finds and unions discussed in Appendix B, the set operations require $O(V \log V + E)$ time. Combining these facts gives the lemma.

The set unions and finds done in Step b actually require less time than the bound in Lemma 21 indicates, but Step b is not the slowest part of the dominators algorithm, and the bound in Lemma 21 is good enough.

6. Calculating semidominators. In this section, we implement Steps g and h, the conversion of cross-links to reverse fronds and the calculation of semidominators. A straightforward algorithm for these steps requires $O(V^2)$ time, but we can do better by using good data structures. First, consider the conversion of a cross-link to a reverse frond. Suppose we are processing vertex v, and we want to convert cross-link (u, v) to a reverse frond. For any $w > v$, either $\text{SDOM}(v, w) \overset{*}{\to} v$ or $\text{SDOM}(v, w) = \text{IDOM}(w)$, by Lemma 16. We can apply Lemma 15 to discover whether $\text{SDOM}(v, w) = \text{IDOM}(w)$; if $\text{SDOM}(v, w)$ is not a proper ancestor of v, we will know the value of $\text{IDOM}(w)$.

The semidominator calculations build the dominator tree from the leaves downward; at any given time, the part of the dominator tree which we know will consist of several vertices and all their descendants in the dominator tree. Let this set of subtrees be F. We shall use sets numbered 1 through V_1 (called ISET's) to contain information about F. If v is a vertex, v will be in the ISET whose number is the root of the subtree in F which contains v. If v is in no subtree in F, then v will be in $\text{ISET}(v)$. Initially each $\text{ISET}(v)$ contains exactly one element, v itself. To update the ISET's, each time we calculate $\text{IDOM}(v)$ for a new vertex, we let $\text{ISET}(\text{IDOM}(v)) = \text{ISET}(v) \cup \text{ISET}(\text{IDOM}(v))$. We can use the set union algorithm in Appendix B to keep track of the sets.

To convert cross-link (u, v) to a reverse frond, we find the set x which contains u. Then (u, v) may be converted to (x, v) by repeated cross-link replacement. (None of the elements of the set $x = \text{ISET}(u)$ can be an ancestor of v, since if $y \overset{*}{\to} v$, then $y < v$ and $y \in \text{ISET}(y)$ when v is being processed.) Either (x, v) is a reverse frond or $\text{SDOM}(v, x) \overset{*}{\to} v$, and applying one more cross-link replacement gives a reverse frond. This is the crux of our implementation of Step g; now we must see how to keep track of semidominator values.

For fixed i, we do not want to calculate $\text{SDOM}(i - 1, v)$ for all vertices v, since for most vertices $\text{SDOM}(i - 1, v) = \text{SDOM}(i, v)$. We only want to calculate semidominators which change when i changes. We use a set of *priority queues* to keep track of the semidominators. A priority queue contains a set of items, each with an attached numeric *priority*. We need to be able to add an item with any

priority to a queue and to remove the item with highest priority from a queue. (If two or more items have the same highest priority, we do not care which is removed first.) We also need to be able to combine two priority queues to give a large queue containing all items from both old queues.

Several good methods for implementing priority queues are known [14], [15]. They all use some sort of tree representation, and have a time bound of $O(n \log n)$ to perform n operations of the three types discussed above, starting with initially empty queues. We shall not discuss here how to implement priority queues; let us assume that we have some good implementation on hand.

To implement the semidominator calculations using priority queues, we set up a queue for each vertex v. Queue v will contain items, each of which is a set of descendants of v. (These sets we call QSET's.) All vertices w in a QSET will have the same value of SDOM(v, w), and this value will be the priority of the QSET in the queue. Only vertices whose IDOM values are not known are included in QSET's; thus if SDOM(v, w) is the priority of some QSET on the queue for v, SDOM$(v, w) \overset{*}{\to} v$.

To update the semidominators when processing vertex v, let (LOWPT(v), v) be the reverse frond (if any) entering vertex v. All sons of v have already been processed. First we construct a priority queue for v by combining the queues of the sons of v. Each QSET in the new queue has a priority corresponding to some ancestor of v. We remove each QSET having priority v (the highest possible priority). Each vertex in a removed QSET has IDOM value equal to v, by Lemma 15. We label these vertices with IDOM values and update the ISET's as described above. If v has no reverse frond entering it, we add to the queue a new QSET $\{v\}$ with priority equal to the father of v. If v has a reverse frond entering it, we remove each QSET with priority equal to or greater than LOWPT(v), we compute the union of all these QSET's, we add v as an element to this QSET, and we add the new QSET to the queue with priority LOWPT(v). This implements Lemma 19 for updating the semidominator calculations.

We handle the QSET unions using the set union algorithm described in Appendix B. Each vertex appears in at most one QSET which is on the priority queue of some vertex. For convenience, we assign each QSET a name consisting of its priority and some number distinguishing QSET's with the same priority. We now can finish the implementation of repeated cross-link replacement. Once a cross-link (u, v) has been converted into an edge (x, v) with IDOM(x) undefined, if (x, v) is not a reverse frond, then x must be in some QSET. Let y be the priority of this QSET in its queue. Then (y, v) is a reverse frond and may be substituted for (x, v).

Steps g and h are given below in ALGOL-like notation.

comment we need some initialization to set up the ISET's; **for** $i := 1$ **until** V_1 **do** ISET$(i) := \{i\}$;
g: **begin comment** convert (u, i) into a reverse frond by repeated cross-link replacement;
$x :=$ IFIND(u);
if $x \overset{*}{\to} i$ **then** replace (u, i) by (x, i);
else

```
        begin
          (y, v) := QFIND(x);
          replace (u, i) by (y, i);
        end;
```

We shall assume for convenience in Step h that the priority queue operations are implemented so that if we try to remove a set from an empty queue, we get an empty set called QSET(0, 0), and if we try to add QSET(0, 0) to a queue, nothing gets added to the queue.

```
h: begin comment calculate SDOM(i − 1, v) for all v such that SDOM(i − 1, v)
       ≠ SDOM(i, v). The semidominator of a vertex v is the priority of the
       QSET containing it, if v ≧ i and IDOM(v) is yet unknown;
     QUEUE(i) := the empty queue;
     for w a son of i do
       QUEUE(i) = QUEUE(i) ∪ QUEUE(w);
     remove QSET(z, j) with highest priority z from QUEUE(i);
     while z = i do
       begin
         for each element e ∈ QSET(z, j) do
           begin
             IDOM(e) := i;
             ISET(i) := ISET(e) ∪ ISET(i);
           end;
         remove QSET(z, j) with highest priority z from QUEUE(i);
       end;
     add QSET(z, j) to QUEUE(i);
     if LOWPT(i) ≧ i then
       begin comment (LOWPT(i), i) is the reverse frond (if any) entering
         vertex i;
         QSET (FATHER(i), i) := {i};
         add QSET (FATHER(i), i) to QUEUE(i);
       end
     else
       begin
         QSET (LOWPT(i), i) = {i};
         remove QSET(z, j) with highest priority z from QUEUE(i);
         while LOWPT(i) ≦ z do
           begin
             QSET(LOWPT(i), i) := QSET(z, j) ∪ QSET(LOWPT(i), i);
              remove QSET(z, j) with highest priority z from QUEUE(i);
            end;
          add QSET(z, j) to QUEUE(i);
          add QSET(LOWPT(i), i) to QUEUE(i);
        end;
end;
```

Step h is a straightforward implementation using the preceding ideas, and it is easy to prove the following hypothesis by induction on the number of times

that Step h is executed: if $v \in \text{ISET}(x)$, x is the highest numbered ancestor of v in the dominator tree such that IDOM(i) has not yet been calculated; if v is in QSET(z, j) and QSET(z, j) is in QUEUE(i), then SDOM(i, v) = z; if IDOM(v) $\neq 0$, then IDOM(v) has the correct value. It follows that Steps g and h correctly calculate dominators.

LEMMA 22. *If the dominators algorithm uses Steps* g *and* h *as implemented above, then the total running time of Steps* g *and* h *is* $O(V \log V + E)$.

Proof. Ignoring set and queue operations, the total running time of Steps g and h is $O(V + E)$. $O(V)$ unions of ISET's and $O(E)$ finds on ISET's will be carried out in Steps g and h. $O(V)$ unions of QSET's and $O(V)$ finds on QSET's will be carried out. The total cost of all the set operations is thus $O(V \log V + E)$ by the timing result in Appendix B. $O(V)$ unions of QUEUE's, $O(V)$ additions to QUEUE's, and $O(V)$ deletions from QUEUE's are carried out. By Crane's results [14], the priority queue operations may be carried out in $O(V \log V)$ time. Combining these results gives the lemma.

7. The complete dominators algorithm. This section contains the entire dominators algorithm in ALGOL-like notation. Several of the steps which have been discussed separately are combined; for instance, the initial depth-first search can be used to sort the fronds by the value of their second vertex and to begin the process of reverse frond deletion. The search can also be used to calculate the father of each vertex in the generated tree; this information is needed to initialize the semidominator calculations. The set and priority queue operations are not implemented here, but are assumed to be primitive operations. (The time required by these operations is included in the time bound for the entire algorithm, however.) Here is the dominators algorithm:

```
procedure DOMINATORS(G, s);
  begin comment we assume that the graph G is represented as a set of adjacency
      lists A(v);
    procedure SEARCH(v);
      begin comment this is the modified version of DFSEARCH used to
          initially explore the graph. It numbers the vertices, classifies the
          edges, deletes all but one reverse frond (LOWPT(v), v) entering each
          vertex v, sorts the fronds using a radix sort, and computes the number
          of descendants and the father of each vertex in the generated tree.
          Vertices not reached during the search have no dominators. Variable
          m denotes the last NUMBER assigned to any vertex. Variable n
          denotes the last SNUMBER assigned to any vertex;
        m := NUMBER(v) := m + 1;
        ND(v) := 1;
        for w ∈ A(v) do
          if NUMBER(w) = 0 then
            begin
              FATHER(w) := v;
              SEARCH(w);
              ND(v) := ND(v) + ND(w);
            end
```

```
      else if SNUMBER(w) = 0 then
        begin comment vertex w is stacked and (v, w) is a frond;
          1: add (v, w) to BUCKET(w);
        end
      else if NUMBER(v) < NUMBER(w) then
        begin (v, w) is a reverse frond;
          d: if NUMBER(v) < LOWPT(w) then LOWPT(w) := NUM-
               BER(v);
        end
      else add (v, w) to list of cross-links entering w;
    n := SNUMBER(v) := n - 1;
    comment v is now unstacked;
  end;
  integer m, n;
  a: comment to classify the edges we initialize and call SEARCH;
  m := 0;
  n := V + 1;
  for each vertex v do
    begin
      NUMBER(v) := SNUMBER(v) := 0;
      BUCKET(v) := the empty list;
      IDOM(v) := 0;
      LOWPT(v) := V;
    end;
  SEARCH(s);
  V1 := m;
  comment henceforth for convenience we assume that the program refers
    to each vertex by its number;
  modify all data structures so that vertices are named by their number;
  B: comment calculate HIGHPT(v) for every reachable vertex v;
  for i := 1 until V1 do
    begin
      HIGHPT(i) := 0;
      SET(i) := the empty set;
    end;
  for i := 2 until V1 do
    SET(FATHER(i)) := SET(FATHER(i)) ∪ SET(i);
  m: for w := V1 step - 1 until 1 do
    while BUCKET(w) is not empty do
      begin
        let (u, w) ∈ BUCKET(w);
        delete (u, w) from BUCKET(w);
        n: while ¬(u ⇢* w) do begin
          x := FIND(u);
          if HIGHPT(u) = 0 then
```

```
                begin
                  SET(x) := SET(x) ∪ SET(u);
                  HIGHPT(u) := w;
                end;
              u := x;
            end;
          end;
    c: comment calculate semidominators;
    for i := 1 until V1 do ISET(i) = {i};
    for i := V1 step − 1 until 1 do
      f: begin
        for each cross-link (u, i) do
          g: begin comment convert (u, i) to a reverse frond by repeated
               cross-link replacement;
            x := IFIND(u);
            if (x ⇢* i) then replace (u, i) by (x, i)
            else begin
              (x, v) := QFIND(x);
              replace (u, i) by (x, i);
            end;
            comment if two reverse fronds now enter i then delete one;
              if x ⇢* LOWPT(i) then LOWPT(i) := x;
          end;
        comment if a frond leaves i then delete it and add a reverse frond if
          necessary;
        if HIGHPT(v) < v and (LOWPT(v) < v) and
          (LOWPT(LOWPT(v)) > HIGHPT(v)) then
          LOWPT(LOWPT(v)) := HIGHPT(v);
        h: comment calculate SDOM(i − 1, v) for all v such that SDOM(i − 1,
          v) ≠ SDOM(i, v). The semidominator of a vertex v is the priority
          of the QSET containing it, if v ≧ i and IDOM(v) is not yet known;
        QUEUE(i) := the empty queue;
        for w a son of i do
          QUEUE(i) := QUEUE(i) ∪ QUEUE(w);
        remove QSET(z, j) with highest priority z from QUEUE(i);
        while z = i do
          begin
            for each element v ∈ QSET(z, j) do
              begin
                IDOM(v) := i;
                ISET(i) := ISET(i) ∪ ISET(v);
              end;
            remove QSET(z, j) with highest priority z from QUEUE(i);
          end;
        add QSET(z, j) to QUEUE(i);
        if LOWPT(i) ≧ i then
```

```
            begin
              QSET(FATHER(i), i) := {i};
              add QSET(FATHER(i), i) to QUEUE(i);
            end
          else
            begin
              QSET(LOWPT(i), i) := {i};
              remove QSET(z, j) with highest priority z from QUEUE(i);
              while LOWPT(i) ≦ z do
                begin
                  QSET(LOWPT(i), i) := QSET(LOWPT(i), i) QSET(z, j);
              remove QSET(z, j) with highest priority z from QUEUE(i);
                end;
              add QSET(z, j) to QUEUE(i);
              add QSET(LOWPT(i), i) to QUEUE(i);
            end;
        end;
    end;
```

This gives a complete algorithm for calculating dominators. Figure 5 shows the graph which results when all the dominator-preserving transformations are applied to the graph in Fig. 4. It is easy to verify that the algorithm has an $O(V + E)$ space bound. Combining the timing results in §§ 2, 4, 5 and 6, we see that the dominators algorithm has an $O(V \log V + E)$ time bound if the set and priority queue operations are implemented efficiently. The slowest parts of the algorithm are those which require priority queues; the set union operations run faster than the queue operations. If we could somehow handle the semidominator calculations using set unions (as we handled the HIGHPT calculations), then we could construct an even faster algorithm. The next section outlines how this may be done for a special case.

8. Toward a faster algorithm. The slow part of the dominators algorithm is the use of priority queues. If we could somehow use sets in place of priority queues (as we could for the HIGHPT calculations), then we could construct a faster dominators algorithm. This section outlines the construction of such an algorithm for the case when G has no cross-links.

Suppose G is a graph which has no cross-links when explored from vertex s using depth-first search. To find dominators in G, we classify the edges and number the vertices of G as before. Then we calculate HIGHPT values using the method described in § 4. Next, we apply reverse frond deletion and frond deletion repeatedly, until we have converted G into a graph with no fronds, no cross-links, and at most one reverse frond entering each vertex. Now we don't have to bother with calculating semidominators; we can calculate the dominators directly.

To calculate the dominators, we sort the remaining reverse fronds (u, v) of G so that if (u_2, v_2) follows (u_1, v_1), then $u_2 > u_1$ or $u_2 = u_1$ and $v_2 < v_1$. This may

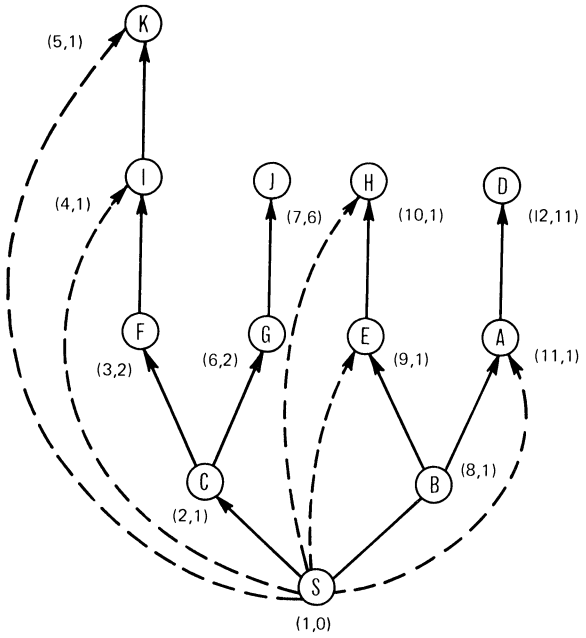


FIG. 5. *The graph in Fig.* 4 *after all dominator-preserving transformations have been applied. The resultant graph contains only tree arcs and reverse fronds. Vertex numbers and immediate dominators are in parentheses.*

be done using a two-pass radix sort with V buckets, similar to the sorting for the calculation of HIGHPT values. We then process the reverse fronds in sorted order, using the following lemma.

LEMMA 23. *Let G be a graph such that each vertex is reachable from s, each vertex has at most one reverse frond entering it, and G contains no cross-links or fronds. Suppose $v \neq 1$. If v has no reverse frond entering it,* IDOM(v) *is the father of v in the spanning tree of G. If v has a reverse frond (u, v) entering it and no reverse frond (x, w) satisfies $x \overset{*}{\to} u \overset{*}{\to} w \overset{*}{\to} v$ and $x < u < w < v$, then* IDOM(v) $= u$. *Otherwise let (x, w) be the reverse frond with smallest x (largest w) satisfying $x \overset{*}{\to} u \overset{*}{\to} w \overset{*}{\to} v$ and $x < u < w < v$. Then* IDOM(v) $=$ IDOM(w).

Proof. If v has no reverse frond entering it, every path to v must pass through the father of v, and IDOM(v) is the father of v. Suppose (u, v) enters v but no reverse frond (x, w) satisfies $x \overset{*}{\to} u \overset{*}{\to} w \overset{*}{\to} v$ and $x < u < w < v$. Suppose path p leads from s to v but doesn't contain u. Let (x, w) be the first edge on p with $w > u$. (x, w) must be a reverse frond with $x \overset{*}{\to} u \overset{*}{\to} w \overset{*}{\to} v$ and $x < u < w < v$. But this is a contradiction, so u dominates v. Since (u, v) is an edge, IDOM(v) $= u$.

If some reverse frond (x, w) satisfies $x \overset{*}{\to} u \overset{*}{\to} w \overset{*}{\to} v$ and $x < u < w < v$, let (x, w) be one with smallest x (largest w). It is clear that any vertex which does not dominate w cannot dominate v. Thus every vertex which dominates v must also dominate w. Now we need only show that IDOM(w) dominates v.

Suppose, to the contrary, that p is a path from s to v which doesn't contain IDOM(w). Let (i, j) be the first edge on this path satisfying $j \geqq x (j \geqq u)$. Then IDOM(w) cannot dominate w because $j \leqq u$, $x \overset{*}{\to} j \overset{*}{\to} u (j \leqq w, u \overset{*}{\to} j \overset{*}{\to} w)$, and

we may form a path from s to w which doesn't contain IDOM(w). This contradiction gives the rest of the lemma.

To calculate dominators, we start with one set numbered 0 containing all the vertices. Then we process the reverse fronds in the order described above. At any given time, a vertex v will be in a set labeled x if x is the largest vertex such that $x \overset{*}{\rightarrow} v$ and a reverse frond (u, x) has been processed. To process a reverse frond (u, v), we locate u and v in sets. If they are in the same set, IDOM(v) = u by Lemma 23. If they are in different sets, IDOM(v) = IDOM(FIND(v)) by Lemma 23. It happens that in this case IDOM(FIND(v)) will already have been computed, but even if this were not true we could fill in the value of IDOM(v) later, once we knew the value of IDOM(FIND(v)). In any case, we split the set containing v into two parts; a set containing descendants of v, having label v, and a set containing nondescendants of v, having the same label as the old set containing v. This algorithm computes dominators, if we can implement the set-splitting operation.

Actually, we don't have to split sets; we can run this algorithm backwards, and turn the splits into union operations. Then we can use the algorithm described in Appendix B. The correct labels for all the resultant sets and the IDOM values must be filled in after the set union operations are carried out, but this is no great problem. The resultant dominators algorithm has a time bound of $O(V + E)$ not counting set unions and finds. There are $O(V)$ unions in the HIGHPT and dominator calculations and $O(V + E)$ finds. If $E \geqq V \log V$, we get the same overall bound as the algorithm in § 7: $O(E)$. If E is substantially smaller than $V \log V$, we get a better bound than that for the algorithm in § 7, namely, $O((V + E) \cdot \log^*(V + E))$, where $\log^* x = \min \{i | \log^i(x) \leqq 1\}$ (see [12], [13].) It seems possible that this faster algorithm may be generalized to handle arbitrary graphs.

9. Conclusions. This paper has presented an algorithm for finding dominators in directed graphs. The algorithm illustrates the use of depth-first search for revealing the connectivity structure of a graph and the use of sophisticated data structures in building efficient graph algorithms. The algorithm requires $O(V + E)$ space and $O(V \log V + E)$ time to find dominators in a graph with V vertices and E edges. The time bound compares favorably with the $O(V(V + E))$ time bound of previously known algorithms such as Aho and Ullman's [1] and Purdom and Moore's [3] for finding dominators in arbitrary graphs, and with the $O(E \log E)$ time bound of Aho, Hopcroft and Ullman's algorithm for finding dominators in reducible graphs [6]. If $E \geqq V \log V$, then the time bound is $O(E)$, and the new algorithm is optimal to within a constant factor, since every edge must be examined to determine dominators. Although the algorithm is based on some delicate graphical transformations, it is easy to program.

Still open is the question of whether a faster algorithm exists if $E < V \log V$. Section 8 gives a faster algorithm for graphs which have no cross-links when they are explored using depth-first search. Many, but not all program flow graphs have this special form; in particular, the **if** $\cdots$ **then** $\cdots$ **else** construction produces a cross-link in the resultant flow graph. By adding vertices, any program flow graph may be converted into a computationally equivalent program flow graph which has no cross-links. However, the number of vertices in the graph may

grow enormously. The slower but more general algorithm thus seems more useful than the faster algorithm. However, it may be that the algorithm in § 8 can be extended to reducible graphs or even to arbitrary graphs.

Appendix A. Basic definitions. A *directed graph* $G = (\mathscr{V}, \mathscr{E})$ is an ordered pair consisting of a set of *vertices* $\mathscr{V}$ and a set of *edges* $\mathscr{E}$. Each edge is an ordered pair (v, w) of distinct vertices. We say edge (v, w) *leaves* v and *enters* w. A graph contains no loops (edges of the form (v, v)) and no multiple edges, although the algorithms presented in this paper may be easily modified to handle graphs with loops and multiple edges. A graph $G_1 = (\mathscr{V}_1, \mathscr{E}_1)$ is a *subgraph* of a graph $G_2 = (\mathscr{V}_2, \mathscr{E}_2)$ if $\mathscr{V}_1 \subseteq \mathscr{V}_2$ and $\mathscr{E}_1 \subseteq \mathscr{E}_2$.

If G is a graph, a *path* $p: v \overset{*}{\Rightarrow} w$ is a sequence of vertices and edges leading from vertex v to vertex w. A vertex w is *reachable* from vertex v if there is a path from v to w. A path is *simple* if all its edges are distinct. A path $p: v \overset{*}{\Rightarrow} v$ is called a *closed* path. A closed path may contain no edges. A closed path $p: v \overset{*}{\Rightarrow} v$ is a *cycle* if all its edges are distinct and the only vertex to occur twice is v, which occurs exactly twice. A cycle contains at least two edges. Two cycles which are cyclic permutations of each other are considered to be the same cycle. A directed graph is *acyclic* if it contains no cycles.

A (directed, rooted) *tree* T is a graph with one distinguished vertex called the *root* r such that every vertex in T is reachable from r, no edges enter r, and exactly one edge enters every other vertex in T. A tree vertex with no exiting edges is called a *leaf*. The relation "(v, w) is an edge in T" is denoted by $v \rightarrow w$. The relation "there is a path from v to w in T" is denoted by $v \overset{*}{\rightarrow} w$. If $v \rightarrow w$, v is the *father* of w and w is a *son* of v. If $v \overset{*}{\rightarrow} w$, v is an *ancestor* of w and w is a *descendant* of v. Every vertex is an ancestor and a descendant of itself. If $v \overset{*}{\rightarrow} w$ and $v \neq w$, v is a *proper ancestor* of w and w is a *proper descendant* of v. If T_1 is a tree and is a subgraph of tree T_2, then T_1 is a *subtree* of T_2. If T is a tree which is a subgraph of a directed graph G and T contains all the vertices of G, then T is a *spanning tree* of G. References on directed graphs include Busacker and Saaty [18], Harary, Norman and Cartwright [19], and Ore [20].

If f and g are functions of x, we say $g(x)$ is $O(f(x))$ if there are constants k_1 and k_2 such that $|g(x)| \leqq k_1|f(x)| + k_2$ for all x.

Appendix B. A good set union algorithm. Suppose we are given a collection of disjoint sets. We want to carry out operations of two types on the sets: FIND(x), which computes the name of the set containing x as an element, and UNION(A, B, C), which computes the union of sets A and B and names the new set C. Initially we have n distinct elements, each in a singleton set. We then carry out $n - 1$ unions and m intermixed finds. We desire a good method for implementing these operations.

A very simple algorithm will solve the problem. Each set is represented as a tree. Each tree vertex represents an element in the set, and the root of the tree represents the entire set as well as some element in the set. Each tree vertex is represented in a computer by a cell containing either two or three items. A cell representing a nonroot vertex contains the element corresponding to the vertex and a pointer to the cell for the father of the vertex in the tree. A cell corresponding

to a root contains the element corresponding to the root, the name of the set corresponding to the tree with that root and the number of vertices in the set.

To carry out FIND(x), we locate the cell containing element x and follow pointers to the cell for the root of the corresponding tree. This cell contains the name of the set. In addition we *collapse* the tree by changing the father of each vertex reached on the way to the root. The root itself becomes the father of each of these vertices. This collapsing process saves time in later finds. To carry out UNION(A, B, C), we choose the set with fewer elements, say A. Then we make the root of A a son of the root of B. The cell corresponding to the root of A becomes a nonroot cell pointing to the cell for the root of B. The root cell of B is changed to contain the name C and the sum of the number of elements in A and B.

Although this set union algorithm is very simple, it is very hard to analyze [11], [12], [13]. Hopcroft and Ullman [12] have studied the algorithm and shown that its running time is $O((n + m) \log^* (n + m))$, where $\log^* x = \min \{i | \log^i x \leqq 1\}$. Tarjan [13] has derived the same upper bound on the running time using a different method and has also shown that the algorithm does not have a linear upper bound on its running time. The exact running time of the algorithm is still unknown. However, for our purposes, a loose upper bound is all that we need. The bound below is generally known but apparently unpublished.

It is useful to think about the set union algorithm in the following way: suppose we perform all $n - 1$ unions first. Then we have a single tree with n vertices. Each of the original finds is now a "partial" find in the new tree: to carry out FIND(x), we follow fathers from x to the closest ancestor of x corresponding to a union which appears before FIND(x) in the original sequence of operations. In this interpretation of the problem, we are interested in bounding the total length of m partial finds performed on a tree generated by $n - 1$ set unions. (The total time required for the set unions is $O(n)$; the time for a find is proportional to its length.)

Let T be a tree containing n vertices numbered 1 through n which has been constructed using $n - 1$ set unions. Let d_i be the number of descendants of vertex i. Let $C(T)$, the *cost* of tree T, be defined by

$$C(T) = \sum_{i=1}^{n} d_i.$$

Let $\bar{C}(n)$ be the maximum cost of a tree with n vertices constructed by applying set unions. Then we have the following.

LEMMA 28. $\bar{C}(n) \leqq n \log 2n$.

Proof. We prove the lemma by induction on n. $\bar{C}(1) = 1 = 1 \cdot \log 2$. Suppose the lemma is true for $n < k$. Let $n = k$. Let T be a tree such that $C(T) = \bar{C}(n)$ and T is formed by taking the union of trees T_1 with a vertices and T_2 with b vertices, $a \leqq b$, $a + b = n$. Then:

$$\begin{aligned}
\bar{C}(n) = C(T) &= C(T_1) + C(T_2) + n - b \\
&\leqq a \log 2a + b \log 2b + a \\
&\leqq a(\log 2n - 1) + b(\log 2n) + a \\
&\leqq n \log 2n.
\end{aligned}$$

The lemma follows by induction on n.

Now suppose we apply a partial find of length k to a tree T. Assume without loss of generality that the find starts at vertex 1 and causes vertices $1, 2, \cdots, k-1$ to become sons of vertex k. Let T' be the tree after this find is performed, and let d'_i be the number of descendants of vertex i in T'. Then $d'_1 = d_1, d'_i = d_i - d_{i-1}$ for $2 \leqq i \leqq k-1$, and $d'_k = d_k$. Since $d_i \geqq d_{i-1} + 1 \geqq i \geqq 1$, it follows that $C(T') \leqq C(T) - k - 2$, and we have the following result.

LEMMA 29. *If m partial finds are performed on a tree with n vertices formed with $n-1$ unions, the total length of all the finds is $O(n \log n + m)$.*

Proof. Let $k_i, i = 1, \cdots, m$, be the length of the ith find. Since every tree has positive cost and any find of length k decreases the cost of the corresponding tree by at least $k-2$, we have

$$n \log 2n - \sum_{i=1}^{m} (k_i - 2) > 0.$$

It follows that

$$\sum_{i=1}^{m} k_i < 2m + n \log 2n.$$

Lemma 29 implies that $n-1$ unions and m finds require $O(n \log n + m)$ time.

REFERENCES

[1] A. V. AHO AND J. D. ULLMAN, *The Theory of Parsing, Translation, and Compiling, Vol. II: Compiling*, Prentice-Hall, Englewood Cliffs, N.J., 1972.

[2] E. S. LORRY AND C. W. MEDOCK, *Object code optimization*, Comm. ACM, 12 (1969), pp. 13–22.

[3] P. W. PURDOM AND E. F. MOORE, *Algorithm 430: Immediate predominators in a directed graph*, Ibid., 15 (1972), pp. 777–778.

[4] M. SHAEFER, *A mathematical theory of global flow analysis*, unpublished notes, System Development Corp., Santa Monica, Calif., 1971.

[5] J. R. BELL AND N. E. ABEL, *An algorithm for finding immediate predominators in optimizing compilers*, unpublished, Digital Equipment Corp., Maynard, Mass., January, 1972.

[6] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *On finding the least common ancestors in trees*, submitted to the 1973 ACM Symposium on Theory of Computing (Austin, Texas, 1973).

[7] M. S. HECHT AND J. D. ULLMAN, *Flow graph reducibility*, this Journal, 1 (1972), pp. 188–202.

[8] J. E. HOPCROFT AND J. D. ULLMAN, *An n log n algorithm for detecting reducible graphs*, Proc. 6th Annual Princeton Conference of Information Sciences and Systems, 1972, pp. 119–122.

[9] F. E. ALLEN, *Control Flow Analysis*, SIGPLAN Notices, vol. 5, no. 7, July 1970, pp. 1–19.

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

[11] M. J. FISCHER, *Efficiency of equivalence algorithms*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 153–168.

[12] J. E. HOPCROFT AND J. D. ULLMAN, *Set-merging algorithms*, this Journal, 2 (1973), pp. 294–303.

[13] R. TARJAN, *On the efficiency of a good but not linear set union algorithm*, Tech. Rep. 72–148, Computer Science Dept., Cornell Univ., Ithaca, N.Y., 1972.

[14] C. R. CRANE, *Linear lists and priority queues as balanced binary trees*, STAN-CS-72-259, Computer Science Dept., Stanford Univ., Stanford, Calif., 1972.

[15] J. E. HOPCROFT, private communication.

[16] D. KNUTH, *The Art of Computer Programming*, vol. 1, Addison-Wesley, Reading, Mass., pp. 258–265.
[17] R. TARJAN, *An $O(V^2)$ algorithm for finding dominators in directed graphs*, unpublished manuscript, Cornell Univ., Ithaca, N.Y., 1972.
[18] R. G. BUSACKER AND T. L. SAATY, *Finite Graphs and Networks: An Introduction with Applications*, McGraw-Hill, New York, 1965.
[19] F. HARARY, R. Z. NORMAN AND D. CARTWRIGHT, *Structural Models*, John Wiley, New York, 1965.
[20] O. ORE, *Theory of Graphs*, American Mathematical Society, Providence, R.I., 1962.
[21] C. P. EARNEST, K. G. BALKE AND J. ANDERSON, *Analysis of graphs by ordering of nodes*, J. Assoc. Comput. Mach., 19 (1972), pp. 23–42.

SIAM J. COMPUT.
Vol. 3, No. 1, March 1974

FIRST ORDER GRAPH GRAMMARS*

CURTIS R. COOK†

Abstract. In this paper we consider first order context-free, linear, and regular graph grammars and obtain many results similar to those for the corresponding string grammars. We obtain normal forms for context-free and regular graph grammars, simplification lemmas, and algorithms for membership, emptiness, finiteness and infiniteness. We show the relation between regular graph languages and regular sets and show that the set of graphs resembling the nonregular set $\{a^n b^n | n \geqq 1\}$ is not generated by a first order context-free graph grammar. We also give several graphical characterizations of each of the three types of graph grammars.

Key words. graphs, graph grammars, block, cutpoint, cutpoint graphs

Introduction. Graph grammars have been the subject of much recent research because of their syntactic pattern recognition and picture processing applications. The main difference between graph and string grammars is that a symbol may appear in a string with a symbol to its left or right, while a symbol in a graph may be joined to an arbitrary number of other symbols. Some of the different models of graph grammars are web grammars [1], [5], [7], plex grammars [2], graph grammars [8], and mth order context-free graph grammars [6].

The main differences between the various models are: (i) the mth order context-free graph grammar rewriting rules do not include a specification of the embedding of the rewritten graph in the host graph; (ii) variables in the mth order context-free and plex grammars are allowed to be graph structures; in the other models variables are either points or lines; (iii) web grammar terminals are points; in the other models terminals are points, lines, and higher order graph structures.

In this paper we investigate first order context-free, linear, and regular graph grammars. The variables and terminals in our model are points, and the productions do not include an embedding specification. Our model is a special case of the first order case of mth order context-free graph grammars and turns out to be identical to the normal context-free web grammars as defined in [1]. It is interesting to note that in all of the other graph grammar models there is no counterpart to regular string grammars.

We give several graphical characterizations of each of the three types of grammars. We obtain many results similar to those for string grammars: normal forms for context-free and regular graph grammars, simplification lemmas, and algorithms for membership, emptiness, finiteness and infiniteness. The proofs are almost identical to the string grammar proofs. We show the relation between regular graph languages and regular sets. We also show that a set of graphs resembling the nonregular set $\{a^n b^n | n \geqq 1\}$ is not generated by a first order context-free graph grammar.

1. Definitions and examples. It is assumed that the reader is familiar with the standard notation and results of formal language and automata theory [4].

* Received by the editors May 21, 1973.

† Computer Science Department, Oregon State University, Corvallis, Oregon 97331.

In this paper all graphs are unordered, connected, and do not contain multiple lines or loops.

DEFINITION. A *first order context free graph grammar* (1-CFGG) $G = (N, T, P, S)$, where

N is a finite set of variables (point variables),

T is a finite set of terminals (T is a single element for unlabeled graphs),

S is in N, and

P is a finite set of productions or rewriting rules of the form $\overset{A}{\bullet} \to \alpha$, where A is in N, α is a graph whose points are labeled from $N \cup T$, exactly one point in α is the image of A, and α is connected to the rest of the graph through the image of A.

Let $\Rightarrow$, $\overset{*}{\Rightarrow}$, and $L(G)$ be defined in the usual way. Thus $L(G)$ is a set of undirected graphs whose points are labeled with symbols from T.

Pavlidis [6] defined the mth order context-free and linear graph grammars and gave several topological characterizations of second order context-free and linear graph grammars. His variables are mth order structures, subgraphs which are connected to the rest of the graph by m points. The variables in our grammars are the special case of his first order or node structures where the node structure is just a point. His terminals are lines and points, while ours are simply points.

Our definition turns out to be identical to the normal context-free web grammars as defined in [1]. In the embedding specification in [1], points adjacent to the rewritten point are adjacent to the image of the point. A web grammar is *normal* if, in each production, the point rewritten has exactly one image in the right side. The main results in [1] concerned the hierarchy of classes of webs generated by normal and nonnormal grammars.

DEFINITION. A *first order linear graph grammar* (1-LGG) is a 1-CFGG in which the right side of each production contains at most one variable.

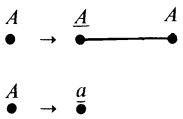
Before we define first order regular graph grammars, we need several graph theory definitions [3]. The *trivial graph* is the graph consisting of a single point. A *cutpoint* of a graph is a point whose removal (includes removing lines incident with point) disconnects the graph. A graph is separable if it contains a cutpoint. A *nonseparable graph* is nontrivial and contains no cutpoint. A *block* of a graph is a maximal nonseparable subgraph.

DEFINITION. A *first order regular graph grammar* (1-RGG) is a 1-LGG in which the right side of each production is either a terminal point or a block containing at most one variable.

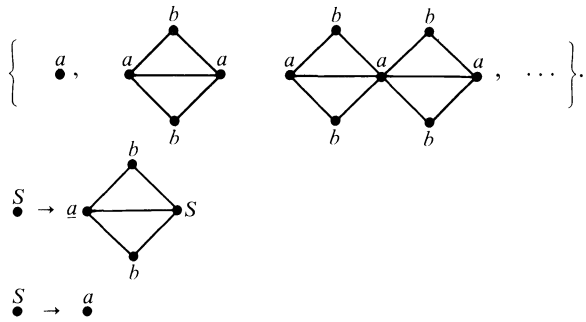
Let 1-CFGL, 1-LGL and 1-RGL denote the set of graphs generated by 1-CFGG, 1-LGG and 1-RGG, respectively.

The following examples illustrate the definitions. The underlined symbol on the right side of a production is the image of the variable on the left. Upper case letters denote variables and lower case denote terminals.

Example 1. 1-CFGG *which generates all trees.*



Example 2.

1-RGG *which generates* 

2. First order context-free graph grammars. In this section we derive many results similar to those for context-free string grammars. We describe several simplifications and obtain a normal form very similar to the Chomsky normal form. We show that there exist algorithms for determining whether the language generated by a 1-CFGG is empty, finite, or infinite. We give several graphical characterizations of 1-CFGL's. Finally, we show that a set of graphs resembling the nonregular set $\{a^n b^n | n \geqq 1\}$ is not generated by a 1-CFGG.

First we give several simplification lemmas whose proofs are identical to the string grammar proofs [4]. Two 1-CFGG's are *equivalent* if they generate the same set of graphs.

LEMMA 1. *Given a* 1-CFGG G, *we can find an equivalent* 1-CFGG G' *in which the start symbol of* G' *does not appear on the right side of a production.*

LEMMA 2. *Given a* 1-CFGG G, *we can find an equivalent* 1-CFGG *with no productions of the form* $\overset{A}{\bullet} \to \overset{B}{\bullet}$, *where* A *and* B *are both variables.*

A variable in G is *useless* if it never appears in the derivation of some graph in $L(G)$.

LEMMA 3. *Given a* 1-CFGG G, *we can find an equivalent* 1-CFGG G' *with no useless variables.*

We include the proof of the next lemma because it yields as a corollary an algorithm for determining whether the language is empty.

LEMMA 4. *Given a* 1-CFGG G, *we can find an equivalent* 1-CFGG G' *such that each variable in* G' *generates a terminal graph.*

Proof. Let $G = (N, T, P, S)$. Let $W_1 = \{A \in N | \overset{A}{\bullet} \to \alpha \text{ in } P \text{ and } \alpha \text{ is a terminal graph}\}$, and for $j \geqq 1$, let $W_{j+1} = W_j \cup \{A \in N | \overset{A}{\bullet} \to \alpha \text{ in } P \text{ and } \alpha \text{ is a graph whose points are labeled from } T \cup W_j\}$.

Construct the grammar $G' = (N', T, P', S)$, where $N' = W_k$, $|N| = k$ and P' consists of those productions $\overset{A}{\bullet} \to \alpha$ in P for which $A \in N'$ and the points of α are labeled from $T \cup N'$. Then G' contains no useless variables, and clearly $L(G) = L(G')$.

It follows immediately from the construction of the W_i's that $L(G)$ is empty if and only if $S \notin N'$.

LEMMA 5. *There is an algorithm for determining if the language generated by a* 1-CFGG *is empty.*

From now on, we assume that all 1-CFGG's are simplified, i.e., satisfy Lemmas 1–4. The graph theory concept of a block plays an important role in the normal form for 1-CFGG's.

THEOREM 1. *Every* 1-CFGL L *can be generated by a grammar in which all productions are of the form* $\overset{A}{\bullet} \to \alpha$, *where* α *is either a terminal point or a block.*

Proof. Let $G = (N, T, P, S)$ be a 1-CFGG such that $L(G) = L$. If a production has a single point or a block on the right side, then it is already in an acceptable form.

Now consider a production $A \to \alpha$, where α is a separable graph with cutpoints $v_1, \cdots, v_i$ and blocks $B_1, \cdots, B_k$. For each cutpoint v_j, let C_j denote the set of all blocks containing v_j. Define the set $D = \{D_{mn} | D_{mn}$ is a nonempty subset of C_n for some $n\}$. Replace $A \to \alpha$ with the set of productions:

1. $A \to \beta$, where β denotes the block B_1 with each cutpoint v_p of B_1 replaced by the variable $C_p - \{B_1\} \in D$.

2. For every $D_{mn} \in D$ and $B_p \in D_{mn}$, $D_{mn} \to \tau_p$, where τ_p denotes the block B_p with each cutpoint v_q, $q \neq n$, replaced by the variable $C_q - \{B_p\}$, and the image of D_{mn} is the variable $D_{mn} - \{B_p\}$ if $D_{mn} - \{B_p\} \neq \varnothing$ and is v_n otherwise.

The variable D_{pq} indicates which of the blocks containing cutpoint v_q have not been generated. This construction guarantees that all of the blocks of α will be generated. The first production $A \to \beta$ initiates the generation with all cutpoints of B_1 replaced by variables indicating block B_1 has been generated. From then on, only productions of type 2 are applied with the cutpoints replaced by variables which indicate the blocks containing that cutpoint which have yet to be generated. A cutpoint variable rewrites as a v_j only after all blocks containing it have been generated.

Let N' and P' denote the new set of variables and productions, respectively. Then $G' = (N', T, P', S)$ is of the proper form. It should be clear from the construction that $L(G) = L(G')$.

From Theorem 1, we obtain a graph grammar version of the Chomsky normal form. For each $\overset{A}{\bullet} \to \alpha$, where α is a block, replace each terminal in α by a new variable which appears on the right side of no other production. Then create a new production in which the new variable rewrites as the terminal it replaced.

THEOREM 2 (Normal form). *Any* 1-CFGL *can be generated by a* 1-CFGG *in which all productions are of the form* $\overset{A}{\bullet} \to \alpha$, *where* α *is either a terminal point or a block in which every point is a variable.*

Two graphical characterizations follow immediately from the normal form theorem (k is the maximum number of points in the right side of a production).

THEOREM 3. *Let* L *be a* 1-CFGL. *Then there exists a number* k *such that no graph in* L *contains a block with more than* k *points.*

Even though Theorem 3 only gives a necessary condition for the characterization of 1-CFGL's, it does provide a simple test for assuring that a set of graphs is not generated by a 1-CFGG.

COROLLARY 3.1. *The following graphs cannot be generated by a* 1-CFGG:

(a) *all nonseparable graphs*;

(b) *all complete graphs* (a complete graph has every pair of its points adjacent);

(c) *all bigraphs* (a bigraph is a graph whose point set can be partitioned into two sets such that every line joins a point in one set with a point in the other set);

(d) *all cycles* C_n *for* $n \geqq 3$.

Pavlidis [6] characterized 2-CFGL by a k-reduction process which is the

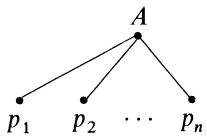
graph counterpart of string grammar parsing. For 1-CFGL's define the k-reduction process as follows.

Step 1. Replace by a single point any subgraphs with no more than k points which are connected to the rest of the graph by exactly one point.

Step 2. Repeat Step 1 until it is no longer applicable. If the reduced graph has exactly one point, then the original graph is *k-reducible*; otherwise it is *k-irreducible.*

THEOREM 4. *If L is a* 1-CFGL, *then there exists a number k such that every member of L is k-reducible.*

There are derivation trees for 1-CFGG's. If $\overset{A}{\bullet} \to \alpha$ is a production and α contains the points $p_1, \cdots, p_n$, then the corresponding nodes in the derivation tree are



Note that the points are unordered, so there is no unique derivation tree for a derivation in a 1-CFGG. The derivation trees are used to show that there exist algorithms which determine, for any graph H, whether H is generated by the 1-CFGG G and which determine whether a 1-CFGL is finite or infinite. The proofs are almost identical to the corresponding string grammar proofs [4].

THEOREM 5. *Let G be a* 1-CFGG. *Given a terminal graph H, there is an algorithm for determining whether H is in $L(G)$.*

Proof. Let $G = (N, T, P, S)$ be a normal form 1-CFGG and let H be a terminal graph with n points. Since G is in normal form, the right side of each production is either a block or a single terminal point. Then at most n productions whose right sides are single terminal points and at most $n - 1$ productions whose right sides are blocks can be applied in the derivation. Hence the length of the derivation of H is less than $2n$.

Thus the algorithm consists of generating all derivations in G of length less than $2n$ and comparing each graph with H.

THEOREM 6. *Let G be a normal form* 1-CFGG *with k variables and with at most n points on the right side of a production. Then $L(G)$ is infinite if it contains a graph with more than n^{k-1} points.*

Proof. Let $p = n^{k-1}$. It is easy to see that if a derivation tree has no path of length greater than j, then the terminal graph derived contains no more than n^{j-1} points.

Hence if H is in $L(G)$ and H contains more than p points, then the derivation tree for H must contain a path of length greater than k. Choosing a path P of longest length, we observe that there must be two nodes n_1 and n_2 satisfying the following conditions:

1. Nodes n_1 and n_2 have the same label, say A.
2. Node n_1 is closer to the root than node n_2.
3. The portion of the path P from n_1 to the leaf is of length at most $k + 1$.

Let α be the subgraph of H generated from node n_1 and β be the subgraph generated from node n_2. Then β must be a proper subgraph of α because G is in normal form. If we replace node n_2 with a copy of the subtree rooted at n_1, we obtain the derivation tree of a terminal graph H' in $L(G)$, and H' properly contains H. Repeating this process, we can obtain an infinite number of terminal graphs, each of which is in $L(G)$.

THEOREM 7. *There is an algorithm for determining whether a given* 1-CFGG *generates a finite or infinite number of graphs.*

Proof. Let G, n, k and p be defined as in Theorem 6. Let $q = n^{k+1}$. Suppose that $L(G)$ is infinite. Then there is a graph H in $L(G)$ with more than $p + q$ points, and the derivation tree for H must contain a path of length greater than k. Choosing a path P of longest length, we observe that there must be two nodes n_1 and n_2 satisfying the following conditions:

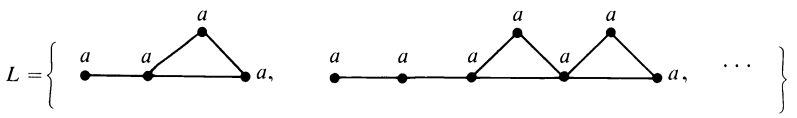
1. Nodes n_1 and n_2 have the same label, say A.
2. Node n_1 is closer to the root than node n_2.
3. The portion of the path P from n_1 to the leaf is of length at most $k + 1$.
4. The subtree rooted at n_1 contains at most q terminal nodes (leaves).

The subtree rooted at n_2 is a proper subtree of the subtree rooted at n_1. Replacing the subtree rooted at n_1 with the subtree rooted at n_2, we obtain a smaller derivation tree for a graph H' in $L(G)$. Clearly H' contains more than p points, and if H' contains more than $p + q$ points, we repeat the above reduction process until we obtain a graph H'' in $L(G)$ with m points, $p < m \leqq p + q$. Thus L is infinite if and only if it contains a graph with m points, $p < m \leqq p + q$.

The algorithm consists of testing all graphs with between p and $p + q$ points for membership in $L(G)$. If there is such a graph, then $L(G)$ is infinite; otherwise, there is no graph with more than p points in $L(G)$, and $L(G)$ is finite.

The last theorem in this section shows that a set of graphs resembling the nonregular set $\{a^n b^n | n \geqq 1\}$ is not generated by a 1-CFGG.

THEOREM 8. *The set of graphs*

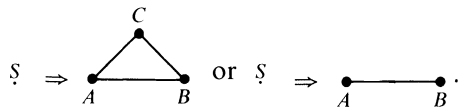


is not a 1-CFGL.

Proof. Suppose L is generated by the normal form 1-CFGG $G = (N, T, P, S)$.

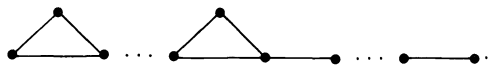
Let $|N| = k$. Consider the derivation tree for the graph H in L with $2k + 1$ lines and triangles. We will show that if H is generated by G, then so are several graphs not in L.

The first step in the derivation of H is either



Suppose S rewrites as a triangle. Then one of the variables, say C, must rewrite as the terminal a. Let n_A and n_B denote the nodes labeled A and B, respectively, in the derivation tree. Then either n_A or n_B must be the root of a derivation

tree with a path P of length greater than k. Since each cutpoint of H lies in exactly two blocks, the subgraph derived from the subtree rooted $n_A(n_B)$ must be of the form



If the subgraph consisted entirely of lines or triangles, two nodes on P have the same label, and if we replaced the node closer to $n_A(n_B)$ with the other node we have the derivation tree of a graph with too few triangles or lines to be in L.

A subgraph of this form must be derived from the subtree rooted at either n_A or n_B and not both. If the subtree is rooted at n_A, then the subgraph contains the $2k + 1$ lines of H. But this leads to a contradiction, as two nodes in the subtree have the same label, and by the same argument as above we can obtain the derivation tree of a graph not in L. Similarly a contradiction is reached if the subtree is rooted at n_B.

The same argument holds if "S rewrites as a line" is the first step in the derivation of H.

Therefore L is not a 1-CFGL.

3. First order linear graph grammars. The first lemma will be used in § 4 to show that the 1-LGL's properly contain the 1-RGL's.

LEMMA 6. *Every finite set of graphs is a* 1-LGL.

Next we give a first order version of a topological characterization which appeared in [6].

THEOREM 9. *If L is a* 1-LGL, *then there is a number k such that any cutpoint of a graph in L separates the graph into components such that no more than two of them have more than k points.*

Proof. Let k be the maximum of points in the right side of a production. The result follows from the following observation about the cutpoints of a graph in L. There are two types of cutpoints in a terminal graph in L. The first type is a terminal cutpoint in the right side of a production. This type of cutpoint separates the graph into several components, only one of which contains more than k points. The second type is a point which was a variable at some time in the derivation. In this case, since the right side of each production contains at most one variable, only one component with more than k points can be generated from the point. Hence, including the portion of the graph generated prior to the appearance of this point, this point separates the graph into at most two components with more than k points.

COROLLARY 9.1. *The set of all trees* (*Example* 1) *is a* 1-CFGL, *but is not a* 1-LGL.

4. First order regular graph grammars. In this section we describe a normal form for 1-RGG's, give a simple graphical characterization of 1-RGL's, and show the relation between 1-RGL's, cutpoint graphs, and regular sets.

THEOREM 10 (Normal form). *Every* 1-RGL *can be generated by a* 1-RGG *in which the right side of each production is a block, except the start symbol may rewrite as a terminal point.*

Proof. Let L be generated by a 1-RGG $G = (N, T, P, S)$ satisfying Lemmas 1–4. Let $G' = (N, T, P', S)$, where P' is constructed from P as follows:

1. For each pair of productions $\overset{A}{\bullet} \to \alpha$ and $\overset{B}{\bullet} \to \overset{b}{\bullet}$, where α is a block containing the variable B, $B \in N - \{S\}$ and $b \in T$, create a new production $\overset{A}{\bullet} \to \beta$, where β is the block α with the variable B replaced by b.

2. Remove all productions of the form $\overset{A}{\bullet} \to \overset{a}{\bullet}$, $A \neq S$, from P. It should be clear that G' satisfies the conditions of the theorem and that $L(G) = L(G')$.

The cutpoint graph plays a central role in the study of 1-RGL's. The points of the *cutpoint graph* $c(H)$ of a graph H are the cutpoints of H. Two points of $c(H)$ are adjacent whenever the corresponding cutpoints in H lie in the same block.

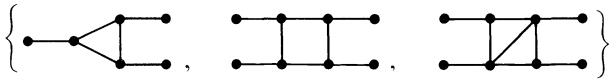
THEOREM 11. *If L is a* 1-RGL, *then the cutpoint graph of each graph in L is a path.*

Proof. Let $G = (N, T, P, S)$ be a normal form 1-RGG generating L. Observe that except for the start symbol, each variable which appears in a derivation of a graph in L is a cutpoint of the graph. Hence at each step after the first in a derivation, either a new cutpoint is generated or another block containing the last created cutpoint is generated. Thus the cutpoint graph is a path.

COROLLARY 11.1. *If L is a* 1-RGL, *then each block of H in L contains at most two cutpoints of H.*

Recall that every finite set of graphs is a 1-LGL (Lemma 6).

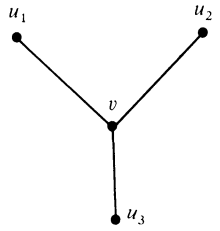
COROLLARY 11.2. *The set of graphs*



is a 1-LGL, *but is not a* 1-RGL.

COROLLARY 11.3. *Let L be a* 1-RGL. *If H is in L, then for any three blocks of H with a common cutpoint, at most two of them have another cutpoint of H.*

Proof. Suppose blocks B_1, B_2 and B_3 contain the cutpoint v of H and B_i also contains the cutpoints u_i, $i = 1, 2, 3$. Then in the cutpoint graph of H, these four points would give rise to the graph



contradicting Theorem 11.

Note that the cutpoint graphs of the non-1-CFGL L in Theorem 8 are paths. Hence Theorem 11 only gives a necessary condition for a set of graphs to be a 1-RGL.

The next theorem shows that the set of the cutpoint graphs of a 1-RGL is a 1-RGL.

THEOREM 12. *Let L be a* 1-RGL. *Then there is a* 1-RGL *which generates the cutpoint graphs of L.*

Proof. Let L be generated by a normal form 1-RGG $G = (N, T, P, S)$. We will construct a 1-RGG $G' = (N, T, P', S)$ from G where the productions of P'

are:

1. $\overset{S}{\bullet} \to \overset{A}{\bullet}$ if $\overset{S}{\bullet} \to \alpha$ is in P, and A is the variable in α.
2. $\overset{A}{\bullet} \to \overset{B}{\bullet}$ if $\overset{A}{\bullet} \to \alpha$ is in P, $A \neq S$, and B is the image of A.
3. $\overset{A}{\bullet} \to \overset{a}{\bullet}\!\!-\!\!\!-\!\!\overset{B}{\bullet}$ if $\overset{A}{\bullet} \to \alpha$ is in P, $A \neq S$, a is the image of A, and B is the variable in α.
4. $\overset{A}{\bullet} \to \overset{a}{\bullet}$ if $\overset{A}{\bullet} \to \alpha$ is in P, $A \neq S$, α is a terminal block, and a is the image of A.

The set of productions of G' generate the labels of the cutpoints of a graph in $L(G)$ and joins two points with a line if the two corresponding cutpoints of the graph in $L(G)$ lie in the same block.

There is a natural relation between strings and graphs. The string $a_1 a_2 \cdots a_n$ *corresponds* to the path

$$\overset{a_1}{\bullet}\!\!-\!\!\!-\!\!\overset{a_2}{\bullet} \cdots \overset{a_{n-1}}{\bullet}\!\!-\!\!\!-\!\!\overset{a_n}{\bullet}$$

generated by the 1-RGG where a_1 is the first point generated. The condition that a_1 be the first point generated enables us to associate exactly one string with the undirected graph.

THEOREM 13. *Let L be a* 1-RGL. *Then the set of strings corresponding to the cutpoint graphs of the graphs in L is a regular set.*

Proof. The proof follows immediately from the correspondence between the productions of the 1-RGG constructed in the proof of Theorem 12 and the regular string grammar productions. That is, the regular productions $A \to aB$, $A \to B$, $A \to a$ correspond to the 1-RGG productions $\overset{A}{\bullet} \to \overset{a}{\bullet}\!\!-\!\!\!-\!\!\overset{B}{\bullet}$, $\overset{A}{\bullet} \to \overset{B}{\bullet}$ and $\overset{A}{\bullet} \to \overset{a}{\bullet}$, respectively.

The converse of this theorem does not hold, as the set of strings corresponding to the cutpoint graphs of L in Theorem 8 is a regular set.

Theorem 14 follows from the observation that the cutpoint graph of a path is the same path with the two endpoints removed.

THEOREM 14. *Let R be a regular set. Then there is a* 1-RGG *G such that R is the set of strings corresponding to the cutpoint graphs of $L(G)$.*

Conclusions. We have defined three classes of first order graph grammars and have obtained results analogous to those for their string grammar counterparts. The same sort of results should hold for higher order graph grammars.

Our investigations of first order context-sensitive graph grammars have led us to conclude that, like context-sensitive web grammars, their rewriting rules must include some type of embedding specification. This appears to be the reason Pavlidis allowed his variables to be graph structures and not simply points and lines.

Finally, we mention that the results of the preceding sections can readily be extended to directed graphs.

REFERENCES

[1] N. ABE, M. MIZUMOTO, J. TOYODA AND K. TANAKA, *Web grammars and several graphs*, J. Comput. System Sci., 7 (1973), pp. 37–65.

[2] J. FEDER, *Plex languages*, Information Sci., 3 (1971), 225–241.

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

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

[5] U. MONTANARI, *Separable graphs, planar graphs and web grammars*, Information and Control, 16 (1970), pp. 243–267.

[6] T. PAVLIDIS, *Linear and context-free graph grammars*, J. Assoc. Comput. Mach., 19 (1972), pp. 11–22.

[7] J. PFALTZ AND A. ROSENFELD, *Web grammars*, Proc. Joint Internat. Conf. on Artificial Intelligence, Washington, D.C., May 1969, pp. 609–618.

[8] T. PRATT, *Pair grammars, graph languages and string-to-graph translations*, J. Comput. System Sci., 5 (1971), pp. 560–595.

SIAM J. COMPUT.
Vol. 3, No. 2, June 1974

OPTIMAL BINARY SEARCH TREES WITH RESTRICTED MAXIMAL DEPTH*

M. R. GAREY†

Abstract. An algorithm is given for constructing a binary tree of minimum weighted path length for n nonnegative weights under the constraint that no path length exceed a given bound L. The number of operations required is proportional to Ln^2. Such problems, which impose an additional constraint on the usual Huffman tree, arise in many applications, including computer file searching and the construction of optimal prefix codes under certain practical conditions.

Key words. algorithms, probabilistic search, variable-length codes

1. Introduction. Binary trees of minimum weighted path length have application in many areas, including searching computer files, merging sorted lists, diagnosing machine failures, and constructing variable-length codes. The well-known Huffman algorithm [5] can be used to efficiently construct such optimal binary trees. In this paper, we consider this problem under the additional restriction that no path length in the tree is permitted to exceed a given bound L. Karp [6] described a rather complicated integer linear programming technique for solving this problem in the context of constructing restricted-length prefix codes. Gilbert [1] proposed using such codes when source probabilities are not accurately known and gave an essentially enumerative method for finding a solution. In [3], Hu and Tan derive an improved nonenumerative algorithm for solving this problem, which, however, requires a computing time which grows exponentially with L. We shall present an algorithm which is considerably more efficient than any of these, requiring only $O(Ln^2)$ operations to construct an optimal tree for n weights.

2. Definitions. Our terminology essentially follows [3]. A binary tree (called an extended binary tree in [6]) consists of a distinguished node, called the *root*, and two disjoint binary trees, called the left and right *subtrees* of the root, either both empty or both nonempty. Nodes occurring in the two subtrees are called *descendants* of the root, and all nodes having a given node as a descendant are *ancestors* of that node. Nodes which have no descendants are called *terminal* nodes, and all nodes which are not terminal are called *nonterminal* nodes. Notice that every nonterminal node has at least two descendants. The *path length* of a node in a binary tree is simply the number of ancestors of that node.

It is convenient to consider the terminal nodes of a binary tree to be ordered, from left to right, as follows: given a binary tree T, the terminal node V_i is *left* of terminal node V_j if and only if there exists a subtree T' of T which has V_i occurring in its left subtree and V_j occurring in its right subtree. This is merely the left-to-right order of occurrence of the terminal nodes in the usual planar representation of the binary tree.

A *weighted binary tree* for the set[1] of weights $\{w_1, w_2, \cdots, w_n\}$ is a binary tree with n terminal nodes, each labeled with a different one of the n given weights.

* Received by the editors February 12, 1973, and in final revised form September 28, 1973.

† Bell Laboratories, Murray Hill, New Jersey 07974.

[1] This is actually a multiset, since we permit different weights to have identical values.

We use $l(w_i)$ to denote the path length of the terminal node labeled by w_i. The *total weighted path length* of a weighted binary tree is given by

$$\sum_{i=1}^{n} w_i \cdot l(w_i).$$

Given a set of nonnegative weights $\{w_1, w_2, \cdots, w_n\}$ and a positive integer L, we consider the problem of constructing a weighted binary tree for this set of weights which has no path length exceeding L and which has minimal total weighted path length among all such trees. We shall call such a tree an *optimal L-restricted tree* for $\{w_1, w_2, \cdots, w_n\}$.

3. Preliminary lemmas. In this section, we present four preliminary lemmas which will be useful later. The straightforward inductive proofs are omitted.

LEMMA 1. *A binary tree with terminal node path lengths $\{l_1, l_2, \cdots, l_n\}$ exists if and only if*

$$\sum_{i=1}^{n} 2^{-l_i} = 1.$$

(See Knuth [7, p. 404, Prob. 3].)

LEMMA 2. *Given positive integers $l'_1, l'_2, \cdots, l'_n$ satisfying*

$$\sum_{i=1}^{n} 2^{-l'_i} \leqq 1,$$

there exists a binary tree with terminal node path lengths $\{l_1, l_2, \cdots, l_n\}$ satisfying $l_i \leqq l'_i$, $1 \leqq i \leqq n$.

LEMMA 3. *An optimal L-restricted tree for $\{w_1, w_2, \cdots, w_n\}$ exists if and only if $L \geqq \log_2 n$.*

LEMMA 4. *If $L \geqq \log_2 n$ and $w_1 \geqq w_2 \geqq \cdots \geqq w_n$, then there exists an optimal L-restricted tree for $\{w_1, w_2, \cdots, w_n\}$ such that $l(w_1) \leqq l(w_2) \leqq \cdots \leqq l(w_n)$ and such that the terminal node labels $w_1, w_2, \cdots, w_n$ appear consecutively from left to right.* (See Schwartz and Kallick [9].)

4. A dynamic programming solution. We now use the results of § 3 to derive a dynamic programming algorithm which constructs an optimal L-restricted tree for $\{w_1, w_2, \cdots, w_n\}$ in time Ln^3. This basic algorithm will then be improved to time Ln^2 in § 5.

For the rest of the paper, we assume that the given weights have been indexed so that $w_1 \geqq w_2 \geqq \cdots \geqq w_n$.

Let T be any optimal L-restricted tree for $\{w_1, w_2, \cdots, w_n\}$ of the form described by Lemma 4. Consider any nonterminal node V in T, and let K denote the path length of V. By choice of T, the set of weights labeling the terminal descendants of V must equal $\{w_t | i \leqq t \leqq j\}$, for some integers i and j, $1 \leqq i < j \leqq n$. The subtree of T with root V must be an optimal $(L - K)$-restricted tree for that set of weights, for otherwise we could replace it by such an optimal tree to improve upon T, a contradiction. This latter fact, along with the special structure of the terminal descendant sets, allows us to give an Ln^3 algorithm for our problem using a "dynamic programming" technique, similar to the method of [2].

Let $[u, v, k]$, with $1 \leqq u \leqq v \leqq n$ and $0 \leqq k \leqq L$, denote the subproblem of finding an optimal k-restricted tree for $\{w_u, w_{u+1}, \cdots, w_v\}$. Let $K[u, v, k]$ be the total weighted path length of such an optimal tree for $[u, v, k]$. Whenever $k < \log_2 (v - u + 1)$, no solution to $[u, v, k]$ exists, and we set $K[u, v, k] = \infty$. Otherwise, we have

$$K[u, u, k] = 0, \qquad 0 \leqq k \leqq L;$$

$$(*) \qquad K[u, v, k] = \sum_{i=u}^{v} w_i + \min_{u \leqq I \leqq v-1} (K[u, I, k-1] + K[I+1, v, k-1]),$$

$$1 \leqq u < v \leqq n, \quad \log_2 (v - u + 1) \leqq k \leqq L.$$

If $I = I_0$ is a solution of (*), then an optimal tree for $[u, v, k]$ consists of a root, a left subtree which is optimal for $[u, I_0, k-1]$, and a right subtree which is optimal for $[I_0 + 1, v, k-1]$. Solving (*) for all required u, v, k in order of increasing $v - u$ results in an optimal tree for the original problem $[1, n, L]$.

It is convenient to define $I[u, v, k]$ to be a specific choice of I satisfying (*), with $I[u, u, k] = u$. By saving $I[u, v, k]$ it becomes unnecessary to save an optimal tree for $[u, v, k]$, since upon determination of $I[1, n, L]$, the optimal tree can be reconstructed directly from the saved $I[u, v, k]$ values.

The resulting algorithm requires a number of operations proportional to Ln^3. The analysis is similar to the analysis of the algorithm given in [2] and will not be detailed here.

5. An improved algorithm. The algorithm of § 4 can be improved with a technique similar to that used by Knuth in [7]. The main result for obtaining this improvement is a corollary to the following theorem, which is proved in the Appendix.

THEOREM 1. *For* $2 \leqq x + 1 < y \leqq n$ *and* $k \geqq 0$,

$$K[x, y, k] - K[x+1, y, k] \geqq K[x, y-1, k] - K[x+1, y-1, k].$$

This essentially states that the increase in total weighted path length of the optimal k-restricted tree caused by adding a new large weight $w_{\max}$ to a set S of nonnegative weights is greater than the increase caused by adding $w_{\max}$ to the smaller set $S - \{w_{\min}\}$, where $w_{\min}$ is the least weight in S. Since the Huffman tree is an optimal k-restricted tree when k is suitably large, we have the following immediate corollary.

COROLLARY 1. *Let* $H(X)$ *denote the total weighted path length of the Huffman tree for the set* X *of nonnegative weights. If* T *is a set of two or more nonnegative weights, then*

$$H(T) - H(T - \{w_{\max}\}) \geqq H(T - \{w_{\min}\}) - H(T - \{w_{\max}, w_{\min}\}),$$

where $w_{\max}$ *and* $w_{\min}$ *are, respectively, the largest and smallest weights in* T.

Though Corollary 1 is itself of some interest, it is stated merely as an interesting consequence of Theorem 1 and is not directly relevant to the problem at hand. The next corollary, however, applies directly to our problem.

COROLLARY 2. *Whenever* $I[u, v-1, k] \leqq I[u+1, v, k]$, *there always exists a solution to* (*) *such that*

$$I[u, v-1, k] \leqq I[u, v, k] \leqq I[u+1, v, k].$$

Proof. We prove that we can choose $I[u, v, k] \leqq I[u+1, v, k]$; the other half follows symmetrically. More specifically, we prove that if (*) is minimized for some $I > I[u+1, v, k]$ then (*) is also minimized for $I = I[u+1, v, k]$. Suppose (*) is minimized for I_1, where $I_1 > I_2 = I[u+1, v, k]$. From repeated application of Theorem 1, we obtain

$$K[u, I_1, k-1] - K[u+1, I_1, k-1] \geqq K[u, I_2, k-1] - K[u+1, I_2, k-1],$$

which we rewrite as

$$K[u, I_1, k-1] - K[u, I_2, k-1] \geqq K[u+1, I_1, k-1] - K[u+1, I_2, k-1].$$

Adding $K[I_1+1, v, k-1] - K[I_2+1, v, k-1]$ to each side and using the optimality of I_2 for $[u+1, v, k]$, we have

$$\begin{aligned}(K[u, I_1, k-1] + K[I_1+1, v, k-1]) - (K[u, I_2, k-1] + K[I_2+1, v, k-1]) \\ \geqq (K[u+1, I_1, k-1] + K[I_1+1, v, k-1]) \\ - (K[u+1, I_2, k-1] + K[I_2+1, v, k-1]) \geqq 0.\end{aligned}$$

Therefore, (*) is also minimized for $I = I_2$, and we may choose $I[u, v, k] = I[u+1, v, k]$, completing the proof.

We now show how Corollary 2 simplifies the previous algorithm. Consider all the subproblems $[u, v, k]$ for a fixed k and fixed $v - u = p$. In solving (*) for $[i, p+i, k]$, we need only consider

$$1 + I[i+1, p+i, k] - I[i, p+i-1, k]$$

possible values for I. Summing this over i, we obtain

$$\begin{aligned}\sum_{i=1}^{n-p} (1 + I[i+1, p+i, k] - I[i, p+i-1, k]) \\ = (n-p) + I[n-p+1, n, k] - I[1, p, k] \leqq 2n - p - 1.\end{aligned}$$

Thus, in solving all the subproblems for fixed $v - u = p$ and fixed k, we need only consider a total of at most $2n - p - 1$ tentative solutions. Since there are only $n+1$ possibilities for p and $L+1$ possibilities for k, the revised algorithm, which simply uses Corollary 2 to reduce the values of I considered in solving (*), requires only $O(Ln^2)$ operations. Note that, in order to achieve this efficiency, it is also necessary to compute the weight sums using previously computed weight sums, that is, according to

$$\sum_{i=u}^{v} w_i = \sum_{i=u}^{v-1} w_i + w_v,$$

which requires only a single addition. A FORTRAN version of this algorithm, implemented on the HIS 6070 by Mary Ann Gatto, required approximately $Ln^2/10$ milliseconds.

An additional feature of this algorithm is that one actually obtains the optimal codes and their costs for all length restrictions less than or equal to L with a single application of the algorithm. This provides the user with potentially useful information concerning the effects of various length restrictions on the optimal solution cost.

6. An open problem. The unimproved dynamic programming algorithm of § 4 can be used to solve a more general problem. Define an *optimal alphabetic binary tree* [4] for a given sequence of weights $w_1, w_2, \cdots, w_n$ (not necessarily ordered by magnitude) to be a weighted binary tree having n terminal nodes labeled by $w_1, w_2, \cdots, w_n$ in consecutive left-to-right order and having minimal total weighted path length among all such trees. One can then consider the analogous problem of constructing an optimal L-restricted alphabetic binary tree. It is easy to see that the Ln^3 algorithm of § 4 can be applied directly to solve this problem. However, an interesting open problem is to determine whether one can improve this algorithm with a result similar to Corollary 2. The proof of Theorem 1, which is used in proving Corollary 2, cannot be extended directly to this more general problem, because it depends strongly on the magnitude ordering of the given weights.

Appendix. Proof of Theorem 1.

THEOREM 1. $K[x, y, k] - K[x + 1, y, k] \geqq K[x, y - 1, k] - K[x + 1, y - 1, k]$.

Proof. If no tree exists for $[x, y, k]$, then $K[x, y, k] = \infty$, and the inequality holds trivially. Otherwise, let $T[x, y, k]$ denote a fixed optimal tree for $[x, y, k]$ with $l_1(r)$ denoting the length of w_r in $T[x, y, k]$, $x \leqq r \leqq y$. Similarly, let $T[x + 1, y - 1, k]$ denote a fixed optimal tree for $[x + 1, y - 1, k]$ with $l_2(r)$ denoting the length of w_r in that tree, $x < r < y$. By Lemma 4, we may assume that $l_1(r) \leqq l_1(r + 1)$ and $l_2(r) \leqq l_2(r + 1)$, for all r satisfying $x \leqq r < y$ or $x < r < y - 1$, respectively.

Define m as the largest element of $\{r | l_2(r) < k,\ x < r < y\}$.[2] Construct a tree T' for $[x + 1, y, k]$ from $T[x + 1, y - 1, k]$ by replacing the terminal node for w_m by a nonterminal node with two terminal descendants, one labeled w_m and one labeled w_{m+1}, and replacing each old label w_r by w_{r+1} for $m + 1 \leqq r \leqq y - 1$. With $l'(r)$ denoting the lengths in T', we then have

$$l'(r) = l_2(r), \qquad x < r < y, \quad r \neq m, \quad r \neq m + 1,$$

$$l'(m) = l'(m + 1) = l_2(m) + 1,$$

$$l'(y) = \max(l_2(y - 1), l_2(m) + 1).$$

Define $\Delta(r) = l_1(r) - l'(r)$ for $x < r \leqq y$. Notice that $\Delta(m) \leqq \Delta(m + 1)$, because $l_1(m) \leqq l_1(m + 1)$, and $\Delta(r) \leqq 0$ for $m + 2 \leqq r \leqq y$, since then $l'(r) = k \geqq l_1(r)$. We now consider a number of cases.

(a) Suppose $l'(x + 1) < l_1(x)$. Construct a tree T^* for $[x, y, k]$ from T' by replacing the terminal node for w_{x+1} by a nonterminal node with two terminal descendants, one labeled w_x and one labeled w_{x+1}. Let $\Delta^*(r) = l_1(r) - l^*(r)$, $x \leqq r \leqq y$, where $l^*(r)$ denotes the length of w_r in T^*. Then $\Delta^*(x + 1) \geqq \Delta^*(x) \geqq 0$. Let T'' be a tree for $[x + 1, y, k]$ with lengths $l''(r)$ satisfying $l''(r) \leqq l'(r) + \Delta^*(r)$, $x + 1 \leqq r \leqq y$. To see that such a tree exists, we need only show that

$$\sum_{r=x+1}^{y} 2^{-l'(r)-\Delta^*(r)} \leqq 1.$$

[2] Note that this set is nonempty since $T[x, y, k]$ exists.

However, we have

$$\sum_{r=x+1}^{y} 2^{-l'(r)-\Delta^*(r)} = \sum_{r=x+2}^{y} 2^{-l^*(r)-\Delta^*(r)} + 2^{1-l^*(x+1)-\Delta^*(x+1)}$$

$$= \sum_{r=x+1}^{y} 2^{-l_1(r)} + 2^{-l_1(x+1)}$$

$$\leqq \sum_{r=x+1}^{y} 2^{-l_1(r)} + 2^{-l_1(x)} = 1.$$

Thus there exists such a T''. Letting $K(T)$ denote the total weighted path length of T, we have

$$K(T') - K(T'') = \sum_{r=x+1}^{y} w_r(l'(r) - l''(r))$$

$$\geqq -\sum_{r=x+1}^{y} w_r\Delta^*(r)$$

$$\geqq -w_x\Delta^*(x) - \sum_{r=x+1}^{y} w_r\Delta^*(r)$$

$$= K(T^*) - K[x, y, k] \geqq 0.$$

Therefore

$$K[x, y, k] - K[x+1, y, k] \geqq K[x, y, k] - K(T'')$$

$$\geqq K(T^*) - K(T')$$

$$= w_{x+1} + w_x(1 + l'(x+1)).$$

Now we can construct a tree $\overline{T}$ for $[x, y-1, k]$ from $T[x+1, y-1, k]$ by replacing the terminal node for w_{x+1} by a nonterminal node with two terminal descendants, one labeled w_x and one labeled w_{x+1}. This is legal because $l'(x+1) \geqq l_2(x+1)$. We then have

$$K[x, y, k] - K[x+1, y, k] \geqq w_{x+1} + w_x(1 + l'(x+1))$$

$$\geqq w_{x+1} + w_x(1 + l_2(x+1))$$

$$= K(\overline{T}) - K[x+1, y-1, k]$$

$$\geqq K[x, y-1, k] - K[x+1, y-1, k],$$

proving the theorem for case (a).

(b) Suppose $\Delta(m+1) \leqq 0$ and $l'(x+1) \geqq l_1(x)$. We claim that then there exists a tree T^* for $[x, y, k]$ with lengths $l^*(r)$ satisfying

$$l^*(x) = l_1(x),$$

$$l^*(r) = l'(r) \quad \text{if } \Delta(r) \leqq 0,$$

$$l_1(r) \geqq l^*(r) \geqq l'(r) \quad \text{if } \Delta(r) > 0.$$

We give an explicit construction for T^*. Initially, set $l^*(x) = l_1(x)$ and $l^*(r) = l'(r)$

for $x < r \leqq y$. At this point, we have

$$\sum_{r=x}^{y} 2^{-l^*(r)} = 2^{-l_1(x)} + \sum_{r=x+1}^{y} 2^{-l'(r)}$$
$$= 2^{-l_1(x)} + 1 > 1.$$

As long as $\sum_{r=x}^{y} 2^{-l^*(r)} > 1$, choose s such that $l^*(s) < l_1(s)$ and $l^*(s)$ is as small as possible; increase $l^*(s)$ by 1. Since

$$\sum_{r=x}^{y} 2^{-\max(l_1(r), l'(r))} \leqq \sum_{r=x}^{y} 2^{-l_1(r)} = 1,$$

we must eventually obtain values for $l^*(r)$ satisfying $\sum_{r=x}^{y} 2^{-l^*(r)} \leqq 1$. Furthermore, since increasing $l^*(s)$ by one decreases $\sum_{r=x}^{y} 2^{-l^*(r)}$ by $2^{-l^*(s)-1} < 2^{-l_1(x)}$, and since the choice of s dictates that each reduction is no larger than the previous reduction, it is easy to see that we in fact will obtain $\sum_{r=x}^{y} 2^{-l^*(r)} = 1$.

We now proceed as in case (a). Define $\Delta^*(r) = l_1(r) - l^*(r)$, $x \leqq r \leqq y$. Let T'' be a tree for $[x+1, y, k]$ with lengths $l''(r)$ satisfying $l''(r) \leqq l'(r) + \Delta^*(r)$, $x+1 \leqq r \leqq y$. The existence of such a T'' follows from Lemma 2, since, using the fact that either $l^*(r) = l'(r)$ or both $l^*(r) > l'(r)$ and $\Delta^*(r) \geqq 0$, we have

$$1 - \sum_{r=x+1}^{y} 2^{-l'(r)-\Delta^*(r)} = \sum_{r=x+1}^{y} 2^{-l'(r)}(1 - 2^{-\Delta^*(r)})$$
$$\geqq \sum_{r=x+1}^{y} 2^{-l^*(r)}(1 - 2^{-\Delta^*(r)})$$
$$= \sum_{r=x+1}^{y} 2^{-l^*(r)} - \sum_{r=x+1}^{y} 2^{-l_1(r)} = 0.$$

Exactly as in case (a), we obtain $K(T') - K(T'') \geqq K(T^*) - K[x, y, k]$, yielding

$$K[x, y, k] - K[x+1, y, k] \geqq K(T^*) - K(T').$$

Now we construct a tree $\bar{T}$ for $[x, y-1, k]$ which has lengths $\bar{l}(r)$ satisfying

$$\bar{l}(r) = l^*(r), \qquad x \leqq r < m,$$
$$\bar{l}(r) = l_2(r), \qquad m \leqq r < y.$$

This tree exists because

$$\sum_{r=x}^{y-1} 2^{-\bar{l}(r)} = \sum_{r=x}^{m-1} 2^{-l^*(r)} + \sum_{r=m}^{y-1} 2^{-l_2(r)}$$
$$= \sum_{r=x}^{m-1} 2^{-l^*(r)} + \sum_{r=m}^{y} 2^{-l'(r)}$$
$$= \sum_{r=x}^{y} 2^{-l^*(r)} = 1.$$

We now have

$$\sum_{r=x}^{y-1} w_r(l(r) - l_2(r)) = \sum_{r=x}^{m-1} w_r(l(r) - l_2(r))$$
$$= \sum_{r=x}^{m-1} w_r(l^*(r) - l'(r))$$
$$= \sum_{r=x}^{y} w_r(l^*(r) - l'(r)).$$

Therefore

$$K[x, y, k] - K[x+1, y, k] \geqq K(T^*) - K(T')$$
$$= K(\bar{T}) - K[x+1, y-1, k]$$
$$\geqq K[x, y-1, k] - K[x+1, y-1, k],$$

completing the proof for case (b).

(c) Suppose $\Delta(m) \leqq 0$, $\Delta(m+1) > 0$ and $l'(x+1) \geqq l_1(x)$. We claim that then there exists a tree T^* for $[x, y, k]$ with lengths $l^*(r)$ satisfying

$$l^*(x) = l_1(x);$$
$$l^*(r) = l'(r) \quad \text{if } r = m+1 \quad \text{or} \quad \Delta(r) \leqq 0;$$
$$l_1(r) \geqq l^*(r) \geqq l'(r) \quad \text{if } r \neq m+1 \quad \text{and} \quad \Delta(r) > 0.$$

If we can show that

$$\sum_{\substack{r=x \\ r \neq m+1}}^{y} 2^{-\max(l_1(r), l'(r))} + 2^{-l'(m+1)} \leqq 1,$$

then the same method for constructing T^* as in case (b) can be used. We now show that this is the case.

There exist positive integers A_r such that $2^{-l'(r)} - 2^{-l_1(r)} = A_r \cdot 2^{-l'(m)}$ for all $r \neq m+1$ satisfying $\Delta(r) > 0$, because $l'(m) \geqq l_1(m) \geqq l_1(r) > l'(r)$ for all such r. Let A denote the sum of all the A_r. If the desired inequality were not satisfied, then

$$1 < \sum_{\substack{r=x \\ r \neq m+1}}^{y} 2^{-\max(l_1(r), l'(r))} + 2^{-l'(m+1)}$$
$$= \sum_{r=x+1}^{y} 2^{-l'(r)} + 2^{-l_1(x)} - A \cdot 2^{-l'(m)}$$
$$= 1 + 2^{-l_1(x)} - A \cdot 2^{-l'(m)},$$

which implies that

$$2^{-l_1(x)} - A \cdot 2^{-l'(m)} \geqq 2^{-l'(m)}.$$

But then we have

$$1 = \sum_{r=x}^{y} 2^{-l_1(r)}$$

$$> \sum_{r=x}^{y} 2^{-\max(l_1(r),l'(r))}$$

$$= \sum_{\substack{r=x \\ r \neq m+1}}^{y} 2^{-\max(l_1(r),l'(r))} + 2^{-l_1(m+1)}$$

$$= 1 + 2^{-l_1(x)} - A \cdot 2^{-l'(m)} + 2^{-l_1(m+1)} - 2^{-l'(m+1)}$$

$$\geqq 1 + 2^{-l'(m)} + 2^{-l_1(m+1)} - 2^{-l'(m)} > 1,$$

a contradiction. Thus, we can construct such a T^* using the method of case (b). The rest of the proof for case (c) follows exactly as in case (b).

(d) Suppose $\Delta(m+1) \geqq \Delta(m) > 0$ and $l'(x+1) \geqq l_1(x)$. Using the same method as in case (c), it is not hard to show that we can use the method of case (b) to construct a tree T^* for $[x, y, k]$ with lengths $l^*(r)$ satisfying

$$l^*(x) = l_1(x),$$
$$l^*(r) = l'(r) \quad \text{if } \Delta(r) \leqq 0,$$
$$l_1(r) \geqq l^*(r) \geqq l'(r) \quad \text{if } \Delta(r) > 0 \quad \text{and} \quad x < r \leqq m,$$
$$l^*(m+1) = l^*(m).$$

As in case (b), we also obtain

$$K[x, y, k] - K[x+1, y, k] \geqq K(T^*) - K(T').$$

Now we construct a tree $\bar{T}$ for $[x, y-1, k]$ which has lengths $\bar{l}(r)$ satisfying

$$\bar{l}(r) = l^*(r), \qquad x \leqq r < m,$$
$$\bar{l}(m) = l^*(m) - 1,$$
$$\bar{l}(r) = l_2(r), \qquad m < r < y.$$

This tree exists because

$$\sum_{r=x}^{y-1} 2^{-\bar{l}(r)} = \sum_{r=x}^{m-1} 2^{-l^*(r)} + 2^{-l^*(m)+1} + \sum_{r=m+1}^{y-1} 2^{-l_2(r)}$$

$$= \sum_{r=x}^{m-1} 2^{-l^*(r)} + 2^{-l^*(m)} + 2^{-l^*(m+1)} + \sum_{r=m+2}^{y} 2^{-l^*(r)}$$

$$= \sum_{r=x}^{y} 2^{-l^*(r)} = 1.$$

We then have

$$\begin{aligned}\sum_{r=x}^{y-1} w_r(\bar{l}(r) - l_2(r)) &= \sum_{r=x}^{m} w_r(\bar{l}(r) - l_2(r)) \\ &= \sum_{r=x}^{m} w_r(l^*(r) - l'(r)) \\ &< \sum_{r=x}^{m+1} w_r(l^*(r) - l'(r)) \\ &= \sum_{r=x}^{y} w_r(l^*(r) - l'(r)).\end{aligned}$$

Therefore

$$\begin{aligned}K[x, y, k] - K[x+1, y, k] &\geqq K(T^*) - K(T') \\ &> K(\bar{T}) - K[x+1, y-1, k] \\ &\geqq K[x, y-1, k] - K[x+1, y-1, k],\end{aligned}$$

completing the proof for case (d) and proving the theorem.

REFERENCES

[1] E. N. GILBERT, *Codes based on inaccurate source probabilities*, IEEE Trans. Information Theory, IT-17 (1971), pp. 304–314.

[2] E. N. GILBERT AND E. F. MOORE, *Variable-length binary encodings*, Bell System Tech. J., 38 (1959), pp. 933–967.

[3] T. C. HU AND K. C. TAN, *Path length of binary search trees*, SIAM J. Appl. Math., 22 (1972), pp. 225–235.

[4] T. C. HU AND A. C. TUCKER, *Optimal computer search trees and variable-length alphabetic codes*, Ibid., 21 (1971), pp. 514–532.

[5] D. A. HUFFMAN, *A method for the construction of minimum redundancy codes*, Proc. Inst. Radio Engineers, 40 (1952), pp. 1098–1101.

[6] R. M. KARP, *Minimum redundancy coding for the discrete noiseless channel*, Inst. Radio Engineers Trans. Information Theory, IT-7 (1961), pp. 27–38.

[7] D. E. KNUTH, *The Art of Computer Programming*, vol. 1, Addison-Wesley, Reading, Mass., 1968, pp. 399–404.

[8] ———, *Optimum binary search trees*, Acta Informat., 1 (1971), pp. 14–25.

[9] E. S. SCHWARTZ AND B. KALLICK, *Generating a canonical prefix encoding*, Comm. ACM, 7 (1964), pp. 166–169.

SIAM J. COMPUT.
Vol. 3, No. 2, June 1974

JUMP PDA'S AND HIERARCHIES OF DETERMINISTIC CONTEXT-FREE LANGUAGES*

S. A. GREIBACH†

Abstract. A jump pushdown store acceptor can in one step erase its store through the first occurrence of one of its pushdown store symbols. Every deterministic context-free language can be accepted by a deterministic jump pushdown store acceptor operating with finite delay (semirealtime). For deterministic jump pushdown store acceptors operating with finite delay, $n + 1$ types of pushdown store symbols are more powerful than n types of pushdown store symbols. As a consequence, it can be shown that the family of deterministic context-free languages does not form a principal AFDL.

Key words. deterministic context-free, pushdown store, hierarchies, jump pda, semirealtime

1. Introduction. The family of deterministic context-free languages has been extensively studied from the point of view of both grammars and machines [1], [2], [3], [4]. Various subfamilies have been found by placing restrictions on grammars or machines or both [5], [6], [7], [8], [9], [10]. Recently there have been various studies of the structure of context-free languages and other nondeterministic systems [11], [12], [13], but no satisfactory algebraic or operation-based theory has been developed for deterministic context-free languages; the results of this paper indicate such a theory might not be possible.

The AFDL (Def. 2.7) has been suggested by Chandler as a model for deterministic systems [14]. He showed that any family of one-way deterministic acceptors with input endmarkers which satisfies a few minor conditions (any closed class of one-way deterministic balloon automata [15]) is closed under certain operations and forms an AFDL, that every AFDL can be so expressed, and further that every principal AFDL can be deterministically accepted by machines with one-way input and a finite number of working tape symbols and instructions (finitely encoded—Def. 2.1). So far, this parallels the results for AFL's and nondeterministic systems—every family of one-way nondeterministic acceptors operating with finite delay (closed class of one-way nondeterministic balloon automata) forms an AFL containing $\{e\}$ and vice versa [16].[1] However, in the nondeterministic case, finitely encoded storage always yields a principal AFL [17].[2] A corresponding result does *not* hold for one-way deterministic machines.

The family $\mathscr{C}$ of deterministic context-free languages can be accepted by deterministic pushdown store acceptors with only two pushdown store symbols and a finite number of instruction types. However, we shall show that $\mathscr{C}$ is *not* a

* Received by the editors December 1, 1972, and in final revised form November 12, 1973.

† Department of System Science, University of California at Los Angeles, Los Angeles, California 90024. The research represented in this paper was supported in part by the National Science Foundation under Grant GJ-803.

[1] An AFL is a family of languages containing at least one nonempty set and closed under union, concatenation, Kleene +, homomorphism, inverse homomorphism and intersection with regular sets.

[2] The least AFL containing a language L is the *principal* AFL *generated by* L.

principal AFDL by expressing $\mathscr{C}$ as $\bigcup_n \mathscr{C}_n$ for a strictly increasing chain of AFDL's $\mathscr{C}_n$, all proper subfamilies of the family of deterministic context-free languages.

Informally, our result says that there is no deterministic context-free language "universal" with respect to the AFDL operations. That is, there is no deterministic context-free language from which all others can be obtained using AFDL operations. By contrast, in the one-way nondeterministic case, finitely encoded storage always implies the existence of a "universal" language. In particular, there is even a context-free language L_0 "universal" with respect to inverse homomorphism—every context-free language is an inverse homomorphic image of L_0 (or L_0-$\{e\}$); thus L_0 is the "hardest" context-free language to recognize (or "parse") deterministically. This point and its implications are discussed further in [20]. No analogous result can hold for deterministic context-free languages.

The bulk of this paper is concerned with defining the $\mathscr{C}_n$ and proving that they form an infinite hierarchy of deterministic context-free languages. First we define jump pushdown store automata or JPDA's—pushdown store automata with instructions such as "erase up to and including the first A on the pushdown store." Then we extend a result of Schützenberger [18] to show that every deterministic context-free language can be accepted with finite delay by a deterministic JPDA. This is *not* true for ordinary pda's where the best we can do is linear time.

Now in linear time, a deterministic pda or JPDA with only two pushdown store symbols is as powerful as any such machine. But if we restrict JPDA's to acceptance with finite delay (semirealtime in [8]), we find that we need more and more types of jumps and hence more and more pushdown store symbols. Thus each $\mathscr{C}_n$ is formed of languages accepted with finite delay by deterministic JPDA's using n pushdown store symbols.

To prove that $n + 1$ symbols are more powerful than n in this context, we define a corresponding type of Dyck set—a Dyck set with "wipeouts". In an ordinary Dyck set, a "negative" symbol $\bar{a}_i$ cancels a corresponding "positive" symbol a_i immediately to its left; $a_i\bar{a}_i \overset{*}{\sim} e$. A "wipeout" E_i cancels everything up to and including the *first* corresponding "positive" symbol a_i to its left, *provided* only positive symbols intervene; $a_iyE_i \overset{*}{\sim} e$ if y contains neither a_i nor any $\bar{a}_j$ or E_j. A formal definition appears in § 2. We obtain our hierarchy by showing that the $n + 1$ symbol Dyck set with wipeouts is in $\mathscr{C}_{n+1} - \mathscr{C}_n$.

So the deterministic context-free languages give us a natural example of a deterministic family defined by a family of finitely encoded one-way deterministic acceptors which is *not* a principal AFDL. Also, $\mathscr{C}_n$ is a natural example of a family of deterministic languages not closed under union with regular sets.

In § 2 we define JPDA's, Dyck sets with wipeouts, and the $\mathscr{C}_n$, and show that $\mathscr{C} = \bigcup_n \mathscr{C}_n$. In § 3 we establish the hierarchy results. Section 4 contains conclusions and open problems.

2. Jump pda's and Dyck sets with wipeouts. In this section we define the family of pushdown store acceptors with "jump" instructions, and show that they can recognize with finite delay all and only deterministic context-free languages. Then we define the associated extended Dyck sets which we use in the next section to obtain our hierarchy results.

A comparison of the definition of deterministic pushdown acceptors used by Schützenberger [18] with the results of Ginsburg and Greibach [1][3] shows that every deterministic context-free language can be accepted in realtime by a pushdown store acceptor which at one move can either write a symbol on the pushdown store, leave the store alone, or else "jump" down the store to erase the first member of some fixed regular set. In this paper we shall use variants of this model.

We shall be introducing several modifications of pushdown store machines, all of which are deterministic, have endmarkers and accept by final state, but which vary in the types of instructions they can use. To avoid redefining machines, instantaneous descriptions, computations and so on for each modification, we introduce the following general definition of a storage schema.[4] It consists of a working tape symbol set, an instruction set and partial functions g and f indicating how the storage is accessed ($g(x)$) and changed ($f(x, u)$). For our purposes, the access will always be in the pushdown mode reading from the right. When $f(x, u)$ is undefined, it indicates that instruction u applied to tape contents x is illegal; that $g(x)$ is undefined indicates an illegal storage configuration. Let $\uparrow$ abbreviate "is undefined" and $\downarrow$ abbreviate "is defined".

First we define a storage schema in terms of a working tape vocabulary Γ, instruction set I and storage access (g) and transition (f) functions, and a few technical restrictions on f and g. The first restriction says that there is a designated initial storage condition (assumed for convenience to be empty storage) which can be distinguished from all other storage configurations. The second implies the existence of "stand still" instructions allowing the state of a machine to be changed or the input tape advanced without altering the storage. The third restriction is a very weak finiteness condition limiting the new symbol types an instruction can add to storage.

DEFINITION 2.1. Let Γ be a set of symbols, I a set, f and g partial functions from $\Gamma^* \times I$ into Γ^* and from Γ^* into Γ^*, respectively.[5] Then $\Omega = (\Gamma, I, f, g)$ is a *storage schema* if (i) $g(e) = e$ and $g(x) \neq e$ for $x \neq e$, (ii) for each γ in $g(\Gamma^*)$ there is a $u_\gamma \in I$ such that $g(x) = \gamma$ implies $f(x, u_\gamma) = x$, and (iii) for each $u \in I$ there is a finite $\Gamma_u \subseteq \Gamma$ such that if $x \in \Gamma_1^*$ and $f(x, u)\downarrow$, then $f(x, u) \in (\Gamma_1 \cup \Gamma_u)^*$. If $I \cup g(\Gamma^*)$ is finite, then Ω is *finitely encoded*.

Thus a finitely encoded schema has a finite number of instructions and a finite amount of information available from the storage at a given step. For example, a finitely encoded pushdown store schema might have two pushdown store symbols (A and B), a few instructions (for example: no change, add A to top, add B to top, erase top symbol) and, although there are infinitely many possible tape contents, the access function only reads the top symbol and learns one of three things (top is A, top is B, or tape is empty).

Now we define a deterministic Ω-machine as an acceptor with a finite state control (state set K, transition function δ), one-way input tape and an endmarker \$, employing a storage tape of type Ω.

DEFINITION 2.2. For storage schema $\Omega = (\Gamma, I, f, g)$, $D = (K, \Sigma, \delta, q_0, F, \$)$ is a deterministic Ω-machine if K and Σ are finite sets of *states* and *inputs*, $\$ \notin \Sigma$ is

[3] This has been pointed out by P. C. Fisher [23] and is implicit in his 1966 paper [24].

[4] Also called an AFA schema in [16] and [17].

[5] Let e be the empty word, L^* the closure of L under concatenation and $L^+ = LL^*$.

an *endmarker*, $q_0 \in K$, $F \subseteq K$, and δ is a partial function from $K \times (\Sigma \cup \{\$, e\}) \times \Gamma^*$ into $K \times I$ such that

(i) $G_D = \{\gamma | \exists q, a, \delta(q, a, \gamma)\downarrow\}$ is finite, and

(ii) for all $q \in K$, $a \in \Sigma \cup \{\$\}$, $\gamma \in G_D$, if $\delta(q, e, \gamma)\downarrow$, then $\delta(q, a, \gamma)$ is undefined.

A member of $K \times \Sigma^* \times \Gamma^*$ is an *instantaneous description* (ID); we define relations $\overset{k}{\vdash}$ and $\overset{*}{\vdash}$ on ID's as follows. If $\delta(q, a, \gamma) = (q', u)$, $w \in \Sigma^*$, $g(x) = \gamma$ and $f(x, u)\downarrow$, then $(q, aw, x) \overset{1}{\vdash} (q', w, f(x, u))$. For ID's C_1, C_2 and C_3, $C_1 \overset{0}{\vdash} C_1$, and if $C_1 \overset{k}{\vdash} C_2$ and $C_2 \overset{1}{\vdash} C_3$, then $C_1 \overset{k+1}{\vdash} C_2$; if $C_1 \overset{k}{\vdash} C_2$ for any k, then $C_1 \overset{*}{\vdash} C_2$. If there is a k such that $(q, e, x) \overset{n}{\vdash} (q', e, x')$ implies $n \leqq k$, then D is *finite delay* and of *delay* k. If $\delta(q, e, \gamma)\uparrow$ for all $q \in K$ and $\gamma \in G_D$, then D is *realtime*. The *language accepted by* D is $L(D) = \{w \in \Sigma^* | \exists q \in F, (q_0, w\$, e) \overset{*}{\vdash} (q, e, e)\}$.

If Ω is not finitely encoded, the totality of Ω-machines might make infinitely many distinctions among storage tapes, but a given Ω-machine D should only recognize a finite number, hence G_D must be finite. Thus if Ω represents a pushdown store scheme with countably many pushdown store symbols $A_1, A_2, \cdots$, a given machine D will only deal with a finite subset $G_D = \{A_{j_1}, \cdots, A_{j_n}\}$.

The definition of an instantaneous description of D is standard: state, input to be processed, and storage contents. If D is in state q, the storage contains x, the information sent to D is $\gamma = g(x)$, the input is a, and $\delta(q, a, g(x)) = \delta(q, a, \gamma) = (q', u)$, then D changes state to q' and the storage is changed to $f(x, u)$. If $a \in \Sigma$, this implies that D reads input symbol a and advances the input tape; if $a = e$, D acts without consulting or advancing its input tape. If D has no transitions $\delta(q, e, \gamma)$, it must advance its input tape at each move and so acts in realtime. The machine D accepts input w if it can start with initial state q_0 and empty storage, read all of $w\$$ (w plus endmarker), and end in a final state with empty storage.

We have only defined deterministic Ω-machines with endmarkers (\$). A *nondeterministic* Ω*-machine* $D = (K, \Sigma, \delta, q_0, F)$ would be similarly defined except that δ would be a function from $K \times (\Sigma \cup \{e\}) \times \Gamma^*$ into the finite subsets of $K \times I$ and the endmarker would be dropped; G_D is still finite, and

$$L(D) = \{w | q \in F, (q_0, w, e) \overset{*}{\vdash} (q, e, e)\}.$$

Unless otherwise stated, by Ω-machine we shall mean a deterministic acceptor.

DEFINITION 2.3. For a storage schema Ω, the family of deterministic languages defined by Ω is

$$\mathscr{L}_d(\Omega) = \{L(D) | D \text{ is an } \Omega\text{-machine}\}$$

and the family of *almost realtime* languages is

$$\mathscr{L}_r^t(\Omega) = \{L(D) | D \text{ is a finite delay } \Omega\text{-machine}\}.$$

The storage schemas we are interested in are pushdown-like. We always have $g(e) = e$ and $g(xA) = A$, indicating that the machine always knows whether the pds (pushdown store) is empty and, if nonempty, what the topmost (rightmost) symbol is. First we define Ω_0-machines, pda's which can erase at one step members of a regular set.

DEFINITION 2.4. Let Γ be a countably infinite set of distinct symbols. For each regular set R in $\mathscr{R}(\Gamma)$, the family of regular sets over Γ^*, let $\langle R \rangle$ be a new symbol. Let $I = \Gamma \cup \{e\} \cup \{\langle R \rangle | R \in \mathscr{R}(\Gamma)\}$. Let $\Omega_0 = (\Gamma, I, f, g)$ be the storage schema

defined by $g(e) = e$, $g(xA) = A$, and $f(x, A) = xA$, $f(x, e) = x$ for A in Γ and x in Γ^*, and $f(xy, \langle R \rangle) = x$ if $x \in \Gamma^*$, $y \in R$ and if $y = y_1 y_2$ and $y_2 \in R$, then $y_1 = e$; $f(x, \langle R \rangle)$ is undefined if $x \notin \Gamma^* R$. Let $\mathscr{C} = \mathscr{L}_r^t(\Omega)$.

An Ω_0-machine has three types of instructions. The instruction e leaves the pds alone. The instruction A adds A to the top of the pds (without erasing anything already on the pds). If $R \in \mathscr{R}(\Gamma)$, the instruction $\langle R \rangle$ erases from the pds the rightmost member of R; if the pds contents is not a member of $\Gamma^* R$, the computation blocks.

It should be clear that by adding new states to compute the reversal of R for $\langle R \rangle$ instructions, we can get a deterministic pda to simulate an Ω_0-machine. Similarly we can simulate any deterministic pda with an Ω_0-machine, so $\mathscr{L}_d(\Omega_0)$ is the family of deterministic context-free languages. Using a very different notation, Schützenberger has shown [18] that every deterministic context-free language can be accepted in realtime by a deterministic Ω_0-machine, and so $\mathscr{L}_r^t(\Omega_0) = \mathscr{L}_d(\Omega_0)$. Hence $\mathscr{C}$ is the family of deterministic context-free languages.

To establish our hierarchy, we shall use a subclass of Ω machines where the erase instructions $\langle R \rangle$ say, in effect, "jump back past the first occurrence of A" for some letter A.

DEFINITION 2.5. For $n \geqq 1$, let $\Gamma_n = \{A_1, \cdots, A_n\}$ and $I_n = \Gamma_n \cup \{e\} \cup \{E_1, \cdots, E_n\}$, where the A_i and E_j are all distinct. Let $\Omega_n = (\Gamma_n, I_n, f_n, g_n)$ be the storage schema defined by $g_n(xA_i) = A_i$, $g_n(e) = e$, $f_n(x, A_i) = xA_i$ for $x \in \Gamma_n^*$, $1 \leqq i \leqq n$, $f_n(x, e) = x$ for $x \in \Gamma_n^*$ and $f_n(xA_i y, E_i) = x$ for $x \in \Gamma_n^*$, $y \in (\Gamma_n - \{A_i\})^*$, $1 \leqq i \leqq n$; if $x \in (\Gamma_n - \{A_i\})^*$, then $f_n(x, E_i)$ is undefined. Call Ω_n-machines pda's *with jumps* or JPDA's. Let $\mathscr{C}_n = \mathscr{L}_r^t(\Omega_n)$.

We now wish to argue that $\mathscr{C} = \bigcup_{n \geqq 1} \mathscr{C}_n$, so that simple jumps suffice to accept with finite delay any deterministic context-free language. This result is not really novel or surprising. It is closely related to Cole's result using tabulator machines [8]. The primary difference is that a JPDA uses the same symbols both as pushdown store symbols and as tabs, so that while an n tab machine can add to n tabs any number of pds symbols, a member of Ω_n has just n symbols. Other less important distinctions are that jpda's have endmarkers and accept by final state and empty storage and jump *below*, not *to* the first occurrence of a "tab". In general, a finite delay tabulator machine can simulate a finite delay Ω_n-machine using n tabs; a finite delay n tab machine can be simulated by a finite delay Ω_{n+3}-machine, using two extra symbols to handle the tabulator's extra pds symbols and one to empty the store at the end. Reference [8] restricts attention to machines that always halt, while we do not; for these pda variants, this makes no difference.

Thus we state Theorem 2.1 without proof, using [8] for justification. A complete, detailed and independent proof can be found in the original version of this paper [32].

THEOREM 2.1. $\mathscr{C} = \bigcup_n \mathscr{C}_n$.

Now one convenience of using the general notion of storage schema is that we can immediately deduce certain closure properties of $\mathscr{L}_r^t(\Omega)$ and $\mathscr{L}_d(\Omega)$ for every storage model Ω we use.

DEFINITION 2.6. A gsm is a sextuple $M = (K, \Sigma, \Delta, \delta, \lambda, q_0)$, where K, Σ and Δ are finite sets of *states*, *inputs* and *outputs*, δ, the *transition* function, is a function

from $K \times \Sigma$ into K, and λ, the *output* function, is a function from $K \times \Sigma$ into Δ^* and $q_0 \in K$. We extend δ and λ to $K \times \Sigma^*$ by $\delta(q, e) = q$, $\lambda(q, e) = e$, $\delta(q, xa) = \delta(\delta(q, x), a)$ and $\lambda(q, xa) = \lambda(q, x)\lambda(\delta(q, x), a)$, for $q \in K$, $x \in \Sigma^*$ and $a \in \Sigma$. If $\delta(q, a) = p$, $\lambda(q, a) = y$, we sometimes write $(q, a) \to (p, y)$ and if $\delta(q, w) = p$, $\lambda(q, w) = y$, we write $(q, w) \overset{*}{\Rightarrow} (p, y)$ for $p, q \in K$, $a \in \Sigma$, $w \in \Sigma^*$, $y \in \Delta^*$. We define $M(w) = \lambda(q_0, w)$, $M^{-1}(w) = \{y | M(y) = w\}$, $M(L) = \{M(w) | w \in L\}$ and $M^{-1}(L) = \{y | M(y) \in L\}$ for a word w or a language L, and call $M(L)$ a gsm *mapping* and $M^{-1}(L)$ an *inverse* gsm *mapping* of L.

DEFINITION 2.7. An AFDL is a family of languages containing at least one nonempty e-free[6] language and closed under inverse gsm mappings, marked union and marked star,[7] and removal of endmarkers.[8] The least AFDL containing a language L is denoted by $\mathscr{F}_d(L)$ and is called a *principal* AFDL.

PROPOSITION 2.1 (Chandler [14]). *For any storage schema* Ω, $\mathscr{L}_d(\Omega)$ *and* $\mathscr{L}_r^t(\Omega)$ *are* AFDL's. *Any* AFDL *is closed under derivatives,*[9] *intersection with regular sets, and concatenation with single symbols.*

COROLLARY. $\mathscr{C}$ *is an* AFDL; *for each* $n \geqq 1$, $\mathscr{C}_n$ *is an* AFDL.

We shall show that $\mathscr{C}_n \subsetneq \mathscr{C}_{n+1}$ for each n, and hence $\mathscr{C}$ is not a principal AFDL. This is rather surprising since context-free languages do form a principal AFDL [20]. It highlights some of the distinctions between deterministic and nondeterministic systems. Context-free languages can be accepted in quasirealtime using only finitely many symbols and instructions.

3. The hierarchy theorems. In this section we show that $\mathscr{C}_{n+1} - \mathscr{C}_n \neq \varnothing$ for each n. The language L_{n+1} we show to be in $\mathscr{C}_{n+1} - \mathscr{C}_n$ is the obvious generalization of a Dyck set on $n + 1$ letters to add "wipeout" or "long range cancel" symbols.

DEFINITION 3.1. For $n \geqq 1$, let $\Sigma_{n,+} = \{a_1, \cdots, a_n\}$, $\Sigma_{n,-} = \{\bar{a}_1, \cdots, \bar{a}_n\}$, $\Sigma_{n,e} = \{E_1, \cdots, E_n\}$ and $\Sigma_n = \Sigma_{n,+} \cup \Sigma_{n,-} \cup \Sigma_{n,e}$. We define a relation $\overset{*}{\sim}$ on $\Sigma_n^* \times \Sigma_n^*$ inductively as follows: (i) $\overset{*}{\sim}$ is an equivalence relation; (ii) for $1 \leqq i \leqq n$, $w_1, w_2 \in \Sigma_n^*$, $w_1 a_i \bar{a}_i w_2 \overset{*}{\sim} w_1 w_2$; (iii) for $1 \leqq i \leqq n$, $w_1, w_2 \in \Sigma_n^*$; if $x \in (\Sigma_{n,+} - \{a_i\})^*$, then $w_1 a_i x E_i w_2 \overset{*}{\sim} w_1 w_2$. Let $L_n = \{w \in \Sigma_n^* | w \overset{*}{\sim} e\}$.

It is evident that L_n is in $\mathscr{C}_n$ for all n. The intuitive reason why L_{n+1} is not in $\mathscr{C}_n$ should be clear. In order for deterministic finite delay machines to suffice, we must have "jumps" E_k [1], [21], [22]. Now since we are allowed finite delay, there is no problem in using a small number of symbols—even two—to encode a_i or $\bar{a}_i$ in a standard way. But the "jumps" cause a problem since we jump to a *symbol* and not to the encoding *string*. So if, say, a_i is encoded by $A_2^i A_1$ on the pds and a_j by $A_2^j A_1$ and on seeing E_i we want to jump through $A_2^i A_1$, the best we can do is execute an E_1 jump and see how many A_2's lie below, until we have passed $A_2^i A_1$. But since any number of a_j's can intervene, this destroys the finite delay property. The reader can try other more sophisticated encodings and verify that the same

---

[6] An *e-free* language does not contain the empty word.

[7] If $L_1 \cup L_2 \subseteq \Sigma^*$ and $c \notin \Sigma$, then $L_1 \cup cL_2$ is a *marked union* of L_1 and L_2 and $(L_1 c)^*$ a *marked star* of L_1.

[8] If $c \notin \Sigma$ and $L \subseteq \Sigma^* c$, then c is an *endmarker* of L and removing c yields $L/c = \{w | wc \in L\}$.

[9] If L is a language and w a word, the *left derivative* of L by w is $w/L = \{y | wy \in L\}$ and the *right derivative* $L/w = \{y | yw \in L\}$.

problem occurs. The advantage of finite delay in permitting encodings not possible in realtime is destroyed by the disadvantage of jumping to a symbol not a string.

This is, alas, not a formal proof. As a first step we might like to show that $\mathscr{C}_n$ is not closed under union with regular languages because it does not contain $L = L_n d \cup \Sigma_n^* c$. One's intuition says that one pds symbol at the bottom must be "reserved" in order to empty the store and accept whenever c appears, so in effect, only $n - 1$ pds symbols can be used to recognize L_n. There are some subtle difficulties involved, and in Lemma 3.1 we show, rather painfully, that if L is in $\mathscr{C}_n$, then L_n can be done by an Ω_{n-1}-machine which is also "allowed" up to k occurrences of A_n on the pds for some fixed k. We formally define this concept.

DEFINITION 3.2. For $n \geqq 2$, $k \geqq 1$, let $\Omega_{n,k} = (\Gamma_n, I_n, f_{n,k}, g_n)$, where for $x \in \Gamma_n^*$, $f_{n,k}(x, A_i) = xA_i$ for $1 \leqq i \leqq n-1$, $f_{n,k}(x, A_n) = xA_n$ if x contains at most $k-1$ occurrences of A_n, $f_{n,k}(x, A_n)$ is undefined if $x \in (\Gamma_{n-1}^* A_n)^k \Gamma_n^*$, and $f_{n,k}(x, e) = x$ and $f_{n,k}(x, E_i) = f_n(x, E_i)$ for all $x \in \Gamma_n^*$ and all i, $1 \leqq i \leqq n$. Let $\mathscr{C}_{n,k} = \mathscr{L}_r^t(\Omega_n)$.

Thus an $\Omega_{n+1,k}$-machine, like an Ω_n-machine, has at its disposal $n-1$ pds symbols $(A_1, \cdots, A_n)$ and so n kinds of jumps $(E_1, \cdots, E_n)$, which can be used without restriction. In addition, it can have on its tape up to k occurrences (but no more) of an $(n+1)$st symbol (A_{n+1}) and thus use up to k occurrences in a row of an $(n+1)$st jump (E_{n+1}).

We shall show that as k grows and more type $n+1$ jumps are allowed, we can handle more and more of L_{n+1} but never all of L_{n+1}. Again, it is intuitively clear that if a "large" word in L_{n+1} contains k nested occurrences of E_{n+1}, we will need k jumps of type E_{n+1} to recognize it correctly. Thus words in L_{n+1} of the form

$$a_{n+1} x_1 \cdots a_{n+1} x_k b_k y_{k-1} b_{k-1} \cdots y_1 b_1,$$

with $b_1, \cdots, b_k \in \{\bar{a}_{n+1}, E_{n+1}\}$ and $x_1, \cdots, x_k, y_1 \cdots y_{k-1} \in \Sigma_n^*$, "need" k E_{n+1} jumps and form a language in $\mathscr{C}_{n+1,k}$ but not $\mathscr{C}_{n+1,k-1}$. Indeed, Lemma 3.2 allows us to conclude that if this is true for $k-1$, it is true for k; however, the formal argument is far from simple.

We shall show that for $n \geqq 2$ and all k, $\bigcup_r \mathscr{C}_{n,r} \subsetneq \mathscr{C}_t \subsetneq \mathscr{C}_{n+1,1}$ and $\mathscr{C}_{n,k} \subsetneq \mathscr{C}_{n,k+1}$. The strategy is to observe that the inequalities hold at the bottom ($\bigcup_r \mathscr{C}_{2,r} \subsetneq \mathscr{C}_2$ and $\mathscr{C}_{2,k} \subsetneq \mathscr{C}_{2,k+1}$), because they hold in the nondeterministic case, and then use two lemmas, 3.1 and 3.2, to obtain the general case by a double induction (on n and k).

We shall find convenient the fact that by Proposition 2.1 each $\mathscr{C}_{n,k}$ is an AFDL and hence $\bigcup_{k \geqq 1} \mathscr{C}_{n,k}$ is an AFDL for each $n \geqq 2$.

First we notice that $\mathscr{C}_1$ contains only one counter languages and that $\mathscr{F}_{k,\omega}$, the family of nondeterministic k counter languages described in [12], is the homomorphic closure of $\mathscr{C}_{2,k}$ [16], [25], so since $\mathscr{F}_{k,\omega} \neq \mathscr{F}_{k+1,\omega}$, obviously $\mathscr{C}_{2,k} \subsetneq \mathscr{C}_{2,k+1}$ [12]. Finally, L_2 is in $\mathscr{C}_2 - \bigcup_k \mathscr{C}_{2,k}$ since $L_2 \cap (\Sigma_{2,+} \cup \Sigma_{2,-})^*$ is not in any $\mathscr{F}_{k,\omega}$ and so a fortiori not in any $\mathscr{C}_{2,k}$ [12].

As a first step in chaining our way up, we show that if L_{n+1} is in $\mathscr{C}_n$, then L_n is in $\bigcup_k \mathscr{C}_{n,k}$ and that L_n is in $\bigcup_k \mathscr{C}_{n,k}$ if and only if $\mathscr{C}_n$ is closed under union with regular sets. These results are immediate consequences of the following technical lemma.

LEMMA 3.1. *Let* $L_1, L_2 \subseteq \Sigma^*$ *be languages such that* Init $L_1 \subseteq L_2$.[10] *Let* c, d *be new. For* $n \geqq 2$, *if* $L_1c \cup L_2d$ *is in* $\mathscr{C}_n$, *then there is a* k *such that* L_1 *is in* $\mathscr{C}_{n,k}$.

Proof. If $L = L_1c \cup L_2d$ is in $\mathscr{C}_n$, then $L = L(D)$ for some deterministic finite delay Ω_n-machine D. Say D is of delay t. Because the prefixes of L_1 are all in L_2, as long as D hasn't "learned" that it is outside L_1 (that is, as long as D reads initial subwords of any member of L_1), D must be prepared to receive d and wipe out its pds. Since D is of delay t, it can use at most $2t + 2$ jump instructions after reading d (it must process d and the endmarker \$ in at most $2t + 2$ steps). Thus, while reading any word in Init L_1, the pds contents of D must be of the form

$$B_1y_1B_2y_2 \cdots B_ly_l$$

for $l \leqq 2t + 2$, $B_i \in \Gamma_n$, $y_i \in (\Gamma_n - \{B_i\})^*$, $1 \leqq i \leqq l$. In particular, this implies that if the pds contents can be factored as $z_1 \cdots z_l$, where each z_j contains all n symbols of Γ_n (so $z_j \notin (\Gamma_n - \{A_i\})^*$, $1 \leqq i \leqq n$), then $l \leqq 2t + 2$.

Let $k_1 = 2t + 2$. The natural first thought is to construct a deterministic finite delay Ω_{n,k_1}-machine $\bar{D}$ to imitate D on Σ^*c and so accept L_1. The new machine would recode the first $n - 1$ symbols on the pds into Γ_{n-1}, use A_n when n symbols have been used, then recode again. At most k_1 disjoint sections on the pds can contain all n symbols, so $\bar{D}$ can remember all k_1 recodings in its finite state control. There is, however, a difficulty in this construction. $\bar{D}$ must know at all times how many A_n's it has on the pds and how many kinds of symbols follow the last A_n. But with no ability to "mark" pds symbols, $\bar{D}$ can lose track of this when erasing. Worse yet, when a "jump" is executed, $\bar{D}$ doesn't know how many or which symbols are wiped out—so, for example, E_1 could erase several A_n's without $\bar{D}$'s being the wiser.

So we let $k = (k_1 + 1)(n + 1)$, construct $\bar{D}$ as a deterministic finite delay $\Omega_{n,k}$-machine and use the extra A_n's allotted as markers. Call a part of the pds of D containing all n symbols a "section". Each section will be marked in $\bar{D}$ by $A_nA_nA_1A_2 \cdots A_{n-1}$; within a section, the first time symbol A_i appears and is encoded by A_j is marked by $A_jA_nA_1A_2 \cdots A_{n-1}$. Thus when $\bar{D}$ applies, say, E_j, it will jump either to an "ordinary" A_j or to one within a marker. Which case occurs can be determined by examining up to n symbols on the pds. If, for example, $\bar{D}$ finds $A_nA_nA_1 \cdots A_{j-1}$ below, it knows that it must execute another E_j, and it will then be in the section below, where a different coding may obtain. This will cause at most $n - 1$ extra jumps, so $\bar{D}$ will still be finite delay.

Thus the actions of $\bar{D}$ may be roughly sketched below; the full construction is exceedingly long and would not aid the reader's understanding.

1. $\bar{D}$ simulates D on input Σ^*; when it reaches its endmarker, \$, it simulates D on c\$. If D adds symbols to the pds on c\$, $\bar{D}$ carries out this part of the simulation in its finite state control alone.

2. When D puts down its first symbol, say A_3, $\bar{D}$ places $A_1A_nA_1 \cdots A_{n-1}$ on its pds and remembers that in this section, A_1 is A_3. Subsequent appearances of A_3 in this section are simply encoded as A_1. The next symbol to be used by D, say A_4, is recorded by $\bar{D}$ as $A_2A_nA_1 \cdots A_{n-1}$ and subsequently as A_2, and $\bar{D}$ remembers the encoding: "A_2 is A_4". During this procedure, $\bar{D}$ knows which section it is dealing with and how many distinct symbols have appeared.

[10] Init $L = \{w | \exists y, wy \in L\}$.

3. Before each simulation of a step of D, if the top symbol of $\bar{D}$ is A_{n-1}, $\bar{D}$ examines the top $n+1$ symbols to see if they are $A_jA_nA_1 \cdots A_{n-1}$; if not, $\bar{D}$ acts like D decoding A_{n-1}; if it is, it decodes A_j to simulate D.

4. When the nth symbol within a section appears—say A_2—$\bar{D}$ marks it as $A_nA_nA_1 \cdots A_{n-1}$ and remembers that a section has been completed in which A_n is A_2. Subsequent appearances of A_2 will *not* be encoded as A_n, for now the count of symbol types begins anew; if A_2 appears again as the ith type, $i < n$, it will be encoded as $A_iA_nA_1 \cdots A_{n-1}$.

5. If $\bar{D}$ ever completes $k_1 + 1$ sections, it blocks.

6. If D wishes to execute an instruction E_j but A_j has not appeared in this section, there are three possibilities. If the current section is the first one, $\bar{D}$ blocks since there is no A_j on the pds of D. If there is a complete section below in which A_j was *not* the nth symbol, $\bar{D}$ executes the jump E_n until the top symbol is A_n (at most $n-1$ jumps are needed) and then executes one more E_n (to erase A_n); now it proceeds as in step 7, knowing that it is in the section below. If there is a complete section below in which A_j was the nth symbol, then $\bar{D}$ jumps through $A_nA_nA_1 \cdots A_{n-1}$ as described above and resumes simulating D.

7. If D wishes to execute an instruction E_j where A_j is encoded by D in this (incomplete) section as A_i, $i \neq n$, then $\bar{D}$ first executes E_i. Next $\bar{D}$ checks to see if the top of the pds is of the form $A_kA_nA_1 \cdots A_{i-1}$. If it is not, $\bar{D}$ restores these $i+1$ symbols and proceeds as before in the knowledge that the right jump has been made, and the section and encoding within the section is unchanged. If the top, however, is $A_kA_nA_1 \cdots A_{i-1}$, clearly $n > k \geqq i$. If $k \neq i$, then $\bar{D}$ executes another E_i jump, and repeats step 7, knowing now that the remainder of this section contains exactly $k-1$ symbols; coding for the other $n-k$ symbols may be changed in accordance with step 2. If the top is $A_iA_nA_1 \cdots A_{i-1}$, $\bar{D}$ removes it and resumes simulating D, knowing that the current section has exactly $i-1$ symbol types.

8. If D is to execute E_j where A_n currently encodes A_j, $\bar{D}$ executes E_n until it has A_n as top symbol, then erases A_n and resumes simulating D, knowing it is in the section below. Note that since each section contains appearances of all n symbols, the procedures in steps 6, 7 and 8 require fewer than n^2 steps, so $\bar{D}$ is still finite delay.

Thus $L_1 = L(\bar{D}) \in \mathscr{C}_{n,k}$ as claimed. □

Now observe that $\bigcup_k \mathscr{C}_{n,k}$ is closed under union with regular sets. If $\mathscr{C}_n$ is closed under union with regular sets, it must contain $Lc \cup \Sigma^* d$ whenever it contains $L \subseteq \Sigma^*$ and $c, d \notin \Sigma$, so $\mathscr{C}_n \subseteq \bigcup_k \mathscr{C}_{n,k}$. Thus we have the following corollary.

COROLLARY 1. *For $n \geqq 2$, $\mathscr{C}_n$ is closed under union with regular sets if and only if $\mathscr{C}_n = \bigcup_k \mathscr{C}_{n,k}$.*

Since $\mathscr{C}_{n+1,1}$ contains the closure of $\mathscr{C}_n$ under union with regular sets, we have another corollary.

COROLLARY 2. *For $n \geqq 2$, $\mathscr{C}_n = \mathscr{C}_{n+1,1}$ if and only if $\mathscr{C}_{n+1,1} = \bigcup_k \mathscr{C}_{n,k} = \mathscr{C}_n$.*

Now

$$L_{n+1} \cap (A_{n+1}\Sigma_n^*\{\bar{A}_{n+1}, \bar{E}_{n+1}\}) = A_{n+1}L_n\bar{A}_{n+1} \cup A_{n+1} \operatorname{Init} L_n\bar{E}_{n+1},$$

so we also have the following.

COROLLARY 3. *For $n \geqq 2$, if $L_{n+1} \in \mathscr{C}_n$, then $L_n \in \bigcup_k \mathscr{C}_{n,k}$.*

So we can conclude that L_3 is not in $\mathscr{C}_2$ because L_2 is not in $\bigcup_k \mathscr{C}_{n,k}$. However, to show that L_4 is not in $\mathscr{C}_3$ requires us to first establish that L_3 is not in $\bigcup_k \mathscr{C}_{3,k}$. The next lemma does this for us. It shows that if L_n is not in $\bigcup_k \mathscr{C}_{n,k}$, then L_{n+1} is not in $\bigcup_k \mathscr{C}_{n+1,k}$ and $\mathscr{C}_{n+1,k} \subsetneq \mathscr{C}_{n+1,k+1}$ for $k \geqq 1$.

The following companion lemma to Lemma 3.1 uses a dichotomy argument similar to those in [26], [27] and [28], but with a twist to take care of the fact that we are dealing here with deterministic systems. The idea behind the proof is quite transparent, but the detailed verification is tedious, relying heavily on special properties of Dyck sets and on Ogden's lemma for deterministic pda's [29]. The interested reader will find the proof in the Appendix.

LEMMA 3.2. *Let $p \geqq 1$, $n \geqq 2$, $k \geqq 1$, $L \subseteq \Sigma^*$ and c, $\not\in$, and d be symbols not in $\Sigma \cup \Sigma_p^*$. Let $\hat{L} = \{\not\in x \not\in w \not\in zc | xz \in \operatorname{Init} L_p, w \in L\} \cup \{\not\in x \not\in w \not\in zd | xz \in L_p, w \in L\}$. If $\hat{L}$ is in $\mathscr{C}_{n+1,k+1}$, then either L_p is in $\bigcup_r \mathscr{C}_{n,r}$ or L is in $\mathscr{C}_{n+1,k}$.*

Finally we establish our main hierarchy result.

THEOREM 3.1. *For all $n \geqq 2, k \geqq 1$, $\bigcup_r \mathscr{C}_{n,r} \subsetneq \mathscr{C}_n \subsetneq \mathscr{C}_{n+1,1}$, and $\mathscr{C}_{n,k} \subsetneq \mathscr{C}_{n,k+1}$, and $L_n \in \mathscr{C}_n - \bigcup_r \mathscr{C}_{n,r} \subsetneq \mathscr{C}_n - \mathscr{C}_{n-1}$.*

Proof. These results have been already established for $n = 2$ and all k. Corollary 2 of Lemma 3.1 tells us that $\bigcup_r \mathscr{C}_{n,r} \neq \mathscr{C}_n$ implies $\mathscr{C}_n \neq \mathscr{C}_{n+1,1}$. Thus we need only show by induction on n and k that $L_n \in \mathscr{C}_n - \bigcup_r \mathscr{C}_{n,r}$ and $\mathscr{C}_{n,k+1} \neq \mathscr{C}_{n,k}$.

Suppose we have shown this for some $n \geqq 2$ and all k. Now we consider $n + 1 \geqq 3$. We define a series of regular sets R_k for $k \geqq 0$ and show that $L_{n+1} \cap R_0 \in \mathscr{C}_{n+1,1} - \mathscr{C}_n$, and show by induction on k that $L_{n+1} \cap R_k \in \mathscr{C}_{n+1,k+1} - \mathscr{C}_{n+1,k}$. This clearly will complete the proof.

Let

$$R_0 = a_{n+1}\Sigma_n^*\{\bar{a}_{n+1}, E_{n+1}\},$$

and for $k \geqq 1$, let

$$R_k = a_{n+1}\Sigma_n^* R_{k-1}\Sigma_n^*\{\bar{a}_{n+1}, E_{n+1}\}.$$

Now

$$L_{n+1} \cap R_0 = a_{n+1}L_n\bar{a}_{n+1} \cup a_{n+1} \operatorname{Init} L_n E_{n+1},$$

so if $L_{n+1} \cap R_0 \in \mathscr{C}_n$, then $a_{n+1}L_n \in \bigcup_r \mathscr{C}_{n,r}$ by Lemma 3.1. Since $\bigcup_r \mathscr{C}_{n,r}$ is an AFDL, it also contains L_n (by Prop. 2.1), which contradicts the induction hypothesis. So $L_{n+1} \cap R_0$ is in $\mathscr{C}_{n+1,1} - \mathscr{C}_n$.

To show by induction on k that $L_{n+1} \cap R_k \in \mathscr{C}_{n+1,k+1} - \mathscr{C}_{n+1,k}$, we notice that each word w in $L_{n+1} \cap R_l$ contains exactly $l + 1$ occurrences of a_{n+1}, one of which starts w, and exactly $l + 1$ occurrences of $\bar{a}_{n+1}$ or E_{n+1}, one of which ends w. Thus there is a gsm M such that

$$M^{-1}(L_{n+1} \cap R_k) = L,$$

where

$$L = \{\not\in x \not\in w \not\in yc | w \in L_{n+1} \cap R_{k-1}, xy \in \operatorname{Init} L_n\}$$
$$\cup \{\not\in x \not\in w \not\in yd | w \in L_{n+1} \cap R_{k-1}, xy \in L_n\}.$$

If $L_{n+1} \cap R_k$ is in $\mathscr{C}_{n+1,k}$, so is L, since $\mathscr{C}_{n+1,k}$ is an AFDL. Since $L_n \notin \bigcup_r \mathscr{C}_{n,r}$, Lemma 3.2 implies that $L_{n+1} \cap R_{k-1}$ is in $\mathscr{C}_{n+1,k-1}$. Hence $L_{n+1} \cap R_{k-1} \notin \mathscr{C}_{n+1,k-1}$ implies $L_{n+1} \cap R_k \notin \mathscr{C}_{n+1,k}$, □

COROLLARY. *For $n \geqq 1$, $\mathscr{C}_n$ is not closed under union with regular sets.*

The next corollary is important enough to state as a theorem.

THEOREM 3.2. *The family of deterministic context-free languages is not a principal* AFDL.

4. Conclusions and open problems. We have shown that the deterministic context-free languages do not form a principal AFDL, although the family of context-free languages is a principal AFDL [20].

We have defined acceptance by empty store and final state. If $\mathscr{C}'_n$ is the family of languages accepted by final state along by Ω_n-machines, clearly $\mathscr{C}_n \subseteq \mathscr{C}'_n \subseteq \mathscr{C}_{n+1,1}$. Since $\mathscr{C}_n$ is not closed under union with regular sets, but $\mathscr{C}'_n$ obviously is, it follows that $\mathscr{C}'_n \neq \mathscr{C}_n$; the reader may use the methods of this paper to show that $\mathscr{C}'_n \subsetneq \mathscr{C}_{n+1,1}$. Thus the fact that $\mathscr{C}'_n \subseteq \mathscr{C}_{n+1,1} \subsetneq \mathscr{C}_{n+1} \subseteq \mathscr{C}'_{n+1}$ shows that we get a similar, though not identical, hierarchy if we consider acceptance by final state alone. (If we define $\mathscr{C}'_{n,k}$ and $\mathscr{C}'$ analogously, we can also show that $\mathscr{C}'_{n,k} \subsetneq \mathscr{C}_{n,k+1} \subsetneq \mathscr{C}'_{n,k+1}$ and $\mathscr{C}'_n \neq \bigcup_k \mathscr{C}'_{n,k}$; of course $\mathscr{C} = \mathscr{C}'$.) So our results are valid under either interpretation; they appear to the author somewhat easier to obtain using $\mathscr{C}_n$.

If we let $\mathscr{T}_n$ be the family of languages accepted by deterministic finite delay n tab tabulator machines in the sense of Cole [8], one can use the methods of this paper to show that $\mathscr{C}_n \subsetneq \mathscr{T}_n \subsetneq \mathscr{C}_{n+2,1}$ and that $\mathscr{T}_n$ and $\mathscr{C}_{n+1}$ are incomparable. Again this establishes the $\mathscr{C}_n$ hierarchy since $\mathscr{T}_n \subsetneq \mathscr{T}_{n+1}$ [8]. This approach seems no simpler than the direct proof that $\mathscr{C}_n \subsetneq \mathscr{C}_{n+1}$. For example, $\mathscr{C}_n \subsetneq \mathscr{T}_n$ because $\mathscr{T}_n$ is closed under union with regular languages but $\mathscr{C}_n$ is not; to show $L_{n+1} \in \mathscr{C}_{n+1} - \mathscr{T}_n$ is at least as tedious as showing $L_{n+1} \in \mathscr{C}_{n+1} - \mathscr{C}_n$.

It is also open whether the family of realtime context-free languages or the various $\mathscr{C}_n$, $\mathscr{C}'_n$ or $\mathscr{C}_{n,k}$ are principal AFDL's. Again we conjecture that they are not. One could ask similar questions about deterministic stack [30], checking automata [27], or nested stack [31] languages and expect similar results. For example, one would expect that deterministic finite delay stack automata with jumps would lead to similar hierarchies.

Two rather more interesting questions suggested by our results are whether similar "jump" hierarchies exist for single tape Turing machines, and what an appropriate algebraic structure theory for the deterministic context-free languages would be.

Suppose we consider deterministic finite delay machines with one Turing tape as working tape and add some instructions of the form $J_L(A)$ or $J_R(A)$—jump to the first A to the left or the right. Do we get a hierarchy by restricting the *types* (i.e., the symbol A jumped to) of jumps or the number of jumps performed? An analogue of Lemma 3.1 seems plausible for Turing machines, but Lemma 3.2 appears harder. For multitape Turing machines, the extra speed afforded by any fixed number of tabs can be achieved without tabs by adding extra tapes [33]; to the best of the author's knowledge this is the best result along these lines to date.

The results of this paper and of [6] make it apparent that AFDL's are not a sufficient framework for studying the structure of the deterministic context-free languages. It might be useful, for example, to find a reasonable set of operations $\mathcal{O}$ such that $\mathscr{C}$ is the least $\mathcal{O}$-closed family containing, say, the Dyck sets. A similar question concerns building up deterministic context-free languages from regular sets without going out of $\mathscr{C}$—finding an analogue to nested iterated substitution as discussed in [11].

Appendix. In this section we give the proof of Lemma 3.2. First, however, we need some more notation and a technical lemma regarding Dyck sets with wipeouts. For w in Σ_n^*, let $\mu(w)$ be the smallest x in Σ_n^* with $\mu(w) \overset{*}{\approx} x$. The usual arguments regarding Dyck sets show that $\mu(w)$ is uniquely defined. Let $v(w) = \max\{|\mu(x)| \; \exists y, w = xy\}$.

We need a lemma which says that if we "pinch out" of words in L_n subwords w in L_n with $v(w)$ "small", then we can use an inverse gsm mapping to put them back in place. This is necessary, for it is only a "pinched" version of L_p that we shall show to be in $\bigcup_k \mathscr{C}_{n,k}$.

Note that for t and n fixed, the set

$$R_{n,t} = \{w \in L_n | v(w) < t, |w| \geqq 2t\}$$

of words "small" in "height" ($v(w) < t$) but long in "width" ($|w| \geqq 2t$) is regular. In general, an AFDL is not closed under substitution by regular sets; our lemma shows that in this special case, we can put $R_{n,t}$ back. The reason is that a gsm can pinch out $R_{n,t}$. The condition $|w| \geqq 2t$ is necessary because every nonempty word in L_n must contain a subword w with $v(w) < t$, so we can only eliminate cases with $|w|$ large.

LEMMA A. *Let $t, n \geqq 2$ and $R_{n,t} = \{w \in L_n | v(w) < t, |w| \geqq 2t\}$ and $L'_{n,t} = L_n \cap (\Sigma_n^* - \Sigma_n^* R_{n,t} \Sigma_n^*)$. There is a* ·gsm *$M$ such that $M^{-1}(L_n d) \subseteq L_n d$ and $M(L_n d) \subseteq L'_{n,t} d$. Hence if L is any language with $L'_{n,t} \subseteq L \subseteq L_n$, then $L_n d = M^{-1}(Ld)$ and L_n is in $\mathscr{F}_d(L)$.*

Proof. The basic idea is that M runs over w in Σ_n^* "pinching out" subwords in the regular set $R_{n,t}$ and making sure that $M(w)$ always "climbs" in steps of exactly t. To this end, M stores $\mu(w)$ in its finite state control as long as $\mu(w) \in \Sigma_{n,+}^*$ and $|\mu(w)| \leqq 2t$. If $\mu(w) \in \Sigma_{n,+}^*$ and $\mu(w) = uv$, where $|u| = t$ and $|v| = t + 1$, then M outputs u, the first t symbols of $\mu(w)$, stores $\mu(v)$ and continues computing $\mu(vz)$ for new input z. If $\mu(w) = w'E_i$, $w' \in \Sigma_{n,+}^*$, M outputs E_i, forgets w', and computes $\mu(z)$ for further input z; the justification for this action is that if the output from or input to M to date lies in Init L_n, then M has produced as output an occurrence of a_i to match E_i, and E_i "jumps over" and cancels w' in between. If $\mu(w) = w'\bar{a}_i$, then M will either output a "garbage" symbol c for $w' \neq e$ or output $\bar{a}_i$ for $w' = e$. Observe that since $e \notin R_{n,t}$, e is in $L'_{n,t}$, so $R_{n,t}$ itself will be mapped into $d \in L'_{n,t} d$.

Since the construction of M is fairly clean, we give it in full. The state set of M is

$$\{\langle \text{STOP} \rangle\} \cup \{\langle u \rangle | 1 \leqq |u| \leqq 2t, u \in \Sigma_{n,+}^*\}$$

with initial state $\langle e\rangle$; the input vocabulary is $\Sigma_n \cup \{d\}$ and the output vocabulary, $\Sigma_n \cup \{c, d\}$. The transitions of M are given below, in notation explained in Definition 2.6.

For $1 \leqq i \leqq n$, state $\langle u\rangle$:

$$(\langle u\rangle, a_i) \to \begin{cases} (\langle ua_i\rangle, e), & |u| \leqq 2t-1, \\ (\langle va_i\rangle, u'), & u = u'v, |u'| = |v| = t, \end{cases}$$

$$(\langle u\rangle, a_i) \to \begin{cases} (\langle u'\rangle, e), & u = u'a_i, \\ (\langle e\rangle, \bar{a}_i), & u = e, \\ (\langle \text{STOP}\rangle, c), & u = u'a_k, \quad k \neq i \end{cases}$$

$$(\langle u\rangle, E_i) \to \begin{cases} (\langle \mu(uE_i)\rangle, e), & \mu(uE_i) \in \Sigma_{n,+}^*, \\ (\langle e\rangle, E_i) & \text{otherwise}, \end{cases}$$

$$(\langle u\rangle, d) \to \begin{cases} (\langle \text{STOP}\rangle, d), & u = e, \\ (\langle \text{STOP}\rangle, c), & u \neq e, \end{cases}$$

$$(\langle \text{STOP}\rangle, b) \to (\langle \text{STOP}\rangle, c), \qquad \text{all } b \in \Sigma_n \cup \{d\}.$$

We leave to the reader the detailed verification that $M^{-1}(L_n d) \subseteq L_n d$ and $M(L_n d) \subseteq L'_{n,t} d$. The points to make are that for $u \in \Sigma_{n,+}^*$, w, $z \in \Sigma_n^*$, if $(\langle e\rangle, w) \overset{*}{\Rightarrow} (\langle u\rangle, z)$ and either w or z is in Init L_n, then $\mu(w) = \mu(zu)$, and that M gives output in $\Sigma_{n,+}^*$ only in blocks of exactly t symbols. □

LEMMA 3.2. *Let $p \geqq 1, n \geqq 2, L \subseteq \Sigma^*$ and $c, \not\!c$, and d be symbols not in $\Sigma \cup \Sigma_p$. Let $\hat{L} = \{\not\!cx\not\!cw\not\!czc | xz \in \text{Init } L_p,\ w \in L\} \cup \{\not\!cx\not\!cw\not\!czd | xz \in L_p,\ w \in L\}$. If $\hat{L}$ is in $\mathscr{C}_{n+1,k+1}$, then either L_p is in $\bigcup_k \mathscr{C}_{n,k}$ or L is in $\mathscr{C}_{n+1,k}$.*

Proof. We can clearly assume $L \neq \varnothing$. Suppose $\hat{L} = L(D)$ for a deterministic finite delay $\Omega_{n+1,k+1}$ machine D with endmarker \$. Consider the behavior of D on input in $\not\!c\Sigma_p^*$. If D never uses pds symbol A_{n+1} before reading the second $\not\!c$, we let w_0 be any member of L and construct a deterministic finite delay Ω_n-machine D_1 as follows. Machine D_1 receives input in $\not\!c\Sigma_p^*\{c, d\}$, and imitates D until it reads c or d. When it reads either c or d, it imitates D on $\not\!cw_0\not\!cc\$$ or $\not\!cw_0\not\!cd\$$, respectively. Since $|w_0| + 4$ is finite, and D is finite delay, D_1 can remember any further pds additions in its finite state control. Hence $L(D_1) = \not\!cL_p d \cup \not\!c$ Init $L_p c \in \mathscr{C}_n$, so $L_p \in \bigcup_k \mathscr{C}_{n,k}$ by Lemma 3.1 and the fact that $\mathscr{C}_n$ and $\bigcup_k \mathscr{C}_{n,k}$ are AFDL's [14].

So now we let D use pds symbol A_{n+1} at least once before the second $\not\!c$. Suppose that $xy \in$ Init L_p, the pds contents of D is $\alpha_1 A_{n+1}$ after reading $\not\!cx$ with $\alpha_1 \in \Gamma_n^*$, and that D does not erase A_{n+1} during the scan of y.

First suppose that for some such x and y and every w in L, D does not erase this first A_{n+1} during the subsequent scan of $\not\!cw\not\!c$. This means that D has at most k additional instances of A_{n+1} on its pds during any time in its scan of $\not\!cw\not\!c$. Hence we can build a deterministic finite delay $\Omega_{n+1,k}$-machine D_2 to accept cLc. The machine D_2 starts out by storing in its finite state control the pds and state of D after reading $\not\!cxy$. Then it simulates D on input $\not\!cw\not\!c$, $w \in \Sigma^*$; during this time it does not touch the pds "bottom" stored in its finite state control. Finally, when D_2 sees its endmarker, it simulates D on $c\$$, accepting when D accepts. Since

$xy \in \text{Init } L_p$, $L(D_2) = ¢L¢ \in \mathscr{C}_{n+1,k}$. Because AFDL's are closed under derivatives, L is in $\mathscr{C}_{n+1,k}$.

Thus if L is *not* in $\mathscr{C}_{n+1,k}$, whenever the pds of D contains A_{n+1} after the scan of a member of Init L_p, there is some w_0 in L such that D erases A_{n+1} during the scan of $¢w_0¢$.

We want to show that in this case, D cannot do much work after using A_{n+1}, since A_{n+1} is erased by $¢w_0¢$, and D only sees subwords with v small. To do so, we notice that D can be imitated by a deterministic pda with some extra states for "jumping". Hence we can appeal to a strong form of Ogden's lemma for deterministic context-free languages obtained by a careful scrutiny of the proof of [27].

Essentially, the result is as follows. There is an integer $m_0 \geqq 2$ with these properties. Suppose $x'ywz$ is in $L(D)$, the pds contents after reading x' is αA, and this A is not erased during the scan of y but is erased during the scan of w. Further, let $|y| \geqq m_0$ and $z \neq e \neq w$. If any m_0 or more places in y are distinguished, then one of two situations obtains.

I. We have $y = y_1y_2y_3$, $w = w_1w_2w_3$, y_1, y_2 and y_3 all contain distinguished positions but no more than m_0 lie within y_2y_3, and $x'y_1y_2^ly_3w_1w_2^lw_3'z'$ is in $L(D)$ for all $l \geqq 0$ and for all z' and w_3' such that $x'yw_1w_2w_3'z' \in L(D)$; further, *D does not erase A while scanning $y_1y_2^ly_3$*.

II. We have $y = y_1y_2y_3y_4y_5$ where y_3 and either y_1 and y_2 or y_4 and y_5 contain distinguished positions but no more than m_0 lie within $y_2y_3y_4$, and $x'y_1y_2^ly_3y_4^ly_5'w'z' \in L(D)$ for all $l \geqq 0$ and for all y_5', w' and z' with $x'y_1y_2y_3y_4y_5'w'z' \in L(D)$; *further, D does not erase A during the scan of $y_1y_2^ly_3y_4^l$*.

In our application, we let x' be $¢x$, and $A = A_{n+1}$, w be $¢w_0¢$ and let $z = z_1d \in \Sigma_p^*d$ where $¢x$ deposits pds contents α_1A_{n+1}; A_{n+1} is not erased during the scan of y but is erased during the scan of $¢w_0¢$, and $z_1 \in \Sigma_{p,-}^*$ is such that $xyz_1 \in L_p$. We want to show that $|\mu(u)| < m_0$ for any subword u of y.

First suppose that $y = u_1B_1u_2B_2u_3 \cdots B_mu_{m+1}$ for $m \geqq m_0$, $B_1, \cdots, B_m \in \Sigma_{p,+}$ and $\mu(B_1u_2 \cdots B_m) = B_1 \cdots B_m$; that is, y has a subword with μ "positive" (in $\Sigma_{p,+}^*$) and of size m_0 or greater. Then $u_2, \cdots, u_m \in L_p$. Let $y' = u_1B_1u_2 \cdots B_m$. Since D does not erase A_{n+1} during the scan of y', it does erase it during $¢w_0¢$ for some $w_0 \in L$. Let $z_1 \in \Sigma_{p,-}^*$ be such that $xy'z_1 \in L_p$, call the B_j distinguished, and apply Ogden's lemma to $¢xy'¢w_0¢z_1d \in L(D)$.

If case I obtains, $y' = y_1y_2y_3$, $¢w_0¢ = w_1w_2w_3$, where y_1, y_2 and y_3 all contain B_j's and $¢xy_1y_2^ly_3w_1w_2^lw_3z_1d \in L(D)$ for all $l \geqq 0$. So $xy_1y_2^*y_3z_1 \subseteq L_p$. But z_1 is in $\Sigma_{p,-}^*$, and y_2 and y_3 lie within the B_j, so $|\mu(y_2^l)| \geqq l$, and y_3z_1 does not contain any E_k which can "jump" over y_2^l. This is a contradiction of the definition of L_p.

So we must have case II. In this case, $y' = y_1y_2y_3y_4y_5$, y_1, y_2 and y_3 contain B_j's or y_3, y_4 and y_5 contain B_j's and $¢xy_1y_2^ly_3y_4^ly_5'¢w_0¢z_1d \in L(D)$ and $xy_1y_2^ly_3y_4^ly_5'z_1 \in L_p$ for all $l \geqq 0$ and y_5' with $xy_1y_2y_3y_4y_5'z_1$ in L_p. In particular, we can have $y_5'z_1 \in \Sigma_{p,-}^*$. Now $y_5'z_1$ contains some $\bar{a}_k$ canceled by an $a_k = B_j$ in y_2 or y_4, and so $\mu(xy_1y_3y_5'z_1) \neq e$, a contradiction.

Thus y cannot have a subword u with $\mu(u) \in \Sigma_{p,+}^*$ and $|\mu(u)| \geqq m_0$. Suppose y has a subword u with $|\mu(u)| \geqq m_0$, that is, $y = u_1B_1u_2 \cdots B_mu_{m+1}$, $u = B_1u_2 \cdots B_m$, $\mu(u) = B_1 \cdots B_m$, $m \geqq m_0$, and $B_1, \cdots, B_m \in \Sigma_p$. Let $y' = u_1u$. Again, D does not erase A_{n+1} during y' but does so during some $¢w_0¢$, $w_0 \in L$, and we call the B_j

distinguished and apply Ogden's lemma to $\not\in xy'\not\in w_0\not\in z_1 d \in L(D)$, where $xy'z_1 \in L_p$ and $z_1 \in \Sigma^*_{p,-}$. In case I, we have $y' = y_1y_2y_3$, y_1, y_2 and y_3 contain B_j's and, as before, $xy_1y_2^*y_3z_1 \subseteq L_p$. Now if y_2 contains $B_j \in \Sigma_{p,-} \cup \Sigma_{p,e}$ (i.e., B_j is an $\bar{a}_k$ or E_k) and $l = |xy_1| + 1$, then B_j is not canceled within y_2y_2, and so $\mu(xy_1y_2^l) \notin \Sigma^*_{p,+}$, a contradiction. Suppose this does not occur. Then y_2 has a $B_j = a_k$ which is not canceled or "jumped" within $y_1y_2^l$. Now y_3 may contain an E_k which can "jump" over y_2. In this case we notice that Ogden's lemma also tells us that A_{n+1} is not erased during the scan of $y_1y_2^l$. Further, $y'' = y_1y_2^{m_0}$ will contain a subword v with $\mu(v) \in \Sigma^*_{p,+}$ and $|\mu(v)| \geqq m_0$. So we obtain a contradiction by applying our previous argument to $\not\in xy''$.

Finally, in case II we arrive at $y' = y_1y_2y_3y_4y_5$, $y'_5z_1 \in \Sigma^*_{p,-}$, $xy_1y_2^ly_3y_4^ly'_5z_1 \in L_p$ for all $l \geqq 0$ and either y_1, y_2 and y_3 or y_3 and y_4 contain B_j's. Further, D does not erase A_{n+1} during the scan of $y_1y_2^ly_3y_4^l$. Running through all possibilities, we either directly contradict the definition of L_p (e.g., if y_2 contains a B_j in $\Sigma_{p,-} \cup \Sigma_{p,e}$ and y_1 contains a $B_{j'}$) or have a subword v with $\mu(v) \in \Sigma^*_{p,+}$ and $|\mu(v)| \geqq m_0$ in $y_1y_2^{m_0}$ or in $y_3y_4^{m_0}$ (e.g., if y_2 has a B_j in $\Sigma_{p,+}$ and no $B_{j'}$ in $\Sigma_{p,-} \cup \Sigma_{p,e}$ or if y_4 has a B_j in $\Sigma_{p,-} \cup \Sigma_{p,e}$) and obtain a contradiction using our previous argument.

Now there is an integer s_t such that if $|y| > s_t$, and y is in[11] $\mathrm{Sub}(L_p)$ and $|\mu(y')| < t$ for each subword y' of y, then y contains a subword v in L_p with $v(v) < t$ and $|v| \geqq 2t$ (e.g., $s_t = 4t^3$). Let $t = m_0$.

Now we know that if D places A_{n+1} on its pds and then reads y in Σ_p^*, all *before* reading the second $\not\in$ or erasing A_{n+1}, either the input to date is not in $\not\in \mathrm{Init}\, L_p$ or $|\mu(y')| < t$ for each subword y' or y, and either $|y| \leqq s_t$ or y contains a subword in $R_{n,t}$. So either D will erase A_{n+1} in a "short" time (before reading s_{t+1} inputs), or it encounters a member of $R_{n,t}$.

We construct a deterministic finite delay Ω_n-machine D_3 to imitate D as follows. Let w_0 be *any* member of L. D_3 reads input in $\not\in\Sigma_p^*\{c, d\}$. First D_3 simulates D as long as D does not use A_{n+1} on its pds. If D_3 reads c or d without using A_{n+1}, it concludes by imitating D on $\not\in w_0\not\in c\$$ or $\not\in w_0\not\in d\$$ as D_1 did. However, if D puts A_{n+1} in its pds, D_3 remembers A_{n+1} in its finite state control. Then D_3 continues the simulation of D wholly within its finite state control until either (i) it has read $s_t + 1$ input symbols in Σ_p, (ii) before $s_t + 1$ inputs in Σ_p occur, D erases through A_{n+1}, or (iii) before $s_t + 1$ inputs in Σ_p occur, c or d occurs as input. In case (i), D_3 simply blocks in the knowledge that either it has left $\not\in \mathrm{Init}\, L_p$ or else it has encountered a subword in $R_{n,t}$. In case (ii), D_3 can now resume the direct simulation of D as before, since now D has pds contents in Γ_n^*. In case (iii), D_3 now concludes by simulating D on $\not\in w_0\not\in c\$$ or $\not\in w_0\not\in d\$$ as discussed before; in all cases, this is what D_3 does on encountering input c or d.

Notice that the only blocking situation in D_3 not in D is in case (i) above, which always occurs *before* the occurrence of c or d. Hence there is some language $L' \subseteq \Sigma_p^*$ such that $L(D_3) = \not\in L'd \cup \not\in \mathrm{Init}\, L'c$. Since D_3 always rejects or blocks if D does, we certainly have $L' \subseteq L_p$. If D_3 blocks and D does not, we have (i) above and know a subword in $R_{n,t}$ has been encountered. Hence $L'_{p,t} \subseteq L'$ where $L'_{p,t}$ is defined in Lemma A.

[11] $\mathrm{Sub}(L) = \{y \mid \exists x, z, xyz \in L\}$.

By Lemma 3.1, since $L(D_3) \in \mathscr{C}_n$, $\not\subset L' \in \bigcup_k \mathscr{C}_{n,k}$ so $L' \in \bigcup_k \mathscr{C}_{n,k}$. But $\bigcup_k \mathscr{C}_{n,k}$ is an AFDL [14], so by Lemma A, L_p is in $\bigcup_k \mathscr{C}_{n,k}$, □

Acknowledgments. An earlier version of this paper appears in [32] together with material in [20]. The author would like to thank the referees of [32] for helpful suggestions and for pointing out the close connection with work in [8].

REFERENCES

[1] S. GINSBURG AND S. A. GREIBACH, *Deterministic context-free languages*, Information and Control, 9 (1966), pp. 602–648.
[2] S. L. GRAHAM, *Extended precedence languages, bounded right context languages, and deterministic languages*, Conf. Record 1970 IEEE 11th Annual Sympos. on Switching and Automata Theory, Santa Monica, California, pp. 175–180.
[3] L. H. HAINES, *Generation and recognition of formal languages*, Ph.D. thesis, Mass. Inst. of Tech., Cambridge, Mass., 1965.
[4] D. E. KNUTH, *On the translation of languages from left to right*, Information and Control, 8 (1965), pp. 607–639.
[5] I. M. HAVEL, *Strict deterministic languages*, Ph.D. thesis, Univ. of Calif. at Berkeley, 1971.
[6] S. A. GREIBACH, *Characteristic and Ultrarealtime Languages*, Information and Control, 18 (1971), pp. 65–98.
[7] A. J. KORENJAK AND J. E. HOPCROFT, *Simple deterministic languages*, Conf. Record of IEEE 7th Annual Sympos. on Switching and Automata Theory, Berkeley, Calif., 1966, pp. 36–46.
[8] STEPHEN V. COLE, *Deterministic pushdown store machines and real-time computation*, J. Assoc. Comput. Mach., 18 (1971), pp. 306–328.
[9] R. MCNAUGHTON, *Parenthesis grammars*, Ibid., 14 (1967), pp. 490–500.
[10] D. J. ROSENKRANTZ AND R. E. STEARNS, *Properties of deterministic top down grammars*, Information and Control, 17 (1970), pp. 226–256.
[11] S. GREIBACH, *Full AFLs and nested iterated substitution*, Ibid., 16 (1970), pp. 7–35.
[12] ———, *An infinite hierarchy of context-free languages*, J. Assoc. Comput. Mach., 16 (1969), pp. 91–106.
[13] J. GRUSKA, *A characterization of context-free languages*, J. Comput. System Sci., 5 (1971), pp. 353–364.
[14] W. J. CHANDLER, *Abstract families of deterministic languages*, Proc. 1st ACM Conf. on Theory of Computing, Marina del Rey, Calif., 1969, pp. 21–30.
[15] J. E. HOPCROFT AND J. D. ULLMAN, *An approach to a unified theory of automata*, Bell System Tech. J., 46 (1967), pp. 1793–1829.
[16] S. GINSBURG AND S. GREIBACH, *Abstract families of languages*, Studies in Abstract Families of Languages, Mem. Amer. Math. Soc., 87 (1969), pp. 1–32.
[17] ———, *Principal AFL*, J. Comput. System Sci., 4 (1970), pp. 308–338.
[18] M. P. SCHÜTZENBERGER, *On context-free languages and pushdown automata*, Information and Control, 6 (1963), pp. 217–255.
[19] ———, *Finite counting automata*, Ibid, 5 (1967), pp. 91–107.
[20] S. GREIBACH, *The hardest context-free language*, this Journal, 2 (1973), pp. 304–310.
[21] A. L. ROSENBERG, *On the independence of real-time definability and certain structural properties of context-free languages*, J. Assoc. Comput. Mach., 15 (1968), pp. 672–679.
[22] ———, *Real-time definable languages*, Ibid., 14 (1967), pp. 645–662.
[23] P. C. FISCHER, private communication.
[24] ———, *Turing machines with restricted memory access*, Information and Control, 9 (1966), pp. 364–379.
[25] S. GINSBURG, S. GREIBACH AND J. HOPCROFT, *PreAFL*, Studies in Abstract Families of Languages, Mem. Amer. Math. Soc., 87 (1969), pp. 41–51.
[26] S. GREIBACH, *Chains of Full AFLs*, Math. Systems Theory, 14 (1970), pp. 231–242.
[27] ———, *Checking automata and one-way stack languages*, J. Comput. System Sci., 3 (1969), pp. 196–217.

[28] ———, *Syntactic operators on full semiAFLs*, Ibid., 6 (1972), pp. 30–76.
[29] W. F. OGDEN, *Intercalculation theorems for pushdown store and stack languages*, Ph.D. thesis, Stanford Univ., Stanford, Calif., 1968.
[30] J. E. HOPCROFT AND J. D. ULLMAN, *Deterministic stack automata and the quotient operator*, J. Comput. System Sci., 2 (1968), pp. 1–12.
[31] A. V. AHO, *Nested stack automata*, J. Assoc. Comput. Mach., 16 (1969), pp. 383–406.
[32] S. GREIBACH, *Jump PDA's, deterministic context-free languages, principal AFDL's and polynomial time recognition*, Proc. 5th Annual ACM Conf. on Theory of Computing, Austin, Texas, 1973.
[33] M. J. FISCHER AND A. L. ROSENBERG, *Limited random access Turing machines*, Proc. 9th Annual IEEE Sympos. on Switching and Automata Theory, 1968, pp. 356–367.

SIAM J. COMPUT.
Vol. 3, No. 2, June 1974

POLYNOMIALS WITH RATIONAL COEFFICIENTS WHICH ARE HARD TO COMPUTE*

VOLKER STRASSEN†

Abstract. We present specific polynomials in $\mathbb{C}[x]$ with algebraic or rational coefficients which are hard to compute (even though arbitrary complex numbers are allowed as inputs for the computation). Examples are: $\sum_{\delta=0}^{d} e^{2\pi i/2^\delta} x^\delta$, $\sum_{\delta=0}^{d} 2^{2^\delta} x^\delta$. We also show that the minimum number of arithmetic operations to compute polynomials in $\mathbb{C}[x]$ is itself computable. Finally, we study computational complexity in finite-dimensional algebras over an algebraically closed field.

Key words. concrete complexity, polynomial evaluation, rational coefficients, trade-off

1. Introduction. Motzkin [5] and Belaga [1] have shown that the computation (evaluation) of polynomials $\alpha_d x^d + \cdots + \alpha_0 \in \mathbb{C}[x]$ using infinite-precision arithmetic "in general" requires d additions/subtractions and $d/2$ multiplications, even if arbitrary auxiliary complex numbers can be used without extra cost (the choice of these numbers—as well as the whole computation—will of course depend on the polynomial $\alpha_d x^d + \cdots + \alpha_0$, i.e., on its coefficients; in the literature, one therefore often speaks of the possibility of "preconditioning" the coefficients, or of the "preconditioning model"; in the terminology of [10], one is simply dealing with computations in $\mathbb{C}[x]$ considered as a ring over $\mathbb{C} \cup \{x\}$). "In general" may here be interpreted in two ways:

1. the quoted lower bounds hold for all polynomials of degree d except for a set of Lebesgue measure 0;

2. they hold for all polynomials with algebraically independent coefficients (over the rationals).

Since arbitrary polynomials in $\mathbb{C}[x]$ of degree d may in fact be computed with $3d/2 + 2$ arithmetical operations (see again Motzkin [5], Belaga [1]), the first interpretation gives rather precise information on the typical computational complexity of polynomials of degree d. On the other hand, if one is interested in the complexity of specific polynomials, the second interpretation applies, and the information is much less complete: most polynomials occurring in mathematics or in applications have rational or algebraic coefficients. For such polynomials, so far only the trivial lower bound $\log_2 d$ is available.

In the present paper we derive lower bounds for the computational complexity of polynomials with rational or algebraic coefficients. Here are a few examples: let β be a computation (for definitions and notation, see [10]) that computes

$$\sum_{\delta=0}^{d} 2^{2^{\delta d^2}} x^\delta$$

in the field $\mathbb{C}(x)$ over $\mathbb{C} \cup \{x\}$ (i.e., β may use division and may fetch arbitrary complex numbers). Then either β contains at least $d - 4$ additions/subtractions and at least $(d/2) - 2$ multiplications/divisions, or the total number of arithmetic operations in β is ridiculously large, namely $\geqq d^2/\log_2 d$ (Cor. 2.12). Thus in view

* Received by the editors January 17, 1973, and in final revised form September 21, 1973.

† A first version of this paper was written at the Matematisk Institut, University of Aarhus, Denmark, during the summer 1970. Now at Universität Zürich, Zürich, Switzerland.

of the results of Motzkin and Belaga, the above polynomial is practically as hard to compute as one with algebraically independent coefficients. The intuitive reason for this is, of course, the tremendous growth rate of the coefficient sequence. If we moderate this growth rate, we obtain weaker lower bounds. For example: the total number of arithmetic operations needed for computing

$$\sum_{\delta=0}^{d} 2^{2^\delta} x^\delta$$

is at least $\sqrt{d/(3 \log d)}$ for large d (Cor. 2.11). By contrast, $\sum_{\delta=0}^{d} 2^\delta x^\delta$ can obviously be computed with $O(\log d)$ operations.

Polynomials with fast growing integer coefficients occur as initial segments of generating functions in combinatorial mathematics. A slightly different example of a polynomial of complexity at least $\sqrt{d/\log d}$ is

$$\sum_{\delta=0}^{d} e^{2\pi i/2^\delta} x^\delta.$$

Concerning the proofs, we remark that already the investigations of Motzkin and Belaga imply the existence of a polynomial in $d+1$ indeterminates $P(y_0, \cdots, y_d) \neq 0$ such that if $P(\alpha_0, \cdots, \alpha_d) \neq 0$, then $\alpha_d x^d + \cdots + \alpha_0$ is hard to compute (roughly speaking). It seems very difficult to actually exhibit such a polynomial P. We show, however, with the help of the Dirichlet–Siegel pigeonhole principle, that polynomials P exist which have moderate degree and integer coefficients of absolute value $\leqq 3$. Even such fragmentary information yields the abovementioned lower bounds for the complexity of specific polynomials.

So far we have talked about the results of § 2 of this paper. Before turning to § 3, let us discuss a somewhat different question in a slightly more general context. Let k be an algebraically closed field, $x_1, x_2, \cdots, x_n$ indeterminates over k. We interpret $k(\mathbf{x}) = k(x_1, \cdots, x_n)$ as a field over $k \cup \{x_1, \cdots, x_n\}$ (see [10]). Let z be an integer-valued proper operation-time (cost function) for the type of $k(\mathbf{x})$, which is strictly positive on $+$, $-$, $*$, $/$. The computational complexity L is a function from the set of finite subsets of $k(\mathbf{x})$ to $\mathbb{N} = \{0, 1, 2, \cdots\}$ (intuitively, $L(F)$ is the minimal amount of time that is sufficient to evaluate the set of rational functions F on a computer with a single processor when it takes $z(\omega)$ units of time to perform the operation ω; for details see [10]). We would like to know if L itself is computable. In other words: is it possible to decide if $L(F) \leqq t$ for finite subsets F of $k(\mathbf{x})$ and $t \in \mathbb{N}$?

We treat this question in more detail, even though the rest of the paper is logically independent of the following discussion. First we encode the elements of $k(\mathbf{x})$ into finite sequences over k: we represent rational functions as pairs of polynomials (this representation is not unique since we do not require that the polynomials are relatively prime; further, only pairs whose "denominator-polynomial" is not zero represent rational functions). Let t_0 be a large natural number. If $t \leqq t_0$, then in investigating the problem "$L(F) \leqq t$?" we can restrict our consideration to rational functions which allow a representation by polynomials of degree $\leqq 2^{t_0}$. We represent such polynomials by the $\binom{2^{t_0}+n+1}{n+1}$-

tuples of their coefficients. For $M = 2 \cdot \binom{2^{t_0} + n + 1}{n + 1}$, every $f \in k^M$ whose last $M/2$ coordinates are not all zero corresponds to a rational function $\tilde{f}$.

Now we construct for every $(t, r) \in \mathbb{N}^2$ with $-\max\{z(+), z(-), z(*), z(/)\} \leqq t \leqq t_0$ a formula $\mathscr{A}_{t,r}(\phi_1, \cdots, \phi_r)$ of the elementary theory of fields (the ϕ_ρ's stand for pairwise distinct M-tuples of free variables) such that

$$\begin{aligned} &\forall t \leqq t_0 \quad \forall r \quad \forall f_1, \cdots, f_r \in k^M, \\ &[(\vDash_k \mathscr{A}_{t,r}(f_1, \cdots, f_r)) \Leftrightarrow (\tilde{f}_1, \cdots, \tilde{f}_r \text{ are defined and } L(\{\tilde{f}_1, \cdots, \tilde{f}_r\}) \leqq t)]. \end{aligned} \tag{1.1}$$

We do this by induction on (t, r) with respect to the lexicographical ordering:

$$\mathscr{A}_{t,r}(\phi_1, \cdots, \phi_r) :\equiv 0 = 0 \qquad \text{if } 0 \leqq t \leqq t_0, \quad r = 0,$$
$$\mathscr{A}_{t,r}(\phi_1, \cdots, \phi_r) :\equiv 0 \neq 0 \qquad \text{if } t < 0,$$

and for $0 \leqq t \leqq t_0, r > 0$:

$$\mathscr{A}_{t,r}(\phi_1, \cdots, \phi_r) :\equiv \bigvee_{\substack{\pi \text{ permutation} \\ \text{of } \{1, \cdots, r\}}} \mathscr{A}'_{t,r}(\phi_{\pi 1}, \cdots, \phi_{\pi r}),$$

where

$$\begin{aligned} \mathscr{A}'_{t,r}(\phi_1, \cdots, \phi_r) :\equiv \\ \bigvee \psi_1 \bigvee \psi_2 (\tilde{\phi}_r = \tilde{\psi}_1 + \tilde{\psi}_2 \wedge \mathscr{A}_{t-z(+),r+1}(\phi_1, \cdots, \phi_{r-1}, \psi_1, \psi_2)) \\ \vee (\tilde{\phi}_r = \tilde{\psi}_1 - \tilde{\psi}_2 \wedge \mathscr{A}_{t-z(-),r+1}(\phi_1, \cdots, \phi_{r-1}, \psi_1, \psi_2)) \\ \vee (\tilde{\phi}_r = \tilde{\psi}_1 * \tilde{\psi}_2 \wedge \mathscr{A}_{t-z(*),r+1}(\phi_1, \cdots, \phi_{r-1}, \psi_1, \psi_2)) \\ \vee (\tilde{\phi}_r = \tilde{\psi}_1 / \tilde{\psi}_2 \wedge \mathscr{A}_{t-z(/),r+1}(\phi_1, \cdots, \phi_{r-1}, \psi_1, \psi_2)) \\ \vee ((\tilde{\phi}_r = x_1 \vee \cdots \vee \tilde{\phi}_r = x_n \vee \tilde{\phi}_r \text{ const.}) \wedge \mathscr{A}_{t,r-1}(\phi_1, \cdots, \phi_{r-1})). \end{aligned}$$

Here the ϕ_ρ, ψ_ρ always stand for pairwise distinct M-tuples of logical variables. The expressions "$\tilde{\phi}_r = \tilde{\psi}_1 + \tilde{\psi}_2$, $\tilde{\phi}_r = \tilde{\psi}_1 - \tilde{\psi}_2$, $\tilde{\phi}_r = \tilde{\psi}_1 * \tilde{\psi}_2$, $\tilde{\phi}_r = \tilde{\psi}_1 / \tilde{\psi}_2$, $\tilde{\phi}_r = x_j$, $\tilde{\phi}_r$ const." are abbreviations of formulas of the elementary theory of fields; the exact (and obvious) definition of these formulas is left to the reader. These formulas should also state that the rational functions are defined and that in the case of division, the denominator does not vanish.

In order to prove (1.1), one shows first by induction on (t, r) (for $t \leqq t_0$)

$$\begin{aligned} &\forall f_1, \cdots, f_r \in k^M, \\ &(\vDash_k \mathscr{A}_{t,r}(f_1, \cdots, f_r) \Rightarrow \tilde{f}_1, \cdots, \tilde{f}_r \text{ are defined, and } L(\{\tilde{f}_1, \cdots, \tilde{f}_r\}) \leqq t), \end{aligned}$$

and then by L-induction (see [10]) on the set variable F,

$$\begin{aligned} &\forall t \leqq t_0 \quad \forall r \quad \forall f_1, \cdots, f_r \in k^M, \\ &(\tilde{f}_1, \cdots, \tilde{f}_r \text{ are defined and } F = \{\tilde{f}_1, \cdots, \tilde{f}_r\} \text{ and } L(F) \leqq t \Rightarrow \vDash_k \mathscr{A}_{t,r}(f_1, \cdots, f_r)). \end{aligned}$$

We apply quantifier-elimination to the formulas $\mathscr{A}_{t,r}(\phi_1, \cdots, \phi_r)$ (in the theory of algebraically closed fields; see [9]). This procedure assigns a quantifier-free

formula $\mathscr{B}_{t,r}(\phi_1, \cdots, \phi_r)$ to every (t, r) with $t \leqq t_0$ such that $\vdash(\mathscr{A}_{t,r} \leftrightarrow \mathscr{B}_{t,r})$ and therefore

$$\forall f_1, \cdots, f_r \in k^M(\models_k \mathscr{A}_{t,r}(f_1, \cdots, f_r) \Leftrightarrow \models_k \mathscr{B}_{t,r}(f_1, \cdots, f_r)).$$

Therefore (1.1) remains valid if we replace $\mathscr{A}_{t,r}$ by $\mathscr{B}_{t,r}$. Since quantifier-elimination and the construction of $\mathscr{A}_{t,r}(\phi_1, \cdots, \phi_r)$ are effective procedures, the function

$$(t, r) \mapsto \mathscr{B}_{t,r}(\phi_1, \cdots, \phi_r)$$

is computable. Hence if we can decide if a rational expression in $f_1, \cdots, f_r \in k^M$ vanishes, then we can check for $\models_k \mathscr{B}_{t,r} f_1, \cdots, f_r)$ and hence for $L(\{\tilde{f}_1, \cdots, \tilde{f}_r\}) \leqq t$ (if the $f_1, \cdots, f_r$ are originally given as quotients of polynomials of some degree γ, then choose $t_0 \geqq \max\{t, \log_2 \gamma\}$).

In particular, this is the case for $k = \mathbb{C}$ if the coordinates of $f_1, \cdots, f_r$ are elements of $\mathbb{Q}$. The computational complexity over $\mathbb{C}(\mathbf{x})$ of a system of rational functions with rational coefficients is therefore a computable function of these coefficients. If we encode finite subsets F of $\mathbb{Q}(\mathbf{x})$ by means of the reduced quotient representation of rational functions bijectively by natural numbers $\langle F \rangle$, then L induces a number theoretic function. This function is recursive.

Of course, the situation is completely different if we replace the computational complexity over $\mathbb{C}(\mathbf{x})$ by the complexity over $\mathbb{Q}(\mathbf{x})$. Problem: is the diophantine relation

$$\{(\langle F \rangle, t) : L_{\mathbb{Q}(\mathbf{x})}(F) \leqq t\} \tag{1.2}$$

recursive?

In the discussion above, we can obviously replace the algebraically closed field k by a real closed field. Further, we can use the depth T instead of L (see [10]). We can also weaken the assumptions about z (e.g., we can allow z to assume values in $\mathbb{Q}$: $\mathbb{Q}$-valued operation times with finite range can always be changed to integer-valued operation times by multiplication with a natural number). Finally, we can interpret $k(\mathbf{x})$ as a k-field over $\{x_1, \cdots, x_n\}$ and choose, e.g., $z = 1_{\{*,/\}}$.

Statement (1.1) with $\mathscr{B}_{t,r}$ instead of $\mathscr{A}_{t,r}$ implies that the set $\{(f_1, \cdots, f_r) : L(\{\tilde{f}_1, \cdots, \tilde{f}_r\}) \leqq t\} \subset k^{Mr}$ is constructible in the sense of algebraic geometry (see [6]). In § 3 we prove an analogous statement (Thm. 3.1), without using logic, by means of algebraic geometry; we apply the theorem of Chevalley ([6, p. 97]) instead of quantifier-elimination (which we could have used as well). The situation in § 3 differs from the previous discussion as follows:

1. Division is not allowed, hence $k(\mathbf{x})$ simplifies to $k[\mathbf{x}]$.
2. $k[\mathbf{x}]$ is replaced by $k[\mathbf{x}]/(x_1, \cdots, x_n)^M$. $(x_1, \cdots, x_n)^M$ denotes the ideal which is generated by the monomials of degree M (since M can be chosen to be arbitrarily large, this does not restrict generality).
3. Besides $k[\mathbf{x}]/(x_1, \cdots, x_n)^M$, arbitrary finite-dimensional k-rings A are considered (which do not have to be commutative nor associative); these rings are interpreted as k-rings over a given system of generators.
4. The operation time is $z = 1_{\{*\}}$.

The encoding of A does not create difficulties: A is a finite dimensional k-vector space and therefore an affine space in the sense of algebraic geometry.

After showing that the set

$$\{(a_1, \cdots, a_r): L(\{a_1, \cdots, a_r\}) \leqq t\} \subset A^r$$

is constructible for every r, t, it is desirable to close the set in the sense of the Zariski topology. We define a new function $\mathbf{L}$ from the set of finite subsets of A to $\mathbb{N}$ such that

$$(1.3)\quad \{(a_1, \cdots, a_r): \mathbf{L}(\{a_1, \cdots, a_r\}) \leqq t\} = \overline{\{(a_1, \cdots, a_r): L(\{a_1, \cdots, a_r\}) \leqq t\}}$$

(overlining denotes closure). If we disregard the difference between $(a_1, \cdots, a_r)$ and $\{a_1, \cdots, a_r\}$, then $\mathbf{L}$ is the largest Zariski-lower-semicontinuous function $\leqq L$. Analogously we define $\mathbf{L}(F \bmod E)$ in terms of $L(F \bmod E)$ (see [10]) by forming the closure with respect to F. $\mathbf{L}(\cdot \bmod \cdot)$ has again the formal properties of L, i.e., $\mathbf{L}$ is a relative L-bound (Thm. 3.4). Theorem 3.5 is a general analogue of Theorem 1 of Paterson–Stockmeyer [7].

$\mathbf{L}$ is more manageable than L: e.g., the knowledge of L is equivalent to the knowledge of the formulas $\mathscr{B}_{t,r}(\phi_1, \cdots, \phi_r)$ or the constructible sets defined by them, whereas $\mathbf{L}$ is known if the much simpler closed sets (1.3) or the corresponding polynomial ideals are known. We will denote these polynomial ideals by $J_r(t)$.

In analogy with § 2, we show in § 3 that for $k = \mathbb{C}$, the ideal $J_r(t)$ contains a nonzero form of relatively small degree with integer coefficients which have absolute value $\leqq 3$. This yields, e.g., the following lower bounds: if we interpret $\mathbb{C}[\mathbf{x}]$ as a $\mathbb{C}$-ring over $\{x\}$, then for large d,

$$(1.4)\quad L\left(*\left|\sum_{\delta=0}^{d} e^{2\pi i/2^\delta} x^\delta\right.\right) > \frac{1}{2} d^{1/3},$$

$$(1.5)\quad L\left(*\left|\sum_{\delta=0}^{d} 2^{2^\delta} x^\delta\right.\right) > \frac{1}{2} d^{1/3},$$

and

$$(1.6)\quad L\left(*\left|\sum_{\delta=0}^{d} 2^{2^{\delta d^3}} x^\delta\right.\right) > \sqrt{d} - 3$$

(see Corollary 3.7).

The lower bound (1.6) is of optimal order as follows from Paterson–Stockmeyer [7, Thm. 4]. The existence of polynomials of degree d with integer coefficients which require at least $\sqrt{d}$ nonscalar multiplications for their computation is already proved in that paper [7, Thm. 1].

The emphasis of § 3 is on general developments. The language is therefore more abstract than in § 2. A reader who is mostly interested in concrete results may skip Theorems 3.3, 3.4 and 3.5 without logical loss.

The most interesting open problems in the present context are perhaps (i) to exhibit a "reasonable" polynomial with coefficients 0 or 1 which is hard to compute, (ii) to derive nontrivial lower bounds for the complexity of initial segments of the classical power series, e.g., $\sum(x^\delta/\delta!)$.

This paper assumes knowledge of [10]. In § 3 we also use notions and theorems of algebraic geometry (see [6, Chap. 1]).

We denote finite sequences $(a_1, \cdots, a_r)$ by $\mathbf{a}$, Im $\mathbf{a}$ stands for $\{a_1, \cdots, a_r\}$. $\mathscr{E}(A)$ is the set of finite subsets of A. $\mathbb{N}'$, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{C}$ stand for the set of positive natural numbers, nonnegative natural numbers, integers, rational numbers and complex numbers, respectively. Let x be a real number; $\exp(x)$ stands for e^x, $\lfloor x \rfloor$ is the greatest integer $\leqq x$, $\lceil x \rceil$ is the least integer $\geqq x$. If f and g are number-theoretic functions, then "$f(d) \lesssim g(d)$ for $d \to \infty$" means that for every $\varepsilon > 0$, finally $f(d) < (1 + \varepsilon)g(d)$. The abbreviation log always stands for $\log_2$.

2. Construction of polynomials over $\mathbb{Q}$ which are hard to compute (counting all operations). We start with preliminaries.

DEFINITION 2.1. The *height* of a polynomial with integer coefficients is the maximum of the absolute values of the coefficients. The *weight* of such a polynomial is the sum of the absolute values of the coefficients.

The weight is subadditive and submultiplicative.

LEMMA 2.2 (pigeonhole lemma). *Let $N > M$. M linear forms $\in \mathbb{Z}[B_1, \cdots, B_N]$, each of weight $\leqq G$, have a common nontrivial zero $\langle b_1, \cdots, b_N \rangle \in \mathbb{Z}^N$ such that*

$$|b_i| \leqq \lfloor G^{M/(N-M)} \rfloor + 2$$

for all i.

Proof. See Schneider [8, p. 140].

LEMMA 2.3. *Let $m \geqq 1$, $c \geqq 2$, $f \geqq 4$ and $q \geqq 5$ be natural numbers, let $z_1, \cdots, z_m$ be indeterminates over $\mathbb{Z}$ and let*

$$P_1, \cdots, P_q \in \mathbb{Z}[z_1, \cdots, z_m]$$

be such that

$$\text{degree } P_\kappa \leqq c, \quad \text{weight } P_\kappa \leqq f$$

for all κ. If g is a natural number such that

$$g^{q-m-2} > c^m q^q \log f, \tag{2.1}$$

then there is a nontrivial form $H \in \mathbb{Z}[y_1, \cdots, y_q]$ of degree g and height $\leqq 3$ such that

$$H(P_1, \cdots, P_q) = 0.$$

Proof. To show: there are integers $b_{i_1,\cdots,i_q}$ (not all $= 0$) such that $|b_{i_1,\cdots,i_q}| \leqq 3$ and

$$\sum_{i_1+\cdots+i_q=g} b_{i_1,\cdots,i_q} P_1^{i_1} \cdots P_q^{i_q} = 0. \tag{2.2}$$

First we replace the $b_{i_1,\cdots,i_q}$'s by indeterminates $B_{i_1,\cdots,i_q}$ and consider

$$Q := \sum_{i_1+\cdots+i_q=g} B_{i_1,\cdots,i_q} P_1^{i_1} \cdots P_q^{i_q} \in \mathbb{Z}[\mathbf{B}, \mathbf{z}].$$

Evidently,

$$\text{degree}_{\mathbf{z}} Q \leqq gc,$$

and

$$\text{weight } Q \leqq f^g \binom{g+q-1}{q-1}.$$

If we write Q as a polynomial in $z_1, \cdots, z_m$, then the coefficients are linear forms $\in \mathbb{Z}[\mathbf{B}]$, of weight $\leqq f^g\binom{g+q-1}{q-1}$. There are at most $\binom{gc+m}{m}$ such forms. (2.2) says that the $b_{i_1,\cdots,i_q}$'s are common zeros of these. The pigeonhole lemma implies the existence of a nontrivial zero $b_{i_1,\cdots,i_q}$ with $|b_{i_1,\cdots,i_q}| \leqq 3$ if

$$\binom{g+q-1}{q-1} > \binom{gc+m}{m}$$

and

$$\left(f^g\binom{g+q-1}{q-1}\right)^{\binom{gc+m}{m}} < 2^{\binom{g+q-1}{q-1}-\binom{gc+m}{m}}. \tag{2.3}$$

The first inequality follows from the second one. The assumptions about m, c, q and $g \geqq 2$ (because of (2.1)) imply

$$\binom{gc+m}{m} \leqq (gc)^m + 1$$

and

$$\left(\frac{g}{q}\right)^{q-1} < \binom{g+q-1}{q-1} < g^q.$$

Suppose now that (2.3) is false. Then

$$(f^g g^q)^{(gc)^m+1} \geqq 2^{(g/q)^{q-1}} - (gc)^m - 1,$$

and therefore

$$((gc)^m + 1)g(\log f + q) \geqq \left(\frac{g}{q}\right)^{q-1} - (gc)^m - 1,$$

which can be written as

$$((gc)^m + 1)(g(\log f + q) + 1) \geqq \left(\frac{g}{q}\right)^{q-1}.$$

This implies (note that $g \geqq 2$, $c \geqq 2$, $\log f \geqq 2$, $q \geqq 5$)

$$(gc)^m gq \log f \geqq \left(\frac{g}{q}\right)^{q-1}.$$

This inequality contradicts (2.1).

Let k be a field; $k[[x]]$ is the ring of formal power series in x interpreted as a ring with division by units over $k \cup \{x\}$. Therefore $k[[x]]$ has the type $\Omega = \{+, -, *, /\} \cup k \cup \{x\}$.

LEMMA 2.4. *Let β be a Ω-computation whose execution in $k[[x]]$ yields the sequence $(a_1, \cdots, a_l)$ of intermediate results. Assume*

$$a_i = \sum_{\delta \geqq 0} \alpha_{i\delta} x^\delta.$$

Let $L(\pm|\beta) = u$, $L(*|\beta) = v_1$, $L(/|\beta) = v_2$. *Put* $v = v_1 + v_2$ *and* $m = \min\{u, 2v\}$. *Then there are polynomials*

$$P_{i\delta} \in \mathbb{Z}[z_1, \cdots, z_m]$$

$(1 \leqq i \leqq l, 1 \leqq \delta < \infty)$ *such that*

$$\begin{aligned} \max_{1 \leqq \delta \leqq d} \text{degree } P_{i\delta} &\leqq (u+1)2^{v_1}(d^{v_2} + 3d^{v_2-1} + \cdots + 3d + 2) \\ &\leqq (u+1)2^{v_1+1}d^{v_2}, \end{aligned} \tag{2.4}$$

$$\sum_{1 \leqq \delta \leqq d} \text{weight } P_{i\delta} \leqq 3^{(u+1)2^{v_1}(d^{v_2}+\cdots+1)} - 1 \leqq 2^{(u+1)2^{v_1+1}d^{v_2}} \tag{2.5}$$

for $1 \leqq i \leqq l$, $d \geqq 5$, *and there are* $\gamma_1, \cdots, \gamma_m$, $\lambda_i \in k$ *such that* $\lambda_i \neq 0$ *and*

$$\lambda_i P_{i\delta}(\gamma_1, \cdots, \gamma_m) = \alpha_{i\delta}$$

for $1 \leqq i \leqq l$, $\delta \geqq 1$.

Proof. First we prove the lemma with $2v$ instead of m by induction on l arranging $\lambda_i = 1$ for all i. The initial step of the induction is obvious (empty computation). Let $\beta = (\beta_1, \cdots, \beta_l)$. By the induction hypothesis, there are $P_{i\delta}$ for $1 \leqq i \leqq l-1$, $1 \leqq \delta < \infty$, having the desired properties.

If $\omega_l \in k$, then we set $P_{l\delta} = 0$ for all δ.

If $\omega_l = x$, then we set $P_{l1} = 1$ and $P_{l\delta} = 0$ for $\delta > 1$.

If $\beta_l = (\pm, i, j)$, then we set $P_{l\delta} = P_{i\delta} \pm P_{j\delta}$.

If $\beta_l = (*, i, j)$, then we define the $P_{l\delta}$'s by

$$\begin{aligned} &\left(z_{2v-1} + \sum_{\delta \geqq 1} P_{i\delta}(z_1, \cdots, z_{2v-2})x^\delta\right)\left(z_{2v} + \sum_{\delta \geqq 1} P_{j\delta}(z_1, \cdots, z_{2v-2})x^\delta\right) \\ &\quad = z_{2v-1}z_{2v} + \sum_{\delta \geqq 1} P_{l\delta}(z_1, \cdots, z_{2v})x^\delta. \end{aligned}$$

Further, we set $\gamma_{2v-1} = \alpha_{i0}$, $\gamma_{2v} = \alpha_{j0}$.

If $\beta_l = (/, i, j)$ then we define the $P_{l\delta}$'s by

$$\begin{aligned} &\left(z_{2v-1} + \sum_{\delta \geqq 1} P_{i\delta}(z_1, \cdots, z_{2v-2})x^\delta\right) \Big/ \left(1/z_{2v} + \sum_{\delta \geqq 1} P_{j\delta}(z_1, \cdots, z_{2v-2})z^\delta\right) \\ &\quad = z_{2v-1}z_{2v} + \sum_{\delta \geqq 1} P_{l\delta}(z_1, \cdots, z_{2v})x^\delta. \end{aligned} \tag{2.6}$$

The $P_{l\delta}$'s are integer polynomials because of

$$\frac{1}{1/z_{2v} + \sum_{\delta \geqq 1} P_{j\delta}x^\delta} = z_{2v} \sum_{\sigma \geqq 0} \left(-z_{2v} \sum_{\delta \geqq 1} P_{j\delta}x^\delta\right)^\sigma. \tag{2.7}$$

Further, we set $\gamma_{2v-1} = \alpha_{i0}$, $\gamma_{2v} = 1/\alpha_{j0}$.

It is not difficult to see that the $P_{l\delta}$'s have the desired properties; we only show how to estimate degree and weight in the case of division. (2.6) and (2.7)

imply

$$\left(z_{2v-1} + \sum_{\delta=1}^{d} P_{i\delta}x^{\delta}\right) z_{2v} \sum_{\sigma=0}^{d-1} \left(-z_{2v} \sum_{\delta=1}^{d} P_{j\delta}x^{\delta}\right)^{\sigma} + z_{2v-1}z_{2v}\left(-z_{2v} \sum_{\delta=1}^{d} P_{j\delta}x^{\delta}\right)^{d}$$

$$\equiv z_{2v-1}z_{2v} + \sum_{\delta=1}^{d} P_{l\delta}x^{\delta} \pmod{x^{d+1}}.$$

The bound for the degree follows immediately. If $\sum_{\delta=1}^{d}$ weight $P_{i\delta} \leqq t-1$ and $\sum_{\delta=1}^{d}$ weight $P_{j\delta} \leqq t-1$ for some $t \geqq 3$, then

$$\left(\sum_{\delta=1}^{d} \text{weight } P_{l\delta}\right) + 1 = \text{weight}\left(z_{2v-1}z_{2v} + \sum_{\delta=1}^{d} P_{l\delta}x^{\delta}\right)$$

$$\leqq t \sum_{\sigma=0}^{d-1} t^{\sigma} + t^{d} \leqq \frac{t^{d+1}}{t-1} + t^{d} \leqq 3t^{d}.$$

The induction step for the weight follows. We now prove the lemma with u instead of m. For simplicity, we assume that $a_i \neq 0$ for all i (it should not cause any problems for the reader to drop this assumption). Then every a_i can be written as

$$a_i = \sum_{\delta \geqq \delta_i} \alpha_{i\delta}x^{\delta}, \qquad \alpha_{i\delta_i} \neq 0.$$

We construct not only the polynomials $P_{i\delta}$ for $\delta \geqq 1$, but also polynomials P_{i0} with integer coefficients such that (2.4) and (2.5) are satisfied with $\max_{0 \leqq \delta \leqq d}$ and $\sum_{0 \leqq \delta \leqq d}$. We also achieve that

$$P_{i0} = \cdots = P_{i,\delta_i-1} = 0, \qquad P_{i\delta_i} = 1.$$

Obviously, this forces $\lambda_i = \alpha_{i\delta_i}$. The proof goes again by induction on l.

If $\omega_l \in k - \{0\}$, then we set $P_{l0} = 1$ and $P_{l\delta} = 0$ for $\delta > 0$.

If $\omega_l = x$, then we set $P_{l1} = 1$ and $P_{l\delta} = 0$ for $\delta \neq 1$.

If $\beta_l = (*, i, j)$, then we define the $P_{l\delta}$'s by

$$\sum_{\delta \geqq 0} P_{i\delta}x^{\delta} \sum_{\delta \geqq 0} P_{j\delta}x^{\delta} = \sum_{\delta \geqq 0} P_{l\delta}x^{\delta}.$$

If $\beta_l = (/, i, j)$, then we define the $P_{l\delta}$'s by

$$\sum_{\delta \geqq 0} P_{i\delta}x^{\delta} \Big/ \sum_{\delta \geqq 0} P_{j\delta}x^{\delta} = \sum_{\delta \geqq 0} P_{l\delta}x^{\delta}.$$

Since a_j is a unit, we have $\alpha_{j0} \neq 0$ and hence $P_{j0} = 1$. Therefore the $P_{l\delta}$'s have integer coefficients.

If $\beta_l = (+, i, j)$ then we distinguish 3 cases.

Case 1. $\delta_i = \delta_j = \delta_l$. We define the $P_{l\delta}$'s by

$$z_u \sum_{\delta \geqq 0} P_{i\delta}(z_1, \cdots, z_{u-1})x^{\delta} + (1 - z_u) \sum_{\delta \geqq 0} P_{j\delta}(z_1, \cdots, z_{u-1})x^{\delta}$$

$$= \sum_{\delta \geqq 0} P_{l\delta}(z_1, \cdots, z_u)x^{\delta}.$$

Then $P_{l\delta} = 0$ for $\delta < \delta_l$ and $P_{l\delta_l} = 1$. Furthermore, we set

$$\gamma_u = \alpha_{i\delta_i}/(\alpha_{i\delta_i} + \alpha_{j\delta_j})$$

(the denominator is $= \alpha_{l\delta_l}$ and therefore $\neq 0$).

Case 2. $\delta_i = \delta_j < \delta_l$. Then $\alpha_{i\delta_i} = -\alpha_{j\delta_j}$. We set $P_{l\delta} = 0$ for $\delta < \delta_l$, $P_{l\delta_l} = 1$, and define the remaining $P_{l\delta}$'s by

$$z_u \sum_{\delta > \delta_l} P_{i\delta} x^\delta - z_u \sum_{\delta > \delta_l} P_{j\delta} x^\delta = \sum_{\delta > \delta_l} P_{l\delta} x^\delta.$$

Furthermore, we set $\gamma_u = \alpha_{i\delta_i}/\alpha_{l\delta_l}$.

Case 3. $\delta_i \neq \delta_j$, say $\delta_i < \delta_j$. Then $\delta_l = \delta_i$. We define the $P_{l\delta}$'s by

$$\sum_{\delta \geqq 0} P_{i\delta} x^\delta + z_u \sum_{\delta \geqq 0} P_{j\delta} x^\delta = \sum_{\delta \geqq 0} P_{l\delta} x^\delta.$$

Again $P_{l\delta} = 0$ for $\delta < \delta_l$ and $P_{l\delta_l} = 1$. Furthermore, we set $\gamma_u = \alpha_{j\delta_j}/\alpha_{i\delta_i}$. The case $\beta_l = (-, i, j)$ is treated analogously. The proof that the $P_{l\delta}$'s possess the desired properties goes along the lines of the first part of this proof.

THEOREM 2.5. *Let β compute*

$$a = \sum_{\delta=0}^{d} \alpha_\delta x^\delta, \qquad \alpha_\delta \in k,$$

in $k(x)$ interpreted as a field over $k \cup \{x\}$. Let $L(\pm|\beta) = u > 0$, $L(*/|\beta) = v > 0$,

$$m = \min\{u, 2v\}, \qquad s = u + v.$$

Let $q \geqq 5$, and let $\delta_1, \cdots, \delta_q$ be pairwise distinct natural numbers $\leqq d$. If g is a natural number such that

$$g^{q-m-2} > d^{s(m+1)} q^q, \tag{2.8}$$

then there is a nontrivial form $H \in \mathbb{Z}[y_1, \cdots, y_\gamma]$ of degree g and height $\leqq 3$ such that

$$H(\alpha_{\delta_1}, \cdots, \alpha_{\delta_q}) = 0.^1$$

Proof. We can assume without loss of generality that β is executable and that k has infinite degree of transcendence over its prime field. Let $(a_1, \cdots, a_l)$ be the sequence of results of β, $a_i \in k(x)$. In order to simplify the argument, we assume $a = a_l$. Let $\theta \in k$ be transcendental over $\alpha_1, \cdots, \alpha_d$ and be different from the zeros and poles of the nontrivial a_i. Then we can develop each a_i as a formal power series in $x - \theta$:

$$a_i = \sum_{\delta \geqq 0} \alpha_{i\delta} (x - \theta)^\delta.$$

For $a_i \neq 0$ we have $\alpha_{i0} \neq 0$. If we interpret the ring $k[[x - \theta]]$ of formal power series as a ring with division by units over $k \cup \{x\}$, then β is executable in this ring and has the sequence $(a_1, \cdots, a_l)$ of results. Therefore we can apply Lemma 2.4 (with $x - \theta$ instead of x). Lemma 2.4 implies the existence of polynomials $P_1, \cdots, P_q \in \mathbb{Z}[z_1, \cdots, z_m]$ and $\gamma_1, \cdots, \gamma_m, \lambda \in k$ with $\lambda \neq 0$ such that

$$\lambda P_\kappa(\gamma_1, \cdots, \gamma_m) = \alpha_{l\delta_\kappa},$$

$$\text{degree } P_\kappa \leqq 2(u+1)d^v \leqq d^s,$$

$$\text{weight } P_\kappa \leqq 2^{2(u+1)d^v} \leqq 2^{d^s}$$

---

[1] Obviously the theorem will only be relevant for $q > m + 2$.

(the right inequalities follow from $d \geqq q \geqq 5$ and $u \geqq 1$). If we set $c = d^s, f = 2^{d^s}$ then by (2.8),

$$g^{q-m-2} > c^m g^q \log f.$$

By Lemma 2.3, there is a nontrivial form $H \in \mathbb{Z}[y_1, \cdots, y_\gamma]$ of degree g and height $\leqq 3$ such that

$$H(P_1, \cdots, P_q) = 0,$$

and therefore

$$\begin{aligned} 0 &= H(\alpha_{l\delta_1}, \cdots, \alpha_{l\delta_q}) \\ &= H\left(\sum_{\delta=\delta_1}^{d} \binom{\delta}{\delta_1} \alpha_\delta \theta^{\delta-\delta_1}, \cdots, \sum_{\delta=\delta_q}^{d} \binom{\delta}{\delta_q} \alpha_\delta \theta^{\delta-\delta_q}\right). \end{aligned}$$

The constant term of this polynomial in θ is $H(\alpha_{\delta_1}, \cdots, \alpha_{\delta_q})$. Since θ is transcendental over $\alpha_1, \cdots, \alpha_q$, we conclude

$$H(\alpha_{\delta_1}, \cdots, \alpha_{\delta_q}) = 0.$$

LEMMA 2.6. *Let* $\operatorname{char} k \notin \{2, 3\}$, $k_0 =$ *prime field of* k, $\tau_1, \cdots, \tau_q \in k$, *and let* g *be a natural number such that*

$$[k_0(\tau_1, \cdots, \tau_\kappa) : k_0(\tau_1, \cdots, \tau_{\kappa-1})] > g$$

for all κ. *Then there is no form* $H \in \mathbb{Z}[y_1, \cdots, y_q]$, $H \neq 0$ *of degree* g *and height* $\leqq 3$ *such that*

$$H(\tau_1, \cdots, \tau_q) = 0.$$

Proof. Assume there is such a form. Let $\tilde{H}$ be the image of H under the canonical homomorphism $\mathbb{Z}[y_1, \cdots, y_q] \to k_0[y_1, \cdots, y_q]$. Since $\operatorname{char} k \notin \{2, 3\}$ and height $H \leqq 3$, $\tilde{H} \neq 0$. Choose κ such that

$$\tilde{H}(\tau_1, \cdots, \tau_\kappa, y_{\kappa+1}, \cdots, y_q) = 0,$$

but

$$\tilde{H}(\tau_1, \cdots, \tau_{\kappa-1}, y_\kappa, y_{\kappa+1}, \cdots, y_q) \neq 0.$$

We interpret $\tilde{H}(\tau_1, \cdots, \tau_{\kappa-1}, y_\kappa, y_{\kappa+1}, \cdots, y_q)$ as a polynomial in $y_{\kappa+1}, \cdots, y_q$ with coefficients $\in k_0(\tau_1, \cdots, \tau_{\kappa-1})[y_k]$. Let Q be a nonvanishing coefficient. Then $Q(\tau_\kappa) = 0$, and degree $Q \leqq$ degree $H = g$, and therefore

$$[k_0(\tau_1, \cdots, \tau_\kappa) : k_0(\tau_1, \cdots, \tau_{\kappa-1})] \leqq g,$$

which is a contradiction.

COROLLARY 2.7. *Let* β *compute*

$$a = \sum_{\delta=0}^{d} \exp(2\pi i/2^\delta) x^\delta$$

in the field $\mathbb{C}(x)$ *over* $\mathbb{C} \cup \{x\}$. *If we set* $L(\pm|\beta) = u$, $L(*/|\beta) = v$, $m = \min\{u, 2v\}$, $s = u + v$, *then*

$$s(m+2) \gtrsim d/\log d$$

for $d \to \infty$. *In particular,* $s > \sqrt{d/\log d}$ *for large* d.

Proof. Let

$$\begin{aligned}\alpha_\delta &= \exp(2\pi i/2^\delta),\\ q &= \lfloor d/((\log d)^2 + 3)\rfloor,\\ \delta_\kappa &= \kappa(\lceil \log d)^2\rceil + 2), \qquad \kappa = 1, \cdots, q,\\ g &= 2\lceil(\log d)^2\rceil.\end{aligned}$$

Then

$$[\mathbb{Q}(\alpha_{\delta_1}, \cdots, \alpha_{\delta_\kappa}) : \mathbb{Q}(\alpha_{\delta_1}, \cdots, \alpha_{\delta_{\kappa-1}})] = \begin{cases} 2^{\delta_\kappa - \delta_{\kappa-1}} & \text{for } \kappa > 1,\\ 2^{\delta_1 - 1} & \text{for } \kappa = 1,\end{cases}$$

which is $> g$ for all κ. Lemma 2.6 and Theorem 2.5 imply

$$g^{q-m-2} \leqq d^{s(m+1)} q^q.$$

The assertion follows by substituting.

COROLLARY 2.8. *Let β compute*

$$a = \sum_{\delta=0}^{d} \exp(2\pi i/2^{\delta^3}) x^\delta$$

in the field $\mathbb{C}(x)$ over $\mathbb{C} \cup \{x\}$. Then for large d, either

$$m > d/5,$$

or

$$s > d^2/(5 \log d).$$

(We use the notation of Corollary 2.7.)

Proof. Let

$$\begin{aligned}\alpha_\delta &= \exp(2\pi i/2^{\delta^3})\\ q &= \lfloor d/4\rfloor\\ \delta_\kappa &= \lceil (4\kappa d^2)^{1/3}\rceil\\ g &= 2^{d^2-4d}.\end{aligned}$$

Then

$$I_\kappa := [\mathbb{Q}(\alpha_{\delta_1}, \cdots, \alpha_{\delta_\kappa}) : \mathbb{Q}(\alpha_{\delta_1}, \cdots, \alpha_{\delta_{\kappa-1}})] = \begin{cases} 2^{\delta_\kappa^3 - \delta_{\kappa-1}^3} & \text{for } \kappa > 1\\ 2^{\delta_1^3 - 1} & \text{for } \kappa = 1.\end{cases}$$

Since

$$\begin{aligned}\delta_\kappa^3 - \delta_{\kappa-1}^3 &\geqq 4\kappa d^2 - ((4(\kappa-1)d^2)^{1/3} + 1)^3\\ &\geqq 4d^2 - 3d^2 - 3d - 1\\ &> d^2 - 4d\end{aligned}$$

and

$$\delta_1^3 - 1 > d^2 - 4d,$$

we infer

$$I_\kappa > g \quad \text{for all } \kappa.$$

Lemma 2.6 and Theorem 2.5 imply

$$g^{q-m-2} \leqq d^{s(m+1)} q^q.$$

The assertion follows easily.

LEMMA 2.9. (See Schneider [8, § 5].) *Let*

$$Q = \sum_{j=0}^{l} \sigma_j y^j \in \mathbb{Z}[y]$$

with $\delta_l \neq 0$ *and*

$$|\sigma_j| \leqq h c^{l-j} \quad \textit{for all } j.$$

Then for any root $\tau \in \mathbb{C}$ *of* Q,

$$|\tau| \leqq (h+1)c.$$

Proof. If $|\tau| \leqq c$, then we have nothing to prove. Therefore assume $|\tau| > c$.

$$\sigma_l \tau^l = -\sum_{j=0}^{l-1} \sigma_j \tau^j$$

implies

$$\begin{aligned} |\tau|^l &\leqq \sum_{j=0}^{l-1} |\sigma_j| |\tau|^j \\ &\leqq h c^l \sum_{j=0}^{l-1} (|\tau|/c)^j \\ &\leqq h c^l (|\tau|/c)^l / ((|\tau|/c) - 1), \end{aligned}$$

which in turn implies the assertion.

LEMMA 2.10. *Let* $\tau_1, \cdots, \tau_q \in \mathbb{Z}$, $q \geqq 5$, *and* $g \in \mathbb{N}$, $g \geqq 3$, *with*

$$|\tau_1| > 4$$

and

$$|\tau_\kappa| > |q\tau_{\kappa-1}|^g \quad \text{for } \kappa > 1.$$

Then there is no form $H \in \mathbb{Z}[y_1, \cdots, y_q]$, $H \neq 0$, *of degree* g *and height* $\leqq 3$ *such that*

$$H(\tau_1, \cdots, \tau_q) = 0.$$

Proof. Assume there is such a form. Choose κ such that

$$H(\tau_1, \cdots, \tau_\kappa, y_{\kappa+1}, \cdots, y_q) = 0,$$

but

$$H(\tau_1, \cdots, \tau_{\kappa-1}, y_\kappa, \cdots, y_q) \neq 0.$$

We interpret $H(\tau_1, \cdots, \tau_{\kappa-1}, y_\kappa, \cdots, y_q)$ as a polynomial in $y_{\kappa+1}, \cdots, y_q$ with coefficients in $\mathbb{Z}[y_\kappa]$. Let Q be a nonvanishing coefficient. The coefficients of Q are polynomials with integer coefficients in $\tau_1, \cdots, \tau_{\kappa-1}$ of degree $\leqq g$ and of height $\leqq 3$. Therefore

$$\text{height } Q \leqq \begin{cases} 3 & \text{if } \kappa = 1, \\ 3\binom{g+\kappa-1}{\kappa-1}|\tau_{\kappa-1}|^g & \text{for } \kappa > 1. \end{cases}$$

$Q(\tau_\kappa)$ is equal to zero. By Lemma 2.9 (with $c = 1$, $h = \text{height } Q$), we infer

$$|\tau_\kappa| \leqq \text{height } Q + 1 \leqq \begin{cases} 4 & \text{if } \kappa = 1, \\ |q\tau_{\kappa-1}|^g & \text{if } \kappa > 1. \end{cases}$$

This contradicts the assumptions about τ_κ.

COROLLARY 2.11. *Let β compute*

$$a = \sum_{\delta=0}^{d} 2^{2^\delta} x^\delta$$

in the field $\mathbb{C}(x)$ over $\mathbb{C} \cup \{x\}$. We set $u = L(\pm|\beta)$, $v = L(*/|\beta)$, $m = \min\{u, 2v\}$, $s = u + v$. Then

$$s(m+2) \gtrsim d/(3 \log d)$$

for $d \to \infty$. In particular, $s > \sqrt{d/(3 \log d)}$ for large d.

Proof. Let $\varepsilon > 0, \alpha_\delta = 2^{2^\delta}, q = \lfloor d/(\log d)^2 \rfloor, \delta_\kappa = \kappa \lfloor (\log d)^2 \rfloor, g = \lfloor d^{(1-\varepsilon)\log d} \rfloor$. Then for large d,

$$|\alpha_{\delta_1}| > 4,$$

$$|\alpha_{\delta_\kappa}| > |q\alpha_{\delta_{\kappa-1}}|^g \qquad (\kappa > 1).$$

Lemma 2.10 and Theorem 2.5 imply

$$g^{q-m-2} \leqq d^{s(m+1)} q^q$$

Substitution yields $s(m+2) \gtrsim (1-\varepsilon)d/\log d$. The assertion follows since ε may be arbitrarily small.

COROLLARY 2.12. *Let β compute*

$$a = \sum_{\delta=0}^{d} 2^{2^{\delta d^3}} x^\delta$$

in the field $\mathbb{C}(x)$ over $\mathbb{C} \cup \{x\}$. Then for large d, either

$$m \geqq d - 4$$

or

$$s > d^2/\log d.$$

(We used the notations of the preceding corollary).

Proof. Let $\alpha_\delta = 2^{2^{\delta d^3}}$, $\delta_\kappa = \kappa$, $q = d$, $g = 2^{d^3-d^2}$. Then for large d,

$$|\alpha_1| > 4$$

and

$$|\alpha_\kappa| > |q\alpha_{\kappa-1}|^g \qquad (\kappa > 1).$$

Lemma 2.10 and Theorem 2.5 imply

$$g^{q-m-2} \leqq d^{s(m+1)}q^q.$$

By substituting and taking logarithms, we get

$$d^4 - d^3 - d\log d \leqq (s\log d + d^3 - d^2)(m+2).$$

If $s \leqq d^2/\log d$, then $m \geqq d - 4$.

If we do not permit divisions, then the proofs are simpler and the results somewhat sharper. If one is only interested in s, i.e., in $L(1|\alpha)$, then one can eliminate all but one division by computing with unreduced numerators and denominators and executing one division at the end of the computation.

The proof of Theorem 2.5 shows that one can replace d by $\max_\kappa \delta_\kappa$ in (2.8). This implies that every lower bound for the computational complexity of a specific polynomial a of degree d which is proved using Theorem 2.5 is equally valid for all polynomials of higher degrees which have a as the initial segment of degree d.

Furthermore, Theorem 2.5 is valid for the values of a polynomial a at the points $\delta_1, \cdots, \delta_q$ (instead of the coefficients $\alpha_{\delta_1}, \cdots, \alpha_{\delta_q}$) if one assumes $\alpha_0 = 0$ and uses a stronger version of (2.8). This follows from the fact that Lemma 2.4 immediately implies a corresponding lemma for the values of the polynomials $\sum_{\delta=1}^{d} \alpha_{i\delta}x^\delta$ at the arguments $1, \cdots, d$. Using this version of the lemma, one can show, for example, that every polynomial which approximates the function e^{e^x-1} (the generating function of the number of partitions) at the points $1, \cdots, d$ reasonably well has to be hard to compute.

3. The computation of polynomials when linear operations are free. In this section, $k_0 \subset k$ are fields, k_0 is perfect and k is algebraically closed. Let $x_1, \cdots, x_n$ be indeterminates over k. The Galois group $\mathrm{Gal}(k/k_0)$ operates on $k[\mathbf{x}] = k[x_1, \cdots, x_n]$ as well as on k^n.

We need some definitions and facts about rationality.

R1. Let V be a Zariski-closed subset of k^n. V is called k_0-*closed* (or *defined over* k_0) if one of the following equivalent conditions is satisfied:

(i) the ideal of V is generated by polynomials belonging to $k_0[\mathbf{x}]$;

(ii) V is the set of common zeros of polynomials belonging to $k_0[\mathbf{x}]$;

(iii) V is stable with respect to $\mathrm{Gal}(k/k_0)$, i.e.,

$$\forall\sigma \in \mathrm{Gal}(k/k_0),\ \sigma V = V.$$

Proof. (i) $\Rightarrow$ (ii) $\Rightarrow$ (iii) is trivial. For (iii) $\Rightarrow$ (i), see [2, §§ 12, 14] or [4, p. 74].

R2. Let C be a constructible subset of k^n. We say that C is *defined over* k_0 if one of the following equivalent conditions is satisfied:

(i) C is a member of the Boolean algebra generated by the k_0-closed sets;

(ii) $\forall\sigma \in \mathrm{Gal}(k/k_0),\ \sigma C = C$.

Proof. (ii) $\Rightarrow$ (i): Let C be constructible and invariant with respect to $\mathrm{Gal}(k/k_0)$. Let U be the largest subset of C which is open in $\bar{C}$, i.e.,

$$U = \bigcup_{\substack{U' \subset C \\ U' \text{ open in } \bar{C}}} U'.$$

$\bar{C}$ and U are invariant, and therefore $\bar{C} - U$ is invariant. Hence $U = \bar{C} - (\bar{C} - U)$ belongs to the Boolean algebra generated by the k_0-closed sets. It is therefore sufficient to prove (i) for $C - U$ instead of C. It is easy to prove that the maximal dimension of the components of $C - U$ is strictly smaller than the maximal dimension of the components of C. We can therefore conclude by induction.

R3. Let V be a k_0-closed, irreducible subset of k^n and let $\alpha \in V$. Then

$$\dim V \geqq \operatorname{trdg}_{k_0} k_0(\alpha)$$

(where trdg stands for transcendence degree).

Proof. Let $\alpha_1, \cdots, \alpha_m$ be algebraically independent over k_0 and let J be the ideal of V. It is sufficient to show that $x_1 + J, \cdots, x_m + J$ are algebraically independent over k in $k[x_1, \cdots, x_n]/J$. Assume otherwise. Then there is a non-trivial $P \in k[x_1, \cdots, x_m]$ which is an element of J. Let $P_1, \cdots, P_l$ be the different polynomials which are generated by the application of the Galois group on P; $Q = \prod_{\lambda=1}^{l} P_\lambda$. Q is invariant under $\operatorname{Gal}(k/k_0)$. Since k_0 is perfect, Q is a non-trivial polynomial in $k_0[x_1, \cdots, x_m] \cap J$. In particular, $Q(\alpha_1, \cdots, \alpha_m) = 0$; this contradicts the algebraic independence of $\alpha_1, \cdots, \alpha_m$ over k_0.

R4. We say a morphism $k^n \to k^m$ is *defined over* k_0 if it is given by polynomials belonging to $k_0[x_1, \cdots, x_n]$.

It is clear how to apply these concepts to a finite-dimensional k-vector space V if some basis is distinguished. If V is given as scalar extension of a k_0-vector space V_0, $V = V_0 \otimes_{k_0} k$, then it is easy to prove that the definitions do not depend on the basis chosen, if one restricts oneself to bases coming from V_0: one says that V possesses a k_0-structure by virtue of the representation $V = V \otimes_{k_0} k$.

Let A_0 be a p-dimensional k_0-ring, i.e., a k_0-ring (not necessarily commutative) which has finite dimension p as a k_0-vector space (the associative law is not essential for the following). All computations will be done in

$$A = A_0 \otimes_{k_0} k;$$

A is interpreted as a k-ring. We use the operation-time $z = 1_{\{*\}}$. We can assume that A_0 is a subset of A. If we neglect the multiplication in A, then A is a k-vector space with k_0-structure.

THEOREM 3.1. *Let* $\mathbf{a} = (a_1, \cdots, a_r)$, $\mathbf{b} = (b_1, \cdots, b_s)$, $N = (s + t + 1)(s + t + r) - s(s + 1)$. *The set*

$$C_{r,s}(t) := \{(\mathbf{a}, \mathbf{b}) \in A^r \times A^s : L(\operatorname{Im} \mathbf{a} \bmod \operatorname{Im} \mathbf{b}) \leqq t\}$$

is the projection to $A^r \times A^s$ *of the graph of the morphism*

$$\phi : A^r \leftarrow A^s \times k^N$$

defined over k_0. *In particular,* $C_{r,s}(t)$ *is an irreducible constructible subset of* $A^r \times A^s$ *defined over* k_0. *Also, for* $E \in \mathscr{E}(A)$,

$$C_{r,E}(t) := \{\mathbf{a} \in A^r : L(\operatorname{Im} \mathbf{a} \bmod E) \leqq t\}$$

is irreducible and constructible, and

$$\dim \overline{C_{r,E}(t)} \leqq N.$$

If $E \in \mathscr{E}(A_0)$, *then* $C_{r,E}(t)$ *is defined over* k_0.

Proof. Let $(e_1, \cdots, e_p)$ be a basis of the k_0-vector space A_0 and therefore also a basis of the k-vector space A. Then

$$e_i e_j = \sum_{l=1}^{p} \sigma_{ij}^{(l)} e_l, \qquad \sigma_{ij}^{(l)} \in k_0. \tag{3.1}$$

Further, let $y_j^{(i)}$ $(i = 1, \cdots, p, j = 1, \cdots, s)$, $z_{1j}^{(i)}$, $z_{2j}^{(i)}$ $(i = 0, \cdots, j-1, j = s+1, \cdots, s+t)$ and $z_j^{(i)}$ $(i = 0, \cdots, s+t, j = s+t+1, \cdots, s+t+r)$ be indeterminates over k. Then $B = A \otimes_k k[\mathbf{y}, \mathbf{z}]$ is a $k[\mathbf{y}, \mathbf{z}]$-ring. Again we can assume that A is a subset of B. $(e_1, \cdots, e_p)$ is then also a basis of the $k[\mathbf{y}, \mathbf{z}]$-module B, and (3.1) is valid in B. We define a sequence $f_0, \cdots, f_{s+t+r}$ of elements of B (a "generic" sequence of intermediate results):

$$f_0 = 1,$$

$$f_j = \sum_{i=1}^{p} y_j^{(i)} e_i \qquad \text{for } 1 \leqq j \leqq s,$$

$$f_j = \sum_{i=0}^{j-1} z_{1j}^{(i)} f_i \sum_{i=0}^{j-1} z_{2j}^{(i)} f_i \qquad \text{for } s+1 \leqq j \leqq s+t,$$

and

$$f_j = \sum_{i=0}^{s+t} z_j^{(i)} f_i \qquad \text{for } s+t+1 \leqq j \leqq s+t+r.$$

We will write

$$f_{s+t+\rho} = \sum_{v=1}^{p} P_\rho^{(v)} e_v, \qquad P_\rho^{(v)} \in k[\mathbf{y}, \mathbf{z}] \quad \text{for } 1 \leqq \rho \leqq r.$$

(3.1) implies

$$P_\rho^{(v)} \in k_0[\mathbf{y}, \mathbf{z}]. \tag{3.2}$$

These polynomials define a morphism

$$\phi : A^r \leftarrow A^s \times k^N;$$

we interpret the y's as coordinate variables of A^s and the z's as coordinate variables of k^N. In order to prove the first assertion of the theorem, we have to show: let $b_1, \cdots, b_s \in A$ with

$$b_j = \sum_{i=1}^{p} \eta_j^{(i)} e_i.$$

Then

$$L(a_1, \cdots, a_r \bmod b_1, \cdots, b_s) \leqq t \tag{3.3}$$

if and only if there are $\zeta_{1j}^{(i)}, \zeta_{2j}^{(i)} \in k$ $(0 \leqq i \leqq j-1,\ s+1 \leqq j \leqq s+t)$ and $\zeta_j^{(i)}$ $(i = 0, \cdots, s+t, j = s+t+1, \cdots, s+t+r)$ such that

$$a_\rho = \sum_{v=1}^{p} P_\rho^{(v)}(\boldsymbol{\eta}, \boldsymbol{\zeta}) e_v.$$

In other words (by the definition of the $P_\rho^{(\nu)}$'s): (3.3) is valid iff there are $\bar{f}_{s+1}, \cdots, \bar{f}_{s+t} \in A$ such that $\bar{f}_{s+\tau}$ is the product of two k-linear combinations of $1, b_1, \cdots, b_s, \bar{f}_{s+1}, \cdots, \bar{f}_{s+\tau-1}$ $(1 \leqq \tau \leqq t)$ and such that a_ρ is a k-linear combination of $1, b_1, \cdots, b_s, \bar{f}_{s+1}, \cdots, \bar{f}_{s+t}$ $(1 \leqq \rho \leqq r)$. This is easy to prove as follows: if (3.3) is valid and if β computes $a_1, \cdots, a_r \bmod b_1, \cdots, b_s$ in A and $L(*|\beta) = t$, then let $\bar{f}_{s+1}, \cdots, \bar{f}_{s+t}$ be the results of the computational steps β_i with $\omega_i = *$ in the given order. On the other hand, if there exist $\bar{f}_{s+1}, \cdots, \bar{f}_{s+t}$ with the desired properties, then (3.3) is a consequence of the transitivity theorem (using the trivial Lemma 2.1 of [11]). Therefore $C_{r,s}(t)$ is the projection of graph(ϕ) on $A^r \times A^s$. Hence $C_{r,s}(t)$ is irreducible and constructible (by the theorem of Chevalley). (3.2) implies that ϕ is defined over k_0; hence graph(ϕ) is stable with respect to Gal(k/k_0). Therefore $C_{r,s}(t)$ is stable with respect to Gal(k/k_0); i.e., $C_{r,s}(t)$ is defined over k_0. Let $E \in \mathscr{E}(A)$, $E = \{b_1, \cdots, b_s\}$ with $b_j = \sum_{i=1}^{p} \eta_j^{(i)} e_i$. Then $C_{r,E}(t) = \{\mathbf{a} : (\mathbf{a}, \mathbf{b}) \in C_{r,s}(t)\}$ is evidently the image of the morphism

$$\phi(\eta, \cdot) : k^N \to A^r. \tag{3.4}$$

The remaining assertions follow.

DEFINITION 3.2. A function

$$\lambda(\cdot \bmod \cdot) : \mathscr{E}(A)^2 \to \mathbb{N} \cup \{\infty\}$$

is *lower semicontinuous* (in the first variable) if the set

$$\{\mathbf{a} \in A^r : \lambda(\operatorname{Im} \mathbf{a} \bmod E) \leqq t\}$$

is Zariski-closed for all $r \in \mathbb{N}'$, $E \in \mathscr{E}(A)$ and $t \geqq 0$.

THEOREM 3.3. *If*

$$\lambda(\cdot \bmod \cdot) : \mathscr{E}(A)^2 \to \mathbb{N} \cup \{\infty\}$$

is monotonic in its first argument, then there is a function

$$\boldsymbol{\lambda}(\cdot \bmod \cdot) : \mathscr{E}(A)^2 \to \mathbb{N} \cup \{\infty\}$$

such that

$$\{\mathbf{a} \in A^r : \boldsymbol{\lambda}(\operatorname{Im} \mathbf{a} \bmod E) \leqq t\} = \overline{\{\boldsymbol{\alpha} \in A^r : \lambda(\operatorname{Im} \mathbf{a} \bmod E) \leqq t\}} \tag{3.5}$$

for all $r \in \mathbb{N}'$, $E \in \mathscr{E}(A)$ and $t \geqq 0$. $\boldsymbol{\lambda}$ is the largest lower semicontinuous function $\leqq \lambda$.

Proof. We set

$$D_r(t) := \{\mathbf{a} \in A^r : \lambda(\operatorname{Im} \mathbf{a} \bmod E) \leqq t\}.$$

(3.5) is equivalent to

$$\boldsymbol{\lambda}(\operatorname{Im} \mathbf{a} \bmod E) = \min \{t : \mathbf{a} \in \overline{D_r(t)}\}$$

for all $\mathbf{a} \in A^r$. We can use this line as definition of $\boldsymbol{\lambda}$, if we know that the right side depends only on Im $\mathbf{a}$. Therefore it is sufficient to show:

$$(a_1, \cdots, a_r) \in D_r(t) \Rightarrow (a_{\pi 1}, \cdots, a_{\pi r}) \in D_r(t)$$

for arbitrary permutations π,

$$(a_1, \cdots, a_r) \in \overline{D_r(t)} \Rightarrow (a_1, \cdots, a_r, a_r) \in \overline{D_{r+1}(t)},$$

$$(a_1, \cdots, a_r) \in \overline{D_r(t)} \Rightarrow (a_1, \cdots, a_{r-1}) \in \overline{D_{r-1}(t)}.$$

The three implications follow from the observation that the set of $\mathbf{a} \in A^r$, for which the right sides are valid, is closed and contains $D_r(t)$. The remaining assertions are trivial.

THEOREM 3.4. $\mathbf{L}(\cdot \bmod \cdot)$ *is the largest lower semicontinuous relative L-bound.*

Proof. Because of Theorem 3.3 and [10, Thm. 4.8] it suffices to show that $\mathbf{L}$ is a relative $\mathbf{L}$-bound.

Monotonicity. We only prove that it is monotonic in the first argument.

It suffices to show

$$(a_1, \cdots, a_r) \in \overline{C_{r,E}(t)} \Rightarrow (a_1, \cdots, a_{r-1}) \in \overline{C_{r-1,E}(t)}. \tag{3.6}$$

This follows from the fact that

$$\{(a_1, \cdots, a_r) : (a_1, \cdots, a_{r-1}) \in \overline{C_{r-1,E}(t)}\}$$

is closed and contains $C_{r,E}(t)$.

Transitivity. First we show that $\mathbf{L}(\cdot \bmod D)$ is an L-bound for the canonical D-expansion of A. Since $L(\cdot \bmod D)$ is such an L-bound, the following is valid:

$$L(F \cup \{\omega\mathbf{a}\} \bmod D) \leqq L(F \cup \operatorname{Im}\mathbf{a} \bmod D) + z(\omega)$$

for $\mathbf{a} \in \operatorname{dom}\omega$, $\omega \in \{0, 1, +, -, *\} \cup k \cup D$. For $F = \{f_1, \cdots, f_r\}$, $\mathbf{a} = (a_1, \cdots, a_s)$, this means

$$(f_1, \cdots, f_r, a_1, \cdots, a_s) \in C_{r+s,D}(t) \Rightarrow (f_1, \cdots, f_r, \omega\mathbf{a}) \in C_{r+1,D}(t + z(\omega))$$

for all t. Evidently $\omega : A^s \to A$ is a morphism. Therefore,

$$\{(f_1, \cdots, f_r, a_1, \cdots, a_s) : (f_1, \cdots, f_r, \omega\mathbf{a}) \in \overline{C_{r+1,D}(t + z(\omega))}\}$$

is a closed set containing $C_{r+s,D}(t)$ and hence containing $\overline{C_{r+s,D}(t)}$. This means

$$(f_1, \cdots, f_r, a_1, \cdots, a_s) \in \overline{C_{r+s,D}(t)} \Rightarrow (f_1, \cdots, f_r, \omega\mathbf{a}) \in \overline{C_{r+1,D}(t + z(\omega))},$$

or in other words,

$$\mathbf{L}(F \cup \operatorname{Im}\mathbf{a} \bmod D) \leqq t \Rightarrow \mathbf{L}(F \cup \{\omega\mathbf{a}\} \bmod D) \leqq t + z(\omega).$$

Since this is valid for all t, we infer

$$\mathbf{L}(F \cup \{\omega\mathbf{a}\} \bmod D) \leqq \mathbf{L}(F \cup \operatorname{Im}\mathbf{a} \bmod D) + z(\omega).$$

Therefore $\mathbf{L}(\cdot \bmod D)$ is an L-bound mod D. To prove transitivity we can assume $u := \mathbf{L}(E \bmod D) < \infty$ without loss of generality. Then

$$\mathbf{L}(F \cup E \bmod D) - u$$

(fixed E, D) is also an L-bound mod $E \cup D$; hence

$$\mathbf{L}(F \cup E \bmod D) - u \leqq L(F \bmod E \cup D).$$

This means ($F = \{f_1, \cdots, f_r\}$, $E = \{b_1, \cdots, b_s\}$)

$$(f_1, \cdots, f_r) \in C_{r,E\cup D}(t) \Rightarrow (f_1, \cdots, f_r, b_1, \cdots, b_s) \in \overline{C_{r+s,D}(t + u)}.$$

Obviously

$$\{(f_1, \cdots, f_r) : (f_1, \cdots, f_r, b_1, \cdots, b_s) \in \overline{C_{r+s,D}(t + u)}\}$$

is a closed set containing $C_{r,E\cup D}(t)$ and therefore $\overline{C_{r,E\cup D}(t)}$. This means

$$(f_1, \cdots, f_r) \in \overline{C_{r,E\cup D}(t)} \Rightarrow (f_1, \cdots, f_r, b_1, \cdots, b_s) \in \overline{C_{r+s,D}(t+u)}$$

and thus

$$\mathbf{L}(F \bmod E \cup D) \leqq t \Rightarrow \mathbf{L}(F \cup E \bmod D) \leqq t + u.$$

Since this is valid for all t, we can infer

$$\mathbf{L}(F \cup E \bmod D) \leqq \mathbf{L}(F \bmod E \cup D) + \mathbf{L}(E \bmod D).$$

Normalization. Since $\mathbf{L}(\cdot \bmod D)$ is an L-bound for the canonical D-expansion of A, we have (putting $D = \operatorname{Im} \mathbf{a}$)

$$\mathbf{L}(\omega\mathbf{a} \bmod \operatorname{Im} \mathbf{a}) \leqq \mathbf{L}(\operatorname{Im} \mathbf{a} \bmod \operatorname{Im} \mathbf{a}) + z(\omega) \leqq z(\omega).$$

THEOREM 3.5. *Let* $\mathbf{a} \in A^r$, $\mathbf{b} \in A^s$. *If* $(e_1, \cdots, e_p)$ *is a* k_0*-basis of* A_0 *and therefore a* k*-basis of* A, *and if*

$$a_j = \sum_{i=1}^{p} \alpha_j^{(i)} e_i$$

with $\alpha_j^{(i)} \in k$, *then*

$$\mathbf{L}(\operatorname{Im} \mathbf{a} \bmod \operatorname{Im} \mathbf{b}) \geqq (\operatorname{trdg}_{k_0} k_0(\boldsymbol{\alpha}))^{1/2} - (r + s).$$

Proof. Let

$$\mathbf{L}(\operatorname{Im} \mathbf{a} \bmod \operatorname{Im} \mathbf{b}) = t. \tag{3.7}$$

By Theorem 3.1, $\overline{C_{r,\operatorname{Im}\mathbf{b}}(t)}$ is an irreducible subvariety of A_r such that

$$\begin{aligned} \dim \overline{C_{r,\operatorname{Im}\mathbf{b}}(t)} &\leqq (s + t + 1)(s + t + r) - s(s + 1) \\ &\leqq (t + r + s)^2. \end{aligned} \tag{3.8}$$

Theorem 3.1 also states that $C_{r,\operatorname{Im}\mathbf{b}}(t)$ is stable with respect to $\operatorname{Gal}(k/k_0)$. Since the elements $\sigma \in \operatorname{Gal}(k/k_0)$ are continuous, $\overline{C_{r,\operatorname{Im}\mathbf{b}}(t)}$ is also stable. Therefore $\overline{C_{r,\operatorname{Im}\mathbf{b}}(t)}$ is k_0-closed. (By 3.7)

$$\mathbf{a} \in \overline{C_{r,\operatorname{Im}\mathbf{b}}(t)}.$$

Using R.3, we get

$$\dim \overline{C_{r,\operatorname{Im}\mathbf{b}}(t)} \geqq \operatorname{trdg}_{k_0} k_0(\boldsymbol{\alpha}).$$

The assertion follows now from (3.8).

In the following, we assume $k_0 = \mathbb{Q}$. Further, let $(e_1, \cdots, e_p)$ be a basis of the k_0-vector space A_0 such that $e_1 = 1$ and

$$\sigma_{i,j}^{(l)} \in \mathbb{Z} \tag{3.9}$$

(see (3.1)). We set

$$\lambda := \max_{i,j} \sum_{l} |\sigma_{ij}^{(l)}| \tag{3.10}$$

and assume $\lambda \geqq 1$. Further, let $\mathbf{b} = (b_1, \cdots, b_s) \in A_0^s$, $b_j = \sum_{i=1}^{p} \eta_j^{(i)} e_i$ with $\eta_j^{(i)} \in \mathbb{Z}$. We set

$$\gamma := \sum_{i,j} |\eta_j^{(i)}| \tag{3.11}$$

and assume $\gamma \geqq 1$. Using the basis $(e_1, \cdots, e_p)$, we can identify the affine space A^r with $k^{pr} = k^p \times \cdots \times k^p$. If $y_1, \cdots, y_{pr}$ are its coordinate variables, then its coordinate ring is the polynomial ring $k[y_1, \cdots, y_{pr}]$. Then $\overline{C_{r,\mathrm{Im}\,\mathbf{b}}(t)}$ is a closed irreducible subvariety of k^{pr}; we denote its ideal by $J(t)$.

THEOREM 3.6. *Let $r, s \geqq 1$, $q \geqq 5$, $1 \leqq \delta_1, \cdots, \delta_q \leqq pr$; let g be a natural number such that*

$$g^{q-(t+r+s)^2} > 2^{(t+r+s)^2(t+1)} q^q \log(\lambda(\gamma+1)). \tag{3.12}$$

Then there exists a nontrivial form

$$H \in J(t) \cap \mathbb{Z}[y_{\delta_1}, \cdots, y_{\delta_q}]$$

such that

$$\text{height } H \leqq 3, \quad \text{degree } H = g.$$

Proof. We use the proof of Theorem 3.1. By replacing the $y_j^{(i)}$ of that proof by $\eta_j^{(i)}$, we get polynomials $P_\rho^{(\nu)}(\boldsymbol{\eta}, \cdot) \in \mathbb{Q}[\mathbf{z}]$. The construction implies that these polynomials are elements of $\mathbb{Z}[\mathbf{z}]$. If we set

$$P_{(\rho-1)p+\nu} := p_\rho^{(\nu)}(\boldsymbol{\eta}, \cdot),$$

then by (3.4) the systems of values of $P_1, \cdots, P_{pr}$ form exactly the set $C_{r,\mathrm{Im}\,\mathbf{b}}(t) \subset k^{pr}$. An element $H \in \mathbb{Z}[y_{\delta_1}, \cdots, y_{\delta_q}]$ is therefore contained in $J(t)$ if and only if

$$H(P_{\delta_1}, \cdots, P_{\delta_q}) = 0. \tag{3.13}$$

The construction of the polynomials $P_\rho^{(\nu)}(\boldsymbol{\eta}, \cdot)$ also yields (the proof goes by induction on t) that

$$\max_{1 \leqq \nu \leqq p} \text{degree } P_\rho^{(\nu)}(\boldsymbol{\eta}, \cdot) \leqq 2^{t+1} - 1$$

and

$$\sum_{\nu=1}^{p} \text{weight } P_\rho^{(\nu)}(\boldsymbol{\eta}, \cdot) \leqq 2^{\log(\lambda(\gamma+2))2^t - \log\lambda} - 1$$

for all ρ. We can apply Lemma 2.3 with $m = N, c = 2^{t+1}, f = 2^{\log(\lambda(\gamma+1))2^{t+1}}$. Since (2.1) is a consequence of (3.12), there is a nontrivial form $H \in \mathbb{Z}[y_{\delta_1}, \cdots, y_{\delta_q}]$ of degree g and height $\leqq 3$ satisfying (3.13), i.e., H is an element of $J(t)$.

COROLLARY 3.7. *Interpret $\mathbb{C}[x]$ as $\mathbb{C}$-ring over $\{x\}$. Then for $d \to \infty$ (for large d, respectively)*

$$L\left(*\left|\sum_{\delta=0}^{d} \exp(2\pi i/2^\delta) x^\delta\right.\right) \gtrsim d^{1/3}, \tag{3.14}$$

$$L\left(*\left|\sum_{\delta=0}^{d} 2^{2^\delta} x^\delta\right.\right) \gtrsim d^{1/3} \tag{3.15}$$

$$L\left(*\left|\sum_{\delta=0}^{d} 2^{2^{\delta d^3}} x^\delta\right.\right) > \sqrt{d} - 3. \tag{3.16}$$

Proof. Let $a \in \mathbb{C}[x]$ be of degree d. Then

$$L_{\mathbb{C}[x]}(*|a) \geqq L_A(*|a \bmod \xi),$$

where $A = \mathbb{C}[x]/x^{d+1}\mathbb{C}[x]$ is interpreted as a $\mathbb{C}$-ring and ξ is the residue class of x. Furthermore,

$$A = (\mathbb{Q}[x]/x^{d+1}\mathbb{Q}[x]) \otimes_{\mathbb{Q}} \mathbb{C}.$$

Therefore we can apply Theorem 3.6 with $p = d + 1$, $e_1 = \xi^{i-1}$,

$$\sigma_{ij}^{(l)} = \begin{cases} 1 & \text{if } i + j - 1 = l, \\ 0 & \text{otherwise,} \end{cases}$$

$\lambda = 1$, $s = 1$, $b_1 = \xi$, $\gamma = 1$ and $r = 1$. One concludes now as in the proofs of Corollaries 2.7, 2.11 and 2.12.

In a similar way, Theorem 3.6 applies to polynomial rings in several variables.

Acknowledgments. Many thanks go to Walter Baur and to the referee for their careful reading of the paper and their helpful remarks.

REFERENCES

[1] E. G. BELAGA, *Some problems involved in the computation of polynomials*, Dokl. Akad. Nauk SSSR, 123 (1958), pp. 775–777.
[2] A. BOREL, *Linear Algebraic Groups*, W. A. Benjamin, New York, 1969.
[3] J. EVE, *The evaluation of polynomials*, Numer. Math., 6 (1964), pp. 17–21.
[4] S. LANG, *Introduction to Algebraic Geometry*, Interscience, New York, 1958.
[5] T. S. MOTZKIN, *Evaluation of polynomials and evaluation of rational functions*, Bull. Amer. Math. Soc., 61 (1955), p. 163.
[6] D. MUMFORD, *Introduction to Algebraic Geometry*, Harvard Lecture Notes.
[7] M. PETERSON AND L. STOCKMEYER, *Bounds on the evaluation time for rational polynomials*, IEEE Conference Record of the 12th Ann. Sympos. of Switching and Automata Theory, 1971, pp. 140–143.
[8] T. SCHNEIDER, *Einführung in die Transzendenten Zahlen*, Springer-Verlag, Berlin, 1957.
[9] J. R. SHOENFIELD, *Mathematical Logic*, Addison-Wesley, Reading, Mass., 1967.
[10] V. STRASSEN, *Berechnung und Programm I*, Acta Informat., 1 (1972), pp. 320–335.
[11] ———, *Vermeidung von Divisionen*, Crelle Journal für die Reine und Angew. Mathematik, 264 (1973), pp. 184–202.

SIAM J. COMPUT.
Vol. 3, No. 2, June 1974

AN ALGORITHM FOR THE COMPUTATION OF LINEAR FORMS*

JOHN E. SAVAGE†

Abstract. Many problems, including matrix-vector multiplication and polynomial evaluation, involve the computation of linear forms. An algorithm is presented here which offers a substantial improvement on the conventional algorithm for this problem when the coefficient set is small. In particular, this implies that every polynomial of degree n with at most s distinct coefficients can be realized with $O(n/\log_s n)$ operations. It is demonstrated that the algorithm is sharp for some problems.

Key words. algorithms, linear forms, matrix multiplication, polynomial evaluation

1. Introduction. How many operations are required to multiply a vector by a known matrix or evaluate a known polynomial at one point? Such questions are frequently asked, and Winograd [1] has shown the existence of real matrices and polynomials (containing indeterminates over the rationals, for example) for which the standard matrix-vector multiplication algorithm and Horner's rule for polynomial evaluation are optimal. That is, n^2 real multiplications and $n(n-1)$ additions are required for some $n \times n$ matrices to multiply the matrix by an n-vector, and n multiplications and n additions are required by some polynomials of degree n to evaluate the polynomial. In this paper, we use an algorithm for the computation of "linear forms" to show that the $n \times n$ matrix-vector multiplication problem and the polynomial evaluation problem can be done with $O(n^2/\log_s(n))$ and $O(n/\log_s(n))$ operations, respectively, when the matrix entries and the polynomial coefficients are known and drawn from a set of size s (even when the entries and coefficients are variables). These results are obtained by exhibiting potentially different algorithms for each matrix and each polynomial.

The algorithm presented here for the computation of "linear forms" is very general and can be applied to many problems including matrix-matrix multiplication, the computation of sets of Boolean minterms, of sets of products over a group, as well as the two problems mentioned above. Applications of this sort are discussed in § 3.

We now define *linear forms*. Let S and T be sets and let R a "small" finite set of cardinality $|R| = s$. Let $\cdot : R \times S \to T$ be any map (call it multiplication) and let $+ : T \times T \to T$ be any associative binary operation (call it addition). Then the problem to be considered is the computation of the m *linear forms* in $x_1, x_2, \cdots, x_n$

$$a_{i1} \cdot x_1 + a_{i2} \cdot x_2 + \cdots + a_{in} \cdot x_n, \qquad 1 \leqq i \leqq m,$$

where $a_{ij} \in R$ and $x_j \in S$. The elements in R shall be regarded as symbols which

* Received by the editors January 16, 1973, and in final revised form November 1, 1973.

† The research reported here was completed primarily at the Division of Engineering and Center for Computer and Information Sciences, Brown University, Providence, Rhode Island, and in part at the Jet Propulsion Laboratory, Pasadena, California. The Brown portion was supported in part by National Science Foundation under Grant GJ-32270, and in part by the Advanced Research Projects Agency of the Department of Defense, which was monitored by U.S. Army Research Office, under Grant DA-ARO-D-31-124-73-G65. The JPL portion was supported by the National Aeronautics and Space Administration. Now on leave at the Department of Mathematics, Technological University, Eindhoven, Eindhoven, the Netherlands.

may be given any interpretation later. For example, in one interpretation, R may be a finite subset of the reals and, in another, R may consist of s distinct variables over a set Q, say.

An algorithm is given in the next section which for each $m \times n$ matrix of coefficients $A = \{a_{ij}\}$ evaluates the set $L_A(\mathbf{x}) = \{\sum_{j=1}^{n} a_{ij}x_j | 1 \leqq i \leqq m\}$ of linear forms with $O(mn/\log_s(m))$ operations, when m is large where $s = |R|$. The conventional direct evaluation of $L_A(\mathbf{x})$ involves mn multiplications and $m(n-1)$ additions, so an improvement is seen when s is small relative to m.

Polynomial evaluation is examined in § 4, and the algorithm for linear forms is combined with a decomposition of a polynomial into a vector-matrix-vector multiplication to show that every polynomial of degree n whose coefficients are taken from a set of s elements can be realized with about $\sqrt{n}\, s$ scalar multiplications, $2\sqrt{n}$ nonscalar multiplications and $O(n/\log_s(n))$ additions, when n is large. The polynomial decomposition is similar to one used by Paterson and Stockmeyer [2], and it achieves about the same number of nonscalar multiplications but uses fewer scalar multiplications and additions.

In § 5, a simple counting argument is developed to show that the upper bounds derived in earlier sections are sharp for matrix-vector multiplications by "chains", that is, straight-line algorithms.

2. The algorithm. The algorithms for computing $L_A(\mathbf{x})$, where $|R| = s$, will be given in terms of an algorithm $\mathscr{B}$ for the construction of all distinct linear forms in $y_1, y_2, \cdots, y_k$, with coefficients from R. That is, $\mathscr{B}$ computes $L_B(\mathbf{y})$, where B is the $s^k \times k$ matrix with s^k distinct rows and entries from R. The algorithm $\mathscr{A}$ for $L_A(\mathbf{x})$ will use several versions of $\mathscr{B}$.

The algorithm $\mathscr{B}$ has two steps. Let $R = \{\alpha_1, \alpha_2, \cdots, \alpha_s\}$. Then:

Step 1. Form $\alpha_i \cdot y_j$ $(1 \leqq i \leqq s, 1 \leqq j \leqq k)$.

Step 2. Let $S(i_1, i_2, \cdots, i_l) = \alpha_{i_1} \cdot y_1 + \alpha_{i_2} \cdot y_2 + \cdots + \alpha_{i_l} \cdot y_l$
$(1 \leqq l \leqq k, 1 \leqq i_j \leqq s.)$

Each element of $L_B(\mathbf{y})$ is equal to $S(i_1, i_2, \cdots, i_k)$ for some set $\{i_1, i_2, \cdots, i_k\}$ of not necessarily distinct integers in $\{1, 2, \cdots, s\}$. Construct $S(i_1, i_2, \cdots, i_l)$ recursively from

$$S(i_1) = \alpha_{i_1} \cdot y_1$$

and

$$S(i_1, i_2, \cdots, i_l) = S(i_1, i_2, \cdots, i_{l-1}) + \alpha_{i_l} \cdot y_l$$

for $2 \leqq l \leqq k$.

The first step uses $\pi_B = ks$ scalar multiplications. Let $N(s, l)$ be the number of additions to construct all linear forms $S(i_1, i_2, \cdots, i_l)$. Then from Step 2 it follows that

$$N(s, 1) = 0,$$

$$N(s, l) = N(s, l-1) + s^l.$$

From this we conclude that

$$N(s, l) = [(s^{l+1} - 1)/(s - 1)] - (s + 1) \leqq s^{l+1}$$
$$\geqq s^l$$

for $s \geqq 2$ and $l \geqq 2$. Therefore, the number σ_B of additions to form $L_B(\mathbf{y})$ satisfies $s^k \leqq \sigma_B \leqq s^{k+1}$.

Partition A into

$$A = [B_1 \quad B_2 \quad \cdots \quad B_p],$$

where $B_1, \cdots, B_{p-1}$ are $m \times k$, B_p is $m \times (n - (p-1)k)$ and $p = \lceil n/k \rceil$. Similarly, partition $\mathbf{x} = \mathbf{y}^1, \mathbf{y}^2, \cdots, \mathbf{y}^p$ where $\mathbf{y}^r = (x_{(r-1)k+1}, \cdots, x_{rk})$ for $1 \leqq r \leqq p-1$ and $\mathbf{y}^p$ is suitably defined. It follows that

$$(*) \qquad A\mathbf{x} = B_1\mathbf{y}^1 + B_2\mathbf{y}^2 + \cdots + B_p\mathbf{y}^p,$$

where $+$ denotes column vector addition.

The algorithm $\mathscr{A}$ for $L_A(\mathbf{x})$ has two steps:

Step 1. Construct $L_{B_r}(\mathbf{y}^r)$, $1 \leqq r \leqq p$, using $\mathscr{B}$, that is, identify the linear forms corresponding to rows of B_r and choose the appropriate forms from those generated by $\mathscr{B}$ on $\mathbf{y}^r$.

Step 2. Construct $L_A(\mathbf{x})$ by adding as per (*) above.

The number of multiplications used by $\mathscr{A}$ is $\pi_A = ns$. The number of additions used in Step 1 is no more than $p\sigma_B$, and in Step 2 it is no more than $m(p-1)$. Therefore, the number of additions used by $\mathscr{A}$ satisfies

$$\sigma_A \leqq ps^{k+1} + m(p-1),$$

where $p = \lceil n/k \rceil$. Ignoring diophantine constraints and with $k = \log_s(m/\log_s(m))$, we have the following.

THEOREM 1. *For each $m \times n$ matrix A over a finite set R of cardinality $|R| = s$, the m linear forms*

$$L_A(\mathbf{x}) = \{a_{i1} \cdot x_1 + \cdots + a_{in}x_n : 1 \leqq i \leqq m\}$$

can be computed with $\pi_A = ns$ multiplications and $\sigma_A \leqq O(mn/\log_s(m))$ additions when m is large relative to s.

Proof. Ignoring diophantine constraints, we have

$$\sigma_A \leqq n\left(\frac{s^{k+1} + m}{k}\right)$$

and $s^k = m/\log_s(m)$. Therefore,

$$\sigma_A \leqq \frac{nm}{\log_s(m)}(1 + \varepsilon_1)/(1 - \varepsilon_2),$$

where $\varepsilon_1 = s/\log_s(m)$ and $\varepsilon_2 = \log_s\log_s(m)/\log_s(m)$. If $\varepsilon_2 < \frac{1}{2}$, it is easily shown that $(1 - \varepsilon_2)^{-1} \leqq 1 + 2\varepsilon_2$. Also,

$$(1 + \varepsilon_1)(1 + 2\varepsilon_2) \leqq 1 + 2(\varepsilon_1 + \varepsilon_2)$$

if $\varepsilon_2 < \frac{1}{2}$, which holds for $m \geqq 16$ when $s \geqq 2$. It follows that

$$\sigma_A \leqq \frac{nm}{\log_s(m)}(1 + 2(\varepsilon_1 + \varepsilon_2))$$

where $m \geqq 16$. Since ε_1 and ε_2 approach zero with increasing m, the conclusion of the theorem follows. Q.E.D.

When $m \gg s$, this result represents a distinct improvement over the conventional algorithm for evaluating $L_A(\mathbf{x})$, which uses mn scalar multiplications and $m(n-1)$ additions. It should be noted that the reduction in the number of additions by a factor of $\log_s(m)$ obtained with algorithm $\mathscr{A}$ follows directly from a reduction by a factor of about k in algorithm $\mathscr{B}$. The obvious algorithm for $L_B(\mathbf{y})$ uses ks^k additions, but $\mathscr{B}$ computes it with no more than s^{k+1} additions.

Although algorithm $\mathscr{A}$ (and $\mathscr{B}$) was discovered independently by the author, it does represent a generalization of an algorithm of Kronrod reported in Arlazarov, et al. [3]. His result applies to the multiplication of two arbitrary Boolean matrices. The heart of algorithm $\mathscr{A}$ is algorithm $\mathscr{B}$, and this was known to the author [4] in the context of the calculation of all Boolean minterms in n variables. This will be discussed in the next section.

3. Applications. In the set $L_A(\mathbf{x})$ of linear forms, the elements a_{ij} and x_j are uninterpreted, as are the operations of multiplication and addition. By attaching suitable interpretations, it is seen that algorithm $\mathscr{A}$ for linear forms has applications to many different problems.

Several problems to which algorithm $\mathscr{A}$ may be applied are now described.

3.1. Multiplication of a vector by a known matrix. Let R be a set of s variables over S, $R = \{z_1, z_2, \cdots, z_s\}$, and let $S = T = \{\text{reals}\}$. Let $+$ and $\cdot$ be addition and multiplication on the reals. Then $L_A(\mathbf{x})$ represents multiplication of $\mathbf{x} = (x_1, x_2, \cdots, x_n)$ by a known (but not fixed) matrix A. That is,

$$L_A(\mathbf{x}) = \{z_{k_{i1}} \cdot x_1 + z_{k_{i2}} \cdot x_2 + \cdots + z_{k_{in}} \cdot x_n : 1 \leqq i \leqq m\},$$

where the $m \times n$ matrix of indices $\{k_{ij}\}$ is fixed. For any given matrix $\{k_{ij}\}$, $L_A(\mathbf{x})$ can be computed using ns real multiplications and $O(mn/\log_s(m))$ real additions.

Independent evaluation of the m forms requires a total of at least $m(n-1)$ operations for any s, since each form consists of n functionally independent terms.

Special cases.

(i) $z_1, z_2, \cdots, z_s$ are assigned distinct real values;

(ii) $s = 2, z_1 = 0, z_2 = 1$. Then $L_A(\mathbf{x})$ is a set of subset sums, such as

$$\{x_1 + x_3, x_2 + x_3 + x_4\}$$

Note. Concerning (i), Winograd [1] has shown that there exist fixed real (and unrestricted) $m \times n$ matrices and vectors $\mathbf{x}$ such that mn real multiplications and $m(n-1)$ real addition are required for their computation with "straight-line" algorithms. Thus, a significant savings is possible if the matrix entries can assume at most s distinct real values and s is small relative to m.

3.2. Matrix-matrix multiplication, AX, A known. Let R and S be as above, and let T be the p-fold Cartesian product $Q^p, Q = \{\text{reals}\}$. Let $\cdot$ be conventional scalar multiplication (consisting of p real multiplications), and let $+$ be vector addition on the reals (consisting of p real additions). Then, $L_A(\mathbf{x})$ represents multiplication of the $n \times p$ matrix X over the reals by a known (but not fixed) $m \times n$ matrix A. That is,

$$L_A(\mathbf{x}) = \{z_{k_{i1}} \cdot \bar{x}_1 + z_{k_{i2}} \cdot \bar{x}_2 + \cdots + z_{k_{in}} \cdot \bar{x}_n : 1 \leqq i \leqq m\},$$

where $\bar{x}_l$ denotes the lth row of X and the $m \times n$ matrix of indices $\{k_{ij}\}$ is fixed.

For any given matrix $\{k_{ij}\}$, $L_A(\mathbf{x})$ can be computed using nps real multiplications and $O(mnp/\log_s(m))$ real additions.

Note. When $n = m = p$, Strassen's [5] algorithm for matrix-matrix multiplication can be used at the cost of at most $(4.7)n^{\log_2 7}$ binary operations. As a consequence, Strassen's algorithm is asymptotically superior to algorithm $\mathscr{A}$ for this problem. However, when $s = 2$, algorithm $\mathscr{A}$ is the superior algorithm for $n \leqq 10^{10}$!! Moral: beware of arguments based upon asymptotics.

3.3. Boolean matrix multiplication. Let $R = S = \{0, 1\}$, $T = \{0, 1\}^p$, let $\cdot$ be Boolean vector conjunction and let $+$ be Boolean vector disjunction. Then, $L_A(\mathbf{x})$ represents the multiplication of a known Boolean $m \times n$ matrix A by an arbitrary Boolean $n \times p$ matrix X. That is,

$$L_A(\mathbf{x}) = \{a_{i1} \cdot \bar{x}_1 + a_{i2} \cdot \bar{x}_2 + \cdots + a_{in} \cdot \bar{x}_n : 1 \leqq i \leqq m\},$$

where $\bar{x}_l$ is the lth row of the $n \times p$ matrix X. The algorithm for computing AX uses no multiplications and $O(mnp/\log_2(m))$ additions.

Note. If A is an arbitrary Boolean matrix and if the selection procedure of Step 1 of algorithm $\mathscr{B}$ can be executed without cost, the algorithm of Kronrod [3] results. The number of operations performed, exclusive of selection, equals that given above. The Kronrod algorithm uses more operations because of a poor choice of the parameter k.

3.4. Boolean minterms. Let $R = S = T = \{0, 1\}$, let $+$ be Boolean conjunction, and let $\cdot$ be defined by

$$\alpha : x = \begin{cases} x, & \alpha = 1, \\ \bar{x}, & \alpha = 0, \end{cases}$$

where $\bar{x}$ denotes the Boolean inverse. Then, $L_A(\mathbf{x})$ represents a set of minterms such as

$$\{\bar{x}_1 x_2 x_3, \bar{x}_1 x_2 \bar{x}_3, x_1 \bar{x}_2 \bar{x}_3\}$$

Note. The set of all 2^n distinct minterms, suitably ordered, represents a map from the binary to positional representation of the integers $\{0, 1, 2, \cdots, 2^n - 1\}$. This map can be realized with at most 2^{n+1} conjunctions and is a map which is useful in many constructions, such as in [4].

3.5. Products in a group G. Let $R = \{-1, 0, 1\}$, $S = T = G$, let $\alpha \cdot x = x^\alpha$ (raise to a power), and let $+$ be group multiplication. Then $L_A(\mathbf{x})$ represents a set of m products of n terms each. For example,

$$\{ab^{-1}cd^{-1}, bc^{-1}, a^{-1}c^{-1}d^{-1}\}$$

is such a set, where x^{-1} is the group inverse of x and x^0 is the group identity which is suppressed.

4. Polynomial evaluation. We turn next to the evaluation of polynomials of degree n. Let

$$p(x) = a_0 + a_1 \cdot x^1 + a_2 \cdot x^2 + \cdots + a_n \cdot x^n,$$

where $+, \cdot$ represent vector addition and scalar multiplication and where $x^1 = x$, $x^i = x * x^{i-1}$, and $*$ represents vector multiplication. Let $a_i \in R, x \in T$ and

$$\cdot : R \times T \to T,$$
$$+ : T \times T \to T,$$
$$* : T \times T \to T,$$

where $+$ and $*$ are associative and $*$ distributes over $+$. We shall construct an algorithm $\mathscr{P}$ for polynomial evaluation which employs algorithm $\mathscr{A}$ for linear forms.

Algorithm $\mathscr{P}$ has three steps. Without great loss of generality, let $n = kl - 1$, and assume that $\mathbf{a} = (a_0, a_1, \cdots, a_n)$ has entries from a set of size s.

Step 1. Construct $x^2, x^3, \cdots, x^l$.

Step 2. Construct the k linear forms in $1, x, x^2, \cdots, x^{l-1}$

$$\begin{bmatrix} r_0(x) \\ r_1(x) \\ \cdot \\ \cdot \\ r_{k-1}(x) \end{bmatrix} = \begin{bmatrix} a_0 a_1 \cdots a_{l-1} \\ a_l a_{l+1} \cdots a_{2l-1} \\ \\ a_{(k-1)l} \cdots a_{kl-1} \end{bmatrix} \begin{bmatrix} 1 \\ x^2 \\ \cdot \\ \cdot \\ x_{l-1} \end{bmatrix}$$

using algorithm $\mathscr{A}$.

Step 3. Construct $p(x) = r_0(x) + r_1(x) * x^l + \cdots + r_{k-1}(x) * x^{(k-1)l}$ using Horner's rule.

Let σ_p denote the number of vector additions used by $\mathscr{P}$, π_p the number of scalar multiplications and μ_p the number of vector multiplications. Since the forms required in Step 1 can be realized with $l - 1$ vector multiplications and Step 3 with $k - 1$ such multipications and $k - 1$ additions, we have

$$\sigma_p \leqq O((n+1)/\log_s(k)) + k - 1,$$
$$\pi_p = ls,$$
$$\mu_p = l + k - 2,$$

since $kl = n + 1$. If we ignore diophantine constraints and choose $k = \sqrt{(n+1)}$, to minimize μ_p, we have the following.

THEOREM 2. *For each* $\mathbf{a} = (a_0, a_1, \cdots, a_n) \in R^{n+1}$ *with* R *a set of cardinality* $|R| = s$, $p(x) = a_0 + a_1 x + \cdots + x_n x^n$ *can be evaluated with* σ_p *vector additions,* π_p *scalar multiplications and* μ_p *vector multiplications, where*

$$\sigma_p \leqq O(n/\log_s(n)),$$
$$\pi_p \leqq s\sqrt{n+1},$$
$$\mu_p \leqq 2\sqrt{n+1} - 2,$$

when n *is large relative to* s.

Horner's rule, which is the optimal procedure for evaluating an arbitrary polynomial on the reals, uses n multiplications and n additions. Even when $\mathbf{a}$ and x assume fixed real values, there exist vectors $\mathbf{a}$ and values x for which Horner's

rule is still optimal [1]. When the coefficients are drawn from a set of size s, however, Horner's rule can be improved upon by a significant factor when n is large relative to s.

The decomposition of $p(x)$ used by algorithm $\mathscr{P}$ is very similar to that used by Paterson and Stockmeyer [2] in their study of polynomials with rational coefficients. They have shown that $O(\sqrt{n})$ vector multiplications are necessary and sufficient for such polynomials, but their algorithms use $O(n)$ additions. Algorithm $\mathscr{P}$ achieves $O(\sqrt{n})$ vector multiplications but requires only $O(n/\log_s(n))$ additions when n is large relative to s. Clearly, algorithm $\mathscr{P}$ can be applied to any problem involving polynomial forms.

5. Some lower bounds. The purpose of this section is to demonstrate the existence of problems for which the performance of algorithm $\mathscr{A}$ can be improved upon by at most a constant factor. To do this, we must carefully define the class of algorithms which are permissible. Then we count the number of algorithms using C or fewer operations and show that if C is not sufficiently large, not all problems of a given type (such as matrix-vector multiplication) can be realized with C or fewer operations.

A *chain* β is a sequence of steps $\beta_1, \beta_2, \cdots, \beta_L$ of two types, *data steps*, in which $\beta_i \in \{y_1, y_2, \cdots, y_n\} \cup K (y_i \notin K, y_i \neq y_j, i \neq j$ and $K \subset Q$ is a finite set of constants), or *computation steps*, in which

$$\beta_i = \beta_j \circ \beta_k, \qquad j, k < i$$

and $\circ : Q \times Q \to Q$ denotes an operation in a set Ω.

Associated with each step β_i of a chain is a function $\bar{\beta}_i$ which is β_i if β_i is a data step, and

$$\bar{\beta}_i = \circ(\bar{\beta}_j, \bar{\beta}_k)$$

if β_i is a computation step. Clearly, $\bar{\beta}_i \cdot Q^n \to Q$. A chain β is said to *compute* m functions $f_1, f_2, \cdots, f_m$, $f_i : Q^n \to Q$ if there exists a set of m steps $\beta_{i_1}, \cdots, \beta_{i_m}$ such that $\bar{\beta}_{i_l} = f_l$, $1 \leqq l \leqq m$.

We now derive an upper bound on the number $N(C, m, n)$ of sets of m functions $\{f_1, \cdots, f_m\}$ which can be realized by chains with C or fewer computation steps.

LEMMA. $N(C, m, n) \leqq v^{4v}$, $v = C + n + m + |K| + 1$ *if* $C \geqq |\Omega| \geqq 2$.

Proof. A chain will have $1 \leqq d \leqq n + |K|$ data steps, and without loss of generality, they may be chosen to precede computation steps. Similarly, the order of their appearance is immaterial, so there are most $\binom{n + |K|}{d} \leqq 2^{n+|K|}$ ways to arrange the d data steps.

Let the chain have t computation steps. Each step may correspond to at most $|\Omega|$ operations, and each of the two operands may be one of at most $t + d$ steps. Thus, there are at most $|\Omega|^t (t + d)^{2t}$ ways to assign computation steps and at most $2^{n+|K|}|\Omega|^t(t + d)^{2t}$ chains with d data steps and t computation steps. A set of m functions can be assigned in at most $(t + d)^m$ ways.

Combining these results, we have that the number of distinct sets of m functions which can be associated with chains which have C or fewer computation steps is

at most

$$N(C,m,n) \leqq \sum_{d=1}^{n+|K|} \sum_{t=1}^{C} 2^{n+|K|}|\Omega|^t(t+d)^{2t+m}$$
$$\leqq (n+|K|)C2^{n+|K|}|\Omega|^C(C+n+|K|)^{2C+m}.$$

But

$$(n+|K|)2^{n+|K|} \leqq (C+n+|K|)^{C+n+|K|}$$

if $C \geqq 2$, and

$$|\Omega|^C \leqq (C+n+|K|)^C$$

if $C \geqq |\Omega|$, from which it follows that

$$N(C,m,n) \leqq (C+n+|K|)^{4C+n+|K|+m+1}$$
$$\leqq v^{4v},$$

where $v = C+n+m+|K|+1$. Q.E.D.

In the interest of deriving a bound quickly, the counting arguments given above are loose. Nevertheless, the bound can at best be improved to about v^v. As seen below, this means a loss of a factor of about 4 in the complexity bound.

Consider the computation of m subset sums of $\{x_1, x_2, \cdots, x_n\}$, $x_i \in \{\text{reals}\}$, as defined in special case (ii) of § 3.1. In the chain defined above, let $Q = \{\text{reals}\}$ and let $\Omega = \{+, \text{addition on the reals}\}$. There are 2^n distinct subset sums, and the number of sets of m distinct subset sums is the binomial coefficient $F = \binom{2^n}{m}$. Fix $0 < \varepsilon < 1$. If C, n and m are such that $N(C,n,m) \leqq F^{1-\varepsilon}$, then there exists at least one set of m distinct subset sums which require C or more additions.

THEOREM 3. *Algorithm $\mathcal{A}$ is sharp for some problems, that is, there exist problems, e.g., the computation of m subset sums over the reals, which require $O(mn/\log_2(m))$ operations with any chain or "straight-line" algorithm, when $m = O(n)$.*

Proof. Set $v^{4v} = F^{1-\varepsilon}$, where $v = C+n+m+|K|+1$. Then $N(C,m,n) \leqq F^{1-\varepsilon}$. For large F, the solution for v is

$$v = (\tfrac{1}{4}\ln F^{1-\varepsilon})/\ln(\tfrac{1}{4}F^{1-\varepsilon}).$$

Since $m = O(n)$, it can be shown from Stirling's approximation to factorials and an examination of the binomial coefficient F that $\ln F$ is asymptotic to $nm(\ln s)$. From this the conclusion follows. Q.E.D.

The counting argument given above could also be applied to matrix-vector multiplication (§ 3.1) and to polynomial evaluation on the reals, as described in § 4, to show that the upper bounds given for these problems are also sharp.

6. Conclusions. The algorithm presented here for the evaluation of a set of linear forms derives its importance from the minimal set of conditions required of the two operations. In fact, the only condition required is that addition be associative. As a consequence, the algorithm applies to a large class of apparently disparate problems having in common the fact that they can be represented in terms of linear forms of this general nature.

The algorithm allows us to treat two important problems, matrix multiplication with a known matrix and polynomial evaluation with a known polynomial. In both cases, an algorithm is constructed which depends explicitly on the matrix entries and the polynomial coefficients. When the entry set of the matrix or of the polynomial coefficients is fixed and the dimensions of either problem are large, a sizable savings in the number of required computations is obtained.

The generality of the algorithm for evaluation of linear forms suggests that it may have application to many important problems not mentioned in this paper.

Acknowledgment. The author acknowledges several important conversations with Dr. Charles M. Fiduccia which resulted in the generalization and clarification of the principal algorithm for linear forms.

REFERENCES

[1] S. WINOGRAD, *On the number of multiplications necessary to compute certain functions*, Comm. Pure and Applied Math., 23 (1970), pp. 165–179.

[2] M. PATERSON AND L. STOCKMEYER, *Bounds on the evaluation time for rational polynomials*, Proc. 12th IEEE Symposium on Switching and Automata Theory, 1971, pp. 140–143.

[3] V. L. ARLAZAROV, E. A. DINIC, M. A. KRONROD AND I. A. FARADZEV, *On economical construction of the transitive closure of an oriented graph*, Soviet Math. Dokl., 11 (1970), pp. 1209–1210.

[4] J. E. SAVAGE, *Computational work and time on finite machines*, J. Assoc. Comput. Mach. (1972), p. 673.

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

SIAM J. COMPUT.
Vol. 3, No. 3, September 1974

ANALYSIS OF A FEEDBACK SCHEDULER*

Y. S. CHUA† AND A. J. BERNSTEIN‡

Abstract. A flexible feedback queueing model for a computer system is described. The model consists of a single server and a queue into which jobs are inserted at positions which are functions of their attained service. Special cases of the model include both the round robin and the first-come-first-served disciplines, but a wide variety of other algorithms, having different performance characteristics, can also be obtained. The model is analyzed using Markovian assumptions, and both the finite quantum and processor sharing cases are considered. The relationship of the model to system overhead is also treated.

Key words. scheduling, Markov analysis, feedback, queueing, time-sharing, partial round robin

1. Introduction. During the last decade a considerable amount of work has been done in developing and analyzing queueing models for time-sharing systems [8]. Figure 1 shows a general model which consists of a single server that is shared by a number of customers. A newly arrived customer joins the queue and waits for his turn to be served. The server follows some decision rules in selecting the customer that is to be served next. In order to prevent customers requiring large amounts of service from tying up the server and thus causing delays for other customers with shorter service times, each customer is served for only a short period of time (called a *quantum*) and is asked to rejoin the queue if the service is not completed within the quantum. A customer thus alternates between waiting and receiving service until he has accumulated the total amount of service he requires.

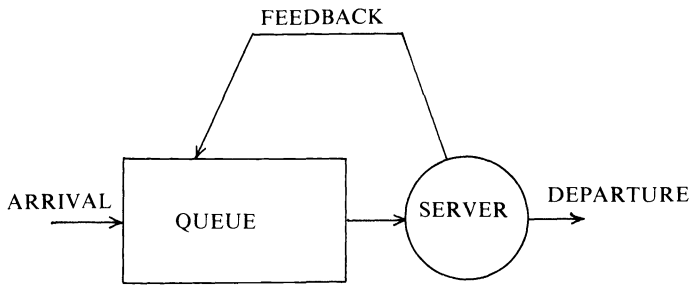


FIG. 1. *A general feedback queueing model*

The set of rules for selecting the customer to be served next is known as a scheduling algorithm or a queueing discipline. A customer who is rejoining the queue to await another quantum of service will be called a *feedback*.

The scheduling algorithm may be a function of one or more of the following:

1. Externally assigned priorities for the jobs.
2. Amount of service so far accumulated by the jobs.

* Received by the editors August 6, 1973. This work was supported in part by the National Science Foundation under Grant GJ 28177.

† Department of Mathematical Sciences, University of North Florida, Jacksonville, Florida 32211.

‡ Department of Computer Science, State University of New York at Stony Brook, Stony Brook, New York 11790.

3. The waiting times of the jobs.
4. The order of arrival of the jobs.

Jobs arrive in a random manner, each job having a service time which is a sample from some distribution function. In order to analyze a given queueing model, the statistics describing the arrival process and the service time distribution are needed. The result of analysis is usually a function, $W(t)$, giving the average waiting time a job experiences as a function of its service time t. The *waiting time for a job* is defined as the total amount of time spent by a job waiting in the queue excluding the time spent in receiving service. The sum of the waiting time and the service time for a job is known as the *average response time*

$$T(t) = W(t) + t. \tag{1}$$

Many models have been developed and analyzed [8], some treating specific models like the round robin scheduler [6], [7], and some providing multiple parameter models like the FB_N system [3].

In this report, we develop a multiple parameter queueing model which covers a wide class of scheduling algorithms. The model bears a strong relationship to the policy driven scheduling algorithm [2] since it involves inserting feedbacks into the queue at positions which are a function of the service they have received. The model is analyzed, yielding a response function. Some examples are presented to illustrate the flexibility of the model. The processor-sharing limit for a subclass of queueing disciplines is examined, and considerations relating to swapping are discussed. A synthesis procedure for the discrete model is described in [9]. This procedure determines a set of parameters for the model so that the resulting scheduling algorithm achieves a specified average response function.

2. The model. The model to be considered involves a single queue. The job at the head of the queue receives one quantum of service. If it does not terminate, it is returned to the queue at a position which depends on its attained service. We make the following assumptions in order to facilitate the analysis.

Assumption 1. Time is quantized into Q second intervals.

Assumption 2. The quantum size to be used by the server is equal to Q seconds, the discrete time interval.

Assumption 3. At most, one new job can arrive at the end of a Q second interval with probability $\lambda_0 Q$. This is the discrete time counterpart of the Poisson process.

Assumption 4. Each job requires exactly nQ seconds of service, where n is an integer. The service time is a random variable which is independent of the arrival process and is a sample from a geometric distribution. Thus the probability that a new arrival requires n quanta is $(1 - \sigma)\sigma^{n-1}$, where σ is the feedback probability or the probability that a job that has just received a quantum of service does not terminate.

Assumption 5. There will be no hold back, i.e., the server will not be idle as long as there is work to be done.

Assumption 6. The system is in the steady state.

Assumption 7. All overhead is negligible.[1]

From Assumption 4, the average service time is given by

$$\bar{t} = Q/(1 - \sigma). \tag{2}$$

Since the average arrival rate is λ_0 jobs per second, it follows that the utilization factor is

$$\rho = \lambda_0 Q/(1 - \sigma). \tag{3}$$

Only systems with $\rho < 1$ will be considered.

The model makes use of a single queue whose positions are numbered as shown in Fig. 2. The head of the queue will be referred to as position 1 and will always contain the job currently being served, if any.

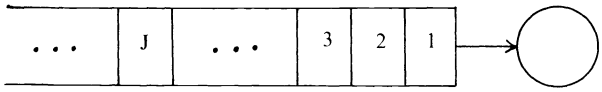


FIG. 2. *Labeling the positions in the queue*

DEFINITION. Jobs which have received exactly iQ seconds of service and have not finished are said to be *in group i* until they have completed the next quantum. A job that has just received iQ seconds of service and does not depart is said to be *joining group i.*

To specify the scheduling algorithm, we associate with group i a position in the queue denoted by π_i. A job joining group i will not be placed farther than π_i positions from the server. At the end of a Q second interval, the following insertions may occur:

1. A feedback and an arrival.
2. A feedback alone.
3. An arrival alone.

The feedback will of course be joining a specific group i, $i \geqq 1$, while the arrival will be joining group 0. Consider the case where there is a feedback as well as an arrival at the end of a Q second interval. If $\pi_i < \pi_0$, then we shall insert the two jobs in the following way. As soon as the job to be fed back leaves position 1, all the jobs occupying positions k, where $k \leqq \pi_i$, will be moved one position closer to the server. The job that is joining group i will then be placed in position $H(i)$, where

$$H(i) = \min \{\pi_i, n' + 1\}, \qquad i \geqq 1,$$

and n' is the total number of jobs currently in the queue (not including the feedback and the arrival). Following the insertion of the feedback, the arrival will then be placed in position $H(0)$, where

$$H(0) = \min \{\pi_0, n' + 2\}.$$

[1] It is possible to include in the quantum size, Q, a fixed amount of overhead, known as a *set up time* [10], without disturbing the analysis.

In the process, jobs that are in position k, where $k \geqq \pi_0$, will be moved one position backward to make room for the arrival. If $\pi_i > \pi_0$, then the order of insertion will be reversed:

$$H(0) = \min \{\pi_0, n' + 1\}$$

and

$$H(i) = \min \{\pi_i, n' + 2\}, \qquad i \geqq 1,$$

where n' is now interpreted as the number of jobs in the queue as seen by the first job to be inserted. In order to avoid ambiguity, we exclude algorithms in which $\pi_0 = \pi_i$ for some $i > 0$.

Consider the case where there is only one job to be inserted at the end of a Q second interval. Such a job may be an arrival or a feedback. Assuming that this job is joining group i, then all the jobs occupying positions $k \leqq \pi_i$ will be moved one position forward followed by the insertion of the tagged job in position $H(i)$, where

$$H(i) = \min \{\pi_i, n' + 1\}, \qquad i \geqq 0.$$

In any case, a job that finishes at the end of a Q second interval will be ejected from the system immediately and will not be included in the determination of n'. Note that the algorithm follows neither the early arrival nor the late arrival rules as defined by Kleinrock [6], but rather a mixture of the two. For a job that is joining group i, where $\pi_i < \pi_0$, the late arrival rule applies (i.e., the feedback is inserted before the new arrival); for jobs joining group i, where $\pi_i > \pi_0$, the early arrival rule applies (i.e., the new arrival is inserted before the feedback). Note also that due to the fact that jobs may be inserted in the queue, some jobs may move forward one position; some may remain in their original positions for another Q second interval, while others may move one position backward. A scheduling algorithm is completely specified by the sequence $\{\pi_0, \pi_1, \pi_2, \cdots\}$, where π_i is an integer indicating a position in the queue. Although, since $\rho < 1$, the queue is not expected to reach an infinite size, we allow ∞ as an acceptable value for π_i, indicating an insertion point at the end of the queue. For the purpose of precisely specifying some scheduling algorithms, we introduce the notation ∞^- to differentiate between two possible insertion points at the end of the queue. Thus if $\pi_i = \infty$ and $\pi_j = \infty^-$, then $\pi_i > \pi_j$. The usage of ∞ and ∞^- will be illustrated in the following examples.

Example 1. First-come-first-served scheduling (FCFS) can be specified by setting $\pi_0 = \infty$ and $\pi_i = 1$, $i \geqq 1$. This simply means that all new arrivals join the end of the queue, while subsequent feedbacks will go to position 1.

Example 2. The late-arrival round robin scheduling system is specified by setting $\pi_0 = \infty$ and $\pi_i = \infty^-$, $i \geqq 1$. Arrivals as well as feedbacks are required to join the end of the queue. However, since $\pi_i < \pi_0$, $i \geqq 1$, a feedback (if any) is allowed to join the end of the queue before an arrival.

The early-arrival round robin scheduling system is obtained by setting $\pi_0 = \infty^-$ and $\pi_i = \infty$, $i \geqq 1$.

Example 3. The algorithm obtained by setting $\pi_0 = \infty$ and $\pi_i = k$ (where k is finite), $i \geqq 1$, is intermediate between the FCFS and the round robin (RR)

schedulers. We shall refer to this as the partial round robin (PRR) scheduler. Up to a maximum of k jobs are allowed to receive service in a round robin fashion; other jobs wait on a FCFS basis until one of the k jobs terminates, at which time the job at the head of the waiting line (i.e., position $k + 1$) will join the $k - 1$ jobs already in service.

Example 4. It is possible to vary the quantum size to some integral multiple of Q by specifying some successive values of π_i to be 1. For instance, we may have a simple RR scheduler which has a varying quantum size depending on the number of passes a job has made through the queue. During the first pass, the server gives the job one quantum of service; during the nth pass, n quanta. The sequence π is given by

$$\pi_i = \begin{cases} \infty, & i = 0, \\ \infty^-, & i = 1,3,6,10,15,\cdots, \\ 1, & i = 2,4,5,7,8,9,11,12,13,14,\cdots. \end{cases}$$

Example 5. As it is the purpose of time-sharing systems to provide faster service for short jobs at the expense of longer ones, a reasonable scheduling algorithm is specified by assigning to π_i a monotonically increasing integer function of i, e.g.,

$$\pi_i = a + bi + ci^2,$$

where a, b and c are positive integers.

3. Analysis. In order to analyze the model, it is necessary to study the behavior of jobs in the system, i.e., the variation of the number of jobs in the system and the movement of jobs in the queue. Since service time is geometrically distributed, the variation of the number of jobs in the system does not depend on the scheduling algorithm (due to the memoryless property of the geometric distribution). However, depending on when the system is observed, we have different probabilities of finding n jobs in the system. If the system is observed in the middle of every Q second interval, the probability of finding n jobs in the system is [6]

$$P'(n) = \begin{cases} 1 - \rho, & n = 0, \\ \dfrac{1 - \rho}{\sigma}\alpha^n, & n > 0, \end{cases} \tag{4}$$

where $\alpha = \rho\sigma/(1 - \lambda_0 Q)$. The expected number of jobs in the system is given by

$$E'(n) = \rho(1 - \lambda_0 Q)/(1 - \rho). \tag{5}$$

Let F be the set whose elements are the integers i such that $\pi_i < \pi_0$. Let λ_i denote the average rate at which jobs join group i, $i \geqq 0$. λ_0, the rate at which jobs join group 0, is known. Out of the λ_0 jobs that arrive every second, only those requiring more than iQ seconds of service will ultimately join group i. Thus, the average rate at which jobs enter group i is given by

$$\lambda_i = \lambda_0 \sum_{n=i+1}^{\infty} (1 - \sigma)\sigma^{n-1}$$

or

$$\lambda_i = \lambda_0 \sigma^i. \tag{6}$$

The probability of a feedback into group i, given that the system is busy, is $\lambda_i Q/\rho$. It can be verified that

$$\sum_{i=1}^{\infty} \frac{\lambda_0 Q \sigma^i}{\rho} = \sigma. \tag{7}$$

We will denote the probability of a feedback into group i, where $i \in F$, by f. It follows that

$$f = \sum_{i \in F} \frac{\lambda_0 Q \sigma^i}{\rho}. \tag{8}$$

DEFINITION. $P_a(n)$ = Prob (at the time a new arrival is inserted in the queue there are n jobs in the system).

DEFINITION. $P_{fi}(n)$ = Prob (at the time a feedback into group i is inserted in the queue there are n jobs in the system).

There will be n jobs in the system as seen by an arrival if during the Q second interval just before the arrival there were n jobs in the system and the job receiving service (if any) is fed back to group i, $i \in F$, at the end of the Q second interval. The arrival will also see n jobs in the system if there were $n + 1$ jobs during the previous Q second interval and the job in service terminates or is fed back to group i, $i \notin F$, at the end of the quantum. Thus

$$P_a(n) = \begin{cases} P'(0) + (1 - f)P'(1), & n = 0, \\ fP'(n) + (1 - f)P'(n + 1), & n > 0. \end{cases} \tag{9}$$

In order to determine $P_{fi}(n)$, we make a simplifying assumption that the queue length at each pass is independent of the queue lengths of previous passes.[2] The probability of a feedback seeing n jobs in the system will depend on i, the number of the group that the job is joining. If $i \in F$, then an arrival does not affect the number of jobs seen by the feedback. On the other hand, if $i \notin F$, then the arrival must be taken into account in determining $P_{fi}(n)$.

$$P_{fi}(n) = \begin{cases} P'(n + 1)/\rho, & i \in F, \quad n \geqq 0, \\ (1 - \lambda_0 Q)P'(1)/\rho, & i \notin F, \quad n = 0, \\ (1 - \lambda_0 Q)P'(n + 1)/\rho + \lambda_0 Q P'(n)/\rho, & i \notin F, \quad n > 0. \end{cases} \tag{10}$$

[2] This approximation is made so that the analysis of the model remains manageable. Another approach would be to adapt the analysis used by Kleinrock [5]. Here the average value of the queue size on the ith pass is determined as a function of its value on the previous pass. This approach, however, does not take into account the probability distribution of the queue size on each pass (cf. (9), (10)) which is important for the model under consideration in this report since jobs are not necessarily fed back to the end of the queue. A more exact model would take into account both the dependence of queue sizes in successive passes and the distribution function for the queue size at each pass. Analysis of such a model would be extremely complicated, however.

DEFINITION. $P_A(i,j)$ = Prob (a job entering group i is inserted into position j under Algorithm A).[3] Using (9) and (10), we find that

$$P_A(i,j) = \begin{cases} P_a(j-1), & i = 0,\ j < \pi_0, \\ \sum_{m=\pi_0-1}^{\infty} P_a(m), & i = 0,\ j = \pi_0, \\ P_{fi}(j-1), & i > 0,\ j < \pi_i, \\ \sum_{m=\pi_i-1}^{\infty} P_{fi}(m), & i > 0,\ j = \pi_i, \\ 0, & i \geqq 0,\ j > \pi_i. \end{cases} \tag{11}$$

DEFINITION. t_j = average time required for a job placed in position j to reach the server.

DEFINITION. w_i = average time required for a job joining group i to receive the next quantum of service.

Given t_j and $P_A(i,j)$, w_i and the response function under Algorithm A, $T_A(nQ)$, can be calculated using the relations

$$w_i = \sum_{j=1}^{\pi_i} t_j P_A(i,j) + Q, \tag{12}$$

$$T_A(nQ) = \sum_{i=0}^{n-1} w_i. \tag{13}$$

To solve for t_j, we consider the random walk shown in Fig. 3. State j in Fig. 3 denotes position j in the queue. The transition period is Q seconds and the transition probabilities are defined as follows:

a_j = Prob (a job currently at position j will advance to position $j-1$ at the end of the current Q second interval);

r_j = Prob (a job currently at position j will remain in position j at the end of the current Q second interval);

b_j = Prob (a job currently at position j will back up to position $j+1$ at the end of the current Q second interval).

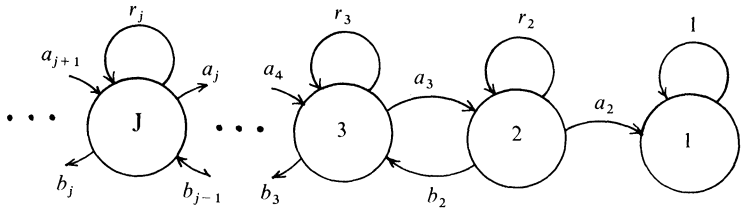


FIG. 3. *Random walk model for the movement of jobs in the queue*

Given a queueing discipline in the form of a sequence π, we can calculate the transition probabilities a_j, r_j and b_j. Let π^j and $\pi^{j'}$ be sets satisfying

$$\pi^j = \{i | \pi_i < j,\ i \neq 0\}; \qquad \pi^{j'} = \{i | \pi_i \leqq j,\ i \neq 0\}.$$

[3] Algorithm A is an arbitrary scheduling algorithm specified by a sequence $\{\pi_0, \pi_1, \pi_2, \ldots\}$.

Thus π^j contains the indices of feedback points other than π_0, which are less than j (i.e., in front of position j) and $\pi^{j'}$ is the union of π^j and the set of indices of feedback points that have value equal to j. Note that π^j and $\pi^{j'}$ may be empty. The transition probabilities are given by

$$a_j = \begin{cases} 1 - g_j, & \pi_0 \geqq j > 1, \\ (1 - \lambda_0 Q)(1 - g_j), & j > \pi_0; \end{cases} \tag{14}$$

$$r_j = \begin{cases} g_j, & \pi_0 > j > 1, \\ (1 - \lambda_0 Q)g_j, & j = \pi_0, \\ (1 - \lambda_0 Q)g_j + \lambda_0 Q(1 - h_j), & j > \pi_0; \end{cases} \tag{15}$$

$$b_j = 1 - a_j - r_j, \qquad j > 1, \tag{16}$$

where

$$g_j = \sum_{i \in \pi^j} \frac{\lambda_0 Q \sigma^i}{\rho}, \qquad h_j = \sum_{i \in \pi^{j'}} \frac{\lambda_0 Q \sigma^i}{\rho}.$$

The mean time to absorption starting at state j in Fig. 3 is exactly the average time required for a job placed in position j to reach the server. The following system of equations [9] describes the situation:

$$t_j = \begin{cases} 0, & j = 1, \\ Q + r_j t_j + b_j t_{j+1} + a_j t_{j-1}, & j > 1. \end{cases} \tag{17}$$

Equation (17) is an infinite set of equations which has no general solution since we have only one boundary condition, $t_1 = 0$. We shall impose restrictions on the sequence π so as to obtain special subclasses of queueing disciplines characterized by solvable sets of equations for t_j.

4. Queueing disciplines with $\pi_0 = \infty$.

DEFINITION. A *unidirectional random walk* (URW) is an absorbing random walk on the integers $1, 2, 3, \cdots$ which satisfies the following conditions:

1. State 1 is absorbing.
2. $a_j > 0$ and $b_j = 0, j > 1$.

Consider the subclass of queueing disciplines in which all new arrivals are required to join the end of the queue, i.e., $\pi_0 = \infty$. In this case, it is impossible for a job to move backwards (i.e., $b_j = 0$). Furthermore, $a_j > 0$ for all j since the probability of having a feedback in front of j is less than unity. This subclass of queueing disciplines can therefore be modeled by a URW. Thus, the system of equations (17) reduces to the following:

$$t_j = \begin{cases} 0, & j = 1, \\ Q + r_j t_j + a_j t_{j-1}, & j > 1. \end{cases} \tag{18}$$

Solving for t_j using $r_j + a_j = 1$, we obtain

$$t_j = Q \sum_{m=2}^{j} \frac{1}{a_m}, \qquad j > 1. \tag{19}$$

Scheduling algorithms that can be modeled by a URW include the round robin, FCFS and the partial round robin schedulers. Analysis of each of these will be presented.

4.1. Round robin. Consider the late-arrival RR algorithm (RRLA). Since arrivals as well as feedbacks are required to join the end of the queue, it follows that $a_j = 1, j > 1$. Thus

$$t_j = (j-1)Q, \qquad j > 1. \tag{20}$$

Because of the late-arrival assumption, it follows from (4), (9), (10) and (11) that

$$P_{\mathrm{RRLA}}(i,j) = (1-\alpha)\alpha^{j-1}, \qquad i \geqq 0, \quad j \geqq 1. \tag{21}$$

Substituting (20) and (21) into (12), we obtain

$$w_i = \frac{Q\rho\sigma}{1-\rho} + Q, \qquad i \geqq 0. \tag{22}$$

The response function is given by

$$T_{\mathrm{RRLA}}(nQ) = nQ\left(\frac{\rho\sigma}{1-\rho} + 1\right), \qquad n \geqq 0. \tag{23}$$

Consider now the early arrival system (RREA); t_j is still given by (20). However,

$$P_{\mathrm{RREA}}(i,j) = \begin{cases} \dfrac{(1-\rho)(1+\rho\sigma)}{1-\lambda_0 Q}, & i = 0, \quad j = 1, \\ \dfrac{1-\rho}{\sigma}\alpha^j, & i = 0, \quad j > 1, \\ 1-\rho, & i > 0, \quad j = 1, \\ \dfrac{1-\rho}{\sigma}\alpha^{j-1}, & i > 0, \quad j > 1. \end{cases} \tag{24}$$

Consequently,

$$w_i = \begin{cases} \dfrac{Q\rho^2\sigma}{1-\rho} + Q, & i = 0, \\ \dfrac{Q\rho(1-\lambda_0 Q)}{1-\rho} + Q, & i > 0. \end{cases} \tag{25}$$

The response function is

$$T_{\mathrm{RREA}}(nQ) = \frac{Q\rho(\rho\sigma + (n-1)(1-\lambda_0 Q))}{1-\rho} + nQ, \qquad n > 0. \tag{26}$$

4.2. First-come-first-served. A job at position j in the FCFS system will move to position $j-1$ at the end of the current Q second interval with probability $1-\sigma$ (i.e., the probability that the job being served terminates after receiving the quantum). Therefore

$$a_j = 1-\sigma, \quad j > 1; \qquad r_j = \sigma, \quad j > 1.$$

Using (19), we obtain

$$t_j = Q(j-1)/(1-\sigma), \qquad j > 1. \tag{27}$$

From (4), (9), (10) and (11), it follows that

$$P_{\mathrm{FCFS}}(i,j) = \begin{cases} (1-\alpha)\alpha^{j-1}, & i = 0, \quad j \geqq 1, \\ 1, & i > 0, \quad j = 1, \\ 0, & i > 0, \quad j > 1. \end{cases} \tag{28}$$

Therefore

$$w_i = \begin{cases} \dfrac{Q\rho\sigma}{(1-\sigma)(1-\rho)} + Q, & i = 0, \\ Q, & i > 0. \end{cases} \tag{29}$$

The response function is given by

$$T_{\mathrm{FCFS}}(nQ) = \frac{Q\rho\sigma}{(1-\sigma)(1-\rho)} + nQ. \tag{30}$$

4.3. Partial round robin. In the PRR system, a job that is in any of the first k positions moves forward one position at the end of a quantum with probability 1, and one that is in position $j, j > k$, will move forward one position at the end of a quantum with probability $1 - \sigma$. Thus

$$a_j = \begin{cases} 1, & 1 < j \leqq k, \\ 1-\sigma, & j > k; \end{cases} \qquad r_j = \begin{cases} 0, & 1 < j \leqq k, \\ \sigma, & j > k. \end{cases}$$

Solving for t_j, we get

$$t_j = \begin{cases} (j-1)Q, & 1 \leqq j \leqq k, \\ \dfrac{(j-k)Q}{1-\sigma} + (k-1)Q, & j > k. \end{cases} \tag{31}$$

Also,

$$P_{\mathrm{PRR}}(i,j) = \begin{cases} (1-\alpha)\alpha^{j-1}, & i = 0, \quad j \geqq 1, \\ (1-\alpha)\alpha^{j-1}, & i > 0, \quad 1 \leqq j < k, \\ \displaystyle\sum_{j=k-1}^{\infty} (1-\alpha)\alpha^{j}, & i > 0, \quad j = k, \\ 0, & i > 0, \quad j > k. \end{cases} \tag{32}$$

Substituting (31) and (32) into (12), we obtain

$$w_i = \begin{cases} \displaystyle\sum_{j=1}^{k} (j-1)Q(1-\alpha)\alpha^{j-1} \\ \qquad + \displaystyle\sum_{j=k+1}^{\infty} Q\left(\frac{j-k}{1-\sigma} + k - 1\right)(1-\alpha)\alpha^{j-1} + Q, & i = 0, \\ \displaystyle\sum_{j=1}^{k} (j-1)Q(1-\alpha)\alpha^{j-1} \\ \qquad + (k-1)Q \displaystyle\sum_{m=k-1}^{\infty} (1-\alpha)\alpha^{j} + Q, & i > 0. \end{cases} \tag{33}$$

Solving for the response function and simplifying, we get

$$T_{PRR}(nQ) = \frac{Q\alpha^k}{(1-\sigma)(1-\alpha)} + nQ\left(\frac{1-\alpha^k}{1-\alpha}\right), \qquad n > 0. \tag{34}$$

It can be verified that as k approaches infinity, $T_{PRR}(nQ)$ approaches $T_{RRLA}(nQ)$. For values of k between 1 and ∞, we have a family of scheduling algorithms having the combined characteristics of the FCFS and the RR systems. An attractive feature of the PRR algorithm is its ability to control the amount of swapping in a time-shared system. If the RR algorithm is employed in a system in which the number of jobs exceeds the capacity of the core, then a swap must occur after each quantum. Overheads can be reduced by employing a PRR system with k equal to the number of jobs that can reside in core simultaneously. In this case, once a job is brought into core, it will not be swapped out until it terminates.

4.4. Comparison. Comparing the FCFS and the RR systems, Kleinrock [6] observed that jobs requiring less than the average amount of service get faster response in the RR system, while those requiring more than the average amount of service get faster service in the FCFS system. The crossover point is $Q/(1-\sigma)$, the average service time. By substituting $1/(1-\sigma)$ for n in (34), it can be shown that a job requiring the average service time will experience the same amount of waiting time regardless of the value of k. Jobs requiring less than the average service time get better response as k increases. We may regard k as a measure of the degree to which shorter jobs are favored at the expense of longer jobs. Typical plots of response functions for the FCFS, RRLA, RREA and PRR systems are shown in Fig. 4. The plots are actually discrete functions but are shown as continuous for illustrative purposes.

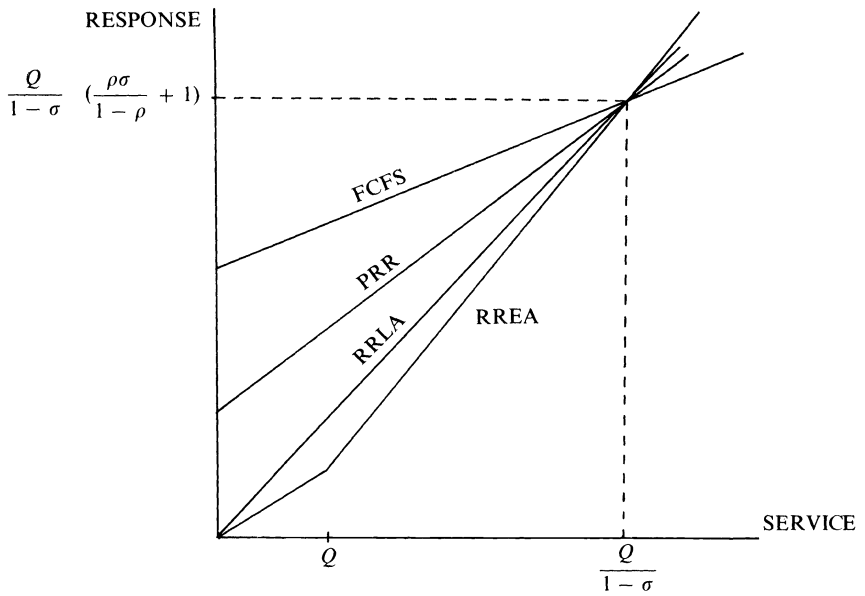


FIG. 4. *Response functions of the* RREA, RRLA, PRR *and* FCFS

5. Queueing disciplines with a finite number of feedback positions and $\pi_0 < \infty$. In this section, queueing disciplines with a finite number of distinct feedback positions will be considered. By a finite number of feedback positions we mean that although the sequence π is infinite, its elements are taken from a finite set of positive integers. Under this constraint, it is possible to relax the restriction $\pi_0 = \infty$. Let $\phi = \{\phi_1, \phi_2, \phi_3, \cdots, \phi_N\}$, where ϕ_i is chosen from the set of positive integers including ∞ and ∞^-. Furthermore, let the sequence ϕ satisfy $0 < \phi_1 < \phi_2 < \cdots < \phi_{N-1} < \phi_N \leqq \infty$. Let ϕ' be the largest finite element of ϕ. A queueing discipline with N distinct feedback positions can be specified by a finite sequence ϕ and an infinite sequence π, the indices of which may be decomposed into N mutually exclusive subsets S_k, $k = 1, 2, \cdots, N$, where $S_k = \{i | \pi_i = \phi_k\}$.

Figure 5 shows the random walk model for the subclass of algorithms that we are considering. In general, the transition probabilities for states j, where $j \leqq \phi'$, are different. Since ϕ' is the maximum finite feedback position, the transition probabilities for the states j, where $j > \phi'$, are identical.

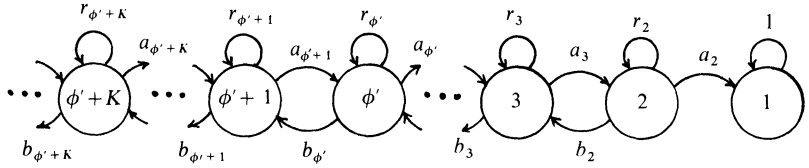


FIG. 5. *Random walk model for the movement of jobs in a queueing discipline with a finite number of feedback positions*

In order to solve for t_j, we split the random walk in Fig. 5 into two parts. A finite random walk is shown in Fig. 6. It contains states 1 through ϕ' of the original chain plus a new reflecting state D. Note that D is entered from state ϕ' with probability $b_{\phi'}$ and thus the transition probabilities of all states j, where $j \leqq \phi'$, of the original chain are preserved. State D represents the infinite portion of the original chain which has been omitted and the number of times it is entered will be equal to the number of times the omitted portion would have been entered in the original chain. We shall refer to the chain shown in Fig. 6 as the *F-chain.*

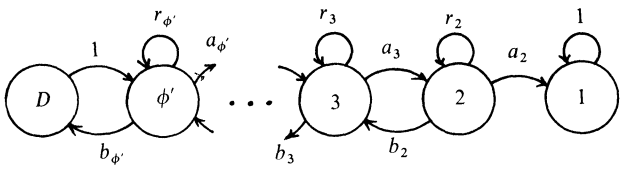


FIG. 6. *The F-chain*

An infinite chain is shown in Fig. 7 which contains states $\phi' + 1, \phi' + 2, \cdots$ of the original chain plus a new absorbing state D'. The average time to absorption, starting in state $\phi' + 1$, will be equal to the average time to go from state $\phi' + 1$ to state ϕ' in the original chain. We shall refer to the chain in Fig. 7 as the *I-chain.*

The *I*-chain is a random walk on the integers $\phi' + i, i \geqq 1$, satisfying $a_j = a$, $r_j = r$ and $b_j = b$ for $j > \phi'$. Note that since $\rho < 1$, a job inserted in the queue

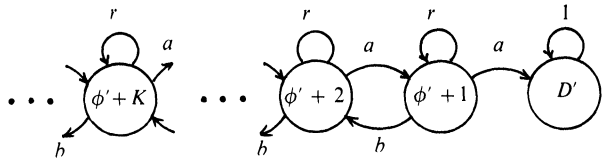


FIG. 7. *The I-chain*

must ultimately reach the server, implying that $a > b$. Under this constraint, it follows from [9] that the mean time to absorption, starting in state $\phi' + k$, is

$$m_{\phi+k} = kQ/(a - b). \tag{35}$$

Using finite Markov chain theory [5] it is possible to analyze the F-chain and, in particular, to calculate the average number of times state D is visited before absorption, given an arbitrary starting state. From (35) it follows that starting in state $\phi' + 1$ of the I-chain, it will take an average time of $Q/(a - b)$ seconds before absorption in state D' (i.e., before reaching state ϕ' of the F-chain). This gives the average amount of time spent in state D for each visit.

Let P be the transition matrix for the F-chain, where $P = \{p_{ij}\}$ and p_{ij} is the probability of being in state j after the next transition given that the current state is i. It follows that

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & & & 0 \\ a_2 & r_2 & b_2 & 0 & \cdots & & & 0 \\ 0 & a_3 & r_3 & b_3 & \cdots & & & 0 \\ 0 & 0 & a_4 & r_4 & \cdots & & & 0 \\ \vdots & \vdots & \vdots & \vdots & \cdots & & & \vdots \\ 0 & 0 & 0 & 0 & \cdots & a_{\phi'} & r_{\phi'} & b_{\phi'} \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 0 \end{pmatrix}. \tag{36}$$

Let G be the $\phi' \times \phi'$ matrix obtained from (36) by deleting the first row and first column, which correspond to the absorbing state, 1. Let $M = \{m_{ij}\}$ be the inverse of $(I - G)$, where I is the $\phi' \times \phi'$ identity matrix. It can be shown [5] that M exists and is given by

$$M = \sum_{k=0}^{\infty} G^k. \tag{37}$$

If we label the rows and columns of M using the integers $2, 3, 4, \cdots, \phi'$ and D, m_{ij} gives the average number of transitions into state j before absorption given that the starting state is i. Summing up the elements of row i of M therefore gives the average number of transitions before absorption, having started in state i. Note that for all states except D, the time between transitions is Q seconds. The

average time spent in D on each visit is $Q/(a - b)$ seconds. Let $\mathbf{v}$ be a vector of ϕ' elements given by

$$\mathbf{v} = \begin{pmatrix} Q \\ Q \\ Q \\ \vdots \\ Q \\ Q/(a-b) \end{pmatrix}. \tag{38}$$

Then, denoting the jth row of M as a row vector $\mathbf{M}_j$, it follows that

$$t_j = \begin{cases} \mathbf{M}_j \cdot \mathbf{v}, & 2 \leqq j \leqq \phi', \\ \dfrac{Q(j-\phi')}{a-b} + t_{\phi'}, & j > \phi'. \end{cases} \tag{39}$$

The response function can therefore be obtained using (11), (12), (13) and (39).

To illustrate the flexibility of the model, we have sketched in Fig. 8 the response function for a queueing discipline which is specified as follows:

$$\pi_i = \begin{cases} 5, & i = 0, \\ 10, & i = 1, 2, 3, \cdots, 10, \\ 20, & i = 11, 12, 13, \cdots, 20, \\ \infty, & i = 21, 22, 23, \cdots. \end{cases}$$

Note that there are four straight-line segments which correspond to the four feedback positions.

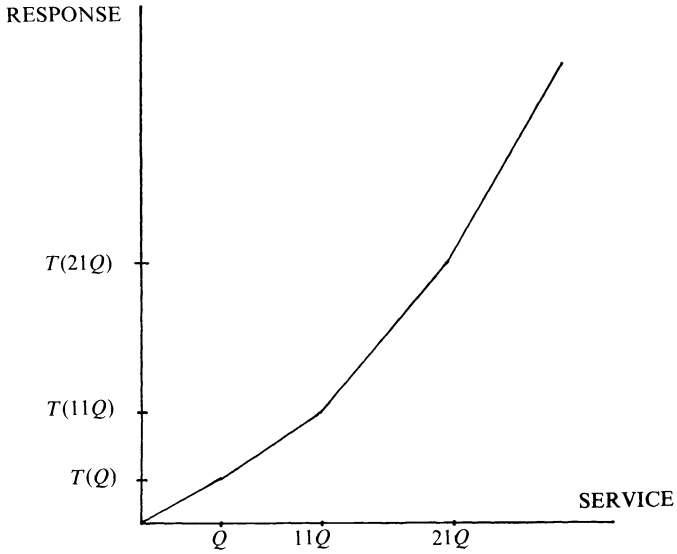


FIG. 8. *Response function of a scheduling algorithm with four distinct feedback positions*

6. Processor sharing. It is interesting to investigate the subclass of PRR algorithms as the quantum size goes to zero [7]. In the limit, the arrival process becomes Poisson and the service time exponential. Let λ_0 be the average arrival rate and $1/\mu$ the average service time. The following probabilities hold [4] for $\rho = \lambda_0/\mu < 1$:

$$\begin{aligned} P(n) &= \text{Prob } (n \text{ jobs in the system}) \\ &= (1 - \rho)\rho^n. \end{aligned} \tag{40}$$

$$\begin{aligned} P_b(n) &= \text{Prob } (n \text{ jobs in the system given that the system is busy}) \\ &= (1 - \rho)\rho^{n-1}. \end{aligned} \tag{41}$$

Consider a PRR system with parameter k. Up to a maximum of k jobs may be sharing the processor equally. A new arrival that finds fewer than k jobs in the system will immediately start service at the rate of 1 second of service for each n seconds, where n is the total number of jobs in the system including the arrival. If, upon arrival, the new job finds k or more jobs in the system, it joins the end of the queue and moves forward at the rate of μ positions per second, which is the rate at which jobs terminate. Thus, if $n \geqq k$, the average waiting time is $(n - k + 1)/\mu$ seconds. The average waiting time is therefore

$$W = \sum_{n=k}^{\infty} (1/\mu)(n - k + 1)(1 - \rho)\rho^n. \tag{42}$$

Let Z denote the average amount of time to receive one second of service once a job has started to receive service. Then

$$Z = \sum_{n=1}^{k-1} n(1 - \rho)\rho^{n-1} + k \sum_{m=k}^{\infty} (1 - \rho)\rho^{m-1}. \tag{43}$$

The response function is given by

$$T_{\text{PRR}}(t) = W + Zt. \tag{44}$$

Substituting the expressions for W and Z into (44) and simplifying, we obtain

$$T_{\text{PRR}}(t) = \frac{\rho^k}{\mu(1 - \rho)} + \frac{(1 - \rho^k)t}{1 - \rho}. \tag{45}$$

Examining $T_{\text{PRR}}(t)$ at extreme values of k, we find that

$$\lim_{k \to \infty} T_{\text{PRR}}(t) = t/(1 - \rho), \tag{46}$$

and, for $k = 1$,

$$T_{\text{PRR}}(t) = \frac{\rho}{\mu(1 - \rho)} + t. \tag{47}$$

Equations (46) and (47), as expected, are the response functions of the processor sharing RR system [4] and the FCFS system [7], respectively. Figure 9 shows the response functions for some members of the PRR family.

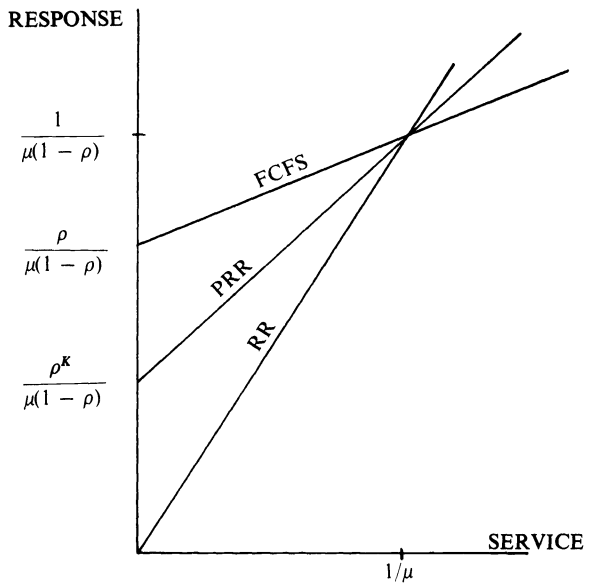


FIG. 9. *Response functions of the processor-shared* RR, PRR *and* FCFS *systems*

7. Swapping. A major source of overhead in time-shared computer systems arises when the size of core is not sufficient to accommodate all the jobs awaiting service. In this case, some jobs are held on a secondary storage device and are periodically exchanged (swapped) with jobs that are in core so that they can receive service in accordance with the scheduling algorithm. Since the model under consideration is an infinite source model, swapping will clearly be necessary. In this section we will obtain a measure of the rate at which swapping occurs as the result of implementing an algorithm specified by a particular sequence π.

We assume that a job which terminates must be swapped out of core. A job that arrives will be swapped in immediately if position $H(0)$ is in core. Otherwise, the arrival will be placed in the portion of the queue which is held on a secondary storage device and will be swapped in at a later time. Clearly, if the core size is infinite, the only swapping that goes on involves arrivals and departures. However, we shall consider a more realistic system that has a finite amount of core so that movement of jobs from the secondary storage device to core and vice versa must be taken into account in determining the swap rate. Since every job that is swapped in must ultimately be swapped out, the average swap in rate is equal to the average swap out rate. It is therefore sufficient to determine the swap in rate and double it in order to arrive at the average swap rate. Let J be the average number of jobs that can reside in core simultaneously. In the following analysis we shall assume that the first J positions in the queue are in core and the rest of the queue is on a secondary storage device.

Consider an algorithm in which $\pi_0 > J$. In this case, an arrival is swapped in immediately only when it sees a queue length that is less than J. The following

are mutually exclusive events which may happen at the end of a Q second interval and which involve swap ins:

1. There is an arrival and the queue size as seen by the arrival is less than J.
2. The queue size is greater than J during the middle of the Q second interval and the job receiving service is not fed back into a position in core, i.e., it either departs or is fed back into a position out of core.

Event 1 results in the swapping in of the arrival and event 2 causes the job at position $J + 1$ to be swapped in. There is no other event which will result in a swap in. Denoting the average swap rate by λ_s, it follows that $\lambda_s Q$ is the average number of swaps that must be performed for each Q second interval. For a scheduling algorithm in which $\pi_0 > J$, we have

$$\lambda_s Q = 2\left[\lambda_0 Q \sum_{n=0}^{J-1} P_a(n) + (1 - h_J) \sum_{n=J+1}^{\infty} P'(n)\right], \tag{48}$$

where

$$h_J = \sum_{i \in \pi^J} \frac{\lambda_0 Q \sigma^i}{\rho}.$$

Consider now the case in which $\pi_0 \leqq J$. An arrival will be swapped in immediately regardless of the queue length. Therefore, the events which involve swap ins are:

3. There is an arrival.
4. There is no arrival; the queue size is greater than J in the middle of the Q second interval; and the job being served is not fed back into a position in core.

Events 3 and 4 are mutually exclusive and cover all possible events that result in swap ins. Thus, for a scheduling algorithm in which $\pi_0 \leqq J$, we have

$$\lambda_s Q = 2\left[\lambda_0 Q + (1 - \lambda_0 Q)(1 - h_J) \sum_{n=J+1}^{\infty} P'(n)\right], \tag{49}$$

where h_J is as indicated for (48).

Equation (48) simplifies to the following:

$$\lambda_s Q = 2[\lambda_0 Q(1 - f\rho\alpha^{J-1} - (1 - f)\rho\alpha^J) + (1 - h_J)\rho\alpha^J], \tag{50}$$

where f is given by (8). Equation (49) can also be simplified and is given by

$$\lambda_s Q = 2[\lambda_0 Q + (1 - \lambda_0 Q)(1 - h_J)\rho\alpha^J]. \tag{51}$$

8. Conclusion. A flexible feedback queueing model has been developed and analyzed. Its wide range of performance is an attractive feature which can be very useful in the design of computer systems. The analysis was made possible using Markovian assumptions. Further work is required in analyzing the model using more realistic assumptions. A more general extension of the model which takes into consideration externally assigned priorities should prove to be an interesting and challenging problem.

REFERENCES

[1] A. J. BERNSTEIN AND J. C. SHARP, *A policy driven scheduler for a time sharing system*, Comm. ACM, 14 (1971), pp. 74–78.

[2] E. G. COFFMAN AND L. KLEINROCK, *Feedback queueing models for time-shared systems*, J. Assoc. Comput. Mach., 15 (1968), pp. 549–576.

[3] J. HSU, *Analysis of a continuum of processor-sharing models for time-shared computer systems*, Ph.D. dissertation, Computer Science Dept., Univ. of Calif. at Los Angeles, 1971.

[4] J. KEMENY AND J. SNELL, *Finite Markov Chains*, Van Nostrand, Princeton, N.J., 1960.

[5] L. KLEINROCK, *Analysis of a time-shared processor*, Naval Res. Logist. Quart., 11 (1964), pp. 59–73.

[6] ———, *Time-shared systems: A theoretical treatment*, J. Assoc. Comput. Mach., 14 (1967), pp. 242–261.

[7] J. M. MCKINNEY, *A survey of analytical time-sharing models*, Comput. Surveys, 1 (1969), pp. 105–116.

[8] E. PARZEN, *Stochastic Processes*, Holden-Day, San Francisco, 1962.

[9] Y. S. CHUA, *Analysis and synthesis of feedback queueing models for time sharing systems*, Ph.D. dissertation, State University of New York at Stony Brook, 1973.

[10] I. ADIRI, *Computer time-sharing queues with priorities*, J. Assoc. Comput. Mach., 16 (1969), pp. 631–645.

SIAM J. COMPUT.
Vol. 3, No. 3, September 1974

ISOMORPH REJECTION ON POWER SETS*

DAVID M. PERLMAN†

Abstract. If X is a finite set and G a finite group acting on X, then an action of G on $P(X)$, the set of all subsets of X, is induced in a natural way. An efficient generating algorithm is described which, when incorporated into a backtrack procedure, produces a system of distinct representatives for the action of G on $P(X)$.

Key words. isomorph rejection, pattern generating, backtrack generators, minimal s.d.r.

Introduction. Let X be a finite set and G a group of permutations of X. The group G defines an equivalence relation on X by

$$x \sim y \text{ if there exists a } g \in G \text{ such that } gx = y.$$

A system of distinct representatives (s.d.r.) for the equivalence classes or orbits is a subset $\Delta \subseteq X$ consisting of exactly one member of each orbit.

If D, R are finite sets and G acts on D, then G acts also on the set of functions R^D as follows:

$$\text{if } f \in R^D, \text{ then let } (gf)(d) = f(gd) \text{ for all } d \in D,$$

and Pólya's theorem [1] provides a means for computing the size of an s.d.r. Δ for G acting on R^D.

Side conditions imposed on this s.d.r. considerably complicate the computation. Let B_0 be any subset of R^D and define

$$B = \{gf \mid f \in B_0, g \in G\}.$$

Clearly B is a union of G-orbits in R^D, and if Δ is any s.d.r. for G acting on R^D, then we may let

$$\Delta_{B_0} = \{f \in \Delta \mid f \notin B\},$$

and it is clear that the size of Δ_{B_0} is well-defined. S. G. Williamson [2] has provided a means for determining the size of Δ_{B_0}; however, the computation involves the generation of an s.d.r. for the action of the group G on the power set of a certain set.

Another application for an algorithm that generates an s.d.r. for the action of a group on a power set is in the generation of Δ itself in the Pólya case, or Δ_{B_0} in the setting of Williamson. For example, if R has only two elements, then there is an obvious correspondence between R^D and $P(D)$. If $R = \{a, b\}$, then for $Y \subseteq D$, we associate $f \in R^D$, where $f^{-1}(a) = Y$. If $R = \{a_1, a_2, \cdots, a_m\}$, then the following recursive procedure can be used to generate Δ:

(i) generate all functions up to the action of G which assume one of the two values a_1 or (not a_1);

* Received by the editors April 23, 1973.

† Computer Center, University of California at San Diego, La Jolla, California 92037. This research was supported in part by the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under AFOSR Contract/Grant 71-2089.

(ii) for each function f generated in step (i), let $D' = D \setminus f^{-1}(a_1)$, let $G' = G_{f^{-1}(a_1)}$, the stabilizer subgroup of $f^{-1}(a_1)$, and let $R' = \{a_2, a_3, \cdots, a_m\}$. Apply the algorithm to G' acting on $R'^{D'}$.

If side conditions are to be imposed on Δ, then they may generally be applied at each stage of the recursive procedure described above, giving rise to a backtrack scheme. Backtracking is an efficient way of searching a tree for all terminal nodes satisfying prescribed conditions. Detailed discussions may be found in [3], [4] and [5]. Fundamentally, a backtrack algorithm consists of a generator which moves in an orderly fashion through the tree and a test phase which ultimately eliminates nodes produced by the generator which do not lie on the path to any desired terminal node. An example is presented in § 2. Section 1 provides the fundamental algorithm, and § 3 gives some experimental results.

1. A theorem.

THEOREM. *Let X be a finite set with an arbitrary ordering defined and G a group acting on X. For each positive integer $k \leqq |X|$, order the k-subsets of X lexicographically, and let Δ_{n-1} contain the minimal s.d.r. for the action on G on the $(n-1)$-subsets of X. Then the following procedure produces a set Δ_n which contains the minimal s.d.r. for the action of G on the n-subsets of X: for each element $U \in \Delta_{n-1}$,*

(i) *compute G_U, the stabilizer subgroup of U;*

(ii) *let X_U be the minimal s.d.r. for the action of G_U on $X \setminus U$;*

(iii) *if $U = (u_1 < u_2 < \cdots < u_{n-1})$, then for each $t \in X_U$ with $u_{n-1} < t$, include $(u_1 u_2 \cdots u_{n-1} t)$ in Δ_n.*

Proof. We first show that the procedure generates an s.d.r. for the n-subsets. Let $T = (s_1 s_2 \cdots s_n)$ be any n-subset of X, and denote $T \setminus \{s_j\}$ by T_j. We must show that an n-set equivalent to T is generated.

Let $U = \min \{V \in \Delta_{n-1} | V \sim T_j \text{ for some } j\}$; the minimum is taken with respect to the lexicographic ordering. Let $U = (u_1 u_2 \cdots u_{n-1})$. By definition of Δ_{n-1}, there exists an integer k and σ in G such that $\sigma : U \to T_k$; therefore $\sigma^{-1} s_k \in X \setminus U$. By definition of X_U there exists a $t \in X_U$ and a $\mu \in G_U$ such that $\mu t = \sigma^{-1} s_k$. We then have

$$\sigma\mu(u_1 u_2 \cdots u_{n-1} t) = \sigma(u_1 u_2 \cdots u_{n-1} \sigma^{-1} s_k) = T.$$

We must now show that $u_{n-1} < t$. Suppose $u_{n-1} > t$ ($u_{n-1} = t$ can never happen). Then there exists a smallest index j, $0 \leqq j < n-1$, such that $t < u_{j+1}$, and we have

$$W = (u_1 u_2 \cdots u_j t u_{j+1} \cdots u_{n-2}) < (u_1 u_2 \cdots u_{n-1}).$$

Let $V = (v_1 v_2 \cdots v_{n-1})$ be the smallest member of Δ_{n-1} equivalent to W, so $V \leqq W < U$, since Δ_{n-1} contains the minimal s.d.r.

But $\sigma\mu : W \to T_i$, where $\sigma\mu(u_{n-1}) = s_i$. Since $V \sim W$, we have $V \sim T_i$, and this violates the minimality of U.

We now show that the set Δ_n so generated in fact contains the smallest member of each orbit. To do this, we shall prove that for some t constructed as in the discussion above, $(u_1 u_2 \cdots u_{n-1} t) \leqq (s_1 s_2 \cdots s_n)$.

Let $V = \min\{W \in \Delta_{n-1} | W \sim T_n\}$. Since Δ_{n-1} contains the minimal s.d.r. for the $(n-1)$-subsets,

$$V = (v_1 v_2 \cdots v_{n-1}) \leqq (s_1 s_2 \cdots s_{n-1}),$$

and by its construction, $U \leqq V$, so we have $U \leqq V \leqq T_n$. If $U < T_n$, then clearly $(u_1 u_2 \cdots u_{n-1} t) < (s_1 s_2 \cdots s_n)$. If, on the other hand, $U = T_n$, then in the original construction of the set equivalent to T we may take $k = n$ and σ to be the identity. We then have $\sigma\mu = \mu \in G_U$ and $t \leqq s_n$, since t and s_n are in the same orbit of G_U and X_U is the minimal s.d.r. for G_U acting on $X \setminus U$. Thus

$$(u_1 u_2 \cdots u_{n-1} t) \leqq (s_1 s_2 \cdots s_{n-1} s_n). \quad \text{Q.E.D.}$$

For any computational environment, the algorithm requires two subprocedures:

(i) given a subset $Y \subseteq X$, compute the stabilizer subgroup G_Y;

(ii) given a subset $Y \subseteq X$ and a subgroup $H \subseteq G$ which acts on Y, compute the minimal s.d.r. for H acting on Y.

Given a set X and a group G acting on X, the generation of an s.d.r. for the action of G on $P(X)$ begins with the application of subprocedure (ii) to obtain a minimal s.d.r. for the 1-subsets. At stage n, note that the theorem does not necessarily produce exactly the minimal s.d.r., but a slightly larger set. For this reason, the theorem may be regarded as a generator for a backtrack scheme to produce the minimal s.d.r. for G acting on $P(X)$. The test phase of the algorithm simply determines whether a subset generated is lexicographically minimal in its own orbit and discards it if it is not. Because the generator eventually produces all minimal representatives, there is no need to check for equivalence with previously accepted subsets—a far more costly test.

Side conditions to be imposed can, of course, be incorporated into the test phase as well.

As a by-product of the generator, the size of the orbit of each subset U generated can be computed immediately after step (i). It is just $|O_U| = |G|/|G_U|$.

A computer program that implements the search just described can be found in [6].

2. Examples.

Example 1. Consider the cube shown in Fig. 1. We wish to generate all configurations of 0, 1, 2, 3 and 4 marks placed at the vertices up to the action of the group of rotations of the cube. We shall also enumerate the patterns by means of Pólya's theorem [1], [5].

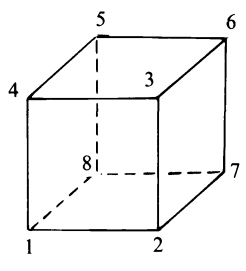


FIG. 1

The 24 elements of the group together with the term of the cycle index polynomial associated with each is as follows:

e	identity	x_1^8
f_1	(1234)(7658)	x_4^2
f_1^{-1}		x_4^2
f_2	(13)(24)(68)(57)	x_2^4
f_3	(2367)(1458)	x_4^2
f_3^{-1}		x_4^2
f_4	(26)(37)(15)(48)	x_2^4
f_5	(3456)(1872)	x_4^2
f_5^{-1}		x_4^2
f_6	(35)(46)(17)(28)	x_2^4
s_1	(12)(56)(38)(47)	x_2^4
s_2	(23)(58)(47)(16)	x_2^4
s_3	(34)(78)(16)(25)	x_2^4
s_4	(14)(67)(38)(25)	x_2^4
s_5	(45)(27)(16)(38)	x_2^4
s_6	(36)(18)(25)(47)	x_2^4
v_1	(1)(6)(248)(357)	$x_1^2x_3^2$
v_1^{-1}		$x_1^2x_3^2$
v_2	(2)(5)(137)(468)	$x_1^2x_3^2$
v_2^{-1}		$x_1^2x_3^2$
v_3	(3)(8)(246)(156)	$x_1^2x_3^2$
v_3^{-1}		$x_1^2x_3^2$
v_4	(4)(7)(135)(268)	$x_1^2x_3^2$
v_4^{-1}		$x_1^2x_3^2$

The generation algorithm begins with the construction of the minimal system of distinct representatives for the subsets of vertices of size one. Since the group is transitive, it consists of a single set, and we have

$$U_1 = \{\{1\}\}.$$

The stabilizer subgroup of the set $\{1\}$ is $\{e, v_1, v_1^{-1}\}$, and the minimal s.d.r. for the action of this subgroup is $\{1, 2, 3, 6\}$. The 1 is discarded, and we have

$$U_2 = \{\{1, 2\}, \{1, 3\}, \{1, 6\}\}.$$

The stabilizer subgroup of $\{1, 2\}$ is $\{e, s_1\}$, and the associated s.d.r. is $\{1, 3, 4, 5\}$, from which we get the sets $\{1, 2, 3\}$, $\{1, 2, 4\}$ and $\{1, 2, 5\}$. The set $\{1, 2, 4\}$ is discarded since it is equivalent to $\{1, 2, 3\}$ by f_1.

The stabilizer subgroup of $\{1, 3\}$ is $\{e, f_2\}$. The associated s.d.r. is $\{1, 2, 5, 6\}$, and the sets generated are $\{1, 3, 5\}$ and $\{1, 3, 6\}$. The set $\{1, 3, 6\}$ is discarded since it is equivalent to $\{1, 2, 6\}$ by s_2. Note that it is not necessary to find the lexicographically least 3-set. The existence of any equivalent, lexicographically smaller set is sufficient.

The stabilizer subgroup of $\{1, 6\}$ is $\{e, s_2, s_3, s_5, v_1, v_1^{-1}\}$, and the s.d.r. is $\{1, 2\}$, so nothing new is generated; hence we have

$$U_3 = \{\{1, 2, 3\}, \{1, 2, 5\}, \{1, 3, 5\}\}.$$

From the set $\{1, 2, 3\}$ we get the subgroup $\{e\}$ and the s.d.r. $\{1, 2, 3, 4, 5, 6, 7, 8\}$. The 4-sets generated are $\{1, 2, 3, 4\}, \{1, 2, 3, 5\}, \{1, 2, 3, 6\}, \{1, 2, 3, 7\}$ and $\{1, 2, 3, 8\}$. All are lexicographically minimal in their respective orbits. The set $\{1, 2, 5\}$ also has a trivial stabilizer subgroup, and the sets produced are $\{1, 2, 5, 6\}$, $\{1, 2, 5, 7\}$ and $\{1, 2, 5, 8\}$. The set $\{1, 2, 5, 7\}$ is equivalent to $\{1, 2, 4, 6\}$ by s_1, and $\{1, 2, 5, 8\}$ is equivalent to $\{1, 2, 4, 7\}$ by f_5, so they are not included.

Finally, the stabilizer subgroup of $\{1, 3, 5\}$ is $\{e, v_4, v_4^{-1}\}$, and the corresponding s.d.r. is $\{1, 2, 4, 7\}$. Only the 7 is used giving $\{1, 3, 5, 7\}$. We have

$$U_4 = \{\{1, 2, 3, 4\}, \{1, 2, 3, 5\}, \{1, 2, 3, 6\}, \{1, 2, 3, 7\}, \{1, 2, 3, 8\}, \{1, 2, 5, 6\}, \{1, 3, 5, 7\}\}.$$

To simply count these sets by means of Pólya's theorem, let D be the vertices of the cube and R the set {yes, no}. Define a weight function by no $\to 1$ and yes $\to x$, so that a function in R^D with k yes's and $8 - k$ no's has weight x^k. The cycle index polynomial is

$$P_G(x_1, \cdots, x_8) = \tfrac{1}{24}[x_1^8 + 9x_2^4 + 6x_4^2 + 8x_1^2x_3^2].$$

The pattern inventory, obtained by setting $x_k = 1 + x^k$, is

$$\tfrac{1}{24}[(1 + x)^8 + 9(1 + x^2)^4 + 6(1 + x^4)^2 + 8(1 + x)^2(1 + x^3)^2].$$

Expanding this expression yields

$$1 + x + 3x^2 + 3x^3 + 7x^4 + \cdots,$$

so there are one null set, one 1-set, three 2-sets, three 3-sets and seven 4-sets.

Example 2. Let D_8, the dihedral group of order 8, act on the hoop shown in Fig. 2. We wish to generate all ways up to the group action of placing any of the three symbols $\{\alpha, \beta, \gamma\}$ on the vertices of the hoop such that no symbol appears twice in succession.

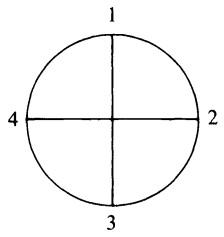


FIG. 2

We first generate an s.d.r. for D_8 acting on $P(\{1, 2, 3, 4\})$ in order to find the possible locations of the symbol α. After the empty set $\varnothing$, we have only one 1-set $\{1\}$. From this 1-set, we get two 2-sets $\{1, 2\}$ and $\{1, 3\}$, but $\{1, 2\}$ is discarded because of the side condition, and the algorithm produces nothing larger containing $\{1, 3\}$. Hence the only preimages of α are $\varnothing$, $\{1\}$ and $\{1, 3\}$.

Applying the theorem to the action of the stabilizer subgroup of $\{1, 3\}$ on the set $\{2, 4\}$, we obtain the three subsets $\varnothing$, $\{2\}$ and $\{2, 4\}$. These describe patterns 1, 2 and 3 shown in Fig. 3.

The action of the stabilizer subgroup of $\{1\}$ acting on $\{2, 3, 4\}$ provides the empty set and two 1-sets, $\{2\}$ and $\{3\}$. From $\{2\}$ we get $\{2, 3\}$, which is discarded,

and $\{2, 4\}$, which is retained. The subsets $\{3\}$ and $\{2, 4\}$ describe patterns 4 and 5 in Fig. 3, but $\varnothing$ and $\{2\}$ are discarded because they represent patterns with adjacent γ's.

Finally, corresponding to those patterns with no α's, we again obtain $\varnothing$, $\{1\}$ and $\{1, 3\}$, but this time only $\{1, 3\}$, which describes pattern 6 of Fig. 3, is kept since the others have adjacent γ's.

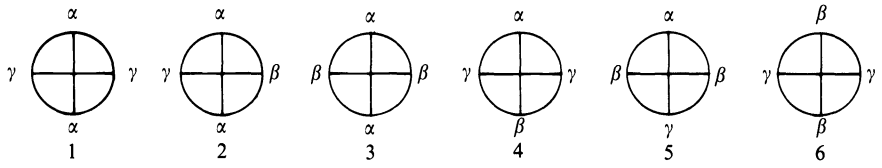


FIG. 3

3. Timing and storage considerations. Estimates of computation time and storage requirements for a pattern generation program based on the theorem presented are difficult to develop, since a good estimate requires some information about the symmetries involved. However, some experimental comparisons have been run against a simpler procedure.

Let D be a set of size d and R a set of size r. Let the group G acting on D be the dihedral group of order $2d$. Two programs, run on the Burroughs B6700 at University of California at San Diego, were written to generate all patterns for G acting on R^D. The first of these was a recursive scheme based on generating s.d.r.'s for a group action induced on a power set. The second algorithm was a sieving procedure that worked as follows: a one-to-one mapping φ is defined from the elements of R^D to the integers $\{0, 1, \cdots, r^d - 1\}$. An array f of length r^d is constructed, with every entry initially having the value 1. The procedure then moves once through the array, and at each location f_i takes the following action: if f_i is 1, then include $\varphi^{-1}(i)$ in the list of patterns being generated, and then set $f_{\varphi g \varphi^{-1}(i)}$ to 0 for each $g \in G$. If f_i is 0, then do nothing.

Most of the time expended in the sieve method is in the computation of $\varphi g \varphi^{-1}$, and it is clear that this will happen $|\Delta||G|$ times, where Δ is the s.d.r. for G acting on R^D. A lower bound for $|\Delta|$ is $r^d/|G|$, and we would expect a lower bound for the computation time to be $c(d)r^d$, where $c(d)$ is the time required to compute $\varphi g \varphi^{-1}(i)$.

Table 1 compares the time in milliseconds used by the recursive and the sieve algorithms for r fixed at 3 and d ranging between 2 and 8. Empirical formulas derived from this data show that the time in milliseconds for the sieve method is roughly $1.2 \times 3^{.96d}$, while the time for the recursive procedure is about $13.5 \times 3^{.69d}$.

Core storage necessary for the sieve method involves an array of length r^d. It can be shown that the recursive method uses less than $\frac{1}{4}(r - 1)d(d + 1)^2$ memory locations for saving intermediate s.d.r.'s.

TABLE 1
Time in milliseconds for $r = 3, |G| = 2d$

d	number of patterns	sieve	recursive
2	6	10	63
3	10	24	123
4	21	72	269
5	39	190	497
6	92	633	1,173
7	198	1,812	2,401
8	498	5,806	5,942

REFERENCES

[1] G. PÓLYA, *Kombinatorische Anzahlbestimmungen für Gruppen, Graphen und Chemische Verbindungen*, Acta Math., 68 (1937), pp. 145–254.

[2] S. G. WILLIAMSON, *The combinatorial analysis of patterns and the principle of inclusion-exclusion*, Discrete Math., 1 (1972), pp. 357–388.

[3] J. P. FILLMORE AND S. G. WILLIAMSON, *On backtracking: A combinatorial description of the algorithm*, this Journal, 3 (1974), pp. 41–55.

[4] S. W. GOLOMB AND L. BAUMERT, *Backtrack programming*, J. Assoc. Comput. Mach., 12 (1965), pp. 516–524.

[5] N. G. DEBRUIJN, *Pólya's theorem of counting*, Applied Combinatorial Mathematics, E. F. Beckenbach, ed., John Wiley, New York, 1964, pp. 144–184.

[6] D. M. PERLMAN, *Computational methods for pattern enumeration and isomorph rejection*, Ph.D. thesis, Univ. of Calif. at San Diego, 1973.

SIAM J. COMPUT.
Vol. 3, No. 3, September 1974

A HIERARCHY THEOREM FOR POLYNOMIAL-SPACE RECOGNITION*

OSCAR H. IBARRA†

Abstract. The effect of increasing the size of the worktape alphabet of Turing machines with a read-only input and a single worktape operating within space $L(n) = n^r$ is investigated. In particular, it is shown that nondeterministic such machines with $m+1$ worktape symbols are more powerful than those with m symbols.

Key words. Turing machines, worktape alphabet, polynomial space

We consider nondeterministic and deterministic Turing machines with a two-way read-only input tape with endmarkers and a single worktape. We denote by 1-NTM (2-NTM) a nondeterministic Turing machine with a one-way (two-way) infinite worktape. The deterministic varieties are denoted by 1-DTM and 2-DTM, respectively. Let $L(n)$ be a function from positive integers into positive integers and $j = 1, 2$. A j-NTM (j-DTM) A is of type $(L(n), m)$ if it has a worktape alphabet of at most m symbols (the blank symbol, b, being one of the symbols) and has the property that if an input x of length n (exclusive of the endmarkers) is accepted, then A has a sequence of moves leading to the acceptance of x without scanning more than $L(n)$ tape squares on its worktape. We denote by j-NSPACE$(L(n), m)$ (j-DSPACE$(L(n), m)$) the class of sets accepted by j-NTM's (j-DTM's) of type $(L(n), m)$.

The purpose of this short note is to prove the following result, a corollary of which resolves an open problem of Seiferas, Fischer and Meyer [5].

THEOREM. *For any integers $r \geqq 1$ and $m \geqq 1$ we have*

(i) 1-NSPACE$(n^r, m) \subsetneq$ 1-NSPACE$(n^r, m+1)$,
(ii) 1-DSPACE$(n^r, m) \subsetneq$ 1-DSPACE$(n^r, m+1)$,
(iii) 2-NSPACE$(n^r, m) \subsetneq$ 2-NSPACE$(n^r, m+1)$,
(iv) 2-DSPACE$(n^r, m) \subsetneq$ 2-DSPACE$(n^r, m+2)$.

The proof involves the following lemma.

LEMMA 1. *Let $m > l \geqq 1$ and $r \geqq 1$. Then*

(i) 1-NSPACE$(n^r, m) \subseteq$ 1-NSPACE(n^r, l) *implies* 1-NSPACE(n^r, m^{k^r}) $\subseteq$ 1-NSPACE$(n^r, l^{k^r} + 3)$ *for all $k \geqq 1$,*
(ii) 1-DSPACE$(n^r, m) \subseteq$ 1-DSPACE(n^r, l) *implies* 1-DSPACE(n^r, m^{k^r}) $\subseteq$ 1-DSPACE$(n^r, l^{k^r} + 3)$ *for all $k \geqq 1$.*

Proof of Lemma 1. We shall only prove (i). A similar proof applies to (ii). Let $k \geqq 1$ and L be in 1-NSPACE(n^r, m^{k^r}). Define the set $Q(L) = \{xd^{(k-1)i} : i = |x|, x \text{ in } L\}$, where d is a new symbol not appearing in any string of L and $|x|$ = length of x.

We first show that $Q(L)$ is in 1-NSPACE(n^r, m). So let A be a 1-NTM of type (n^r, m^{k^r}) accepting L. We shall construct a 1-NTM B of type (n^r, m) accepting $Q(L)$. For inputs of the form $xd^{(k-1)i}$, where $i = |x| \leqq 2$, B accepts by table look-up.

* Received by the editors October 12, 1973, and in revised form January 23, 1974.

† Department of Computer, Information, and Control Sciences, University of Minnesota, Minneapolis, Minnesota 55455. This work was supported by the National Science Foundation under Grant GJ-35614.

For the cases where $i = |x| \geqq 3$, B operates as follows:

1. B uses two symbols on its worktape to check that the input is of the form $xd^{(k-1)i}$ for some x, where $i = |x| \geqq 3$. This is easily done by writing the string $\#b^{i-2}\#$ on the worktape and using this string to check that there are exactly $(k-1)i$ d's following x. B then rewrites the $\#$'s by blanks (i.e., by b's) and positions its worktape head on the square previously occupied by the left $\#$.
2. B simulates the actions of A on input x by encoding each of the m^{k^r} symbols of A as a unique string of length k^r using m symbols.

It is clear that B operating as described accepts $Q(L)$. We need only check that B operates within n^r tape squares for an input of length n. Consider an input $xd^{(k-1)i}$ $(i = |x|)$ of length ki. If A accepts x, then A has a sequence of moves leading to the acceptance of x using no more than i^r tape squares. It follows that B has a sequence of moves leading to the acceptance of $xd^{(k-1)i}$ using no more than $k^r i^r$ tape squares (since each symbol of A is encoded as a string of length k^r). Thus B is of type (n^r, m) and $Q(L)$ is in 1-NSPACE(n^r, m).

By assumption, 1-NSPACE$(n^r, m) \subseteq$ 1-NSPACE(n^r, l). Hence $Q(L)$ is also in 1-NSPACE(n^r, l). Let C be a 1-NTM of type (n^r, l) accepting $Q(L)$. We shall construct a 1-NTM D of type $(n^r, l^{k^r} + 3)$ accepting L. For inputs of length $\leqq 2$, D accepts by table look-up. Now suppose x is an input of length $\geqq 3$. Then D operates as follows:

1. D generates the string $\#b^{i^r-2}\#$ on its worktape, where $i = |x|$. D can do this using no more than 4 symbols.
2. D then simulates on x the computation of C on input $xd^{(k-1)i}$ using only the squares occupied by the string $\#b^{i^r-2}\#$. D can carry out the simulation within this space bound by treating each of its symbols as a k^r-tuple of l symbols from an alphabet of l symbols. Note that D can use the length of x to simulate the motion of the input head of C on symbol d. D does not actually alter the $\#$'s but remembers in its finite control the changes that are made on the k^r-tuples represented by the $\#$'s.

Since D uses k^r-tuples in space i^r, D has enough space to simulate C. Moreover, D can do the simulation using only l^{k^r} symbols (exclusive of the symbol $\#$). It follows that D is of type $(n^r, l^{k^r} + 3)$ and L is in 1-NSPACE$(n^r, l^{k^r} + 3)$. (The $+3$ takes into account the special symbol $\#$ and guarantees that $l^{k^r} + 3 \geqq 4$ for all $l, k, r \geqq 1$, a condition necessary in step 1.)

We remark that Lemma 1 is yet another form of "translation" used in several places in the literature (see, e.g., [1], [2], [3], [4]).

Remarks. For Turing machines with two-way infinite worktape, we make the following observations concerning the applicability of the proof of Lemma 1.

(a) The construction of D from C must be modified if C has a two-way infinite worktape. However, since the workspace of D is bounded at both ends by the special symbol $\#$, the simulation of C can still be accomplished within the desired space bound with only $l^{k^r} + 3$ symbols.

(b) Suppose the Turing machine A in the proof of Lemma 1 has a two-way infinite worktape. We consider two cases:

(i) For the nondeterministic case, the construction of B still works provided that in step 1, B—after rewriting the $\#$'s by blanks—non-

deterministically positions its worktape head on any one of the squares used in checking the input format, before simulating A. (Thus, B guesses that the squares used in checking the input format are among the squares that it may use in the simulation of A, allowing B to operate within the desired space bound.)

(ii) For the deterministic case, there is no guarantee that the squares used for checking the input format are among the squares B may use in the simulation of A. Thus, B may not be able to operate within the desired space bound. However, it should be clear that if B is provided with a special symbol to bound the squares it is allowed to scan during the simulation, B could be made to operate within the desired space bound. Thus, B would have $m + 1$ symbols instead of m.

By appropriately modifying the constructions in the proof of Lemma 1 along the lines mentioned in the remarks above, we get the following lemma.

LEMMA 2. *Let* $m > l \geqq 1$ *and* $r \geqq 1$. *Then*

(i) 2-NSPACE$(n^r, m) \subseteq$ 2-NSPACE(n^r, l) *implies* 2-NSPACE$(n^r, m^{k^r}) \subseteq$ 2-NSPACE$(n^r, l^{k^r} + 3)$ *for all* $k \geqq 1$,

(ii) 2-DSPACE$(n^r, m + 1) \subseteq$ 2-DSPACE(n^r, l) *implies* 2-DSPACE$(n^r, m^{k^r}) \subseteq$ 2-DSPACE$(n^r, l^{k^r} + 3)$ *for all* $k \geqq 1$.

Before we can prove the theorem, we need the following result.

LEMMA 3. *Let* $j = 1, 2$. *Then for each* $m \geqq 1$, *we have*

(i) j-NSPACE$(n^r, m) \subsetneq \bigcup_{l \geqq 1} j$-NSPACE$(n^r, l)$,

(ii) j-DSPACE$(n^r, m) \subsetneq \bigcup_{l \geqq 1} j$-DSPACE$(n^r, l)$.

Proof of Lemma 3. The first part follows from Seiferas, Fischer and Meyer [5, Cor. 2]. The second part can easily be shown using a diagonal argument similar to that of [6].

Proof of Theorem. We only prove part (i) of the theorem. Parts (ii)–(iv) are similarly shown using Lemma 1 (part (ii)) and Lemma 2.

Suppose 1-NSPACE$(n^r, m + 1) \subseteq$ 1-NSPACE(n^r, m) for some $m \geqq 1$. Then by Lemma 1,

$$\text{(1)} \qquad \text{1-NSPACE}(n^r, (m + 1)^{k^r}) \subseteq \text{1-NSPACE}(n^r, m^{k^r} + 3) \quad \text{for all } k \geqq 1.$$

Consider $(m + 1)^{k^r}$ and $m^{(k+1)^r} + 3$. Then

$$\text{(2)} \qquad \frac{(m + 1)^{k^r}}{m^{k^r}} = \left(\frac{m + 1}{m}\right)^{k^r},$$

$$\text{(3)} \qquad \frac{m^{(k+1)^r} + 3}{m^{k^r}} \leqq dm^{ck^{r-1}} \quad \text{for some constants } c \text{ and } d.$$

Moreover,

$$\text{(4)} \qquad \log_2 \left(\frac{m + 1}{m}\right)^{k^r} = k^r \log_2 \left(\frac{m + 1}{m}\right),$$

$$\text{(5)} \qquad \log_2 dm^{ck^{r-1}} = ck^{r-1} \log_2 m + \log_2 d.$$

We conclude from (2)–(5) that there exists $k_0 \geqq 1$ such that

$$\text{(6)} \qquad (m + 1)^{k^r} \geqq m^{(k+1)^r} + 3 \quad \text{for all } k \geqq k_0.$$

It follows from (1) and (6) that 1-NSPACE(n^r, $(m+1)^{k^r}$) $\subseteq$ 1-NSPACE(n^r, $m^{k^6}+3$) for all $k \geqq k_0$. This contradicts Lemma 3.

For the case $r = 1$, we have the following corollary, which settles a question posed in [5].

COROLLARY. *For each* $m \geqq 1$, *we have*

(i) 1-NSPACE(n, m) $\subsetneq$ 1-NSPACE($n, m+1$),
(ii) 1-DSPACE(n, m) $\subsetneq$ 1-DSPACE($n, m+1$),
(iii) 2-NSPACE(n, m) $\subsetneq$ 2-NSPACE($n, m+1$),
(iv) 2-DSPACE(n, m) $\subsetneq$ 2-DSPACE($n, m+2$).

Acknowledgment. I would like to thank Sartaj Sahni for helpful discussions concerning this work.

REFERENCES

[1] S. A. COOK, *A hierarchy for nondeterministic time complexity*, Proc. 4th Annual ACM Sympos. on Theory of Computing, Denver, Colorado, 1972, pp. 187–192.

[2] O. H. IBARRA, *A note concerning nondeterministic tape complexities*, J. Assoc. Comput. Mach., 19 (1972), pp. 608–612.

[3] ———, *On two-way multihead automata*, J. Comput. System Sci., 7 (1973), pp. 28–36.

[4] S. RUBY AND P. C. FISCHER, *Translational methods and computational complexity*, IEEE Conf. Record on Switching Circuit Theory and Logical Design, Ann Arbor, Michigan, 1965, pp. 173–178.

[5] J. I. SEIFERAS, M. J. FISCHER AND A. R. MEYER, *Refinements of the hierarchies of time and tape complexities*, 14th Annual Sympos. on Switching and Automata Theory, Iowa City, Iowa, October 1973.

[6] R. E. STEARNS, J. HARTMANIS AND P. M. LEWIS II, *Hierarchies of memory limited computations*, IEEE Conf. Record on Switching Circuit Theory and Logical Design, Ann Arbor, Michigan, 1965, pp. 179–190.

SIAM J. COMPUT.
Vol. 3, No. 3, September 1974

OPTIMUM COMMUNICATION SPANNING TREES*

T. C. HU†

Abstract. Given a set of nodes N_i ($i = 1, 2, \cdots, n$) which may represent cities and a set of requirements r_{ij} which may represent the number of telephone calls between N_i and N_j, the problem is to build a spanning tree connecting these n nodes such that the total cost of communication of the spanning tree is a minimum among all spanning trees. The cost of communication for a pair of nodes is r_{ij} multiplied by the sum of the distances of arcs which form the unique path connecting N_i and N_j in the spanning tree. Summing over all $\binom{n}{2}$ pairs of nodes, we have the total cost of communication of the spanning tree. Note that the problem is different from the minimum spanning tree problem solved by Kruskal and Prim.

Key words. communication spanning trees, cut-tree

1. Introduction. Suppose we are given a set of n nodes N_i ($i = 1, \cdots, n$) and the distances d_{ij} between N_i and N_j. These n nodes may represent cities which need to communicate with each other. We are also given a set of requirements r_{ij} (which may represent the number of telephone calls between N_i and N_j). The problem is to build a spanning tree connecting these n nodes such that the total cost of communication of the spanning tree is minimum among all spanning trees.

The cost of communication of a given spanning tree is defined as follows. For a pair of nodes N_i and N_j, there is a unique path in the spanning tree between N_i and N_j. The distance of the path is the sum of distances of links in the path. The cost of communication for the pair of nodes N_i and N_j is r_{ij} multiplied by the distance of the path. Summing over all $\binom{n}{2}$ pairs of nodes, we have the cost of the spanning tree. For example, the distances between six nodes are shown in Fig. 1, and the requirements between the six nodes are shown in Fig. 2.

The cost of the spanning tree in Fig. 3 is

$$\begin{aligned} r_{12}(2+2) + r_{13}(2+3) + r_{14}(2+4+3) + \cdots + r_{46}(3+4) + r_{56}(4) \\ = 10(2+2) + 0(2+3) + 0(2+4+3) + \cdots + 2(3+4) + 3 \times 4 \\ = 225. \end{aligned}$$

This problem of constructing optimum communication spanning trees will be referred to as the general problem. In this paper, we shall not deal with the general communication spanning tree, and shall devote ourselves to two special cases of this general problem.

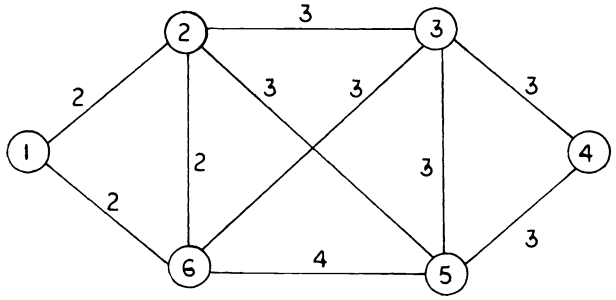
Case A. The distances d_{ij} are all equal to one, while the requirements r_{ij} are arbitrary. We shall call the optimum spanning tree in this case the *optimum requirement spanning tree.*

* Received by the editors October 4, 1973, and in revised form January 31, 1974.

† Mathematics Research Center, University of Wisconsin—Madison, Madison, Wisconsin. Now at University of California at San Diego, La Jolla, California 92037. This work was sponsored by the United States Army under Contract DA-31-124-ARO-D-462 and by the National Science Foundation under Grant GJ 28339.

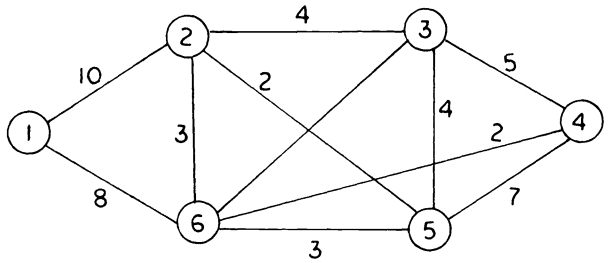
Case B. The distances d_{ij} are arbitrary, while the requirements r_{ij} are all equal to one. We shall call the optimum spanning tree in this case the *optimum distance spanning tree.*

The problem of optimum distance spanning tree was proposed to the author by Professor F. Maffioli. The general problem and Case A were formulated by the author. Both Case A and Case B are much harder problems than the well-known problem of minimum spanning tree (see Kruskal [8] and Prim [9]).



(d_{ij} not shown are assumed to be 4)

FIG. 1. *Distances between six nodes*



(r_{ij} not shown are assumed to be zero)

FIG. 2. *Requirements between six nodes*

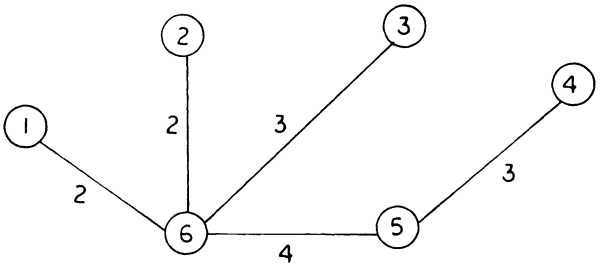


FIG. 3. *A spanning tree*

2. Optimum requirement spanning trees. Throughout this section, we have $d_{ij} \equiv 1$ and r_{ij} arbitrary as mentioned in Case A.

First we shall give a few definitions and deal with some seemingly unrelated notions. All these definitions and concepts are illustrated in more detail in Hu [7, pp. 131–142].

Consider a network N of n nodes with undirected arcs A_{ij} connecting nodes N_i and N_j. An arc has an *arc capacity* b_{ij} ($b_{ij} = b_{ji}$ for all i, j). A *cut*, denoted by $(X, \bar{X})$ where X is a subset of nodes of the network and $\bar{X}$ is its complement, is the set of arcs A_{ij} with $N_i \in X$ and $N_j \in \bar{X}$.

The capacity of a cut $(X, \bar{X})$ is denoted by $b(X, \bar{X})$. The capacity of a cut is the sum of b_{ij} of all arcs in the cut. Since $b_{ij} = b_{ji}$, we have $b(X, \bar{X}) = b(\bar{X}, X)$. Two cuts $(X, \bar{X})$ and $(Y, \bar{Y})$ are said to cross each other if and only if each of the four sets $X \cap Y$, $X \cap \bar{Y}$, $\bar{X} \cap Y$ and $\bar{X} \cap \bar{Y}$ contains at least one node. A set of cuts is said to be noncrossing if no two of them cross each other.

The number of arcwise-disjoint paths between two nodes N_p and N_q is denoted by f_{pq}, which is the value of the maximal flow from N_p to N_q (see Ford and Fulkerson [4]). By the max-flow min-cut theorem [4], there always exists a cut $(X, \bar{X})$ with $N_p \in X$ and $N_q \in \bar{X}$ such that $b(X, \bar{X}) = f_{pq}$. The cut $(X, \bar{X})$ is called a *minimum cut* since it has the least capacity of any cut separating N_p and N_q.

The algorithm of Gomory and Hu [6] constructs a spanning tree with the following properties:

(i) Each link of the spanning tree has a value v_{ij} associated with it. If we remove the link with value v_{ij}, so that the network is disconnected into two components, say X and $\bar{X}$, then $v_{ij} = b(X, \bar{X})$ and $(X, \bar{X})$ is a minimum cut of the original network.

(ii) The maximal flow value f_{pq} between any two nodes N_p and N_q of the original network is

$$f_{pq} = \min (v_{pa}, \cdots, v_{ij}, \cdots, v_{tq}),$$

where $v_{pa}, \cdots, v_{ij}, \cdots, v_{tq}$ are values associated with links in the tree which form the unique path connecting N_p and N_q in the tree.

From now on, the spanning tree constructed by Gomory and Hu [6] will be referred to as the *cut-tree*. The cut-tree is obtained by doing $n - 1$ maximal flow problems, each problem taking at most $O(n^3)$ applications of the Ford–Fulkerson labeling procedures [5], (see Dinic [2] and Edmonds and Karp [3]). Thus the algorithm of constructing the cut-tree can be thought of as $O(n^4)$ algorithm. Now we shall establish a few lemmas.

LEMMA 1. *A spanning tree with $n - 1$ links corresponds to a set of $n - 1$ noncrossing cuts uniquely.*

Proof. Remove any link of the spanning tree; this will disconnect the tree into two components, say T_1 and T_2. Then let this link correspond to the cut (T_1, T_2). Do the same process to the tree T_1 or T_2. Thus from any spanning tree, we get a set of $n - 1$ noncrossing cuts. Conversely, from a set of $n - 1$ noncrossing cuts, we can construct the spanning tree as follows. Take a cut $(X, \bar{X})$; we can draw two supernodes connected by a link (each supernode represents a set of ordinary nodes symbolically); in one supernode, we list the names of nodes in X, and in the other supernode, we list the names of nodes of $\bar{X}$. This creates one

link of the spanning tree. Now consider another cut $(Y, \bar{Y})$. Since $(Y, \bar{Y})$ does not cross $(X, \bar{X})$, we have $Y \subset X$ and $\bar{Y} \supset \bar{X}$ (or $Y \supset X$ and $\bar{Y} \subset \bar{X}$); then we can create a tree with three supernodes Y, $(X - Y)$ and $\bar{X}$ as shown in Fig. 4. After $n - 1$ steps, we create a spanning tree of $n - 1$ links.

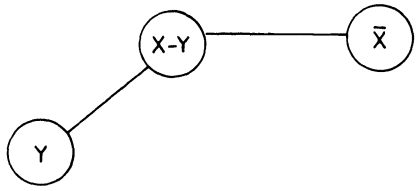


FIG. 4

THEOREM 1. *Given a set of requirements r_{ij} and a spanning tree T, the cost of communication of T for the set r_{ij} is equal to the sum of capacities of the $n - 1$ noncrossing cuts of a network N. (The $n - 1$ cuts are determined by the edges of T). The network N has the same set of nodes as T and has arc capacities $b_{ij} = r_{ij}$.*

Proof. Since $d_{ij} \equiv 1$, the cost of communication of a pair of nodes N_p and N_q in a spanning tree is equal to r_{pq} multiplied by the number of links in the path connecting N_p and N_q. Summing over $\binom{n}{2}$ pairs of nodes, we have the cost of the spanning tree. We can also calculate the cost of the spanning tree in an alternate way. For a given link of the tree, we add the r_{pq}'s which use the link and let the sum be attached to the given link; summing over $n - 1$ links, we have the cost of the spanning tree. In calculating the cost of the spanning tree in this manner, the sum of the r_{pq}'s for a given link of T is the same as the capacity of the cut corresponding to that link. Any single requirement r_{pq} which is not a direct link in the spanning tree T is counted exactly the same number of times as the number of links which join N_p and N_q in T. Thus the cost of communication of the spanning tree T is numerically equal to the sum of the capacities of the $n - 1$ noncrossing cuts in N which correspond to the links of T.

LEMMA 2. *The sum of the capacities of $n - 1$ noncrossing cuts represented by the cut-tree is less than or equal to the sum of the capacities of* any *$n - 1$ noncrossing cuts.*

Proof. See Lemma 1 and Adolphson and Hu [1, Thm. 1].

THEOREM 2. *The cut-tree is an optimum requirement spanning tree.*

Proof. This follows from Lemmas 1 and 2 and Theorem 1.

For example, if a network of six nodes has requirements as shown in Fig. 2, then according to Theorems 1 and 2, we regard these r_{ij} as arc capacities of a network. Then from the algorithm of Gomory and Hu (see [6] or [7], where the example is illustrated in detail), we obtain a cut-tree as shown in Fig. 5, with total cost of 77, which is exactly the sum of the $n - 1$ cut capacities. ($77 = 18 + 17 + 13 + 14 + 15$.)

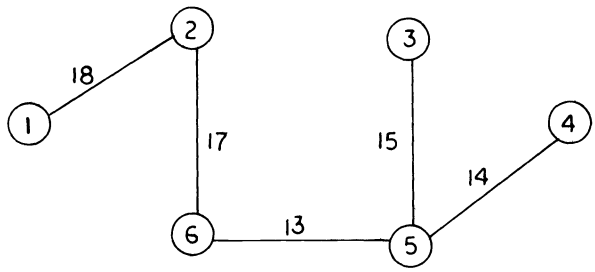


FIG. 5. *Optimum requirement spanning tree*

3. Optimum distance spanning trees. In this case, $r_{ij} \equiv 1$, and d_{ij} are arbitrary except that they satisfy the regular triangular inequality

$$d_{ij} + d_{jk} \geqq d_{ik} \quad \text{for all } i, j \text{ and } k.$$

Throughout this section, we shall assume that there are n nodes and $n \geqq 4$. We define a node in a tree to be an *outer node* if the degree of the node is one, an *inner node* if the degree of the node is two or more. A tree is called a *star-tree* if there is only one inner node in the tree.

In general, an optimum distance spanning tree may not be a star-tree. We shall define a sufficient condition for the optimum distance spanning tree to be a star-tree and then state a simple algorithm for getting the optimum distance spanning tree in this case.

First, we introduce a new way of calculating the cost of a distance spanning tree. For a given spanning tree, we calculate the cost of a link as follows. If the link of length d_{ij} is removed and the network is disconnected into two sets, one containing k nodes and the other $n - k$ nodes, then the cost of the link is $d_{ij}(k)(n - k)$. Summing over all $n - 1$ links of the tree, we have the cost of the tree. Note that the cost of a link depends only on d_{ij} and the number of nodes on both sides of the link, but does not depend on how the two subtrees are arranged on both sides.

LEMMA 3. *If all the d_{ij} are the same, then the optimum distance spanning tree is a star-tree.*

Proof. Let the $d_{ij} = d$ for all the links. For every link in a star-tree, the cost of the link is $d(1)(n - 1)$. If the spanning tree is not a star-tree, then at least one link will have k nodes at one end and $n - k$ nodes at the other end, where $k \geqq 2$ and $n - k \geqq 2$, so the cost of the link is $d(k)(n - k)$. Since $k(n - k) > (1)(n - 1)$ for $n \geqq 4$, this completes the proof.

We shall find a sufficient condition for an optimum distance spanning tree to be a star-tree. Roughly speaking, the sufficient condition requires all links do not differ too much in length.

If we erase all outer nodes from a tree of n nodes (which is not a star-tree), then the remaining is again a tree formed by inner nodes. This tree will be referred to as the *inner tree* T_I. In the tree T_I, there again must be some nodes of degree one, and these nodes are called *extreme* inner nodes. In Fig. 6, there are five inner

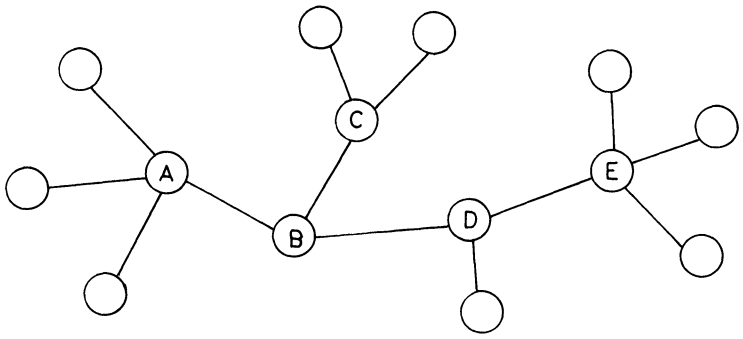


FIG. 6

nodes and N_A, N_C, N_E, are extreme inner nodes. Two nodes are called *neighbors* if they both are adjacent to the same link. For example, N_D and N_E are neighbors.

THEOREM 3. *In the distance spanning tree problem, let a, b and c be the distances of three sides of any triangle in the n-node network* ($n \geqq 4$), *where*

$$a \leqq b \leqq c. \tag{1}$$

If there exists a positive t not larger than $(n-2)/(2n-2)$ *such that*

$$a + tb \geqq c \tag{2}$$

for all triangles in the network, then there exists an optimum distance spanning tree which is a star-tree.

Note that the smaller the value of t, the more restrictive is the inequality. If $t = 0$, then it restricts all sides of any triangle to be of equal length. If $t = 1$, it reduces to the regular triangular inequality. Since the value of t must be less than one, (2) is a stronger condition than the regular triangular inequality. Note also that (1) and (2) imply five other inequalities, namely $ta + b \geqq c$, $tb + c \geqq a$, $b + tc \geqq a$, $ta + c \geqq b$ and $tc + a \geqq b$.

Proof. It is sufficient to show that we can reduce the number of inner nodes in any spanning tree (which is not a star-tree) without increasing the cost. So, let T be any spanning tree which contains at least two inner nodes. Let N_q be an extreme inner node in T with a neighbor N_p which is an inner node. Since N_q is an extreme inner node, all its neighbors (except N_p) must be of degree one in the tree T. Call these nodes N_i ($i = 1, 2, \cdots, k-1$). Without loss of generality, we can assume that $(n/2) \geqq k$. This is shown in Fig. 7, where the distance between N_p and N_i is denoted by c_i.

Let us construct a new spanning tree T' which is the same as T except the nodes N_i ($i = 1, 2, \cdots, k-1$) are connected to N_p directly. In the new tree T', N_q is no longer an inner node. Thus the number of inner nodes is decreased by one. We shall show that the cost of T' is not greater than the cost of T. If we apply this idea recursively to the tree T', T'', $\cdots$, then we will finally get a tree T^* which is a star-tree. Since the part of the tree to the left of N_p is exactly the same for both T and T', we need only calculate the cost for the part to the right of N_p. For the

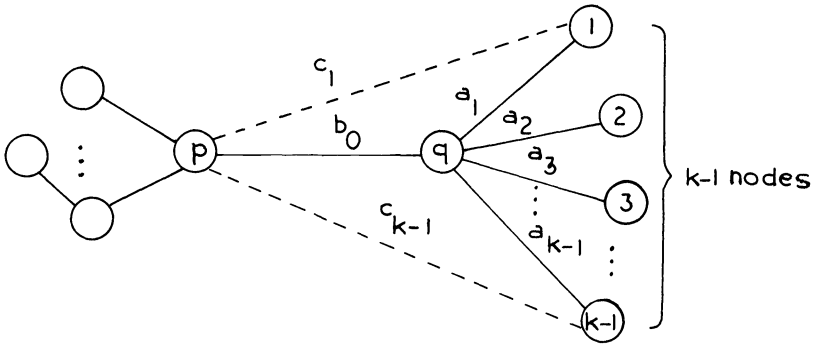


FIG. 7

tree T, this cost is

$$b_0 k(n-k) + \sum_{i=1}^{k-1} a_i(1)(n-1). \tag{3}$$

The corresponding cost for T' is

$$b_0(1)(n-1) + \sum_{i=1}^{k-1} c_i(1)(n-1). \tag{4}$$

The net decrease of cost in changing from T to T' is

$$(n-1)\left[\sum_{i=1}^{k-1}\left(a_i - c_i + \frac{n-k-1}{n-1}b_0\right)\right]. \tag{5}$$

The decrease will be positive if each term in (5) is positive, namely,

$$a_i + \frac{n-k-1}{n-1}b_0 \geqq c_i \quad \text{for all } i. \tag{6}$$

For a fixed n, $(n-k-1)/(n-1)$ is smallest when k is largest; that is, when $k = n/2$. Thus it is sufficient to have

$$a_i + \frac{n-2}{2n-2}b_0 \geqq c_i \quad \text{for all } i. \tag{7}$$

As noted before, assuming $a_i \leqq b_0 \leqq c_i$ gives the strongest inequality, and we have the statement of the theorem.

If the sufficient conditions (1) and (2) are satisfied, then the optimum distance spanning tree will be a star-tree. We can just calculate the n sums $\sum_j d_{ij}$ $(i = 1, \cdots, n)$ and let

$$\sum_j d_{sj} = \min_i \sum_j d_{ij};$$

then the optimum distance spanning tree is a star-tree with N_s as the star. Note that if we have the distances shown in Fig. 1, (1) and (2) are not satisfied, yet the optimum distance spanning tree is a star-tree with N_2 as the star. To check if (1) and (2) are satisfied, we can use a procedure similar to that used in the Appendix of [6].

REFERENCES

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

[2] E. A. DINIC, *Algorithm for solution of a problem of maximal flow in a network with power estimation*, Soviet Math. Dokl., 11 (1970), pp. 1277–1280.

[3] J. EDMONDS AND R. M. KARP, *Theoretical improvements in algorithmic efficiency for network flow problems*, J. Assoc. Comput. Mach., 19 (1972), pp. 248–264.

[4] L. R. FORD, JR. AND D. R. FULKERSON, *Maximal flow through a network*, Canad. J. Math., 8 (1956), pp. 399–404.

[5] ———, *A simple algorithm for finding maximal network flows and an application to the Hitchcock problem*, Ibid., 9 (1957), pp. 210–218.

[6] R. E. GOMORY AND T. C. HU, *Multi-terminal network flows*, SIAM J. Appl. Math., 9 (1961), pp. 551–570.

[7] T. C. HU, *Integer Programming and Network Flows*, Addison-Wesley, Reading, Mass., 1969.

[8] J. B. KRUSKAL, *On the shortest spanning tree of a graph and the traveling salesman problem*, Proc. Amer. Math. Soc., 7 (1956), pp. 48–50.

[9] R. C. PRIM, *Shortest connection networks and some generalizations*, Bell System Tech. J., 36 (1957), pp. 1389–1401.

SIAM J. COMPUT.
Vol. 3, No. 3, September 1974

POLYNOMIAL MULTIPLICATION, POWERS AND ASYMPTOTIC ANALYSIS: SOME COMMENTS*

RICHARD J. FATEMAN†

Abstract. This paper examines multiplication and powering of dense symbolic polynomials, in one or several variables, with "nongrowing" coefficients (e.g., coefficients in a finite field). We use a "completely dense" model for polynomials, in order to present worst-case analyses.

In this context, the use and abuse of asymptotic analysis techniques is discussed. Six algorithms for computing polynomial powers are analyzed in terms of the time required for execution on a typical digital computer, and procedures are derived for choosing the fastest algorithm, exactly, as a function of degree, number of variables, and power to be computed. The case of sparse polynomials is discussed in a separate paper [6].

Key words. polynomial multiplication, polynomial powers, exponentiation, computing time analysis, algorithmic analysis, finite Fourier transform, algebraic manipulation, modular arithmetic

1. Introduction. The choice of an appropriate computer algorithm can have an enormous influence on the size and complexity of problems which can be solved in a reasonable time. In situations where several algorithms are possible, it is frequently important to characterize the problem domains for which each is superior.

This paper discusses asymptotic analysis, and some of the alternatives to it, as a tool to evaluate algorithms. We use the context of computing powers of polynomials, a problem which has been discussed in [2], [4], [5], [6], [9], [11], [13] and [14].

The method of asymptotic analysis of algorithms is to characterize the speed or efficiency of an algorithm by some simple parameterized expression. Thus algorithm A may be order n^2 $(=O(n^2))$ and B may be $O(n \log n)$, where n is a measure of the size of the problem. This implies that for all $n >$ some N, algorithm A is more time-consuming than algorithm B. For smaller n, it is usually the case that algorithm A is faster than B.

It becomes critical, in the cases of interest in this paper, to find N, the cutoff point between algorithms, in order to decide between alternative algorithms. We are, in fact, attempting to determine the best algorithm or combination of algorithms to use as a system program in MACSYMA [1], [17].

It has been suggested that one might empirically find the "constants" C_A and C_B characterizing algorithms A and B as running in times $C_A n^2$ and $C_B n \log n$, respectively, and thus give the asymptotic analysis some precision in predicting the cutoff point, N. This is not necessarily reasonable, simply because empirical data may reflect nonasymptotic terms in the algorithm's cost function. For example, $n^2 + 1000n$ for $0 < n < 10$ can hardly be characterized by Cn^2.

* Received by the editors February 12, 1973.

† Department of Mathematics and Project MAC, Massachusetts Institute of Technology, Cambridge, Massachusetts. This work was supported in part by Project MAC, an MIT interdepartmental laboratory sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract N0014-70-A-0362-0001, and in part by the MIT Department of Mathematics under National Science Foundation Grant GP-22796. Now at Department of Electrical Engineering and Computer Science, University of California at Berkeley, Berkeley, California 94720.

The approach which we follow, as nearly as possible, is to derive exact formulas for the number of operations an algorithm requires, as a function of n, and then compare these formulas. These formulas may be unwieldy to manipulate by hand, but since they can be manipulated by computer (e.g., in MACSYMA [1], [17]), we feel the approach is both accurate and tenable.

We have at our disposal another exact method for finding cutoff points: we simply program the algorithms and time them. This has the major disadvantage that different implementations may have drastically different cutoff points. In the experiments performed for this paper, we compared the results of timings, and the actual counts of multiplications/divisions, additions/subtractions, and exponentiations, to the run times. It was satisfying to note that actual computation times were closely proportional (after removing the perturbing effects of the counting routines) to the total of these counts, and reasonably proportional to just the multiplications/divisions. This implied that our implementations were reasonable and, in a "machine-independent" sense, we were finding cutoff points based on minimizing the number of arithmetic operations used by the algorithms.

Using a computer to actually run algorithms gives us a particularly useful hold on reality. It is easier to see when the true cutoff point is of no practical interest: the problem simply exceeds the bounds of the practically computable. In fact, we have decided that if a computation requires more than 2 minutes of computer time in the system described in [1], it is an exceedingly large and unusual computation of a polynomial power. We believe an algorithm which performs well only outside that 2 minute region is not a good choice for problems encountered by algebraic manipulation systems, and cannot, in good faith, be touted as such, regardless of its asymptotic performance.

In the preceding discussion, we have assumed the existence of some parameter, n, which characterizes a problem. In the case of powering polynomials, a number of parameters emerge. We can use v for the number of variables (or indeterminates) in the polynomial, d for its (maximum) degree in each variable, n for the (integer) power we wish to compute, and other parameters to characterize the sparseness of the polynomial and the size of the coefficients. We use only (n, d, v) as parameters, although the sparseness question is quite important, and is examined in [6] and [14]. The size of the coefficients can be briefly dispensed with: let us specify that all coefficient computations are done in a finite field, so that multiplication and addition costs do not grow with the number of digits in the result. The analysis can be extended to the case of arbitrary integer coefficients by the use of the Chinese remainder algorithm. This is explained, for example, in [7] and [13]. Alternatively, one might imagine the coefficient arithmetic to be "floating-point" finite-precision arithmetic, so that the costs again remain constant.

We will be dealing with a particular model of polynomials which corresponds to "worst-case" assumptions. We need a few definitions.

A univariate (one-variable) polynomial of degree d is *dense* if it has no zero coefficients. That is, it has $d + 1$ terms. A multivariate polynomial with v variables where each variable $x_1, \cdots, x_v$ occurs to maximum degree d is called *uniformly dense to degree d in each of v variables.* That is, $f = f_d x_v^d + \cdots + f_0$, where each of the f_i are polynomials uniformly dense to degree d in each of $v - 1$ variables.

A polynomial in 0 variables is an integer. We define the *size* of a polynomial f, size(f) to be the number of monomial terms in f. For example, size(f) $= (d+1)^v$ if f is uniformly dense to degree d in each of v variables.

The cost of an algorithm can be equated to the number of elementary operations it requires. In the algorithms examined here, this is usually proportional to the number of required coefficient multiplications/divisions, and thus we will derive, in most cases, only cost_M, the multiplication cost for an algorithm. For the most part, this is sufficient, although cost_A, the addition cost, is in one case significant enough to require examination.

We would like to emphasize, as is done in [10] and [19], that without empirical evidence to confirm that timings do correspond to the quantities we are analyzing, the analysis would be irrelevant. In fact, timing data presents a strong confirmation of our techniques, and demonstrates that intuition based on the analysis in this paper is applicable to the problem at hand.

2. Some history. It is fairly well known, now, that two n-bit integers can be multiplied by an "FFT method" to get a $2n$-bit result in $O(n \log(n) \log(\log n))$ operations. Few uses have been made of this algorithm because it is useful only for very large n. It is also well known that the same two integers may be multiplied in $O(n^{1.585\cdots})$ operations, yet uses of this algorithm are also infrequent. Knuth [15] gives an extensive account of the integer multiplication problem.

In fact, to our knowledge, all multiple-precision integer multiplication programs in algebraic manipulation systems use an $O(n^2)$ method which corresponds to "classical" multiplication.

This gap between "best known" algorithms and programming practice is sometimes harmless. It separates "theoretical" computational complexity from "practical" algorithm analysis and programming. Unfortunately, the separation of these two disciplines occasionally causes the use of (in fact) inefficient algorithms, and tends to obscure analyses of algorithms based on less-than-optimal subalgorithms.

If we turn to a logical extension of the n-word integer multiplication problem, namely the multiplication of univariate polynomials of degree $d = n - 1$, we find that a gap of the unfortunate variety prevails: as noted in [15], there are polynomial multiplication algorithms of $O(d^2)$, $O(d^{1.585\cdots})$ and $O(d \log(d))$. Except for our own work, all practical polynomial multiplication programs appearing in the context of an algebraic manipulation system are $O(d^2)$. Polynomial powering algorithms which assume polynomial multiplication of dense polynomials taking $O(d^2)$ have been analyzed by Heindel [11] and Horowitz [13]. In fact, for the class of dense polynomials used in these papers, we will show that using faster multiplication schemes not only yields the answer in less time, but also reverses the choice of the "best" powering algorithm rather dramatically. Specifically, Heindel and Horowitz prove that for a sufficiently large polynomial P, computing $P^n = P \cdot (P \cdot \cdots \cdot P)$ is faster than (assume $n = 2^m$), $P^n = ((P^2)^2 \cdots)^2$. Their proofs depend implicitly on polynomial multiplication being $O(d^2)$. If such is *not* the case, their choice is, for most problems, the slower of the two methods.

Of the six powering algorithms discussed in § 4, RMUL (repeated multiplication) and RSQ (repeated squaring) are the rather obvious ones above, SUMS

is a venerable method usually applied to power series [8], [15], EVAL was first proposed in [13], and the first implementation of BINOM and FFT were described in an earlier version of this paper [5]. In [9], Gentleman suggested the use of the FFT, although no details were supplied.

Another surprising result to one who uses asymptotic analysis is that an evaluation-interpolation homomorphism algorithm with favorable asymptotic growth rates is of little use over most of the region of practical problems.

3. Some results on polynomial multiplication.

THEOREM 3.1 (Classical polynomial multiplication). *The product of two polynomials f and g of size s and t can be computed with st coefficient multiplications, and no more than $st - 1$ coefficient additions. That is, at worst,* $\text{cost}_M = st$, $\text{cost}_A = st - 1$.

Proof. The classical method of polynomial multiplication, term by term, clearly uses st multiplications. Even if all the terms produced "collapse" into a single term, no more than $st - 1$ additions could be required. □

THEOREM 3.2 (Binary splitting multiplication). *Let f and g be univariate polynomials in the variable x. If $s = \text{size}(f) = 2^k$ and $\text{size}(g) \leqq s$, then $f \cdot g$ can be computed with* $\text{cost}_M(s) \leqq s^{\log_2 3}$, $\text{cost}_A(s) \leqq (17/3)s^{\log_2 3} - 8s + 3$.

Proof. Let us first present the algorithm (similar to [15, p. 258]):

Algorithm: Split multiplication of polynomials.

Input: (f, g) polynomials of size $\leqq 2^k$.

Output: $f \cdot g$.

1. If f or g is a monomial, use classical multiplication. Otherwise, let
$$f = f_1 x^{2^{k-1}} + f_0 \quad \text{and} \quad g = g_1 x^{2^{k-1}} + g_0,$$
where sizes of f_1, f_0, g_1 and $g_0 \leqq 2^{k-1}$.
2. Compute $A = f_1 \cdot g_1$, and $C = f_0 \cdot g_0$ by a recursive application of this algorithm. (A and C are of size $2^k - 1$.)
3. Compute $B = (f_1 + f_0) \cdot (g_1 + g_0)$ by a recursive application of this algorithm.
4. Compute $B = B - A - C$. (B is of size $2^k - 1$.)
5. Return $Ax^{2^k} + Bx^{2^{k-1}} + C$. (This is a shifting operation, and requires no multiplication.)

Let us analyze the specific case in which f and g are of degree $2r - 1$, and thus are of size $2r$. Then $f = f_1 x^r + f_0$, where f_1 and f_0 are of degree $r - 1$ (size r) and similarly for g. In step 2, A and C are computed by applying this algorithm to polynomials of degree $r - 1$ (or size r), and similarly in step 3, the multiplication is of polynomials of size r. Thus

$$\text{cost}_M(2r) = 3\,\text{cost}_M(r) \Rightarrow \text{cost}_M(s) = s^{\log_2 3}.$$

For cost_A, if $r = 1$, $\text{cost}_A(2) = 4$ (simply count the additions). For $r > 1$, consider each step separately in $\text{cost}_A(2r)$:

Step 2. $2\,\text{cost}_A(r)$ is incurred by the two smaller multiplications.

Step 3. $\text{cost}_A(r)$ for the lower multiplication, plus $(f_1 + f_0)$ and $(g_1 + g_0)$, each of cost at most r. Total for step 3: $2r + \text{cost}_A(r)$.

Step 4. B, A and C are of size $2r - 1$, so the two polynomial subtractions (costing as much as additions) cost $2(2r - 1) = 4r - 2$.

Step 5. This is a tricky one. The degree of C is sufficient so that some terms from it combine with terms from Bx^r. Specifically, C is of degree $2r - 2$, so that $r - 2$ terms coincide with Bx^r; this addition is of cost $r - 2$. Similarly, B is of degree $2r - 2$, so that $r - 2$ terms coincide with Ax^{2r}. Total addition cost is $2r - 4$.

Thus

$$\begin{aligned}\text{cost}_A(2r) &= 3\,\text{cost}_A(r) + 8r - 6,\\ \text{cost}_A(s) &= 3\,\text{cost}_A(s/2) + 4s - 6\\ &= \tfrac{17}{3}s^{\log_2 3} - 8s + 3. \quad \square\end{aligned}$$

Thus while classical multiplication takes about $\text{cost}_A + \text{cost}_M \cong 2s^2$, "split" takes about $(20/3)s^{1.585}$. Split should become better when s is about 19. Actually, most implementations of split will encounter significant further costs for shifting, etc. We have been unable to implement split so that it requires less time in the univariate case, even for $s > 40$.

Let us examine the situation for several variables.

THEOREM 3.2a. *If f is dense in v variables to degree $d = 2^k - 1$, and g is a polynomial which has no higher degree than f in any variable, $f \cdot g$ may be computed with* $\text{cost}_M(v, d) = (d + 1)^{v \log_2 3}$.

Proof. The proof is trivial by induction on v, the number of variables, and the result of Theorem 3.2. $\square$

To compare these results with those of Theorem 3.1, classical multiplication, if we compare cost_M in each case, and neglect lower order effects, we compare

$$\text{split} = s^{1.585v} \quad \text{to} \quad \text{classical} = s^{2v},$$

For small s and v, the additions and additional bookkeeping slow split down, so that empirical studies are required.

If we really want to multiply polynomials rapidly, it is clear that the best algorithm for a given problem should be chosen. In a recursive algorithm such as "split", in which subproblems are generated, we require a test which can make this choice expeditiously. A simple way of doing this heuristically (in our implementation) is to test at each level of multiplication of polynomials in subsidiary variables, whether either input is univariate, in which case it generally pays to use classical multiplication.

An exact "formula" for the result of Theorem 3.2a has been derived using dense assumptions, when both inputs are characterized by the same (but arbitrary) degree d, and the same (but arbitrary) number of variables, v. The size need not be a power of 2. The formula is phrased in terms of a computer program in MACSYMA [1]:

split$(d, v) :=$ split1$(d, v, d + 1)$\$
split1$(d, v, h) :=$ block ([ ],
if $v = 1$ then return $(d + 1) \uparrow 2$,
if $h = 1$ then return (split$(d, v - 1)$),
return (split1$(d, v, h/2) + 2 *$ split1$(d, v, (h + 1)//2)$))\$

The // indicates integer quotient, and * indicates multiplication. The symbol $:=$ is used to define a function.

The above program reflects the decision to use classical multiplication for univariate cases ($v = 1$), because of the empirical observation that we cannot implement split efficiently for 1-variable cases in MACSYMA.

It is probably worth pointing out that our idealized worst-case situation is not universally appropriate. If the two inputs are of widely disparate sizes, or if they are not dense, the splitting technique tends to push the computation toward the worst-case situation of two equally large and completely dense polynomials. For this reason, caution should be exercised in using this algorithm. In [11] and [13], the worst-case analysis assumptions make this splitting algorithm quite appropriate.

There is yet another possibility for faster multiplication, analogous to the fastest known integer multiplication technique.

THEOREM 3.3 (FFT multiplication). *If* $s = \text{size}(f)$, *and* $\text{size}(g) \leqq s$, *then* $f \cdot g$ *can be computed with* $\text{cost}_M(s) = \text{cost}_A(s) = O(s \log s)$.

Proof. Let us first present a sketch of the algorithm. A detailed description can be found in Bonneau [2] or [4], and the exact method of programming is not relevant to our discussion. Basically it looks like this for univariate f and g:

1. Compute a "discrete Fourier transform" of the sequence of coefficients of f, extended (with zeros) to be of at least the size of the answer. Do the same for g.
2. Multiply, term by term, the corresponding elements of the transformed sequences.
3. Compute the inverse transform of the result.

It can be shown (again, refer to Bonneau [4] or to Pollard [18]) that steps 1 and 3 may be done in $O(s \log s)$ coefficient operations, and step 2 can be done, clearly, in $O(s)$.

Efficient implementation of the Fourier transform has been a problem, but it is clear at this point that for moderately large multiplication problems, or for small problems involving many operations which can be done rapidly in the transform space (e.g., polynomial powering) the "FFT" approach is quite practical. We will return to this in the next section.

4. Algorithms for computing powers of polynomials. We will briefly discuss the algorithms RMUL and RSQ presented in [11] and [13] and show how various multiplication algorithms affect the costs. We then move on to detailed considerations of a number of additional algorithms.

In each of the following sections, we discuss an algorithm which computes a power n of a polynomial f which is completely dense to degree d in each of v variables.

4.1. Algorithm RMUL (repeated multiplication).

4.1.1. Description of RMUL. RMUL successively computes $f^2 = f \cdot f$, $f^3 = f \cdot f^2, \cdots, f^n = f \cdot f^{n-1}$.

4.1.2. Analysis of RMUL. For classical multiplication, the cost_M is easily seen to be $\text{size}(f) \cdot \text{size}(f) + \text{size}(f) \cdot \text{size}(f^2) + \cdots + \text{size}(f) \cdot \text{size}(f^{n-1})$. More precisely, since $\text{size}(f^i) \leqq (id + 1)^v$, the cost for classical multiplication is

$$(4.1.2a) \qquad (d+1)^v \sum_{i=1}^{n-1} (id+1)^v < (d+1)^{2v} \sum_{i-1}^{n-1} i^v < (d+1)^{2v} n^{v+1}/(v+1).$$

Note that the left side of (4.1.2a) is an exact count. It is not "asymptotic". It may be evaluated by hand or computer to predict exactly how many multiplications will be required, given n, d and v. Although it assumes no zero coefficients are generated, this is a reasonable prospect, given the input assumptions. (It is fulfilled, for example, if the input coefficients are floating point numbers greater than 0.)

A computerized formulation of (4.1.2a) has a particular advantage in that MACSYMA can evaluate the program, exactly, when only n, or only v is supplied. Thus in MACSYMA,

$$\text{rmul}(n, d, v) := (d + 1) \uparrow v * \text{SUM}((i * d + 1) \uparrow v, i, 1, n - 1)\$$$

$$\text{rmul}(n, d, 2);$$

$$(1/6)((2d^4 + 4d^3 + 2d^2)n^3 + (-3d^4 + 9d^2 + 6d)n^2 + (d^4 - 4d^3 - 5d^2 + 6d + 6)n - 6d^2 - 12d - 6) \tag{4.1.2b}$$

It would seem reasonable to improve RMUL by using faster multiplication methods, but it can be shown that other methods of polynomial multiplication do not thrive on the unequal-sized multiplicands produced by RMUL. Therefore we move on to other powering methods.

4.2. Algorithm RSQ (repeated squaring).

4.2.1. Description of RSQ. RSQ computes f^n by computing a sequence of polynomials based on the binary expansion of n. For example, if $n = 7$, it computes $f^2 = f \cdot f, f^3 = f \cdot f^2, f^4 = f^2 \cdot f^2, f^7 = f^3 \cdot f^4$. If $n = 8, f^8 = ((f \cdot f)^2)^2$, which is why we call it repeated squaring.

More precisely:

1. Set Q to 1. Set Z to f.
2. Set I to the rightmost bit of n (in binary). Shift n right by one bit. If I is 0, go to step 4.
3. If Q equals 1, set Q to Z, otherwise set Q to $Q \cdot Z$.
4. If n is 0, return Q. Otherwise, set Z to $Z \cdot Z$ and go to step 2.

4.2.2. Analysis of RSQ. Let us consider RSQ only for $n = 2^j$. It may seem that we are giving RSQ some sort of advantage, but closer analysis can be done which shows this is not necessarily so. It "evens out" in the long run.

Heindel [11] and Horowitz [13] show that using classical multiplication, RSQ is $O((dn)^{2v})$. They therefore reject it, since RMUL is only $O(d^{2v}n^{v+1})$.

The cost_M for RSQ using *split* multiplication is *approximated* by

$$\begin{aligned} \sum_{i=0}^{j-1} \text{size}(f^{2^i})^{1.585} &= \sum_{i=0}^{j-1} (2^i d + 1)^{1.585v} \\ &< (d + 1)^{1.585v} \sum_{i=0}^{j-1} (2^{1.585v})^i \\ &< (n(d + 1))^{1.585v}/(3^v - 1) = O((nd)^{1.585v}). \end{aligned} \tag{4.2.2a}$$

A more exact rendition of (4.2.2a) can be given as a MACSYMA program: for $n = 2^j$, an exact formula is

$$\text{rsq}(n, d, v) : \text{SUM}(\text{split}(2 \uparrow i * d, v), i, 0, \text{ceillog2}(n) - 1)\$$$

where ceillog2 computes the smallest integer greater than the base-2 logarithm of n. For n not a power of 2, this is a gross overestimate. An analysis of split for unequal-sized inputs removes the power-of-2 restriction, but entails needless and lengthy explanations.

To see how good this is compared to RMUL, we may look at a typical large case, $n = 4$, $d = 3$, $v = 3$. At this point Horowitz [13] indicates that RSQ is twice as time-consuming as RMUL, when they are both using classical multiplication. In fact, using split multiplication, RSQ uses 31921 coefficient multiplications, while RMUL, using classical multiplication, requires 90048 coefficient multiplications. RSQ is faster than RMUL for almost all problems of smaller size, the exceptions being large n and $v > 2$, combined with small d (e.g., $n = 16$, $v = 3$, $d < 4$).

4.3. Algorithm SUMS.

4.3.1. Description of SUMS. SUMS is a clever method described by Knuth [15] for raising a power series to a power. (See also Fettis [8]).

Briefly, if f is a univariate polynomial, $f = \sum_{i=0}^{d} f_i x^i$, then

$$g = f^n = \sum_{i=0}^{nd} g_i x^i, \quad \text{where } g_0 = f_0^n,$$
$$g_i = \frac{1}{i f_0} \sum_{j=1}^{\min(d,i)} ((n+1)j - i) \cdot f_j \cdot g_{i-j}, \qquad 1 \leqq i \leqq nd., \tag{4.3.1a}$$

The obvious extension to multivariate cases is a poor method, since division by f_0 is expensive. If, however, we multiply f by a new "variable", f_0^*, such that $f_0 \cdot f_0^* = 1$, and compute $g^* = (f \cdot f_0^*)^n$ by SUMS, finally substituting f_0^j for $(f_0^*)^{(n-j)}$ in g^*, we obtain the correct answer.

4.3.2. Analysis of SUMS. For the univariate case, we can itemize cost_M as follows:

computation of g_0: 1 exponentiation.

computation of g_i: 2 multiplications for each of the $\min(d, i)$ iterations of the sum. For $0 < i < d$, the cost to compute each g_i is $2i$, totaling $\sum_{i=1}^{d-1} 2i = d(d-1)$. For $d \leqq i \leqq nd$, the cost to compute each g_i is $2d$, totaling $\sum_{i=d}^{nd} 2d = 2d(nd - d + 1)$. There are also nd divisions by if_0. Adding all these costs gives

$$\text{cost}_M = (2n-1)d^2 + (n+1)d. \tag{4.3.2a}$$

This is *exact*, and not an asymptotic result.

This contrasts strongly with the previous methods in being linear in n, instead of quadratic. Since the leading coefficient is $2n - 1$ rather than $n^2/2$ as in RMUL, univariate SUMS does not become faster than RMUL until $n > 4$. However, for $n > 4$ SUMS rapidly becomes far superior. Only the FFT method is better for large n and d.

For the several variable case, the algorithm is complicated by the back-substitution, and by the variety of sizes of the coefficients. The algorithm loses its superiority, and can be shown [5] to be, for $v > 1$,

$$O(n^v(d+1)^{2v} + n^{v+1}(d+1)^{2v-1}). \tag{4.3.2b}$$

Since SUMS turns out to be both difficult to program for $v > 1$ and not very fast, we can ignore the multivariate consequences.

4.4. Algorithm EVAL (evaluation homomorphism).

4.4.1. Description of EVAL. EVAL computes the nth power of the degree-d polynomial $f(x)$ (say f is univariate) by computing $f(b_i)^n$ for $nd + 1$ integers $b_1 = 0, \cdots, b_{nd+1} = nd$, and then using interpolation to compute $g = f^n$. For several variables, EVAL is used recursively, since $f(b_i)$ would be a polynomial in 1 fewer variables than $f(x)$.

More precisely, EVAL is as follows.

1. If $v = 0$, i.e., f is an integer, return the integer f^n.
2. Set $c(x_1, \cdots, x_{v-1})$ to 0, $h(x_v)$ to 1, b to -1.
3. Set b to $b + 1$.
4. Set $\bar{f}$ to $f(x_1, \cdots, x_{v-1}, b)$ (evaluation).
5. Set g to $\bar{f}^n$ (computed by EVAL).
6. If $h = 1$, set c to g; otherwise, set $c(x_1, \cdots, x_v)$ to
$$\frac{h(x_v)}{h(b)} \cdot (g - c(x_1, \cdots, x_{v-1}, b)) + c(x_1, \cdots, x_v)$$
(interpolation).
7. Set $h(x_v)$ to $(x_v - b)h(x_v)$.
8. If degree of h is less than or equal to n times the degree of x_v in f $(=d)$, then go to 3; otherwise return the value c.

We have stated this algorithm, first proposed by Horowitz [13], with some slight improvements. A further improvement apparent from the analysis, is that recursion down to $v = 0$ is a poor choice; when $v = 1$, faster algorithms can be used.

4.4.2. Analysis of EVAL. We will consider only cost_M for EVAL.

If $v = 0$, EVAL executes only step 1, using 1 exponentation. For $v = 0$, then, $\text{cost}_M(0) = 0$.

To compute $\text{cost}_M(v)$, let us itemize multiplications at each step.

Step 4. This is executed $nd + 1$ times, and all but the first execution requires, using Horner's rule, d multiplications of b by a polynomial of size $(d + 1)^{v-1}$. The first time through, $b = 0$, requiring no multiplications. The cost_M for step 4 is therefore

$$nd^2(d + 1)^{v-1}. \tag{4.4.2a}$$

Step 5. This is executed $nd + 1$ times for a

$$\text{cost}_M = (nd + 1)\,\text{cost}_M(v - 1). \tag{4.4.2b}$$

Step 6. The first time through, $h = 1$, so the cost $= 0$. The ith time through, for $i = 2, \cdots, nd + 1$, the multiplication costs can be itemized as follows (note that at the ith step, $h(x_v) = x_v \cdot (x_v - 1) \cdot \cdots \cdot (x_v - i + 2)$ and has $i - 1$ terms):

$$h(b) : i - 1,$$
$$h^* = h(x_v)/h(b) : i - 1,$$
$$c^* = c(x_1, \cdots, x_{v-1}, b) : (nd + 1)^{v-1}(i - 2),$$
$$h^* \cdot (g - c^*) : (nd + 1)^{v-1}(i - 1),$$

Thus for step 6, cost_M is

$$\sum_{i=2}^{nd+1} [(nd+1)^{v-1}(2i-3) + 2i - 2] = n^2d^2(nd+1)^{v-1} + n^2d^2 + nd. \tag{4.4.2c}$$

Step 7. This step is executed $nd + 1$ times, costing

$$2 + \sum_{i=2}^{nd+1} 2(i-1) = n^2d^2 + nd + 2. \tag{4.4.2d}$$

Including all these cost, we find

$$\begin{aligned} \text{cost}_M(v) = (nd+1)\,\text{cost}_M(v-1) + nd^2(d+1)^{v-1} \\ + n^2d^2(nd+1)^{v-1} + 2n^2d^2 + 2nd + 2. \end{aligned} \tag{4.4.2e}$$

The formula above is *exact*. It is *not* asymptotic. It predicts precisely how many multiplications our EVAL implementation needs given n, d and v, assuming no zero coefficients are generated, a reasonable prospect, given the worst-case input assumptions.

If we assume $\text{cost}_M(0) = 0$, we can show from (4.4.2e) that EVAL is $O((nd)^{(v+1)})$, and is, asymptotically speaking, more efficient than the previous algorithms, at least for $v > 1$.

The cutoff point at which EVAL becomes better than, say, RMUL, reveals how important (or unimportant) this asymptotic result is. Let us compare exact formulas for EVAL and RMUL as computed from (4.4.2e) and (4.1.2a), respectively, for several values of v. Since (4.4.2e) can be represented as a computer program in the same way as (4.1.2a), we can display the result exactly for $v = 2$, and compare with (4.1.2b). It is

$$4d^3n^3 + (d^3 + 8d^2)n^2 + (d^3 + 2d^2 + 5d)n + 3. \tag{4.4.2f}$$

Since most of the formulas themselves are rather large, we will, for the most part, display the leading terms only. However, exact formulas were used to determine the cutoff points in Table 1. The cutoff points are the smallest n and d where EVAL uses fewer multiplications than RMUL.

TABLE 1

v	RMUL	EVAL	Cutoff
1	$(1/2)d^2n^2$	$3d^2n^2$	—
2	$(1/3)(d^4 + 2d^3)n^3$	$4d^3n^3$	$n = 2,\ d = 35$; $n = 4,\ d = 18$; $n = 6,\ d = 15$; $n = 200, d = 10$
3	$(1/4)(d^6 + 3d^5 + 3d^4)n^4$	$5d^4n^4$	$n = 2,\ d = 8$; $n = 3,\ d = 5$; $n = 4,\ d = 5$; $n = 5,\ d = 4$; $n = 29, d = 3$.

From Table 1, we see that for many quite sizable problems, EVAL is worse than RMUL. The time for EVAL is frequently several times that for RMUL.

The timings given in tables in [13] don't go above these cutoff points, and yet EVAL is claimed to be superior to RMUL for many much smaller problems. In fact, on the basis of exact counting of all arithmetic operations, and actual timings of our programs, we believe the tables in [13] reflect only irrelevant factors related to the implementation.

In § 5 we give counts for EVAL modified to use RMUL instead of EVAL when recursion requires powering univariate polynomials. This is a considerable improvement as shown by Table 2, calculated from (4.4.2e) but using cost_M (1) derived from (4.1.2a).

TABLE 2

v	EVAL (+RMUL)	Cutoff points (EVAL better than RMUL)
1	$(1/2)d^2n^2$	—
2	$(3/2)d^3n^3$	$n = 2, d = 10$; $n = 3, d = 6$; $n = 4, d = 5$; $n = 5, d = 4$; $5 < n < 15, d = 3$; $n \geqq 15, d = 2$
3	$(5/2)d^4n^4$	$n = 2, d = 5$; $n = 3, d = 3$; $n = 4, d = 3$; $n \geqq 5, d = 2$

Since SUMS is better than RMUL for univariate polynomials, $n > 4$, we can improve EVAL for $n > 4$ by using SUMS instead of RMUL. Analysis of this situation reveals that EVAL is worsened for small cases, as expected, and improved (but not very much) for large cases. There is really no need to compromise between RMUL and SUMS: if $n < 4$, we could use RMUL, otherwise SUMS.

We can observe, now, that even though we can, for $v = 3$, improve EVAL to $2d^4n^4 + \cdots$, (the result of using SUMS), it is still not better than RMUL for moderate size problems. Since RMUL is inferior to RSQ for most problems, this puts EVAL typically no higher than third best. Its exact standing, of course can be derived from the evaluation of the appropriate counting formulas.

4.5. Algorithm BINOM (binomial expansion).

4.5.1. Description of BINOM. BINOM uses an adaptation of binomial expansion to compute the nth power of the polynomial f.

The f is either a monomial or a sum. If it is a monomial, powers are trivially calculated. If f is a sum, it may be split into two subparts, $R + S$. To compute f^n, compute powers of $R: R^2, \cdots, R^n$ and powers of $S: S^2, \cdots, S^n$, and use the binomial theorem:

$$f^n = \sum_{i=0}^{n} \binom{n}{i} R^i S^{n-i}.$$

While R^2 and S^2 may be calculated by BINOM, it greatly simplifies the analysis if we simply use classical multiplication to compute $R^2 = R \cdot R$, $R^3 = R \cdot R^2$, etc. The actual difference is small, for powers exceeding 2.

Two methods for splitting f come to mind, the first being to try to separate f into two equal-sized parts, and the other is to split off a monomial. Monomial splitting was first proposed in [5]. In either case, the algorithm has four steps:

1. Compute $R^2, \cdots, R^n$.
2. Compute $S^2, \cdots, S^n$.

3. Compute $h_i = \binom{n}{i} R^i S^{(n-i)}$ for $i = 1, \cdots, n-1$.

4. Compute $\sum_{i=1}^{n-i} h_i + R^n + S^n$.

4.5.2. Analysis of BINOM. We first analyze BINOM for equal splitting. Assume, for convenience, that the degree d is odd. Then R and S are the same size. Let $M(i) = \text{size}(R^i) = \text{size}(S^i) = (id+1)^{v-1}(i(d-1)/2 + 1)$.

The multiplication costs are itemized as follows:

Step 1. $M(1)\sum_{i=1}^{n-1} M(i)$.

Step 2. Same as 1.

Step 3. $\sum_{i=1}^{n-1} M(i)(M(n-i)+1)$.

Step 4. 0.

The total is therefore

$$\sum_{i=1}^{n-1} M(i)(2M(1) + M(n-i) + 1). \tag{4.5.2a}$$

This formula is *exact* if d is odd, and BINOM is not used recursively. This formula is $O(n^{2v+1}d^{2v})$.

If we use "monomial" splitting, we separate f at the highest power of the main variable. That is, if $f = f_d x_v^d + \cdots + f_0$, then $R = f_d x_v^d$, and $S = f_{d-1}x_v^{d-1} + \cdots + f_0$. Basically, R is now a polynomial in $v-1$ variables (times x_v^d). In this case, let

$$N(i) = \text{size}(S^i) = (id+1)^{v-1}(i(d-1)+1) \quad \text{and} \quad L(i) = \text{size}(R^i) = (id+1)^{v-1}.$$

The multiplication costs are itemized as follows:

Step 1. $L(1)\sum_{i=1}^{n-1} L(i)$.

Step 2. $N(1)\sum_{i=1}^{n-1} N(i)$.

Step 3. $\sum_{i=1}^{n-1} L(i)(N(n-i)+1)$.

Step 4. 0.

The total is therefore

$$\sum_{i=1}^{n-1} L(i)(N(n-i) + L(1) + 1) + N(i)N(1). \tag{4.5.2b}$$

This formula is *exact*, for any n, d and v, assuming that BINOM is not used recursively for squaring.

This formula is $O(n^{2v}d^{2v-1} + n^{v+1}d^{2v})$. For asymptotic considerations, monomial splitting appears superior. In this case, appearances do not deceive, and (4.5.2b) is usually smaller than (4.5.2a). While half-splitting is better for small cases, by $v = 2$, $n = 4$, monomial splitting is better for all d.

Of course, RMUL is only $O(n^{v+1}d^{2v})$, and EVAL is $O((nd)^{v+1})$, while BINOM is $O(n^{2v}d^{2v-1} + n^{v+1}d^{2v})$, so that for large enough problems, BINOM will be inferior. Yet we can show by (4.5.2b) (and confirm by timings) that BINOM is really rather good for most problems of interest.

If we proceed by evaluating (4.5.2b) for $v = 2$, we find that cost_{M} is exactly

$$\begin{aligned}(1/12)&((d^3 - d^2)n^4 + (4d^4 + 4d^2 - 6d)n^3 \\ &+ (-6d^4 + 11d^3 + 13d^2 + 30d - 6)n^2 \\ &+ (2d^4 - 12d^3 - 4d^2 + 42)n - 12d^2 - 24d - 36).\end{aligned} \tag{4.5.2c}$$

This can be compared directly with RMUL (4.1.2b) and EVAL (4.4.2f) for $v = 2$. To make the comparison more direct, let us use the improved EVAL (at $3/2d^3n^3$) and fix d to be 3. Then

RMUL costs $48n^3 - 24n^2 - 8n - 16$,
EVAL costs $45n^3 + 27n^2 + 28n - 3$,
BINOM costs $(28.5 + 1.5n)n^3 + n^2 - 13n - 18$.

The problem must be fairly large for BINOM to be "bad", as can be seen in § 5, where cost tables are computed. Another facet of BINOM's attractiveness, in spite of its poor dense asymptotic behavior, is that for sparse problems, it has good asymptotic behavior, as discussed in [6].

4.6. Algorithm FFT (fast Fourier transform).

4.6.1. Description of Algorithm FFT. Let p be a prime number. Let F be the multiplicative group consisting of the nonzero elements of $GF(p)$, the integers modulo p. Let h be a divisor of $p - 1$, possibly $h = p - 1$, and let u be an element of order h in F. Then one may transform a sequence $(a_i)_{i=0}^{h-1}$ of elements of $GF(p)$ into another sequence $(\hat{a}_i)_{i=0}^{h-1}$ by the finite Fourier transform

$$\hat{a}_i = \sum_{j=0}^{h-1} a_j u^{ij}. \tag{4.6.1a}$$

The inverse transform is defined by

$$a_i = h' \cdot \sum_{j=0}^{h-1} \hat{a}_i u^{-ij}, \tag{4.6.1b}$$

where h' is the inverse of h in $GF(p)$. Equations (4.6.1a) and (4.6.1b) can be calculated by the "fast Fourier transform" [18] where complex multiplication and addition are replaced by the corresponding operations in F. A detailed description of the FFT in a finite field and proofs of its important properties may be found in [4] or [18].

The property of the FFT which we use is the following.
Let

$$f = \sum_{i=0}^{d} a_i x^i, \qquad a_i \in F.$$

Then

$$f^n = \sum_{i=0}^{nd} c_i x^i, \qquad c_i \in F,$$

where (c_i) may be calculated by FFT's. More precisely, we extend (a_i) to be of length $h > nd$. Then compute $(\hat{c}_i) = (\hat{a}_i n)$, that is, raise each element of $(\hat{a}_i)$ to the nth power in F. (In the multivariate case, the $\hat{a}_i$ would be polynomials in $v - 1$ variables, for which this algorithm could be used recursively to compute $\hat{c}_i = \hat{a}_i^n$).

The inverse transform of $(\hat{c}_i)$ is (c_i). This answer is over the field F.

Ordinarily it is convenient to let the algorithm choose the field F, to take advantage of precomputed values of u, or to require that a member of some particular list of primes be used for the modulus of the field F. Furthermore, h is generally rounded up to a power of 2 or 3. (In § 5, we are timing a power-of-2 FFT).

Some details may be found in [18], but Bonneau [4] gives precise descriptions of an implementation, and specifies how one can overcome the difficulties cited by Pollard. The extension for results over the integers is also described by Bonneau.

4.6.2. Analysis of FFT. Let us assume that a suitable field F, and a $u \in F$ have been computed. The number of operations (multiplications and additions) required to calculate an FFT or inverse FFT of a sequence of integers in $GF(p)$ of K elements is $O(K \log K)$. Between these two transforms one must compute nth powers of the K elements. The powering can be done with K exponentiations.

For a univariate polynomial where $K = nd + 1 < 2^j$, our programs require *exactly* (see [4] for details)

$$j2^{j-1} - 2 \tag{4.6.2a}$$

multiplications for the FFT and

$$j^{2^{j-1}} + 2^j - 2 \tag{4.6.2b}$$

multiplications for the inverse FFT. Thus $\text{cost}_{\mathbf{M}}$ is

$$O(nd \log (nd)). \tag{4.6.2c}$$

This is the best asymptotic bound known for this problem.

If the elements of the sequences are polynomials in subsidiary variables, then the transforms are done on the coefficients (etc.) down to elements of the base field. Then $\text{cost}_{\mathbf{M}}(v)$ can be itemized as follows:

Step 1. If $v = 0$, use one exponentiation. Otherwise compute $(\hat{a}_i)$, the transform of (a_i) of length $> nd + 1$.

Step 2. Compute $(\hat{c}_i)$, the componentwise nth power of $(\hat{a}_i)$ using this algorithm (or one of the others) recursively.

Step 3. Compute the inverse transform of $(\hat{c}_i)$.

For $v = 0$, we can say that cost of computing the nth power of a finite field element is less than $2 \log_2 (n)$ multiplications = one exponentiation. To keep our analysis consistent (and inconsequentially different), we ignore this cost, and count only the multiplications used. If $2^j > nd + 1$,

$$\begin{aligned} &\text{cost}_{\mathbf{M}}(0) = 0, \\ &\text{cost}_{\mathbf{M}}(v) = (d+1)^{v-1}(j2^{j-1} - 2) + 2^j \text{cost}_{\mathbf{M}}(v-1) \\ &\qquad + (nd+1)^{v-1}(j2^{j-1} + 2^j - 2) \qquad (v > 0). \end{aligned} \tag{4.6.2d}$$

This formula is not exact, unfortunately, but an upper bound, since not all transform arithmetic will not be as costly as indicated; many of the coefficients in the input sequence are zero, since the sequence has been extended to size 2^j where $2^j > nd + 1$, so only $d + 1$ elements are nonzero. A similar consideration holds for the inverse transform.

Nevertheless, this formula is still

$$O(v(nd)^v \log (nd)). \tag{4.6.2e}$$

This is the best known asymptotic bound for the dense multivariate case. Again, we refer the reader to [4] for implementation details.

The point at which the FFT algorithm becomes practical can be seen from a few specific cases, as given in § 5.

For $v = 1$, $d = 7$, the FFT method is best for $n > 3$. For $v = 1$, $d \geqq 15$, the FFT is best for any n.

For $v = 2$, $d = 3$, the FFT is best for $n \geqq 7$. For $v = 2$, $d = 6$, the FFT is best for $n \geqq 4$. At $n = 5$ it is almost 4 times faster than the next best.

For $v = 3, d = 4$, it is best for $n > 2$.

These figures are for "pure" FFT methods. Since the algorithm, as implemented, provides convenient points at which other programs can be invoked, a substitution of RSQ (for example) for the FFT method is quite feasible. In fact, by using the FFT in combination with RSQ and perhaps BINOM, one algorithm can be constructed to use the fewest operations over nearly the whole range of problems which are correctly characterized by the completely dense assumptions of this paper.

5. Empirical tests. The following tables give the number of multiplications required to compute the indicated powers using programs in the MACSYMA system for algebraic manipulation [1], [17]. All polynomial multiplications except those in RSQ are done using classical multiplication. RSQ uses split multiplication. EVAL uses RMUL for univariate polynomials, otherwise it would be much slower.

TABLE 3
Univariate polynomials ($V = 1$)

D = 2	2	3	4	5	6	7	8
RMUL	9	24	45	72	105	144	189
RSQ	9	24	32	61	79	112	115
SUMS	18	28	38	48	58	68	78
BINOM	7	18	32	49	69	92	118
FFT	28	28	76	76	76	76	188

D = 7	Power (N)						
	2	3	4	5	6	7	8
RMUL	64	184	360	592	880	1224	1624
RSQ	64	184	289	521	724	1047	1130
SUMS	168	273	378	483	588	693	798
BINOM	57	163	317	519	769	1067	1413
FFT	76	188	444	444	444	444	444

D = 15	Power (N)						
	2	3	4	5	6	7	8
RMUL	256	752	1488	2464	3680	5136	6832
RSQ	256	752	1217	2193	3108	4519	4938
SUMS	720	1185	1650	2115	2580	3045	3510
BINOM	241	707	1397	2311	3449	4811	6397
FFT	188	444	444	1020	1020	1020	1020

TABLE 4
Bivariate polynomials ($V = 2$)

D = 2	Power (N)						
	2	3	4	5	6	7	8
RMUL	81	306	747	1476	2565	4086	6111
RSQ	63	228	488	1190	1703	2732	3971
EVAL	191	542	1175	2174	3623	5606	8207
BINOM	66	233	555	1094	1920	3111	4753
FFT	308	342	1556	1636	1724	1820	7484

D = 3	Power (N)						
	2	3	4	5	6	7	8
RMUL	256	1040	2640	5344	9440	15216	22960
RSQ	144	676	1369	3761	6010	10091	13368
EVAL	522	1540	3422	6438	10858	16952	24990
BINOM	212	843	2154	4442	8040	13317	20678
FFT	261	1397	1505	6903	7137	7497	7749

D = 6	Power (N)			
	2	3	4	5
RMUL	2401	10682	28371	58996
RSQ	1225	5866	13224	36149
EVAL	3327	10258	23435	44910
BINOM	2114	9477	25635	54494
FFT	1790	7830	8430	9102

TABLE 5
Trivariate polynomials ($V = 3$)

D = 1	Power (N)						
	2	3	4	5	6	7	8
RMUL	64	280	792	1792	3520	6264	10360
RSQ	36	186	477	1477	2292	4007	7702
EVAL	184	614	1558	3322	6284	10894	17674
BINOM	52	189	470	974	1804	3091	4998
FFT	172	2068	2208	2616	2806	23836	24460

D = 3	Power (N)		
	2	3	4
RMUL	4096	26048	90048
RSQ	1296	11404	31921
EVAL	5792	24114	69712
BINOM	3344	21441	78474
FFT	2014	21630	25410

For univariate polynomials, we do not give counts for EVAL, since EVAL = RMUL. For multivariate polynomials we do not give counts for SUMS, since its performance is uniformly poor in these cases.

Similar charts using computation time rather than multiplication counts are essentially identical. In almost all cases, the fastest algorithm used the fewest multiplications. (Occasionally RSQ is relatively slower than the multiplication count indicates because of the larger number of additions).

6. Summary. If one is presented with a problem in which the worst-case assumptions of complete denseness do hold, the logical choice for a univariate powering algorithm is the FFT, unless all problems are of trivial size. For several variables of low degree ($d = 2, 3, 4$), to a low power ($n = 2$ or 3), BINOM or RSQ has the edge. For more variables of larger degree and to higher powers, FFT will be superior.

If one is interested in precisely minimizing the number of multiplications (and thus, in our experiments, minimizing the run time), a decision between EVAL, RMUL and BINOM can be made on the basis of formulas provided. If n is a power of 2, an exact formula for RSQ is provided.

Powers of polynomials whose coefficients are multi-precision integers can be computed by any of these algorithms by using the Chinese remainder algorithm [7], [12], [13]. Alternatively, all but the FFT can be used directly on multi-precision integers.

In terms of polynomial multiplication, we have found the classical method may be convenient to use, but inappropriately slow for dense polynomials. It is especially inappropriate for use in asymptotic analysis. Split multiplication or FFT multiplication must be considered as practical algorithms, and certainly should be used in asymptotic analysis under dense assumptions. Implementations in MACSYMA [3] have demonstrated that these methods, in contrast to their analogues in integer arithmetic, are practical even for reasonably small problems.

We believe that the FFT has been unnecessarily neglected by designers of algebraic manipulation systems, and trust that forthcoming work by researchers in the field of algebraic manipulation will find numerous practical applications.

One question, implicit in this work, which we have not attempted to answer concerns the relevance of analysis which assumes "worst cases". It is clear that worst-case problems can be constructed, but it is not at all clear that the best algorithm chosen for these assumptions is the best algorithm for an "average" case. Since there appears to be no satisfactory characterization for average cases (although one is attempted in [14]), one can only point out the problem and hope this work will be relevant in providing intuition. An alternative model, using a concept of "completely sparse" as defined in [9], gives another handle on the problem. Formulas analogous to those of this paper, for sparse polynomials, for a number of algorithms, are given in [6]. The conclusion there is that a method similar to BINOM is a good choice.

In any case, the problems of characterizing polynomials are serious, and have been largely ignored in algorithmic analysis of polynomial algorithms. We hope to see more progress in this area.

Acknowledgments. The MACSYMA system [1] provided the motivation for this study, and was invaluable in supplying data and deriving formulas.

The author wishes to thank his colleagues at Project MAC and elsewhere for their comments on earlier drafts of this paper, and especially to Richard Bonneau, for bringing the FFT to a point where it could be used in "non-asymptotic" situations.

REFERENCES

[1] R. BOGEN, *The MACSYMA Manual*, Project MAC, Mass. Inst. of Tech., Cambridge, Mass., 1973.

[2] R. J. BONNEAU, *Polynomial exponentiation: The fast Fourier transform revisited*, TM34, Project MAC, Mass. Inst. of Tech., Cambridge, Mass., 1973.

[3] ———, *A class of finite computation structures supporting the fast Fourier transform*, TM31, Project MAC, Mass. Inst. of Tech., Cambridge, Mass.; presented at SIAM National Conf., Hampton Beach, Va., 1973.

[4] ———, *Polynomial operations using the fast Fourier transform*, Ph.D. thesis, Dept. of Mathematics, Mass. Inst. of Tech., Cambridge, Mass., 1974.

[5] R. J. FATEMAN, *On the computation of powers of polynomials*, Dept. of Mathematics, Mass. Inst. of Tech., Cambridge, Mass., 1972.

[6] ———, *On the computation of powers of sparse polynomials*, Studies in Appl. Math., 53 (1974), pp. 145–155.

[7] ———, *On the implementation of modular algorithms*, Dept. of Mathematics, Mass. Inst. of Tech., Cambridge, Mass., presented at SIAM National Conf., Hampton Beach, Va., 1973.

[8] H. E. FETTIS, *Algorithm 158*, Comm ACM, 6 (1963), p. 104.

[9] W. M. GENTLEMAN, *Optimal multiplication chains for computing a power of a symbolic polynomial*, Math. Comp., 26 (1972), pp. 935–939.

[10] ———, *On the relevance of various cost models of complexity*, Complexity of Sequential and Parallel Algorithms, J. F. Traub, ed., Academic Press, New York, 1973.

[11] L. E. HEINDEL, *Computation of powers of multivariate polynomials over the integers*, J. Comput. System Sci., 6 (1972), pp. 1–8.

[12] E. HOROWITZ, *Modular arithmetic and finite field theory: A tutorial*, Proc. 2nd Sympos. of Symbolic and Algebraic Manipulation, S. R. Petrick, ed., Association for Computing Machinery, Los Angeles, 1971, pp. 188–195.

[13] ———, *The efficient calculation of powers of polynomials*, 13th Ann. Sympos. on Switching and Automata Theory, IEEE Computer Society, October, 1972.

[14] E. HOROWITZ AND S. SAHNI, *On the Computation of Powers of a Class of Polynomials*, TR72–143, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., 1973; presented at SIAM National Conf., Hampton Beach, Va., 1973.

[15] D. E. KNUTH, *The Art of Computer Programming. Vol. 2: Semi-numerical Algorithms*, 2nd ed., Addison-Wesley, Reading, Mass., 1969.

[16] J. D. LIPSON, *Chinese remainder and interpolation algorithms*, Proc. 2nd Sympos. of Symbolic and Algebraic Manipulation, S. R. Petrick, ed., Association for Computing Machinery, Los Angeles, 1971, pp. 372–391.

[17] W. A. MARTIN AND R. FATEMAN, *The MACSYMA System*, Proc. of the 2nd Sympos. on Symbolic and Algebraic Manipulation, S. R. Petrick, ed., Association for Computing Machinery, Los Angeles, 1971, pp. 59–75.

[18] J. M. POLLARD, *The fast Fourier transform in a finite field*, Math. Comp. 25 (1971), pp. 365–374.

[19] D. Y. YUN, *The Hensel lemma in algebraic manipulation*, Ph.D. thesis, Dept. of Mathematics, Mass. Inst. of Tech., Cambridge, Mass., 1973.

SIAM J. COMPUT.
Vol. 3, No. 3, September 1974

ON HAMILTONIAN WALKS IN GRAPHS*

S. E. GOODMAN AND S. T. HEDETNIEMI†

Abstract. A Hamiltonian walk in a graph G is a closed walk of minimum length which contains every point of G. An Eulerian walk in a graph G is a closed walk of minimum length which contains every line of G. In this paper we establish several relationships between Hamiltonian and Eulerian walks. We also derive a number of bounds on the length of a Hamiltonian walk.

Key words. Hamiltonian walks, Eulerian walks, graph theory spanning traversals

1. Introduction. Given any connected graph G, it is possible to start at an arbitrary point u of G, walk in some sequence along the lines of G and return to the starting point u having passed through every point in G at least once. In general, such a walk might pass through some points, and traverse some lines, more than once. We call such a walk a *closed spanning walk* of G. A *Hamiltonian walk* in G is a closed spanning walk of minimum length. The length of a Hamiltonian walk in G will be denoted by $h(G)$. If G has p points and $h(G) = p$, then G is a Hamiltonian graph.

Similarly, given any connected graph G, it is possible to start at an arbitrary point u, traverse all the lines of G and return to point u. Again, it is possible that some lines might have to be traversed more than once. We call a walk of this kind a *closed covering walk.* An *Eulerian walk* in G is a closed covering walk of minimum length. The length of an Eulerian walk in G will be denoted by $e(G)$. If G has q lines and $e(G) = q$, then G is an Eulerian graph.

The theory of Eulerian walks is fairly complete. Let G be a connected graph with q lines and $2n$ points of odd degree. In 1736, Euler showed that $e(G) = q$ if and only if $n = 0$. For $n > 0$, Eulerian walks can be studied using the theory described in [1], [2] and [3]. Essentially, this theory finds a set of paths in G, of minimum total length, which connects all the odd points of G in pairs. The lines in this set of paths are then added to G to produce a multigraph $\bar{G}$. Since every point of $\bar{G}$ has even degree, it can be traversed as described in Euler's classical result. It is possible to show that the walk so obtained is an Eulerian walk in G. Various theorems and algorithms are available which aid in finding Eulerian walks.

It is well known that there is no satisfactory characterization of Hamiltonian graphs; and we have never seen any work done on the Hamiltonian walk problem. In this paper we will determine some simple bounds on $h(G)$ and present several relationships between Eulerian and Hamiltonian walks.

In the interests of self-containment, we next present a few definitions; any terms not defined here can be found in Harary [4].

A *cutpoint* V of a connected graph G is a point whose removal results in a disconnected graph $G - V$; similarly, a *bridge* uv of a connected graph G is a line

* Received by the editors May 11, 1973, and in revised form November 5, 1973.

† Department of Applied Mathematics and Computer Science, University of Virginia, Charlottesville, Virginia 22901.

whose removal results in a disconnected graph $G - uv$. A *block* of a graph is a maximal, connected subgraph which has no cutpoints.

Finally, the *distance* $d(u, v)$ between two points u and v is the length of a shortest path between u and v; the *diameter* d of a graph G equals the maximum $d(u, v)$ for all points u, v in G.

2. Simple bounds on $h(G)$. Since no graph can be "better" than Hamiltonian and since any covering walk is also spanning, we can immediately establish the following bounds for any graph G with p points and q lines:

$$p \leqq h(G) \leqq e(G) \leqq 2q.$$

These bounds are best possible in the sense that there exist large classes of graphs where $h(G)$ actually attains either the lower or upper bound. For most graphs, however, the above bounds are very coarse.

Our first result shows that in order to find a Hamiltonian walk in a graph G, or to find $h(G)$ or bounds for $h(G)$, it is sufficient to solve these problems for certain subgraphs of G; the proof is obvious and is omitted.

THEOREM 1 (Cutpoint Theorem). *Let G be a connected graph having blocks $B_1, \cdots, B_k$. Then the union of the lines in Hamiltonian walks for each of the B_i forms a Hamiltonian walk for G, and conversely, the lines in any Hamiltonian walk of G form Hamiltonian walks for each of the blocks B_i.*

COROLLARY 1a. *Every bridge of a graph G appears twice in every Hamiltonian walk of G.*

COROLLARY 1b. *If G is a tree with p points, $h(G) = e(G) = 2q = 2(p - 1)$.*

COROLLARY 1c. *If $B_1, B_2, \cdots, B_k$ are the blocks of a graph G and $a_i \leqq h(B_i) \leqq b_i$ for $1 \leqq i \leqq k$, then $a_1 + \cdots + a_k \leqq h(G) \leqq b_1 + \cdots + b_k$.*

If a graph G is not a block, Corollary 1c permits a much better estimate of $h(G)$.

By limiting one's attention to special classes of graphs, it is often possible to get exact values for $h(G)$. One such result is the following, which completely determines $h(G)$ for complete n-partite graphs.

THEOREM 2. *Let $G = K_{m_1, m_2, \cdots, m_n}$ be a complete n-partite graph on $m_1 + m_2 + \cdots + m_n = p$ points, where $m_1 \leqq m_2 \leqq \cdots \leqq m_n$. Then*

(a) *G is Hamiltonian, i.e., $h(G) = p$ if and only if $m_1 + m_2 + \cdots + m_{n-1} \geqq m_n$.*

(b) *if $m_1 + m_2 + \cdots + m_{n-1} < m_n$, then $h(G) = 2m_n$.*

Proof. Let the points of $K_{m, m_2, \cdots, m_n}$ be partitioned into sets $M_1, M_2, \cdots, M_n$, where $|M_i| = m_i$. According to a Theorem of Dirac [5], a graph G with p points is Hamiltonian if every point of G has degree $\geqq p/2$. If we assume that $m_1 + m_2 + \cdots + m_{n-1} \geqq m_n$, then it follows that $m_1 + m_2 + \cdots + m_{n-1} \geqq p/2$. Therefore, every point in the set M_n has degree equal to $m_1 + m_2 + \cdots + m_{n-1} \geqq p/2$.

Furthermore, since $m_i \leqq m_n$, for $1 \leqq i \leqq n - 1$, it follows that every point in M_i has degree larger than or equal to the degree of any point in M_n, which in turn has degree $\geqq p/2$. Thus if $m_1 + m_2 + \cdots + m_{n-1} \geqq m_n$, then G is Hamiltonian, by virtue of Dirac's theorem.

If, on the other hand, $m_1 + m_2 + \cdots + m_{n-1} < m_n$, then G cannot be Hamiltonian, since any Hamiltonian cycle would have to enter the set M_n m_n times from the set $M_1 \cup M_2 \cup \cdots \cup M_{n-1}$. This cycle would also have to enter the set $M_1 \cup M_2 \cup \cdots \cup M_{n-1}$ exactly m_n times from the set M_n. But since

$m_1 + m_2 + \cdots + m_{n-1} < m_n$, it follows that this cycle would have to enter at least one point of $M_1 \cup M_2 \cup \cdots \cup M_{n-1}$ at least twice, a contradiction.

It is a relatively easy matter to construct a Hamiltonian cycle in $K_{m_1,m_2,\cdots,m_n}$ when $m_1 + m_2 + \cdots + m_{n-1} \geqq m_n$. One can show, furthermore, that

$$h(K_{m_1,m_2,\cdots,m_n}) = 2m_n$$

by observing that any Hamiltonian walk must enter and leave M_n at least m_n times each. Thus $h(G) \geqq 2m_n$. It is an easy matter to construct a Hamiltonian walk of length exactly $2m_n$.

A very desirable form of bound is one that would "float" up or down as a function of various easily computable parameters of a graph. One such bound is given in the next theorem.

THEOREM 3. *If G is n-connected and has diameter d, then*

$$h(G) \leqq 2p - [n/2](2d - 2) - 2,$$

where $[n/2]$ denotes the integer part of $n/2$.

The proof of Theorem 3 follows from Corollary 1b and the fact that G contains at least two points which are connected by n point disjoint paths, each of which is at least of length d.

As we have seen in Theorem 1, there is no loss of generality if we only consider graphs with $n \geqq 2$. For any block G with diameter d, we have $h(G) \leqq 2(p - d)$.

Finding lower bounds for $h(G)$ is much more difficult than finding upper bounds. If one is to conclude that $h(G)$ is strictly greater than p, then as part of this effort one has to show that G is not Hamiltonian. In general, this is very difficult to do. We present one general lower bound.

A *clique* of a graph is a maximal complete subgraph. A point is *unicliqual* if it lies on only one clique. We will need the following two lemmas.

LEMMA 4a. *A point v in a graph G is unicliqual if and only if the set of points adjacent to v induces a complete subgraph of G.*

LEMMA 4b. *If v is unicliqual in a graph G and $u \neq v$ is a point in G, then v is unicliqual in $G - u$.*

THEOREM 4. *Let U denote the set of unicliqual points in a graph G. Then $h(G - U) + |U| \leqq h(G)$.*

Proof. Let $W = v_1 v_2 \cdots v_m$ be a Hamiltonian walk in G and let

$$U = \{u_1, u_2, \cdots, u_k\}$$

be the set of unicliqual points in G. Consider the point $u_1 \in U$. Since W is a Hamiltonian walk, u_1 must appear somewhere in W. Consider the two points immediately preceding and following *any* occurrence of u_1 in W, say $v_i u_1 v_{i+2}$. Since by Lemma 4a we know that the set of points adjacent to u_1 in G forms a complete subgraph, we know that either $v_i v_{i+2}$ is a line of G or $v_i = v_{i+2}$.

Hence by deleting every occurrence of u_1 for which $v_i \neq v_{i+2}$, or by deleting u_1 and v_{i+2} if $v_i = v_{i+2}$, we will obtain a closed walk (call it W_1) in the graph $G - u_1 = G_1$ which contains every point of G_1.

By Lemma 4b we know that points $u_2, \cdots, u_k$ are unicliqual in G_1. Therefore by deleting in the same way every occurrence of u_2 in W_1 we will obtain a closed walk which contains every point of $G_1 - u_2 = G_2$.

By iterating this process we will obtain a series of closed walks $W_1, W_2, \cdots, W_k$ which contain every point of the graphs $G_1, G_2, \cdots, G_k$, respectively. Thus we will obtain a closed walk W_k, of length $\leqq h(G) - |U| = h(G) - k$, which contains every point of the graph $G_k = G - \{u_1, u_2, \cdots, u_k\} = G - U$. Therefore $h(G_k) = h(G - U) \leqq h(G) - |U|$.

COROLLARY 4a. *No unicliqual point ever appears more than once in a Hamiltonian walk.*

COROLLARY 4b. *Let U denote the set of unicliqual points in a graph G. Then $h(G - U) + |U| \leqq h(G) \leqq h(G - U) + 2|U|$.*

3. Relationships between Eulerian and Hamiltonian walks. While in general it is very difficult to find Hamiltonian walks in graphs, it is relatively easy to find Eulerian walks in graphs [2], [3]. In this section, we use the existing theory of Eulerian walks to establish several properties of Hamiltonian walks.

In [3] it was shown that no line of a graph G ever appears more than twice in any Eulerian walk in G; the same result is also true for Hamiltonian walks.

LEMMA 5a. *No line of a graph G ever appears more than twice in any Hamiltonian walk in G.*

Proof. Let W be a Hamiltonian walk in a graph G and let uv be a line which appears at least three times in G; then W must have one of the following three forms.

Case 1. $W = AuvBuvCuvD$, where A, B, C and D are subwalks (possibly empty) of W. In this case the walk $Au\overleftarrow{B}vCuvD$, where $\overleftarrow{B}$ denotes the reversal of the subwalk B, is a walk containing every point of G which is shorter than W and hence contradicts the minimality of W;

Case 2. $W = AuvBuvCvuD$. In this case the walk $Au\overleftarrow{B}uCvuD$ contradicts the minimality of W;

Case 3. $W = AuvBvuCuvD$. In this case the walk $AuCuvBvD$ contradicts the minimality of W.

Let W be a Hamiltonian walk in a graph $G = (V, E)$. Let G_W denote the multigraph induced by W; that is, $G_W = (V, E_W)$, where E_W consists of the lines in W. If a line occurs twice in E_W, then it occurs twice in the multigraph G_W. By Lemma 5a we know that no line occurs more than twice in G_W. Since W is a closed walk, it follows that G_W is an Eulerian multigraph.

From the above observations about the multigraph G_W, it is apparent that the problem of finding a Hamiltonian walk is equivalent to finding a connected, spanning, Eulerian multisubgraph of G having a minimum number of lines. By a multisubgraph of G we mean a multigraph, every line of which is a line of G.

If V is an Eulerian walk in a graph G, then the corresponding multigraph G_V will also be a connected, spanning, Eulerian multisubgraph of G. Consequently, if we can perform any transformation on G_V which will result in a smaller connected, spanning, Eulerian multisubgraph of G, say G'_V, then the number of lines in G'_V will be an upper bound for $h(G)$. In particular, if we can delete every line in a cycle of G_V and still have a connected multigraph, then the number of lines in the resulting graph will provide an upper bound for $h(G)$. This suggests the following two results.

LEMMA 5b. *No cycle of a graph G can have more than half of its lines appear twice in any Hamiltonian walk in G.*

Proof. Let G_W be the multigraph produced by a Hamiltonian walk W in G. By a previous observation, we note that G_W is a connected, spanning, Eulerian multisubgraph of G having a minimum number of lines. Let C be a cycle of G, more than half the lines of which appear twice in G_W. If we delete from G_W one of the two occurrences of each of these lines on C and add in their place a line to G_W for every remaining line on C, we can observe that the resulting multigraph will still be connected, spanning and Eulerian, and have fewer lines than G_W: a contradiction.

A cycle C in a connected graph G is a *cutting cycle* if the graph obtained from G by deleting every line of C is disconnected.

THEOREM 5. *In a connected graph G, if $h(G) = e(G)$, then every cycle of G is a cutting cycle.*

Proof. Let $h(G) = e(G)$, and assume that G contains a cycle C of length k, the removal of which does not disconnect G. Let G_V be a multigraph induced by an Eulerian walk V in G, i.e., G_V contains $e(G)$ lines. Since the cycle C is not a cutting cycle, it follows that the multigraph $G_V - C$ obtained from G_V by deleting every line of C is a connected, spanning, Eulerian multisubgraph of G which has fewer lines than G_V. Hence $h(G) \leqq e(G) - k < e(G)$, a contradiction. Thus every cycle of G must be a cutting cycle.

An improvement in Theorem 5 can be obtained by noting the following. Let G be a connected, Eulerian multigraph. Let C be a cycle in G and let the lines of C be partitioned into two subsets C_1 and C_2 such that $G - C_1$ is a connected graph. If we remove from G all the lines in C_1 and add one multiline for every line in C_2, the resulting multigraph, denoted by $G - C_1 + C_2$, will still be connected and Eulerian. Furthermore, if $|C_1| > |C_2|$, then $G - C_1 + C_2$ will have fewer lines than G. This essentially proves our next result.

THEOREM 6. *If a graph G contains a cycle, more than half the lines of which can be removed without disconnecting G, then $h(G) < e(G)$.*

The theta-graph $K_{2,3}$ shows that the converse to Theorem 6 is false. Our next result shows that the converse to Theorem 6 does hold for Eulerian graphs.

THEOREM 7. *If a graph G is Eulerian and $h(G) < e(G)$, then G must contain a cycle of which more than half the lines can be removed without disconnecting G.*

Proof. Assume that $h(G) < e(G)$. Let W be a Hamiltonian walk in G and let G_W be the Eulerian multigraph induced by W. Color in red those lines uv of G which do not appear in W. Color in blue those lines uv of G which appear twice in W. Let G' be the subgraph of G which consists of these colored lines. Since both G and G_W are Eulerian, it follows that every connected component of G' is Eulerian. Furthermore, since $h(G) < e(G)$, it follows that at least one component, say H, of G' has more red than blue lines.

Finally, since every Eulerian graph can be expressed as a line-disjoint union of cycles, it follows that in the Eulerian graph H, which is a subgraph of G' and of G, there must exist at least one cycle containing more red than blue lines.

If we were to remove these red lines from the graph G, the resulting graph would have to be connected since there would still remain in G all the blue lines in W, which serve to connect G. Thus G contains a cycle of which more than half the lines can be removed without disconnecting G.

COROLLARY 7a. *If a graph is Eulerian, then $h(G) < e(G)$ if and only if G contains a cycle of which more than half the lines can be removed without disconnecting G.*

Our next result shows that it is always possible to start with an Eulerian walk V in G and by a sequence of transformations of the cycles of the induced multigraph G_V produce a Hamiltonian walk in G.

By *cycle splitting* in a multigraph, we mean the process of partitioning the lines of a cycle of G into two sets, say C_1 and C_2, where $|C_1| \geqq |C_2|$, deleting from G every line in C_1 and adding one additional multiline for each line in C_2. Note that it is possible for $|C_2| = 0$.

THEOREM 8. *Let V and W be any Eulerian and Hamiltonian, respectively, walks in a graph G, and let G_V and G_W be the corresponding induced multigraphs. Then G_V can be transformed into G_W by a sequence of cycle splittings.*

Proof. Assume that the lines of the multigraphs G_V and G_W have been colored red and blue, respectively. Consider the multigraph $G_V + G_W$ obtained by superimposing these two multigraphs. Between any two adjacent points of the multigraph $G_V + G_W$ there will appear a set of multilines of one of the forms appearing in Fig. 1, where solid lines denote red lines and dashed lines denote blue lines.

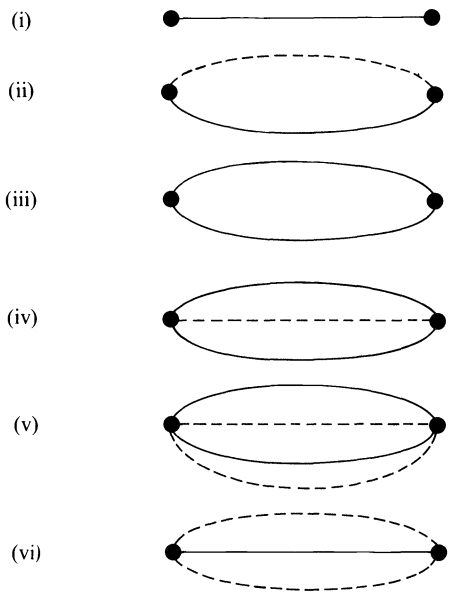


FIG. 1

We will now describe a sequence of transformations of the multigraph $G_V + G_W$. Each such transformation will have associated with it a corresponding transformation of the graph G_V. At the end of this sequence of transformations we will have deleted every line of $G_V + G_W$, leaving it empty; the corresponding sequence of transformations on G_V will produce the multigraph G_W.

Every set of multilines having forms (ii) or (v) in Fig. 1 will be deleted from $G_V + G_W$. The corresponding action in G_V is to leave the corresponding real lines alone.

Multilines of the form (iii) are also to be deleted from $G_V + G_W$. In G_V these same lines will be deleted; this will correspond to a cycle splitting in G_V where $|C_1| = 2, |C_2| = 0$.

Every line of the form (i) will ultimately be deleted from $G_V + G_W$. The corresponding action in G_V will be to delete this line as a part of a cycle splitting. Multilines of the form (iv) in $G_V + G_W$ will ultimately be transformed into form (ii), i.e., one of the two red lines will be deleted; the corresponding action in G_V will also be to delete such a multiline as part of a cycle splitting.

Multilines of the form (vi) in $G_V + G_W$ will ultimately have an additional red line added to them as part of a cycle splitting. The corresponding action in G_V will be to add a multiline as part of a cycle splitting.

Intuitively, the purpose of these transformations is not necessarily to remove every line from $G_V + G_W$, rather to transform $G_V + G_W$ into a multigraph which has only multilines of the forms (ii) or (v). Having done this, the multigraph G_V will have been transformed into the multigraph G_W.

We now proceed to transform $G_V + G_W$.

Step 1. Remove all multilines of forms (ii), (iii) or (v), leaving only multilines of forms (i), (iv) or (vi). Denote the multigraph which results from Step 1 by G'. Note that every component of G' is an Eulerian multigraph.

Step 2. Let G'' be any component of G'. Since G'' is Eulerian, there must exist a cycle of multilines in G''.

If every multiline on C is of the form (i) or (iv), then perform a cycle splitting of red lines on C, where $C_1 = C, |C_2| = 0$. In this process, multilines of the form (i) will be deleted, while multilines of the form (iv) will be transformed into form (ii). Repeat Step 1.

If C has multilines of the form (vi), then perform a cycle splitting of red lines. That is, remove multilines of the form (i), delete one of the two red lines in multilines of the form (iv) and add one red line to all multilines of the form (vi). Repeat Step 1.

The process of repeating Steps 1 and 2 will terminate when and only when all lines of $G_V + G_W$ have been removed. The corresponding actions in G_V will produce the multigraph G_W.

Unfortunately, the proof of Theorem 8 is existential rather than constructive. It resembles a theorem due to Tutte (cf. [4, p. 46]) which proves that any 3-connected graph can be generated from a wheel by a finite sequence of two transformations. Tutte's theorem is also nonconstructive.

Theorem 8 suggests the following procedure for obtaining relatively good upper bounds for $h(G)$, for an arbitrary graph G. By a *half cycle* of a graph G we mean a cycle of which more than half the lines, possibly all, can be removed without disconnecting G.

Step 1. Find an Eulerian walk V in G and construct the induced multigraph G_V.

Step 2. Locate any half cycle C in G_V.

Step 3. Perform a cycle splitting on C in which we remove from G_V as many lines of C as possible, without disconnecting G_V, and we add multilines on the remaining lines of C. Repeat Step 2.

This procedure stops when a graph is produced which has no half cycles. After each cycle splitting we have a new connected multigraph with fewer lines than the preceding multigraph. Any Eulerian walk on this multigraph spans the original graph, and so gives an upper bound to $h(G)$. The ith successive iteration of the algorithm produces a new multigraph G_i with $h(G) \leqq e(G_i) < e(G_{i-1})$.

Theorem 8 proves that there is at least one sequence of G_i such that the sequence $e(G_i)$ converges to $h(G)$. However, not all sequences of half cycles will give $e(G_i)$ which converge to $h(G)$. The process of trying out all possible sequences and choosing the minimum is undoubtedly exponential. In practice, however, it should not take too much effort to make the algorithm yield a reasonably good upper bound.

Acknowledgment. The authors would like to thank Shen Lin for his helpful comments which led us to the realization of Corollaries 4a and 4b and to a strengthening of Theorem 8.

REFERENCES

[1] MEI-KO KWAN, *Graphic programming using odd or even points*, Chinese Math. Acta, 1 (1962), pp. 273–277.

[2] J. EDMONDS AND E. L. JOHNSON, *Matching, Euler tours and the Chinese postman*, Math. Programming, 5 (1973), pp. 88–124.

[3] S. GOODMAN AND S. HEDETNIEMI, *Eulerian walks in graphs*, this Journal, 2 (1973), pp. 16–27.

[4] F. HARARY, *Graph Theory*, Addison-Wesley, Reading, Mass., 1969.

[5] G. A. DIRAC, *Some theorems on abstract graphs*, Proc. London Math. Soc. Ser. 3, 2 (1951), pp. 69–81.

SIAM J. COMPUT.
Vol. 3, No. 3, September 1974

A NOTE ON THE HECHT-ULLMAN CHARACTERIZATION OF NONREDUCIBLE FLOW GRAPHS*

J. M. ADAMS, J. M. PHELAN AND R. H. STARK†

Abstract. An anomaly in the proof of a theorem characterizing nonreducible flow graphs is pointed out. A proof is given which corrects for the anomaly and is fundamentally simpler than the original proof.

Key words. flow graph, reducibility

Hecht and Ullman [1] have recently given an interesting structural characterization of nonreducible flow graphs as follows.

THEOREM. *If a flow graph is nonreducible, then it has a subgraph of the form* (*) *shown in Fig.* 1.

The wavy lines in Fig. 1 denote node-disjoint paths.

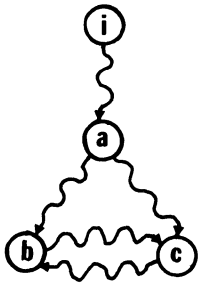


FIG. 1. *The graph* (*)

We submit a proof of the characterization theorem which is fundamentally simpler than that in [1] and corrects for an anomaly in the proof.[1]

In the proof in [1], one assumes the existence of a depth first spanning tree (DFST) with a loop on its spine (right-most branch). Such a loop is created by an edge in the graph but not the DFST from a descendant to an ancestor. However, there exist nonreducible graphs which have DFST's with no loops on their spines as shown in Fig. 2.

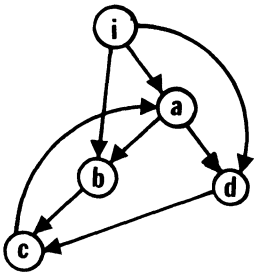


FIG. 2

* Received by the editors July 5, 1973.

† Department of Computer Science, New Mexico State University, Las Cruces, New Mexico 88003.

[1] The anomaly was observed independently by John Beatty of the University of California at Berkeley.

Proof of characterization theorem. We induct on n the number of nodes in $G = (N, E, i)$, the flow graph. As observed in [1], the case $n = 3$ follows by considering all possible three-node flow graphs.

Inductive hypothesis. Any nonreducible flow graph with from three to n nodes has a subgraph of the form (*).

Let $G = (N, E, i)$ be a nonreducible flow graph with $n + 1$ nodes.

As in [1], we assume without loss of generality that T_1 and T_2 are not applicable to G.

Let $T = (N, E')$ be a DFST for G. The node i is the root of T, and we let r denote the right-most successor of i. Since T_1 and T_2 do not apply to G, there exists an edge (k, r) in $E - E'$ with $k \neq r$, and by [1, Lemma 2] there is a path, Q, in T from r to k. Consider the subgraph H_0 of all nodes along the path Q from r to k and all edges in G between these nodes.

Let H be the subgraph consisting of (i) the nodes in H_0 plus the nodes having a path in G to k which does not pass through r, and (ii) all the edges between these nodes.

If i is in H, then there is a path P from i to k that avoids r. In this case let i, a, b and c of (*) be, respectively i, i, r and the first node on P in H_0.

Now assume i is not in H. No edge enters H from outside except those which enter r, thus H with root r is irreducible, since G is. Since H is a proper subgraph of G, it has (*) by induction. Thus G also has (*).

REFERENCE

[1] M. S. HECHT AND J. D. ULLMAN, *Flow graph reducibility*, this Journal, 1 (1972), pp. 188–202.

SIAM J. COMPUT.
Vol. 3, No. 3, September 1974

ON BOUNDED RIGHT CONTEXT LANGUAGES AND GRAMMARS*

SUSAN L. GRAHAM†

Abstract. It is shown using phrase-structure-preserving grammatical transformations that the family of (1, 1) bounded right context languages is the same as the family of context-free deterministic languages. The elimination of λ-rules and a reconciliation of our definition of bounded right context with Floyd's are also discussed.

Key words. deterministic languages, LR(k), parsing, bounded context, bounded right context, context-free grammars, grammatical transformations

1. Introduction. The class of context-free grammars called bounded right context was introduced by Floyd [2]. These grammars have the property that they are unambiguous, that parsing of sentences in such languages can be done left-to-right without backup in linear time and that each parsing decision can be made locally, that is, on the basis of a bounded number of characters. Furthermore, the class includes grammars amenable to many of the efficient parsing techniques used at that time or discovered subsequently, in particular, methods in which analysis algorithms can be mechanically generated from the grammars. Thus there is a concern both for generality and for efficiency. Floyd commented [2]:

> The procedure for bounded context[1] analysis described here for nontrivial languages and contexts of more than one character, may make unreasonable demands on computer time and storage space. It seems unlikely to supplant such efficient specialized techniques as those based on operator precedence; it may serve, however, to illuminate the relationship among existing efficient techniques and to suggest their appropriate generalizations.

A further advance in the direction of unification and generality without extreme loss of efficiency was provided by the LR(k) grammars defined by Knuth [7]. He showed that these grammars generate all of the deterministic context-free languages, yet are "sufficiently simple that an efficient left-to-right parsing algorithm could be mechanically generated." It is easily shown that the family of LR(k) grammars properly contains the family of bounded right context grammars.

In this paper we investigate the language and structural properties of bounded right context grammars. We show that the (1, 1) bounded right context grammars generate all of the deterministic context-free languages. That is, it suffices to use contexts of one character for these methods of syntactic analysis. Moreover, we obtain this result primarily by means of grammatical transformations which can be made on the full class of grammars and which preserve the phrase structure of the generated language. We also examine the role of λ-rules in these grammars and consider the use of a definition of bounded right context which includes such rules.

---

* Received by the editors December 14, 1972, and in final revised form February 13, 1974.

† Department of Computer Science, University of California at Berkeley, Berkeley, California 94720. This work was supported by the National Science Foundation under Grants GJ-474 and GJ-43318. An extended abstract including this work was presented at the IEEE 11th Annual Symposium on Switching and Automata Theory, 1970, and appears in the Conference Record.

[1] Bounded context is a subclass of bounded right context in which the left-to-right assumption is not made.

In § 2 of the paper, we give our definitions and notation. Section 3 contains structure-preserving transformations to take LR(k) and (m, k) BRC grammars into $(1, k)$ BRC grammars. It is from these transformations that we get our characterization of the deterministic context-free languages. In § 4, we investigate the use of λ-rules and show that they can be eliminated from LR(k) and (m, n) BRC grammars. In § 5, we demonstrate that our definition of bounded right context is consistent with that of Floyd. Section 6 contains some concluding remarks.

This paper is one of three based on [3]. In the subsequent papers [4], [5], the transformations and results presented here are used to develop further results about bounded right context and precedence languages and grammars.

2. Definitions and notation. An *alphabet* or *vocabulary* is a finite set of symbols (designated by a possibly primed, subscripted or superscripted upper-case italic letter). *Strings* are sequences of symbols from an alphabet. The *length* of a string x, designated $|x|$, is the number of occurrences of symbols in the string. The string of length zero is denoted by λ. If x is any string, then for any integer $k \geqq 0$,

$$\text{last}_k(x) = \begin{cases} x & \text{if } |x| \leqq k, \\ y & \text{if } x = zy \text{ and } |y| = k; \end{cases}$$

$$\text{first}_k(x) = \begin{cases} x & \text{if } |x| \leqq k, \\ y & \text{if } x = yz \text{ and } |y| = k. \end{cases}$$

For any string x and any k, $0 \leqq k \leqq |x|$, $\text{last}_k(x)$ is a *suffix* of x and $\text{first}_k(x)$ is a *prefix* of x. For any alphabet V, V^* denotes the set of all strings formed from symbols of V and $V^+ = V^* - \{\lambda\}$. For any alphabets V_1 and V_2, a *homomorphism* $h: V_1 \to V_2^*$ is a mapping of elements of V_1 to strings in V_2^*. The homomorphism is extended to strings in V_1^* by the rules that $h(\lambda) = \lambda$ and for every $a \in V_1, x \in V_1^*$, $h(xa) = h(x)h(a)$.

A (*context-free*) *grammar* is a 4-tuple $G = (V, V_T, P, S)$, where $V_T \subseteq V$ is an alphabet of *terminal symbols*, $V_N = V - V_T$ is a nonempty alphabet of *nonterminal symbols*, P is a finite set of *rules* or *productions* $A \to a$, where $A \in V_N$ and $a \in V^*$,[2] and $S \in V_N$ is the *initial symbol*.

As usual, with respect to a grammar $G = (V, V_T, P, S)$ we define the relation $\Rightarrow$ on $V^* \times V^*$ such that for any $a \in V^*$, $b \in V^*$, $a \Rightarrow b$ if and only if there exist $U \in V_N$, σ, π, $u \in V^*$ and $U \to u$ in P such that $a = \sigma U \pi$ and $b = \sigma u \pi$. We represent by $\overset{+}{\Rightarrow}$ ($\overset{*}{\Rightarrow}$) the transitive closure (reflexive-transitive closure) of $\Rightarrow$. For any $n \geqq 0$, $a_i \in V^*$, where $0 \leqq i \leqq n$, we call the sequence $a_0 \Rightarrow a_1 \Rightarrow \cdots \Rightarrow a_n$ a *derivation of a_n from a_0 of length n in G*. We refer to the process of reconstructing a derivation, given a string of terminal symbols and a grammar, as *parsing*. If $a_0 \Rightarrow a_1 \Rightarrow \cdots \Rightarrow a_n$, where for $0 \leqq i < n$, $a_i = \sigma_i U_i \pi_i$ and $a_{i+1} = \sigma_i u_i \pi_i$ for some $\sigma_i \in V^*$, $\pi_i \in V_T^*$, and $U_i \to u_i$ in P, the derivation is a *rightmost derivation*. A string u is a *sentential form* if $S \overset{*}{\Rightarrow} u$. The string u is a *rightmost sentential form* if there is a rightmost derivation such that $S \overset{*}{\Rightarrow} u$. The *language $L(G)$ defined (or generated) by G* is the set of all sentential forms with no nonterminal symbols. Thus

[2] A rule $A \to a$ is called a *λ-rule* if $a = \lambda$.

$L(G) = \{u \in V_T^* | S \overset{+}{\Rightarrow} u\}$. Grammars G_1 and G_2 are said to be *equivalent* if they define the same language (that is, if $L(G_1) = L(G_2)$). Finally, a grammar $G = (V, V_T, P, S)$ is *reduced* either if P is the empty set or if for every $X \in V$, there exist $\alpha, \beta \in V^*$ such that $S \overset{*}{\Rightarrow} \alpha X \beta$ and there exists $t \in V_T^*$ such that $X \overset{+}{\Rightarrow} t$.

We next define the two families of grammars with which we are concerned. The LR(k) grammars were first defined by Knuth [7]. Our definition is essentially the same as his except for our added condition on derivations of the initial symbol.

DEFINITION. Let $G = (V, V_T, P, S)$ be a context free grammar such that there is no rightmost derivation $S \overset{+}{\Rightarrow} S$ in G.[3] Let \$ be a symbol not in V. G is an LR(k) *grammar* ($k \geqq 0$) if the following property holds for every rule $U \to u$ in P and all $\tau, \sigma \in V^*$; $X \in V_N$; $v_1, v_2, v, \omega \in V_T^*\{\$\}^*$ such that $|v| = k$.

If $S\$^k \overset{*}{\Rightarrow} \tau U v v_1 \Rightarrow \tau u v v_1$ and $S\$^k \overset{*}{\Rightarrow} \sigma X \omega \Rightarrow \tau u v v_2$ are rightmost derivations in G,[4] then $\tau = \sigma$, $\omega = v v_2$ and $X = U$.

Our family of bounded right context grammars (which we designate as BRC grammars) is larger than the family of bounded right context grammars defined by Floyd [2] (and designated here as FBRC grammars). We examine the relationship between the two definitions in § 5.

DEFINITION. Let $G = (V, V_T, P, S)$ be a context-free grammar such that there is no rightmost derivation $S \overset{+}{\Rightarrow} S$ in G.[5] Let $\not\subset$, \$ be symbols not in V. G is an (m, n) BRC *grammar* ($m, n \geqq 0$) if the following property holds for every rule $U \to u$ in P and all $\tau_1, \tau_2, \sigma, t \in \{\not\subset\}^* V^*$; $X \in V_N$; $v_1, v_2, v, \omega \in V_T^*\{\$\}^*$ such that $|t| = m$ and $|v| = n$.

If $\not\subset^m S\$^n \overset{*}{\Rightarrow} \tau_1 t U v v_1 \Rightarrow \tau_1 t u v v_1$ and $\not\subset^m S\$^n \overset{*}{\Rightarrow} \sigma X \omega \Rightarrow \tau_2 t u v v_2 = \sigma x \omega$ are rightmost derivations in G,[6] where $|\omega| \leqq |v v_2|$, then $\sigma = \tau_2 t$, $\omega = v v_2$ and $X = U$.

G is a BRC *grammar* if for some $m, n \geqq 0$, G is an (m, n) BRC grammar. We designate by *BRC* the class of languages generated by BRC grammars and by (m, n) *BRC* the class of languages generated by (m, n) BRC grammars.

The restriction on $|\omega|$ is a fundamental part of the definition, which serves to restrict consideration of u to occurrences wholly within σx. Its importance is illustrated by the following example:

Let $G = (\{S, U, A, Y\}, \{a, b, c, u_1, u_2, v\}, P, S)$, where

$$P: \quad S \to aAUv | cYu_2v$$

$$U \to u_1 u_2$$

$$A \to aA | a$$

$$Y \to bAu_1.$$

---

[3] If such derivations are not excluded, the grammar may be ambiguous even if it satisfies the subsequent property.

[4] Without loss of generality, we follow every sentential form in a derivation by the same string taken from a disjoint vocabulary. (Additionally, in derivations for BRC grammars, we precede every sentential form by the same string.) This is done in order to insure that we always have strings of the designated lengths and thereby to simplify notation.

[5] See footnote 3.

[6] See footnote 4.

P has rightmost derivations

$$S \Rightarrow aAUv \Rightarrow aA\underline{u_1u_2}v \tag{2.1}$$

$$S \overset{*}{\Rightarrow} aAu_1u_2v \Rightarrow a\underline{aA}u_1u_2v \tag{2.2}$$

$$S \Rightarrow cYu_2v \Rightarrow \underline{cbAu_1}u_2v, \tag{2.3}$$

where the underlined substrings are right parts of the rules used in the last step of each derivation. If we use the definition as stated, then G is a (1, 1) BRC grammar. However, without the restriction, derivations (2.1) and (2.2) would violate the (1, 1) BRC property, even though (2.2) only extends (2.1) by one step. Also, derivations (2.1) and (2.3) would violate the (1, 1) BRC property.

In Floyd's paper, the bounded right context grammars are introduced by first defining and motivating the class of bounded context grammars. The two classes are not the same. One obtains a definition of Floyd's class of bounded context grammars by removing the restriction to rightmost derivations throughout the FBRC definition in § 5. It can be shown that the bounded context grammars are a proper subclass of the bounded right context grammars and that proper inclusion holds for the corresponding language classes as well.

3. (1, 1) *BRC* and the context-free deterministic languages. In this section, we present transformations to construct for any (m, n) BRC grammar an equivalent $(1, n)$ BRC grammar and for any LR(k) grammar an equivalent $(1, k)$ BRC grammar. Using these results and theorems of Knuth [7], we obtain our characterization of the deterministic languages. Our primary concern in choosing these transformations is that they preserve the "phrase structure" of the language as much as possible. We are also interested in achieving minimal differences in size between the vocabularies and the production sets of the original and transformed grammars. Consequently the transformations we present are more complex than they need be if our interest were only in the equivalence results.

We present a general transformation in Theorem 3.1. Most of the subsequent transformations are special cases of this one. The idea of the transformation is that in the rightmost sentential forms of the transformed grammar, we encode in certain nonterminal symbols information about the symbols to their left. The encoding is determined by an encoding function g, which is a function of a symbol of the original vocabulary and the new symbol which will occur to its left. The encoding function must satisfy three conditions. Condition (g1) says that every X in the original vocabulary is replaced either by itself or by a bracketed symbol $[\mathscr{W}_i, X]$. Conditions (g2) and (g3) are consistency conditions which insure that if the rules of the grammar are transformed using the encoding function, the encoding extends to rightmost sentential forms. That is, every symbol in a rightmost sentential form of the new grammar is a function of the symbol to its left and the symbol to which it corresponds in the original vocabulary.

More precisely, we have the following.

DEFINITION. Let $G = (V, V_T, P, S)$ be a context-free grammar. Let

$$\mathscr{W} = \{\mathscr{W}_i | 0 \leqq i \leqq s, s \geqq 0\}$$

be a set of distinct objects. Then $V_{[]} = \{[\mathscr{W}_i, X] | \mathscr{W}_i \in \mathscr{W}, 0 \leqq i \leqq s \text{ and } X \in V\}$ is

the *new nonterminal alphabet determined by* V *and* $\mathscr{W}$. A function $g:(V \cup V_{[]}) \times V \to V \cup V_{[]}$ is an *encoding function* if

(g1) for every X in V and every Y in $V \cup V_{[]}$,

$$g(Y, X) \in \{X\} \cup \{[\mathscr{W}_i, X], 0 \leqq i \leqq s\};$$

(g2) for every X in V and every Y in $V \cup V_{[]}$, if $g(Y, X) = [\mathscr{W}_i, X]$ for some i, $0 \leqq i \leqq s$, then for each U in V such that $U \overset{*}{\Rightarrow} X\sigma$, where $\sigma \in V^*$, it is also the case that $g(Y, U) = [\mathscr{W}_i, U]$;

(g3) for every U in V and every Y in $V \cup V_{[]}$, if $g(Y, U) = [\mathscr{W}_i, U]$ for some i, $0 \leqq i \leqq s$, then for each X in V such that $U \overset{*}{\Rightarrow} X\sigma$, where $\sigma \in V^*$, either $g(Y, X) = [\mathscr{W}_i, X]$ or for every Z in $V \cup V_{[]}$, $g(Z, X) = X$. In the latter case, X is said to be *fixed under* g.

Example. Given a context-free grammar $G = (V, V_T, P, S)$, the simplest encoding function would be that for every X and Y in V, $g(Y, X) = X$. (In this case, $\mathscr{W}$ and $V_{[]}$ are empty and all of V is fixed under g.) Almost as simple is to let $\mathscr{W} = V$ and $V_{[]} = \{[Y, X] | Y, X \in V\}$ and to define g by:

$$\text{for every } X \text{ and } Y \text{ in } V \text{ and every } [Z, Y] \text{ in } V_{[]},$$

$$g(Y, X) = g([Z, Y], X) = [Y, X].$$

In this second definition, encoded within the second argument of g is the counterpart of the first argument in the original vocabulary.

The transformation is carried out left-to-right on the right parts of each rule of the original grammar and is determined by replacing each symbol after the first by the encoding function of the symbol and the new symbol to its left. (As a consequence of condition (g2), the leftmost symbol is unchanged.) If a terminal symbol x is replaced by a nonterminal symbol $[\mathscr{W}_i, x]$, we add a rule $[\mathscr{W}_i, x] \to x$ to generate the terminal. Additionally for every rule having a left part which is not fixed under g, we add new rules in which the left part is replaced by a corresponding bracketed symbol and the leftmost symbol of the right part, if it is not fixed under g, is replaced by a bracketed symbol containing the same encoding information (thereby insuring that condition (g3) will extend to rightmost derivations). A bracketed symbol generates the same set of terminal strings in the transformed grammar as its second component generates (in either grammar). There is a one-to-one correspondence between rightmost derivations in the original grammar and rightmost derivations in the transformed grammar, and between derivation steps in the original grammar and sequences of derivation steps in the transformed grammar. (The extra steps in the transformed grammar come from the use of rules $[\mathscr{W}_i, x] \to x, x \in V_T$ to "erase" the context information.) Details of the transformation are contained within the theorem, which is followed by an example.

THEOREM 3.1. *Let* $G = (V, V_T, P, S)$ *be an* (m, n) BRC *grammar* $(m > 0)$. *Let* $\mathscr{W} = \{\mathscr{W}_i | 0 \leqq i \leqq s, s \geqq 0\}$ *be a set of distinct objects, let* $V_{[]}$ *be the new nonterminal alphabet determined by* V *and* $\mathscr{W}$, *and let* $g:(V \cup V_{[]}) \times V \to V \cup V_{[]}$ *be an encoding function. Then the following transformation yields an equivalent* (m, n) BRC *grammar* $G' = (V', V_T, P', S)$.

Transformation. Let $V' = \{Z \in V \cup V_{[\,]} \mid$ there is some X in V and some Y in $V \cup V_{[\,]}$ such that $g(Y, X) = Z\} \cup V$. Construct P' as follows:

(3.1) For each rule $X \to X_1 X_2 \cdots X_k$ in P, $k \geqq 0$.

(3.1a) If $k = 0$, then put $X \to \lambda$ in P'. Otherwise put $X \to X'_1 X'_2 \cdots X'_k$ in P', where $X'_1 = X_1$ and for $2 \leqq j \leqq k$, $X'_j = g(X'_{j-1}, X_j)$.

(3.1b) For every $[\mathscr{W}_i, X]$ in $V' - V$, $0 \leqq i \leqq s$, if $k = 0$, then put $[\mathscr{W}_i, X] \to \lambda$ in P'. Otherwise put $[\mathscr{W}_i, X] \to X''_1 X''_2 \cdots X''_k$ in P', where

$$X''_1 = \begin{cases} X_1 & \text{if } X_1 \text{ is fixed under } g, \\ [\mathscr{W}_i, X_1] & \text{otherwise}, \end{cases}$$

and for $2 \leqq j \leqq k$, $X''_j = g(X''_{j-1}, X_j)$.

(3.2) For each $x \in V_T$, for every $[\mathscr{W}_i, x]$ in $V' - V$, $0 \leqq i \leqq s$, put $[\mathscr{W}_i, x] \to x$ in P'.

Proof. The transformation replaces every $X \in V$ in every rule either by itself or by a bracketed symbol $[\mathscr{W}_i, X]$ (because of condition (g1)), where the replacement is a function of the preceding symbol and the $\mathscr{W}_i$ carries information about the symbol or symbols to the left of X. As shown by Fact 1, step (3.1b) of the transformation (together with conditions (g2) and (g3)) extends this encoding of left context to rightmost derivations. As shown by Fact 2, every rightmost derivation of G' contains this context information.

FACT 1. *For every* $A \in V$, $z \in V_T^*$, $x \in V^*$, *where* $x = X_1 X_2 \cdots X_k$, $k \geqq 0$, *and for* $1 \leqq i \leqq k$, $X_i \in V$, *if there is a rightmost derivation* $A \overset{*}{\Rightarrow} xz$ *in* G, *then*

(a) *there is a rightmost derivation* $A \overset{*}{\Rightarrow} X'_1 X'_2 \cdots X'_k z$ *in* G' *of greater or equal length, where* $X'_1 = X_1$ *and for* $2 \leqq j \leqq k$, $X'_j = g(X'_{j-1}, X_j)$;

(b) *for every* $[\mathscr{W}_i, A] \in V' - V$, $0 \leqq i \leqq s$, *there are rightmost derivations* $[\mathscr{W}_i, A] \overset{*}{\Rightarrow} X''_1 X''_2 \cdots X''_k z$ *in* G', *where*

$$X''_1 = \begin{cases} X_1 & \text{if } X_1 \text{ is fixed under } g, \\ [\mathscr{W}_i, X_1] & \text{otherwise}, \end{cases}$$

and for $2 \leqq j \leqq k$, $X''_j = g(X''_{j-1}, X_j)$.

Proof. The proof is done by induction on the length of a derivation in G. Define a bracket-and-context-erasing homomorphism h by:

$$\text{for each } [\mathscr{W}_i, X] \in V' - V,\ 0 \leqq i \leqq s,\ h([\mathscr{W}_i, X]) = X;$$

$$\text{for each } X \in V,\ h(X) = X.$$

FACT 2. *For every* $A \in V'$, $z \in V_T^*$, $x \in (V')^*$, *where* $x = X_1 X_2 \cdots X_k$, $k \geqq 0$, *and for* $1 \leqq i \leqq k$, $X_i \in V'$, *if there is a rightmost derivation* $A \overset{*}{\Rightarrow} xz$ *in* G', *then*

(a) *there is a rightmost derivation* $h(A) \overset{*}{\Rightarrow} Y_1 Y_2 \cdots Y_k z$ *in* G, *where, for* $1 \leqq i \leqq k$, $Y_i = h(X_i)$;

(b) $x = Y''_1 Y''_2 \cdots Y''_k$, *where*

$$Y''_1 = \begin{cases} [\mathscr{W}_i, Y_1] & \text{if } A = [\mathscr{W}_i, U] \text{ for some } \mathscr{W}_i \in \mathscr{W},\ U \in V \\ & \text{and } Y_1 \text{ is not fixed under } g, \\ Y_1 & \text{otherwise}, \end{cases}$$

and for $2 \leqq j \leqq k$, $Y''_j = g(Y''_{j-1}, Y_j)$.

Proof. The proof proceeds by induction on the length of a derivation in G'.

It follows from Fact 1 and (a) of Fact 2 that $L(G) = L(G')$. Suppose G' is not an (m, n) BRC grammar. Since there is no rightmost derivation $S \overset{+}{\Rightarrow} S$ in G (because G is an (m, n) BRC grammar), it follows from Fact 1 that there is no rightmost derivation $S \overset{+}{\Rightarrow} S$ in G'. Therefore for some X, $U \in V' - V_T$; $u, x \in (V')^*$; $\tau_1, \tau_2, t, \sigma \in \{\not\epsilon\}^*(V')^*$; $v_1, v_2, v, \omega \in V_T^*\{\$\}^*$ such that $|t| = m$ and $|v| = n$, there are rightmost derivations

$$\not\epsilon^m S\$^n \overset{+}{\Rightarrow} \tau_1 tUvv_1 \Rightarrow \tau_1 tuvv_1 \tag{3.3}$$

and

$$\not\epsilon^m S\$^n \overset{+}{\Rightarrow} \sigma X\omega \Rightarrow \sigma x\omega = \tau_2 tuvv_2, \tag{3.4}$$

where $|\omega| \leqq |vv_2|$ and either $\sigma \neq \tau_2 t$ or $\omega \neq vv_2$ or $X \neq U$.

Case 1. $h(U)$, $h(X) \in V_N$. By Fact 2, there are corresponding rightmost derivations

$$\not\epsilon^m S\$^n \overset{+}{\Rightarrow} h(\tau_1 tU)vv_1 \Rightarrow h(\tau_1 tu)vv_1$$

and

$$\not\epsilon^m S\$^n \overset{+}{\Rightarrow} h(\sigma X)\omega \Rightarrow h(\sigma x)\omega = h(\tau_2 tu)vv_2$$

in G, where $|\omega| \leqq |vv_2|$. (Notice that h is length-preserving.) Since G is an (m, n) BRC grammar, it cannot be the case that $h(\sigma) \neq h(\tau_2 t)$ or $\omega \neq vv_2$ or $h(X) \neq h(U)$. Suppose $h(\sigma) = h(\tau_2 t)$, $\omega = vv_2$, and $h(X) = h(U)$. Then $|\sigma| = |\tau_2 t|$ and therefore $\sigma = \tau_2 t$ (since σ is a prefix of $\tau_2 tuvv_2$). Thus (3.4) has the form $\not\epsilon^m S\$^n \overset{+}{\Rightarrow} \tau_2 tXvv_2 \Rightarrow \tau_2 tuvv_2$. Since $h(X) = h(U)$, it follows from Fact 2 that $X = U$. Therefore this case can arise only if G is not an (m, n) BRC grammar.

Case 2. $h(U) \in V_N$, $h(X) \in V_T$. Since (3.4) is a rightmost derivation, it must be of the form

$$\not\epsilon^m S\$^n \overset{+}{\Rightarrow} \sigma' X'\omega' \Rightarrow \sigma' x'\omega' \overset{+}{\Rightarrow} \sigma X\omega \Rightarrow \sigma x\omega = \tau_2 tuvv_2, \tag{3.5}$$

where $|\omega'| \leqq |\omega|$, $h(X') \in V_N$, and all steps in the derivation after $\sigma' X'\omega' \Rightarrow \sigma' x'\omega'$ use rules created by step (3.2) of the transformation. It follows that $h(\sigma' x'\omega') = h(\tau_2 tuvv_2)$. The argument of Case 1 holds for (3.3) and (3.5).

Case 3. $h(U) \in V_T$. Let T be the rightmost symbol of t. It follows from Fact 2 and the form of (3.3) that $U = g(T, h(U))$. Since $h(U) \in V_T$ and $U \to u$ is in P', it is a consequence of step (3.2) of the transformation that $h(U) = u$ and therefore that $u \in V_T$ and $U = g(T, u)$. Since $U \in V' - V_T$ and, from condition (g1), $g(T, u) \in V' - V_N$, it must be the case that $g(T, u) \in V' - V$. Therefore no right part of a rule in P' can contain T followed by u. It follows that $|\sigma| \geqq |t_2 t|$. Furthermore, it follows from Fact 2 that u cannot occur in a rightmost sentential form of G' immediately to the right of T, but to the left of the rightmost nonterminal. Therefore $\sigma = \tau_2 t$, $x = uv'$ for some $v' \in V_T^*$, $j \geqq 0$, such that $v' = \text{first}_j(vv_2)$, and $X \to uv'$ is in P'. If this rule is in P' by virtue of step (3.1) of the transformation, then P contains a rule $h(X) \to uv'$. If so, then since u is not fixed under g (because $g(T, u) \in V' - V$), it follows from condition (g2) that $g(T, h(X)) \in V' - V$ and from step (3.1) and the fact that $u \in V_T$ that $X \in V_N$. But by Fact 2, $X = g(T, h(X))$, contradicting this possibility.

Therefore $X \to uv'$ can only be in P' by virtue of step (3.2) of the transformation. Consequently $v' = \lambda$ and $h(X) = u$. It follows that $w = vv_2$ and

$$X = g(T, h(X)) = g(T, u) = U.$$

Therefore this case cannot occur.

Since none of the cases in which G' might fail to be an (m, n) BRC grammar can occur, G' must be an (m, n) BRC grammar. □

COROLLARY 3.2. *Let $G = (V, V_T, P, S)$ be an* LR(k) *grammar. Let the transformation and the associated sets be as in Theorem* 3.1. *Then the transformation yields an equivalent* LR(k) *grammar $G' = (V', V_T, P', S)$.*

Proof. Let $\tau_1 = \tau_2$ and drop the restriction on $|\omega|$ in the proof of the theorem. □

Example. The following grammar $G = (V, V_T, P, S)$ will serve as an example for § 3.

$$V_N = \{S, X, Y, Z, U_1, U_2\}, \quad V_T = \{a, b, c, y, z, u, t\},$$

$$\begin{aligned} P: \quad & S \to aaX|bbX|bbY|ccY|ccZ \\ & X \to tU_1y \\ & Y \to tU_1z \\ & Z \to tU_2y \\ & U_1 \to u \\ & U_2 \to u. \end{aligned}$$

Suppose we use the simple form of encoding that a bracketed symbol contains the symbol to its left in the corresponding string of the original vocabulary. Then $\mathscr{W} = V$ and $V_{[]} = \{[X, Y] | X, Y \in V\}$. We can define g by:

$$\text{for every } X, Y \in V, [Z, X] \in V_{[]},$$

$$g(X, Y) = g([Z, X], Y) = [X, Y].$$

Clearly g satisfies the conditions of the theorem. The reduced transformed grammar has rules P':

(from step (3.1a))

$$S \to a[a, a][a, X] \mid b[b, b][b, X] \mid b[b, b][b, Y] \mid c[c, c][c, Y] \mid c[c, c][c, Z]$$

(from step (3.1b))

$$\begin{aligned} & [a, X] \to [a, t][t, U_1][U_1, y] \\ & [b, X] \to [b, t][t, U_1][U_1, y] \\ & [b, Y] \to [b, t][t, U_1][U_1, z] \\ & [c, Y] \to [c, t][t, U_1][U_1, z] \\ & [c, Z] \to [c, t][t, U_2][U_2, y] \\ & [t, U_1] \to [t, u] \\ & [t, U_2] \to [t, u] \end{aligned}$$

(from step (3.2))

$$[a, a] \to a$$

$$[b, b] \to b$$

$$[c, c] \to c$$

$$[U_1, y] \to y$$

$$[U_2, y] \to y$$

$$[U_1, z] \to z$$

$$[t, u] \to u$$

$$[a, t] \to t$$

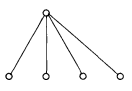
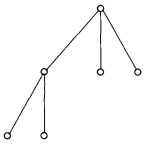
$$[b, t] \to t$$

$$[c, t] \to t.$$

The rightmost derivation $S \Rightarrow bbX \Rightarrow bbtU_1y \Rightarrow bbtuy$ of G becomes

$$S \Rightarrow b[b, b][b, X] \Rightarrow b[b, b][b, t][t, U_1][U_1, y] \Rightarrow b[b, b][b, t][t, U_1]y$$
$$\Rightarrow b[b, b][b, t][t, u]y \Rightarrow b[b, b][b, t]uy \Rightarrow b[b, b]tuy \Rightarrow bbtuy$$

in G'.

The transformation of Theorem 3.1 preserves the "phrase structure" of the generated language in a very strong sense. Given the syntax tree of any sentence generated by an (m, n) BRC grammar G (under the usual correspondence between derivations and trees), the syntax tree for that sentence with respect to the transformed grammar G' will be the same up to the relabeling of some nonterminal nodes in a well-defined way (namely, the first tree may have a node with label Z and the second tree have label $[\mathcal{W}_i, Z]$) and the replacement of some terminal nodes by nonterminals with one terminal successor. (These notions and terminology could be made more precise, but the ideas should be clear. The structure-preservation can be deduced from Fact 1 of the theorem and step (3.2) of the transformation.) It will follow from Theorems 3.4 and 3.5 that any LR(n) grammar or any (m, n) BRC grammar can be transformed to a $(1, n)$ BRC grammar with syntax trees preserved up to labeling and one-level extension of terminal nodes. Therefore a semantic model (such as that of Wirth and Weber [10]) which associates semantics with each rule, rather than with the symbols within the right part, can be carried over directly to the transformed grammar. The key characteristic of these transformations is that of preserving the length and symbol-wise semantic significance of a right part. For instance, a tree  is never mapped into a tree  .

The transformation of Theorem 3.1 works for any choice of g. However, the utility of the transformation depends on the care with which g is defined. Let us consider the definition of g used in the previous example. Associated with the grammar G are the following rightmost derivations:

$$S \Rightarrow aaX \Rightarrow aatU_1y \Rightarrow aatuy$$

$$S \Rightarrow bbX \Rightarrow bbtU_1y \Rightarrow bbtuy$$

$$S \Rightarrow bbY \Rightarrow bbtU_1z \Rightarrow bbtuz$$

$$S \Rightarrow ccY \Rightarrow cctU_1z \Rightarrow cctuz$$

$$S \Rightarrow ccZ \Rightarrow cctU_2y \Rightarrow cctuy.$$

G is not a (1, 1) BRC grammar. For example (surrounding the derivations by end-markers), with respect to the rule $U_1 \rightarrow u$, the second derivation,

$$¢S\$ \overset{+}{\Rightarrow} ¢bbtU_1y\$ \Rightarrow ¢bbtuy\$$$

and the fifth derivation,

$$¢S\$ \overset{+}{\Rightarrow} ¢cctU_2y\$ \Rightarrow ¢cctuy\$$$

violate the (1, 1) BRC condition since there are two different ways to generate u with the same length-1 context on either side (namely, t—y). Furthermore, no matter how much the length of the right context is increased, the two possibilities remain. However, if the length of the left context is increased, then in the second derivation, u is generated preceded by $¢bbt$ and in the fifth derivation, u is preceded by $¢cct$. Similarly, the first and fifth derivations provide two different ways to generate u in the same context t—y. However, in the first derivation, u is generated preceded by $¢aat$. These are the only (1, 1) BRC violations. We need only consider left contexts, at, bt and ct in order to exclude a BRC violation. G is a (2, 1) BRC grammar.

(Notice that with respect to $U_1 \rightarrow u$, the fourth and fifth derivations of G listed after Corollary 3.2 cause a BRC violation for any length left context, so the right context must be at least one. Had we used rules $S \rightarrow abY|acZ$ instead of $S \rightarrow bbY|ccZ$, the grammar could be regarded either as (3, 0) BRC or as (2,1) BRC. For the sake of containment within the family of LR(0) grammars, such a grammar is treated as a (3, 0) BRC grammar.)

In the transformed grammar G', we have encoded within the symbol replacing t the symbol preceding it. Therefore in the corresponding derivations in G', $[t, u]$ (the symbol replacing u) is generated with left contexts $[a, t]$, $[b, t]$ or $[c, t]$. This does not violate the (1, 1) BRC condition, and it turns out that G' is a (1, 1) BRC grammar.

In fact, the transformation with this choice of the function g always reduces the left bound of a BRC grammar. (This is proved by taking $k = 1$ in Lemma 3.3.) Therefore at most $m - 1$ repeated applications of the transformation transform any (m, n) BRC grammar to a $(1, n)$ BRC grammar. By a straightforward change in the way we define g, we can get the $(1, n)$ BRC grammar directly. Instead of encoding in each symbol the symbol preceding it in the original grammar, we encode the k or fewer symbols preceding it, for some k, $0 < k < m$, and thereby reduce

the left bound by k. In Lemma 3.3 we give the appropriate definition of g and prove that the left bound reduction does occur.

LEMMA 3.3. *Let $G = (V, V_T, P, S)$ be an (m, n) BRC grammar $(m > 1)$. For any $k, 0 < k < m$, G can be transformed to an equivalent $(m - k, n)$ BRC grammar.*

Proof. Let k be some fixed value, $0 < k < m$. Let $\mathscr{W} = \{x \in V^* | 1 \leqq |x| \leqq k\}$. (Thus $V_{[]} = \{[x, B] | 1 \leqq |x| \leqq k, x \in V^*, B \in V\}$.) Transform G to G' by the transformation of Theorem 3.1, where for every $X \in V$, $[y, Y] \in V_{[]}$,

$$g([y, Y], X) = [\text{last}_k(yY), X],$$

and for each X and Y in V,

$$g(Y, X) = [Y, X].$$

Clearly, g is an encoding function. Therefore $L(G) = L(G')$ and G' is an (m, n) BRC grammar. It remains to show that G' is an $(m - k, n)$ BRC grammar.

Suppose not. Since G' is an (m, n) BRC grammar, there is no rightmost derivation $S \overset{+}{\Rightarrow} S$. Therefore for some rule $U \to u$ in G' and some X, $U \in V' - V_T$; $u, x \in (V')^*$; $\tau_1, \tau_2, t, \sigma \in \{\not\in\}^*(V')^*$; $v_1, v_2, v, \omega \in (V')^*$; $T \in V' \cup \{\not\in\}$ such that $|t| = m - k - 1$ and $|v| = n$, there are rightmost derivations

$$\not\in^m S\$^n \overset{*}{\Rightarrow} \tau_1 T t U v v_1 \Rightarrow \tau_1 T t u v v_1$$

and

$$\not\in^m S\$^n \overset{*}{\Rightarrow} \sigma X \omega \Rightarrow \sigma x \omega = \tau_2 T t u v v_2$$

in G', where $|\omega| \leqq |vv_2|$ and either $\sigma \neq \tau_2 T t$ or $X \neq U$ or $\omega \neq vv_2$.

We follow the case analysis of Theorem 3.1.

Case 1. $h(U), h(X) \in V_N$. By Fact 2 of Theorem 3.1, there are corresponding rightmost derivations

$$\not\in^m S\$^n \overset{*}{\Rightarrow} h(\tau_1 T t U) v v_1 \Rightarrow h(\tau_1 T t u) v v_1$$

and

$$\not\in^m S\$^n \overset{*}{\Rightarrow} h(\sigma X)\omega \Rightarrow h(\sigma x)\omega = h(\tau_2 T t u) v v_2$$

in G, where $|\omega| \leqq |vv_2|$. Since G is an (m, n) BRC grammar, $\text{last}_k(h(\tau_1)) \neq \text{last}_k(h(\tau_2))$. Therefore, $T \neq \not\in$. Since g maps every pair of arguments into a symbol in $V_{[]}$ and since in at least one of the derivations there are elements of V' to the left of T, $T \in V_{[]}$ (by Fact 2). However, it is easily shown by induction on p that for any string $X_1[x_2, X_2][x_3, X_3] \cdots [x_p, X_p]$ in V^+ and any $0 < k \leqq m$, if for $2 \leqq i \leqq p$, $x_i = \text{last}_k(x_{i-1}X_{i-1})$, then $x_i = \text{last}_k(X_1 X_2 \cdots X_{i-1})$. That is, encoded in each bracketed symbol is the image under h of the length-k string to the left of it. Therefore, for some $z \in V^*$, $Z \in V$, $T = [z, Z]$, where

$$z = \text{last}_k(h(\tau_1)) \quad \text{and} \quad z = \text{last}_k(h(\tau_2)).$$

But this contradicts the fact that $\text{last}_k(h(\tau_1)) \neq \text{last}_k(h(\tau_2))$. Therefore Case 1 cannot occur. The other cases are ruled out by arguments analogous to those of Theorem 3.1. It follows that G' is an $(m - k, n)$ BRC grammar. □

We now have a way to transform (m, n) BRC grammars to equivalent $(1, n)$ BRC grammars. However, as can be seen even from the previous example (where

$m = 2$), the transformed grammar may become significantly larger than the original grammar. The reason is that we have made some changes in the original grammar that were unnecessary in order to achieve the desired BRC property.

Let us reconsider the previous example. The only left contexts of length two that we need to consider have second symbol t. Therefore t is the only symbol which needs to be replaced by a bracketed symbol containing its predecessor. Furthermore, if we look at the two sets of rightmost derivations which violate the (1, 1) BRC condition, we see that it suffices to know when t is preceded by c and when it is preceded by other symbols (namely a or b). Consequently, by partitioning the set $\{a, b, c\}$ of left contexts of t into $\{a, b\}$ and $\{c\}$, we can encode *sets* of left contexts in the bracketed symbols replacing t. Furthermore, we can let the larger of these subsets, $\{a, b\}$, be part of the default case in which the encoding is implicit. Therefore we let $\mathscr{W} = \{\{c\}\}$ and define g by :

$g(c, t) = [\{c\}, t]$;

for every $A \in \{X, Y, Z\}$, $g(c, A) = [\{c\}, A]$ (to satisfy condition (g2));

for every $A \in (V \cup V_{[\,]}) - \{c\}$, for every $B \in \{t, X, Y, Z\}$, $g(A, B) = B$;

for every $A \in V \cup V_{[\,]}$, for every $B \in V - \{t, X, Y, Z\}$, $g(A, B) = B$.

The reduced transformed grammar $G'' = (V'', V_T, P'', S)$ has rules P'':

(from step (3.1a))

$$S \to aaX|bbX|bbY|cc[\{c\}, Y] \mid cc[\{c\}, Z]$$

$$X \to tU_1y$$

$$Y \to tU_1z$$

$$U_1 \to u$$

$$U_2 \to u$$

(from step (3.1b))

$$[\{c\}, Y] \to [\{c\}, t]U_1z$$

$$[\{c\}, Z] \to [\{c\}, t]U_2y$$

(from step (3.2))

$$[\{c\}, t] \to t.$$

We have changed only those symbols whose context-encoding was relevant for the BRC condition (or necessary for transmitting context) and we have grouped together those contexts which occur similarly in rightmost derivations. (Any time t occurs in a rule which is not formed by step (3.2), it has left context in $(VU\{\not\!\!\!c\}) - \{c\}$.) We now specify these conditions in general.

DEFINITION. Let $G = (V, V_T, P, S)$ be an (m, n) BRC grammar, $m > 0$. For all $T_1, T_2 \in V \cup \{\not\!\!\!c\}$, $t \in V^*$, $|t| < m$, if, for some rule $U \to u$ in G and some $X \in V$, $\tau_1, \tau_2, \sigma \in \{\not\!\!\!c\}^*V^*$; $v_1, v_2, \omega, v \in V_T^*\{\$\}^*$ where $|v| = n$, G has rightmost derivations

$$\not\!\!\!c^m S\$^n \overset{*}{\Rightarrow} \tau_1 T_1 tUvv_1 \Rightarrow \tau_1 T_1 tuvv_1$$

and

$$¢^m S\$^n \overset{*}{\Rightarrow} \tau X \omega \Rightarrow \tau_2 T_2 tuvv_2$$

(where $|\omega| \leqq |vv_2|$ and either $\sigma \neq \tau_2 T_2 t$ or $X \neq U$ or $\omega \neq vv_2$) for $T_1 \neq T_2$ but not for $T_1 = T_2$, then we call $(T_1 t, T_2 t)$ a *minimal significant left context pair* (in G). We say the pair is *length-k* if $|T_1 t| = |T_2 t| = k$.

Thus, the minimal significant left context pairs are the shortest pairs of strings that exclude a BRC violation. For instance, in the previous example, the minimal significant left context pairs are (at, ct) and (bt, ct).

By defining an encoding function determined by the minimal significant left context pairs, we get a second proof of Lemma 3.3 which produces much smaller grammars than the transformation used in the first proof.

Second proof of Lemma 3.3. With each $A \in V$ associate a collection of non-empty sets $\mathscr{X}_A = \{X_{A,0}, X_{A,1}, \cdots, X_{A,p_A}, p_A \geqq 0\}$ such that

(i) $¢ \in X_{A,0}$;
(ii) $\bigcup_{j=0}^{p_A} X_{A,j} = V \cup \{¢\}$;
(iii) the components of $\mathscr{X}_A$ are pairwise disjoint;
(iv) if $(T_1 A\pi, T_2 A\pi)$ is any minimal significant left context pair of length two or greater, then T_1 and T_2 are in different components of $\mathscr{X}_A$;
(v) for each $A, B \in V, x \in V^*$ such that $A \neq B$ and $A \overset{*}{\Rightarrow} Bx$, either $\mathscr{X}_B = \{X_{B,0}\}$ or $\mathscr{X}_A = \mathscr{X}_B$.

(In general, there will be more than one way to partition $V \cup \{¢\}$ into the components of each $\mathscr{X}_A$. It is always possible to find such partitions since, at worst, each $\mathscr{X}_A$ can consist only of singletons. The increase both in the number of rules and in the number of nonterminals resulting from the transformation is minimized if each $\mathscr{X}_A$ contains as few components as possible and each $\mathscr{X}_{A,0}$ is as large as possible.)

Let $\mathscr{W} = \{\mathscr{W}_i | \mathscr{W}_i \subseteq V\}$. Transform G to G' by the transformation of Theorem 3.1, where for every $A \in V$, $Y \in V \cup V_{[]}$,

$$g(Y, A) = \begin{cases} [\mathscr{W}_i, A] & \text{if } h(Y) \in X_{A,j},\ 1 \leqq j \leqq p_A \text{ and } \mathscr{W}_i = X_{A,j}, \\ A & \text{otherwise}. \end{cases}$$

Clearly, condition (g1) is satisfied. Conditions (g2) and (g3) follow from condition (v) of the definition of the sets $\mathscr{X}_A$. Therefore g is an encoding function and it follows from Theorem 3.1 that $L(G) = L(G')$ and G' is an (m, n) BRC grammar. It remains to show that G' is an $(m - k, n)$ BRC grammar for some k, $0 < k < m$.[7]

Suppose not. Since G' is an (m, n) BRC grammar, there is no rightmost derivation $S \overset{+}{\Rightarrow} S$ in G'. Therefore G' has some length-m minimal significant left context pair. That is, for some rule $U \to u$ in G' and some $X \in V' - V_T$, $t, u, x \in (V')^*$; $\tau_1, \tau_2, \sigma \in \{¢\}^*(V')^*$, $v_1, v_2, v, \omega \in (V')^*\{\$\}^*$; $T_1, T_2 \in V' \cup \{¢\}$ such that $|t| = m - 1$ and $|v| = n$ there are rightmost derivations

$$¢^m S\$^n \overset{*}{\Rightarrow} \tau_1 T_1 tUvv_1 \Rightarrow \tau_1 T_1 tuvv_1$$

---

[7] All one can guarantee is $k = 1$. However, as a "side effect", the transformation may reduce the left bound further.

and

$$\mathfrak{c}^m S\$^n \overset{*}{\Rightarrow} \sigma X\omega \Rightarrow \sigma x\omega = \tau_2 T_2 tuvv_2$$

in G' where $T_1 \neq T_2$, $|\omega| \leqq |vv_2|$ and either $\sigma \neq \tau_2 T_2 t$ or $X \neq U$ or $\omega \neq vv_2$.

We follow the case analysis of Theorem 3.1.

Case 1. $h(U), h(X) \in V_N$. By Fact 2 of Theorem 3.1, there are corresponding rightmost derivations

$$\mathfrak{c}^m S\$^n \overset{*}{\Rightarrow} h(\tau_1 T_1 tU)vv_1 \Rightarrow h(\tau_1 T_1 tu)vv_1$$

and

$$\mathfrak{c}^m S\$^n \overset{*}{\Rightarrow} h(\sigma X)\omega \Rightarrow h(\sigma x)\omega = h(\tau_2 T_2 tu)vv_2$$

in G, where $|\omega| \leqq |vv_2|$. Since G is an (m, n) BRC grammar, $h(T_1) \neq h(T_2)$ and $(h(T_1 t), h(T_2 t))$ is a minimal significant left context pair in G. Since $|t| > 0$, $h(t) = A\pi$ for some $A \in V$, $\pi \in V^*$. By construction, $h(T_1) \in \mathscr{X}_{A,i}$, $h(T_2) \in \mathscr{X}_{A,j}$, $i \neq j$. Consequently by definition of g, $g(T_1, A) \neq g(T_2, A)$, contradicting the facts (using Fact 2 of Theorem 3.1) that T_1 is followed by $g(T_1, A)$, T_2 is followed by $g(T_2, A)$, and both are followed by t. Hence Case 1 cannot arise. The other cases are ruled out by arguments analogous to those of Theorem 3.1. It follows that G' is an $(m - k, n)$ BRC grammar for some k, $0 < k < m$. □

In fact we can "optimize" the transformation still further. In defining the sets $\mathscr{X}_A$ in the second proof of Lemma 3.3, we can replace condition (v) by condition (v′):

(v′) for every $A, B \in V$, $x \in V^*$ such that $A \neq B$ and $A \overset{*}{\Rightarrow} Bx$,

(a) $X_{A,0} \subseteq X_{B,0}$,

(b) for $1 \leqq j \leqq p_A$, either $X_{A,j} = X_{B,k}$, $1 \leqq k \leqq p_B$ or $X_{A,j} \subseteq X_{B,0}$.

$X_{B,0}$ represents the "don't care" symbols—those symbols which it is not necessary to record as context information. Condition (v) was included in order that the ensuing definition of g satisfy encoding function condition (g3), which is needed for BRC-preservation. However, using condition (v′) above and this particular definition of g, we could modify the transformation of Theorem 3.1 by replacing the definition of X_1'' in step (3.1b) by

$$X_1'' = \begin{cases} X_1 & \text{if for every } Z \in \mathscr{W}_i,\ g(Z, X_1) = X_1, \\ [\mathscr{W}_i, X] & \text{otherwise.} \end{cases}$$

It requires a separate proof (which we do not present here) that the modified transformation with this definition of g transforms an (m, n) BRC grammar to an equivalent $(m - k, n)$ BRC grammar for some k, $0 < k < m$.

Having shown in a variety of ways that the left bound of a BRC grammar can be reduced, we have the following conclusion.

THEOREM 3.4. *Let* $G = (V, V_T, P, S)$ *be an* (m, n) BRC *grammar.* G *can be transformed to an equivalent* $(1, n)$ BRC *grammar* $G' = (V', V_T, P', S)$.

Proof. Either let $k = m - 1$ in the first proof of Lemma 3.3 or repeat the transformation of Theorem 3.1 using the second proof of Lemma 3.3 at most $m - 1$ times. □

Since the structure-preserving transformation approach works for transforming (m, n) BRC grammars to $(1, n)$ BRC grammars, why not use the same sort

of transformations to take LR(k) grammars into (m, k) BRC grammars and LR(k) grammars into LR(1) grammars? We consider both these issues.

The transformation from an LR(k) grammar to an (m, k) BRC grammar is a generalization of the (m, n) BRC-to-($1, n$) BRC transformation. We again wish to modify nonterminal symbols to reflect their immediate left context in rightmost derivations. We can find sets of minimal significant left contexts (see details in [3]). However, the sets of minimal significant left contexts are in general infinite regular sets. Therefore we cannot reduce minimal significant left contexts one symbol at a time (as we did in the second proof of Lemma 3.3), since their lengths are unbounded.

For example, consider a set of rules which is a modification of the set in the BRC example following Theorem 3.1:

$$
\begin{aligned}
P: \quad & S \to aaX \mid bbX \mid bbY \mid ccY \mid ccZ \\
& X \to tX \mid tU_1y \\
& Y \to tY \mid tU_1z \\
& Z \to tZ \mid tU_2y \\
& U_1 \to u \\
& U_2 \to u.
\end{aligned}
$$

Here an occurrence (in rightmost sentential forms) of U_1 generated by $X \to tU_1y$ is distinguished from an occurrence of U_2 by left contexts at^+ and bt^+ for U_1 and ct^+ for U_2. The set of minimal significant left contexts of length two or greater is $\{(at^i, ct^i) \mid i > 0\} \cup \{(bt^i, ct^i \mid i > 0\}$. We can reduce the context information by transforming P to a set of rules P' in which we have replaced occurrences of t following c by a bracketed symbol indicating that fact. For example,

$$
\begin{aligned}
P': \quad & S \to aaX \mid bbX \mid bbY \mid cc[c, Y] \mid cc[c, Z] \\
& X \to tX \mid tU_1y \\
& Y \to tY \mid tU_1z \\
& [c, Y] \to [c, t][c, Y] \mid [c, t]U_1z \\
& [c, Z] \to [c, t][c, Z] \mid [c, t]U_2y \\
& U_1 \to u \\
& U_2 \to u \\
& [c, t] \to t.
\end{aligned}
$$

However, the occurrence of c in a rightmost sentential form may be an unbounded distance from the occurrence of U_1 or U_2.

Although this transformation is in form similar to that of the BRC example, the transformation is not, in general, made solely on the basis of the preceding symbol, but rather on the basis of the preceding context. The left context of U_2 is $cc[c, t]^+$, whereas the left context of U_1, when generated by $X \to tU_1y$, is aat^+.

In fact this transformation has all the combinatorial complexity associated with constructing general LR parsers. We can use the transformation of Theorem 3.1 with yet another encoding function g to incorporate in the bracketed symbols of the transformed grammar the state sets or LR(k) tables which would be used by an LR parser.[8] The encoding is done in such a way that the sequence of nonterminals in any rightmost sentential form of the transformed grammar corresponds to the stack contents of the corresponding configuration of an LR parser for the original grammar. The only difference is that the sets of LR(k) items at the bottom and top of the parsing stack are missing from the encoding and the brackets are present. The sets $\mathscr{W}_i \in \mathscr{W}$ used in Theorem 3.5 correspond to the state sets or LR(k) tables of the LR parser.

THEOREM 3.5. *Let $G = (V, V_T, P, S)$ be an* LR(k) *grammar. G can be transformed to an equivalent* $(1, k)$ BRC *grammar.*

Proof. Let $\mathscr{X} = \{[X \to x_1 \cdot x_2, u] | X \to x_1 x_2$ is in P, $x_1, x_2 \in V^*$, $u \in V_T^*\{\$\}^*$, $|u| = k\}$ be the set of LR(k) states or items. Let $\mathscr{W} = \{\mathscr{W}_i | \mathscr{W}_i \subseteq \mathscr{X}, 0 \leqq i \leqq s\}$. Thus each $\mathscr{W}_i$ is a state set or set of LR(k) items. The $\mathscr{W}_i$ are numbered arbitrarily, except that $\mathscr{W}_0$ is formed as follows:

(a0) for each rule $S \to x$ in P,

$$[S \to \cdot x, \$^k] \in \mathscr{W}_0;$$

(b0) for each $[X \to \cdot Yx, u] \in \mathscr{W}_0$, for every rule $Y \to y$ in P, for each w, $v \in V_T^*\{\$\}^*$ such that $|v| = k$ and $xu \overset{*}{\Rightarrow} vw$,

$$[Y \to \cdot y, v] \in \mathscr{W}_0.$$

Define a function $f: \mathscr{W} \times V \to \mathscr{W}$ by:

$$f(\mathscr{W}_i, Y) = \mathscr{W}_j,$$

where

(a1) for each $[X \to x_1 \cdot Yx_2, u] \in \mathscr{W}_i$, $[X \to x_1 Y \cdot x_2, u] \in \mathscr{W}_j$

(b1) for each $[X \to x_1 \cdot Yx_2, u] \in \mathscr{W}_j$, for each $Y \to y$ in P, for each $w, v \in V_T^*\{\$\}^*$ such that $|v| = k$ and $x_2 u \overset{*}{\Rightarrow} vw$,

$$[Y \to \cdot y, v] \in \mathscr{W}_j.$$

Transform G to G' by the transformation of Theorem 3.1, where for each $X \in V$, $[\mathscr{W}_i, Y] \in V_{[]}$,

$$g([\mathscr{W}_i, Y], X) = [f(\mathscr{W}_i, Y), X],$$

for each $X, Y \in V$,

$$g(Y, X) = [f(\mathscr{W}_0, Y), X].$$

Clearly g is an encoding function. Therefore, by Corollary 3.2, $L(G) = L(G')$ and G' is an LR(k) grammar. It remains to show that G' is a $(1, k)$ BRC grammar.

Since g maps every pair of arguments into a symbol in $V_{[]}$, it follows from Fact 2 of Theorem 3.1 that every rightmost sentential form x is in $VV_{[]}^*V_T^* \cup V_T^*$.

[8] We assume the reader is somewhat familiar with LR parsing. The reader is referred to Aho and Ullman [1, § 5.2] or Knuth [7, § II] for a discussion of this subject.

The first elements of the bracketed symbols carry information about left context in the following ways.

FACT 3.[9]

(a) *For each* $[X \to \cdot x, s] \in \mathscr{W}_0$, $S\$^k \overset{*}{\Rightarrow} Xs\omega$ *is a rightmost derivation in* G *for some* $\omega \in V_T^*\{\$\}^*$.

(b) *Let* $S\$^k \overset{*}{\Rightarrow} U_0[\mathscr{W}_{i_1}, U_1][\mathscr{W}_{i_2}, U_2] \cdots [\mathscr{W}_{i_p}, U_p]v$ *be any rightmost derivation in* G', *where* $v \in V_T^*\{\$\}^k$, $p \geqq 0$, $U_0 \in V$. *Then for every* j, $1 \leqq j \leqq p$, *and for each* $[X \to x_1 \cdot x_2, s] \in \mathscr{W}_{i_j}$, $x_1 = U_{q+1}U_{q+2} \cdots U_{j-1}$, $-1 \leqq q \leqq j-1$, *and* $S\$^k \overset{*}{\Rightarrow} U_0U_1 \cdots U_qXs\omega$ *is a rightmost derivation in* G, *for some* $\omega \in V_T^*\{\$\}^*$.

Proof. (a) The proof is by induction on the number of elements in $\mathscr{W}_0$.

(b) The proof is by induction on j (each case is by induction on the size of $\mathscr{W}_{i_j}$).

Thus each element of each $\mathscr{W}_{i_j}$ corresponds to some rightmost derivation of a sentential form with prefix $U_0U_1 \cdots U_{j-1}$.

FACT 4.[10] *Let*

$$S\$^k \overset{*}{\Rightarrow} tU'vv \Rightarrow tU'_{q+1}[\mathscr{W}_{i_{q+2}}, U_{q+2}][\mathscr{W}_{i_{q+3}}, U_{q+3}] \cdots [\mathscr{W}_{i_p}, U_p]vv$$

be any rightmost derivation in G', $vv \in V_T^*\{\$\}^k$, $|v| = k$, $q = |t| - 1$, $q \leqq p$ *in which the last step uses a rule formed by step* (1) *of the transformation, where*

$$U' = \begin{cases} U & \text{if } t = \lambda, \\ [\mathscr{W}_{i_{q+1}}, U] & \text{otherwise}; \end{cases}$$

$$U'_{q+1} = \begin{cases} U_{q+1} & \text{if } t = \lambda, \\ [\mathscr{W}_{i_{q+1}}, U_{q+1}] & \text{otherwise}; \end{cases}$$

and

$$t = U_0[\mathscr{W}_{i_1}, U_1][\mathscr{W}_{i_2}, U_2] \cdots [\mathscr{W}_{i_q}, U_q].$$

Then:

(a) *For some* $x \in V^*$, $u \in V_T^*\{\$\}^*$, $\mathscr{W}_0$ *contains an item of the form* $[X \to \cdot U_0x, u]$.

(b) *For* $1 \leqq j \leqq q$, $\mathscr{W}_{i_j}$ *contains an item of the form* $[X \to U_{r+1}U_{r+2} \cdots U_{j-1} \cdot U_jx, u]$, *for some* $x \in V^*$, $u \in V_T^*\{\$\}^*$, $-1 \leqq r \leqq j$.

(c) $\mathscr{W}_{i_{q+1}}$ *contains some item of the form* $[X \to U_{r+1}U_{r+2} \cdots U_q \cdot Ux, u]$, *for some* $x \in V^*$, $u \in V_T^*\{\$\}^*$, $-1 \leqq r \leqq q$ *and the item*

$$[U \to \cdot U_{q+1}U_{q+2} \cdots U_p, v].$$

(d) *For* $q < j \leqq p$, $\mathscr{W}_{i_j}$ *contains the item* $[U \to U_{q+1}U_{q+2} \cdots U_{j-1} \cdot U_jU_{j+1} \cdots U_p, v]$.

Proof. The proof is by induction on the length of a derivation.

Thus, if we regard $\mathscr{W}_0$ as implicitly preceding U_0, for $0 \leqq j \leqq p$, $\mathscr{W}_{i_j}$ contains an item or state which indicates the production by which U_j was generated in the rightmost derivation.

Now we are ready to establish the BRC condition. Suppose G' is not a $(1, k)$ BRC grammar. Then for some rule $U \to u$ in G' and some X, $U \in V' - V_T$; u,

[9] See Aho and Ullman [1, Thm. 5.10].

[10] See footnote 9.

$x \in (V')^*$; $\tau_1, \tau_2, \sigma \in \{\not\subset\}^*(V')^*$; $v_1, v_2, v, \omega \in (V')^*$; $T \in V' \cup \{\not\subset\}$ such that $|v| = k$, there are rightmost derivations

$$\not\subset S\$^k \overset{*}{\Rightarrow} \tau_1 TUvv_1 \Rightarrow \tau_1 Tuvv_1$$

and

$$\not\subset S\$^k \overset{*}{\Rightarrow} \sigma X\omega \Rightarrow \sigma x\omega = \tau_2 Tuvv_2$$

in G', where $|\omega| \leqq |vv_2|$ and either $\sigma \neq \tau_2 T$ or $X \neq U$ or $\omega \neq vv_2$.

We follow the case analysis of Theorem 3.1.

Case 1. $h(U)$, $h(X) \in V_N$. Suppose $u = U_1'[\mathscr{W}_{i_2}, U_2][\mathscr{W}_{i_3}, U_3] \cdots [\mathscr{W}_{i_p}, U_p]$, where

$$U_1' = \begin{cases} U_1 & \text{if } T = \not\subset, \\ [\mathscr{W}_{i_1}, U_1] & \text{otherwise,} \end{cases}$$

and $p \geqq 1$. By Fact 4, $[U \to U_1 U_2 \cdots U_{p-1} \cdot U_p, v] \in \mathscr{W}_{i_p}$, where if $T = \not\subset$ and $p = 1$, then $i_p = 0$. It follows from Fact 3 that for some $v_3 \in V_T^*\{\$\}^*$, $\not\subset S\$^k \overset{*}{\Rightarrow} h(\tau_2 TU)vv_3 \Rightarrow h(\tau_2 Tu)vv_3$ is a rightmost derivation in G. By Fact 2 of Theorem 3.1, $\not\subset S\$^k \overset{*}{\Rightarrow} h(\sigma X)\omega \Rightarrow h(\sigma x)\omega = h(\tau_2 Tu)vv_2$ is a rightmost derivation in G, where $|\omega| \leqq |vv_2|$. Since G is an LR(k) grammar, $h(\sigma) = h(\tau_2 T)$, $h(X) = h(U)$ and $\omega = vv_2$. It follows from Fact 2 of Theorem 3.1 that $\sigma = \tau_2 T$, $X = U$, and $\omega = vv_2$, contradicting the non-BRC assumption. Therefore this case cannot arise.

Alternatively, suppose $u = \lambda$. Since $h(X) \in V_N$ and $\sigma X \in VV_{[]}^*$, $\omega = vv_2$ and $|\sigma| \leqq |\tau_2 T|$. Since G' is an LR(k) grammar, $T \neq \not\subset$. Let $\mathscr{W}_i$ be the set of items associated with T (that is, if $T \in V$, then $i = 0$; otherwise $T = [\mathscr{W}_i, Z]$ for some $Z \in V$). Similarly, let $\mathscr{W}_j$ be the set of items associated with X. Let $U = [\mathscr{W}_l, Z]$ for some $Z \in V$. By Fact 2, $[U \to \cdot, v] \in \mathscr{W}_l$. By Fact 2 of Theorem 3.1, $\omega_l = f(\mathscr{W}_i, h(T))$. If $\sigma = \tau_2 T$, then $[X \to \cdot, v] \in \mathscr{W}_j$ and $\mathscr{W}_j = f(\mathscr{W}_i, h(T)) = \mathscr{W}_l$. It follows from Fact 3 that for some $v_3, v_4 \in V_T^*\{\$\}^*$,

$$\not\subset S\$^k \overset{*}{\Rightarrow} h(\tau_2 TU)vv_3 \Rightarrow h(\tau_2 T)vv_3$$

and

$$\not\subset \$S^k \overset{*}{\Rightarrow} h(\tau_2 TX)vv_4 \Rightarrow h(\tau_2 T)vv_4$$

are rightmost derivations in G. Since G is an LR(k) grammar, $h(U) = h(X)$. Therefore, by Fact 2 of Theorem 3.1, $U = X$, contradicting the non-BRC assumption. Therefore this case cannot arise. On the other hand, if $|\sigma| < |\tau_2 T|$, then by Fact 4, $[X \to t \cdot h(T), v] \in \mathscr{W}_i$ for some t which is a suffix of $h(\tau_2)$. Therefore

$$[X \to th(T)\cdot, v] \in f(\mathscr{W}_i, h(T)) = \mathscr{W}_l.$$

It follows from Fact 3 that for some $v_3, v_4 \in V_T^*\{\$\}^*$,

$$\not\subset S\$^k \overset{*}{\Rightarrow} h(\tau_2 TU)vv_3 \Rightarrow h(\tau_2 T)vv_3$$

and

$$\not\subset S\$^k \overset{*}{\Rightarrow} h(\sigma X)vv_4 \Rightarrow h(\tau_2 T)vv_4$$

are rightmost derivations in G, where $h(\sigma) \neq h(\tau_2 T)$, contradicting the fact that G is an LR(k) grammar.

The other cases are ruled out by arguments analogous to those of Theorem 3.1. It follows that G' is a $(1, k)$ BRC grammar. □

We have shown that there is a structure-preserving transformation for transforming LR(k) grammars into $(1, k)$ BRC grammars. Minimizing the increase in grammatical size caused by this transformation can be done by using the ideas of the second proof of Lemma 3.3 together with optimization techniques for LR parsing tables (see, for example, [1]).

The problem with general transformations for reducing right context is that they do not preserve grammatical structure in the strict sense used so far. In reducing left bounds, we were able to change the nonterminals in left contexts according to their immediate left contexts, that is, to have nonterminal symbols carry information about their left contexts in rightmost sentential forms. This technique is not applicable for reducing right bounds because right contexts are composed only of terminal symbols.

Suppose $G = (V, V_T, P, S)$ is an LR(k) grammar with rightmost derivations

$$S \overset{*}{\Rightarrow} xU_1va \Rightarrow xuva$$

and

$$S \overset{*}{\Rightarrow} xU_2vb \Rightarrow xuvb.$$

Given the sentential form $xuva$, in order to determine whether u is generated by U_1 or U_2, it is necessary to look beyond v to see whether v is followed by a or b. In order to decrease the lookahead, either U_1 and U_2 must be made to generate part of v as well as u, or u must be generated by the same rule in both contexts. In the former case, the shape of the syntax tree is changed; in the latter, two distinct cases (possibly having different associated semantics) are transformed to one. (These issues are discussed further in [3], which also contains structure-modifying transformations which reduce right context.)

The language characterization follows easily from Theorem 3.5.

THEOREM 3.6. *Every deterministic context-free language is generated by a* (1, 1) BRC *grammar.*

Proof. Knuth [7] states that every deterministic language is generated by an LR(1) grammar. (More complete proofs appear in [6] and [8].) Theorem 3.5 sharpens the condition to (1, 1) BRC. □

Theorem 3.6 can be sharpened further by putting additional restrictions on the (1, 1) BRC grammars. We have shown [3] that a variation of the transformations presented here preserves the disjointness of the Wirth–Weber simple precedence relations [10] (i.e., if these relations are disjoint for the initial grammar, then they are disjoint for the transformed grammar) and that any LR(k) or (m, n) BRC grammar can be transformed to an equivalent such grammar with disjoint simple precedence relations. (The variation consists of adding $\{[x] | x \in V_T\}$ to V', changing the definition of X'_1 in step (3.1a) of the transformation of Theorem 3.1 to

$$X'_1 = \begin{cases} [X_1] & \text{if } X_1 \in V_T \text{ and } X_1 \text{ is not fixed under } g, \\ X_1 & \text{otherwise,} \end{cases}$$

adding to the transformation of Theorem 3.1 the step

(3.6) For each $X \in V_T$, if step (3.1a) has added to P' a rule containing $[x]$, then put $[x] \to x$ in P',

extending the definition of h so that for every $x \in V_T$, $h([x]) = x$, and adding to the conditions on encoding functions,

(g4) For every $x \in V_T$, if for some $Y \in V \cup V_{[]}$, $g(Y, x) \in V_{[]}$, then for every $Y \in V \cup V_{[]}$, $g(Y, x) \in V_{[]}$.

This insures that if terminals are replaced by nonterminals, the replacement occurs everywhere in the grammar.) Therefore we can add the precedence-disjointness condition to the theorem. Furthermore, we have shown [3], [4] that a subset of the (1, 1) BRC grammars which have the added constraints of no two rules with the same right part and disjoint (2, 1) precedence relations generate all the deterministic languages.

4. Elimination of λ-rules. The definition of BRC grammars given in this paper admits grammars with λ-rules. For purposes of comparison with other classes of grammars generating the deterministic context-free languages, notably various kinds of precedence grammars and Floyd's formulation of bounded right context, it is useful to investigate the necessity for the inclusion of such rules. In Theorem 4.3 we show that it is possible to remove the λ-rules from any (m, n) BRC grammar, at worst removing λ from the generated language.

We prove this result in several stages. We first show (Lemma 4.1) that for right bounds of one or more, the inclusion of occurrences of the initial symbol in right parts is inessential, by showing that a trivial transformation removing this condition is (m, n) BRC-preserving and LR(k)-preserving.

LEMMA 4.1. *Let $G = (V, V_T, P, S)$ be an (m, n) BRC grammar, $n > 0$, and let $S_0 \notin V$. Then $G_1 = (V \cup \{S_0\}, V_T, P \cup \{S_0 \to S\}, S_0)$ is an equivalent (m, n) BRC grammar.*[11]

Proof. For any derivation $S \overset{*}{\Rightarrow} x$ in G, $x \in V^*$, there is a corresponding derivation $S_0 \Rightarrow S \overset{*}{\Rightarrow} x$ in G_1, where the derivation $S \overset{*}{\Rightarrow} x$ in G_1 uses only rules in P. Any derivation in G_1 is of the form $S_0 \Rightarrow S \overset{*}{\Rightarrow} x$, where $S \overset{*}{\Rightarrow} x$ uses only rules in P (since $S_0 \to S$ is the only rule in which S_0 occurs). Therefore, $L(G) = L(G_1)$. Furthermore since G is an (m, n) BRC grammar, and since there is no rightmost derivation $S_0 \overset{+}{\Rightarrow} S_0$ in G_1, G_1 could fail to be an (m, n) BRC grammar only for a derivation of length one, that is, only if there exist rightmost derivations

$$\cent^m S_0 \$^n \Rightarrow \cent^m S \$^n$$

and

$$\cent^m S_0 \$^n \overset{+}{\Rightarrow} \sigma X \omega \Rightarrow \cent^m S \$^n$$

in G_1, where $|\omega| \leqq n$, and $\sigma \neq \cent^m$ or $\omega \neq \$^n$ or $X \neq S$. However, the second derivation must be of the form $\cent^m S_0 \$^n \Rightarrow \cent^m S \$^n \overset{*}{\Rightarrow} \sigma X \omega \Rightarrow \cent^m S \$^n$ and therefore cannot occur, since by hypothesis there is no rightmost derivation $S \overset{+}{\Rightarrow} S$ in G. Hence G_1 is an (m, n) BRC grammar. □

[11] Notice that Lemma 4.1 and Corollary 4.2 are false for $n = k = 0$. A counterexample is provided by $G = (\{S, a\}\{a\}, \{S \to Sa|a\}, S)$.

COROLLARY 4.2.[12] *Let* $G = (V, V_T, P, S)$ *be an* LR(k) *grammar and let* $S_0 \notin V$. *Then* $G_1 = (V \cup \{S_0\}, V_T, P \cup \{S_0 \to S\}, S_0)$ *is an* LR(k) *grammar.*

Proof. The proof is analogous to the proof of Lemma 4.1 (with $\not\in^m$'s deleted). □

We next show that for languages not containing λ, the inclusion of λ-rules in (m, n) BRC grammars is inessential. As the reader can verify, a much simpler transformation suffices to preserve the LR(k) property alone. Therefore an alternate proof of this result would be to use the LR(k) transformation and to appeal to Theorem 3.5. However, a direct proof yields a transformed grammar more closely related to the original and a better understanding of the role of λ-rules in (m, n) BRC grammars.

THEOREM 4.3.[13] *Given an* (m, n) BRC *grammar* $G = (V, V_T, P, S)$ $(m, n \geqq 0)$, *an* (m, n) BRC *grammar without* λ*-rules can be constructed which generates the language* $L(G) - \{\lambda\}$.

Proof. For the sake of clarity, we present the transformation for λ-rule elimination as a sequence of less complex transformations.

We first transform the grammar G to $G_1 = \{V \cup \{S_0\}, V_T, P \cup \{S_0 \to S\}, S_0\}$, where $S_0 \notin V$ and G_1 has no occurrence of the initial symbol S_0 in the right part of any rule. This is a temporary device, to avoid having multiple initial symbols in intermediate stages of the overall transformation. The effect of this first transformation is later "undone". By Lemma 4.1, $L(G_1) = L(G)$, and if $n > 0$, then G_1 is an (m, n) BRC grammar. It follows from the proof of Lemma 4.1 that for $n = 0$, the only possible violations of the $(m, 0)$ BRC condition are cases in which one of the relevant derivations is $\not\in^m S_0 \Rightarrow \not\in^m S$.

Let $\Lambda = \{A \in V - V_T | A \overset{+}{\Rightarrow} \lambda \text{ in } G_1\}$. (Notice that $S_0 \notin \Lambda$ because $S_0 \notin V$). We next transform G_1 to a grammar in which every nonterminal which generates λ generates *only* λ. We do this by creating a dual symbol A_λ for each element A in Λ and transferring the derivation of λ from each element of Λ to its dual. Let $V_\lambda = \{A_\lambda | A \in \Lambda\}$ be the new alphabet of dual symbols. Define a homomorphism $A \to \bar{A}$ from $V \cup \{S_0\} \cup V_\lambda$ to $V \cup \{S_0\}$ by

$$\bar{A} = A \quad \text{for each } A \in V \cup \{S_0\},$$

$$\bar{A}_\lambda = A \quad \text{for each } A_\lambda \in V_\lambda.$$

Thus each element of V_λ is mapped onto its corresponding element of Λ and each element of $V \cup \{S_0\}$ is mapped onto itself.

Transform G_1 to $G_2 = (V \cup \{S_0\} \cup V_\lambda, V_T, P_2, S_0)$, where P_2 is formed as follows: for each rule $A \to x$ in $P \cup \{S_0 \to S\}$, add to P_2 all the productions $A \to y$ such that $y \in (V \cup V_\lambda)^*$ and $\bar{y} = x$. Thus $P \cup \{S_0 \to S\} \subseteq P_2$ and, in addition, P_2 contains all rules obtained by taking a rule in $P \cup \{S_0 \to S\}$ and substituting dual symbols for one or more occurrences of elements of Λ in the right part of the rule. Since $P \cup \{S_0 \to S\} \subseteq P_2$ and since the elements of V_λ are not left parts of any rules in P_2, $L(G_2) = L(G_1)$. Furthermore, it is easily shown that for any rightmost derivation $\not\in^m S_0 \$^n \overset{*}{\Rightarrow} \sigma X \omega \Rightarrow \sigma x \omega$ in G_2 there is a corresponding derivation $\not\in^m S_0 \$^n \overset{*}{\Rightarrow} \bar{\sigma} X \omega \Rightarrow \bar{\sigma} \bar{x} \omega$ in G_1. Since the homomorphism is length-preserving,

[12] See footnote 11.

[13] A somewhat similar construction is used in [9] to prove a similar result for LL(k) grammars.

G_2 fails to be (m, n) BRC only if G_1 fails to be (m, n) BRC. Furthermore, if $n = 0$, the only possible violations of the $(m, 0)$ BRC condition are cases in which one of the relevant derivations is either $\not\vdash^m S_0 \Rightarrow \not\vdash^m S$ or $\not\vdash^m S_0 \Rightarrow \not\vdash^m S_\lambda$.

Having introduced the elements of V_λ into the (unreduced) grammar G_2, we modify this grammar so that λ is generated only by elements of V_λ.

Transform G_2 to $G_3 = (V \cup \{S_0\} \cup V_\lambda, V_T, P_3, S_0)$, where P_3 is formed as follows:

(4.1) For each rule $S_0 \to x$ in P_2, $S_0 \to x$ is in P_3. (Notice that as a consequence of the previous transformations, $x \in \{S, S_\lambda\}$.)

(4.2) For each rule $A \to x$ in P_2 such that $A \in V$, if $x \in V_\lambda^*$, then $A_\lambda \to x$ is in P_3 (where A_λ is the element of V_λ corresponding to A), otherwise $A \to x$ is in P_3.

P_3 has the same number of rules as P_2, and the rules have the same form except that in some instances the left part of a rule is replaced by its dual symbol. By the usual induction on the length of the derivation, it follows that

(a) for any rightmost derivation $X \overset{*}{\Rightarrow} x$ in G_3,
  (a1) if $X \in V$, then $x \in (V \cup V_\lambda)^* - V_\lambda^*$;
  (a2) if $X \in V_\lambda$, then $x \in V_\lambda^*$;
  (a3) if $X \overset{*}{\Rightarrow} x$ in G_3, then there is a rightmost derivation $\overline{X} \overset{*}{\Rightarrow} y$ in G_2 for every $y \in (V \cup V_\lambda)^*$ such that $\bar{x} = \bar{y}$;

(b) for any rightmost derivation $X \overset{*}{\Rightarrow} x$ in G_2,
  (b1) if $x \in V_\lambda^*$, then there is a rightmost derivation $X_\lambda \overset{*}{\Rightarrow} x$ in G_3 (where X_λ is the element of V_λ corresponding to X);
  (b2) if $x \in (V \cup V_\lambda)^* - V_\lambda^*$, then there is a rightmost derivation $X \overset{*}{\Rightarrow} x$ in G_3.

It follows from (4.1) of the transformation and (a) and (b) above that $L(G_2) = L(G_3)$. It follows from (a1) that for any $X \in V$, there is no derivation $X \overset{*}{\Rightarrow} \lambda$ in G_3. It follows from (a2) that for any $X \in V_\lambda$, $\{x \in V_T^* | X \overset{*}{\Rightarrow} x \text{ in } G_3\} = \{\lambda\}$. Suppose G_3 is not an (m, n) BRC grammar. By construction, there is no derivation $S_0 \overset{*}{\Rightarrow} S_0$ in G_3, since S_0 does not occur as the right part of any rule. Therefore, for appropriate $\tau_1, \tau_2, \sigma, t, v, v_1, v_2, \omega, U \to u$ and $X \to x$, there are rightmost derivations

$$\not\vdash^m S_0 \$^n \overset{*}{\Rightarrow} \tau_1 t U v v_1 \Rightarrow \tau_1 t u v v_1$$

and

$$\not\vdash^m S_0 \$^n \overset{*}{\Rightarrow} \sigma X \omega \Rightarrow \sigma x \omega = \tau_2 t u v v_2$$

in G_3, where $|\omega| \leqq |v v_2|$ and either $\sigma \neq \tau_2 t$, or $X \neq U$, or $|\omega| < |v v_2|$. By (a3), there are corresponding derivations

$$\not\vdash^m S_0 \$^n \overset{*}{\Rightarrow} \tau_1 t \overline{U} v v_1 \Rightarrow \tau_1 t u v v_1$$

and

$$\not\vdash^m S_0 \$^n \overset{*}{\Rightarrow} \sigma \overline{X} \omega \Rightarrow \sigma x \omega = \tau_2 t u v v_2$$

in G_2, where $|\omega| \leqq |v v_2|$. If $\sigma \neq \tau_2 t$ or $\omega \neq v v_2$, then G_2 is not an (m, n) BRC grammar. If $\sigma = \tau_2 t$ and $\omega = v v_2$, then $U \to u$ and $X \to u$ are the rules used in the derivations in G_3. If $u \in V_\lambda^*$, then $X, U \in V_\lambda$. If $u \in (V \cup V_\lambda)^* - V_\lambda^*$, then $X, U \in V$. In either case, $U \neq X$ only if $\overline{U} \neq \overline{X}$, in which case G_2 is not an (m, n)

BRC grammar. Since any violation of the (m, n) BRC condition for G_3 leads to a corresponding violation for G_2, we can conclude that if $n > 0$, then G_3 is an (m, n) BRC grammar, and if $n = 0$, then the only possible violations of the $(m, 0)$ BRC condition are cases in which one of the relevant derivations is either $\not\in^m S_0 \Rightarrow \not\in^m S$ or $\not\in^m S_0 \Rightarrow \not\in^m S_\lambda$.

Now we will eliminate λ from the language generated by G_3, and at the same time reverse the change made by the first transformation. Let

$$G_4 = (V \cup V_\lambda, V_T, P_3 - \{S_0 \to S, S_0 \to S_\lambda\}, S\}.$$

The only difference between G_3 and G_4 is that we have changed the initial symbol from S_0 to S and deleted all rules with left part S_0. It is easily seen that for every $x \in (V \cup V_\lambda)^* - V_\lambda^*$, $S \overset{*}{\Rightarrow} x$ is a (rightmost) derivation in G_4 if and only if $S_0 \Rightarrow S \overset{*}{\Rightarrow} x$ is a (rightmost) derivation in G_3. Since S_λ generates only λ in G_3, it follows that $L(G_4) = L(G_3) - \{\lambda\}$ and G_4 is an (m, n) BRC grammar (including the case $n = 0$).

Now that we have isolated the use of λ-rules (i.e., only in derivations from V_λ) and eliminated λ from the language, we are ready to eliminate the λ-rules. However, it is necessary to preserve the left context information transmitted by the elements of V_λ. Therefore, as before, we will incorporate left-context information in the new nonterminal symbols we use. Let

$$V_{[]} = \{[B_{i_1} \cdots B_{i_k} A] | A \in V, 1 \leqq k \leqq m \quad \text{and} \quad 1 \leqq j \leqq k, B_{i_j} \in V_\lambda\}$$

be a new alphabet. Define a mapping $\alpha : (V \cup V_\lambda)^* \to (V \cup V_{[]})^*$ by

(c1) for each $x \in V_\lambda^*$, $\alpha(x) = \lambda$;

(c2) for each xAy, $x \in V_\lambda^+$, $A \in V$, $y \in (V \cup V_\lambda)^*$, $\alpha(xAy) = [\text{last}_m(x)A]\alpha(y)$;

(c3) for each Ay, $A \in V$, $y \in (V \cup V_\lambda)^*$, $\alpha(Ay) = A\alpha(y)$.

The effect of the mapping on a string in $(V \cup V_\lambda)^*$ is to replace any subsequence of one or more elements of V_λ followed by an element of V by a bracketed symbol containing the last m elements of V_λ and the element of V, and to drop all trailing (rightmost) elements of V_λ in the string. More formally, we have the following.

FACT 5. *Let $u \in (V \cup V_\lambda)^+$. That is, for some $k \geqq 0$, $x_1, x_2, \cdots, x_k, x_{k+1} \in V_\lambda^*$ and $A_1, A_2, \cdots, A_k \in V$, $u = x_1 A_1 x_2 A_2 \cdots x_k A_k x_{k+1}$. Then*

(a) $\alpha(u) = \alpha(x_1 A_1)\alpha(x_2 A_2) \cdots \alpha(x_k A_k)$, *where for* $1 \leqq i \leqq k$,

$$\alpha(x_i A_i) = \begin{cases} A_i & \text{if } x_i = \lambda, \\ [\text{last}_m(x_i)A_i] & \text{if } x_i \neq \lambda; \end{cases}$$

(b) *for* $1 \leqq i \leqq k$,

$$\text{last}_{m+1}(x_1 A_1 x_2 A_2 \cdots x_i A_i) = \text{last}_{m+1}(\text{last}_m(x_1)A_1 \, \text{last}_m(x_2)A_2 \cdots \text{last}_m(x_i)A_i).$$

Proof. The proof proceeds by induction on k.

Furthermore, any string x in $(V \cup V_{[]})^*$ is the image under α only of some string which can be obtained by inserting a sequence of elements of V_λ before every bracketed symbol in x and then erasing the brackets. More formally, let h be a bracket-erasing homomorphism. That is, for each $A \in V$, $h(A) = A$; for each $[yA] \in V_{[]}$, $h([yA]) = yA$.

FACT 6. *Let* $u \in (V \cup V_{[]})^*$. *That is, for some* $k \geqq 0$ *and* $C_1, C_2, \cdots, C_k \in V \cup V_{[]}$, $u = C_1 C_2 \cdots C_k$. *Let* u' *be any string in* $(V \cup V_\lambda)^*$ *such that* $u = \alpha(u')$. *Then*

(a) *for some* $x_1, x_2, \cdots, x_k, x_{k+1} \in V_\lambda^*$, $u' = x_1 h(C_1) x_2 h(C_2) \cdots x_k h(C_k) x_{k+1}$, *where, for* $1 \leqq i \leqq k$, *if* $C_i \in V$, *then* $x_i = \lambda$;

(b) *for* $1 \leqq i \leqq k$,

$$\text{last}_{m+1}(x_1 h(C_1) x_2 h(C_2) \cdots x_i h(C_i)) = \text{last}_{m+1}(h(C_1 C_2 \cdots C_i)).$$

Proof. The proof is by induction on k.

Next we transform G_4 to G_5 in such a way that the mapping α extends to rightmost sentential forms.

Let $G_5 = (V \cup V_{[]}, V_T, P_5, S)$, where P_5 is formed as follows.

(4.3) For each $X \to x$ in P_4,

(4.3a) $X \to \alpha(x)$ is in P_5,

(4.3b) for all $[yX] \in V_{[]}$, $[yX] \to \alpha(yx)$ is in P_5.

(4.4) For each $a \in V_T$, for all $[ya] \in V_{[]}$, $[ya] \to a$ is in P_5.

This transformation bears a strong resemblance to the transformation used in the first proof of Lemma 3.3. In that case, encoded within every bracketed nonterminal were the up to m symbols which preceded it in every rightmost sentential form in which it occurred. In this case, encoded in every bracketed nonterminal are the up to m symbols of V_λ which would precede it in each corresponding rightmost sentential form of G_4. However, the symbols of V_λ do not occur in rightmost sentential forms of G_5 except as auxiliary encoding, and furthermore, for any sequence of more than m symbols of V_λ in a rightmost sentential form of G_4, only the last m are even represented by the encoding. (This does not affect the terminal strings which are generated because the elements of V_λ generate only λ in G_4.)

We must now show that $L(G_5) = L(G_4)$, that G_5 has no λ-rules, and that G_5 is an (m, n) BRC grammar.

We know that for every $X \in V - V_T$ and $X \to x$ in P_4, $x \in (V \cup V_\lambda)^* - V_\lambda^*$ (i.e., in the statement of Fact 5, $k > 0$). Therefore, by Fact 5, step (4.3) of the transformation to G_5 creates no λ-rules. Clearly step (4.4) creates no λ-rules. Therefore G_5 has no λ-rules. To prove the other conditions, we must use the correspondence between derivations in G_4 and derivations in G_5. We now formalize the remarks following the transformation.

FACT 7. *For every* $A \in V$ *and* $x \in (V \cup V_\lambda)^*$, *if there is a rightmost derivation* $A \overset{*}{\Rightarrow} x$ *in* G_4, *then*

(a) *there is a rightmost derivation* $A \overset{*}{\Rightarrow} \alpha(x)$ *in* G_5;

(b) *for all* $[yA] \in V_{[]}$, *there is a rightmost derivation* $[yA) \overset{*}{\Rightarrow} \alpha(yx)$ *in* G_5.

Proof. The proof is by induction on the length of a derivation in G_4.

FACT 8. *For every* $A \in V$, $[yA] \in V_{[]}$ *and* $x \in (V \cup V_{[]})^*$,

(a) *if there is a rightmost derivation* $A \overset{*}{\Rightarrow} x$ *in* G_5, *then there is a rightmost derivation* $A \overset{*}{\Rightarrow} x'$ *in* G_4 *for some* $x' \in (V \cup V_\lambda)^*$ *such that* $\alpha(x') = x$;

(b) *if there is a rightmost derivation* $[yA] \overset{*}{\Rightarrow} x$ *in* G_5, *then there is a rightmost derivation* $A \overset{*}{\Rightarrow} x'$ *in* G_4 *for some* $x' \in (V \cup V_\lambda)^*$ *such that* $\alpha(yx') = x$.

Proof. The proof is by induction on the length of a derivation in G_5.

It follows from Fact 5 that for every $x \in V_T^*$, $\alpha(x) = x$. Therefore we can conclude from Fact 7 that $L(G_4) \subseteq L(G_5)$. It follows from Fact 6 that for every

$x \in V_T^*$, $\{x' \in (V \cup V_\lambda)^* | \alpha(x') = x\} = \{x\}$. Therefore we can conclude from Fact 8 that $L(G_5) \subseteq L(G_4)$. Hence $L(G_5) = L(G_4)$. It is easily shown using Facts 6 and 8 that for every $x \in (V \cup V_\lambda)^*$, $y \in (V \cup V_{[\,]})^*$ such that $\alpha(x) = y$, the set of terminal strings generated by x in G_4 is the same as the set of terminal strings generated by y in G_5.

Suppose G_5 is not an (m, n) BRC grammar. Then there are rightmost derivations

$$(4.5) \qquad ¢^m S \$^n \overset{*}{\Rightarrow} \tau_1 t U v v_1 \Rightarrow \tau_1 t u v v_1$$

and

$$(4.6) \qquad ¢^m S \$^n \overset{*}{\Rightarrow} \sigma X \omega \Rightarrow \tau_2 t u v v_2$$

in G_5, where $|\omega| \leqq |vv_2|$ and either $\sigma \neq \tau_2 t$ or $X \neq U$ or $|\omega| < |vv_2|$. The rightmost derivations in G_4 corresponding to (4.5) are:

(4.5.1) If $U \in V$, then

$$¢^m S \$^n \overset{*}{\Rightarrow} \tau'_1 t' U v v_1 \Rightarrow \tau'_1 t' u' v v_1,$$

where $\alpha(\tau'_1) = \tau_1$, $\alpha(\tau'_1 t') = \tau_1 t$, $\text{last}_{m+1}(t') = \text{last}_{m+1}(h(t))$, $\alpha(u') = u$;

(4.5.2) if $U = [yU']$, $U' \in V - V_T$, then

$$¢^m S \$^n \overset{*}{\Rightarrow} \tau'_1 t' y' U' v v_1 \Rightarrow \tau'_1 t' y' u' v v_1,$$

where $\alpha(\tau'_1) = \tau_1$, $\alpha(\tau'_1 t') = \tau_1 t$, $\text{last}_{m+1}(t') = \text{last}_{m+1}(h'(t))$, $\text{last}_m(y') = y$, $y' \in V_\lambda^+$, $\alpha(y'u') = u$;

(4.5.3) if $U = [yu]$, $u \in V_T$, then

$$¢^m S \$^n \overset{*}{\Rightarrow} \tau'_3 U'' v_3 \Rightarrow \tau'_3 u'' v_3 = \tau'_1 t' y' u v'_1 v_2 v_1,$$

where $\alpha(\tau'_1) = \tau_1$, $\alpha(\tau'_1 t') = \tau_1 t$, $v = v_1 v_2$, $\alpha(v'_1) = v_1$, $\text{last}_{m+1}(t') = t$, $\text{last}_m(y') = y$. The last step in this derivation is the step in which u is generated (and corresponds to a previous step in (4.5)).

There is an analogous correspondence between (4.6) and rightmost derivations in G_4. By a rather lengthy case analysis, it can be shown that for every possible pair of derivations (4.5), (4.6), in G_5, the corresponding derivations in G_4 provide a contradiction to the fact that G_4 is an (m, n) BRC grammar. In some cases, the derivations immediately violate the (m, n) BRC condition. In the other cases, the two derivations are seemingly incomparable, but (because of Facts 7 and 8) the differences are only in the occurrence of elements of V_λ. Therefore by extending both derivations (i.e. considering subsequent steps), both derivations are found, in all these cases, to have the form which, by Fact 9 below, contradicts the (m, n) BRC assumption. (Fact 9 says, in essence, that in an (m, n) BRC grammar there cannot be two rightmost derivations of λ in the same (m, n) context.)

FACT 9. *For every* $\tau_1, \tau_2, t_1, t_2 \in (V \cup V_\lambda)^*$, $x_1, y_1 \in V_\lambda^*$, $U_1, U_2 \in V$, *and* $v_1, v_2, \mathrm{v}_1, \mathrm{v}_2 \in V_T^*$, *if* G_4 *contains rightmost derivations*

$$S \overset{*}{\Rightarrow} \tau_1 U_1 \mathrm{v}_1 \Rightarrow t_1 x_1 v_1 \quad \textit{and} \quad S \overset{*}{\Rightarrow} \tau_2 U_2 \mathrm{v}_2 \Rightarrow t_2 y_1 v_2,$$

where $\text{last}_m(t_1) = \text{last}_m(t_2)$, $\text{first}_n(v_1) = \text{first}_n(v_2)$, $|\mathrm{v}_1| \leqq |v_1|$ *and* $|\mathrm{v}_2| \leqq |v_2|$, *then* $x_1 = y_1$.

Proof. Suppose $x_1 \neq y_1$. Since $x_1, y_1 \in V_\lambda^*$, the derivations can be extended as:

$$S \overset{*}{\Rightarrow} \tau_1 U_1 v_1 \Rightarrow t_1 x_1 v_1 \Rightarrow t_1 x_2 v_1 \Rightarrow \cdots \Rightarrow t_1 x_p v_1 \Rightarrow t_1 v_1$$

and

$$S \overset{*}{\Rightarrow} \tau_2 U_2 v_2 \Rightarrow t_2 y_1 v_2 \Rightarrow t_2 y_2 v_2 \Rightarrow \cdots \Rightarrow t_2 y_q v_2 \Rightarrow t_2 v_2,$$

where $p, q \geqq 1$ and for $1 \leqq i \leqq p$, $1 \leqq j \leqq q$, x_1, $y_j \in V_\lambda^*$ and for $1 \leqq i < p$, $1 \leqq j < q$, $x_i \Rightarrow x_{i+1}$, $y_j \Rightarrow y_{j+1}$, and each rule used has left part in V_λ. Compare these derivations step by step from right to left, erasing each step as long as both derivations use the same production. There must be some step at which each derivation uses a different production. Suppose $p \geqq q$ (the case $p < q$ is symmetric). Then at the point at which the derivations differ, the first derivation has the form

$$S \overset{*}{\Rightarrow} \tau_1 U_1 v_1 \overset{*}{\Rightarrow} t_1 z X v_1 \Rightarrow t_1 z w v_1 = t_1 x_i v_1,$$

$1 \leqq i \leqq p$, $z, w \in V_\lambda^*$, and the second derivation has the form

$$S \overset{*}{\Rightarrow} \tau_2 U_2 v_2 \overset{*}{\Rightarrow} \tau_3 Y v_3 \Rightarrow \tau_3 y v_3 = t_2 z w v_2 = t_2 y_j v_2,$$

$1 \leqq j \leqq q$, where $\tau_3, y \in (V \cup V_\lambda)^*$ and $v_3 \in V_T^*$. If $\tau_2 U_2 v_2 = \tau_3 Y v_3$, then $v_3 = v_2$; otherwise $v_3 = v_2$. Therefore $|v_3| \leqq |v_2|$. The existence of such derivations contradicts the fact that G_4 is an (m, n) BRC grammar. □

COROLLARY 4.4. *Given an* LR(k) *grammar* $G = (V, V_T, P, S)$ *where* $k \geqq 0$, *an* LR(k) *grammar without* λ*-rules can be constructed which generates the language* $L(G) - \{\lambda\}$.

Proof. The transformations to G_4 are easily seen to be LR(k)-preserving, since the finiteness of the left context is inessential. In the transformation to G_5, m may be chosen arbitrarily. (If $m = 0$, then no bracketed symbols are needed.) Exploiting the fact that any two elements of $(V \cup V_\lambda)^*$ mapped by α into the same string generate the same set of terminals, an argument analogous to that of the theorem proves that the transformation is LR(k)-preserving. □

Example. We carry out the transformation for $G = (\{S, X, Y, A, B, a, b, c, d\}, \{a, b, c, d\}, P, S)$ where

$$\begin{aligned} P: \quad & S \to AdX \mid Y \\ & A \to aA \mid \lambda \\ & X \to AbX \mid \lambda \\ & Y \to BAcY \mid \lambda \\ & B \to \lambda. \end{aligned}$$

G is a (1, 1) BRC grammar.

P_1: Add rule $S_0 \to S$ to G.

$\Lambda = \{S, A, X, Y, B\}$.

P_2: $S_0 \to S \mid S_\lambda$

$S \to AdX \mid A_\lambda dX \mid AdX_\lambda \mid A_\lambda dX_\lambda \mid Y \mid Y_\lambda$

$A \to aA \mid aA_\lambda \mid \lambda$

$$X \to AbX|A_\lambda bX|AbX_\lambda|A_\lambda bX_\lambda|\lambda$$

$$Y \to BAcY|BAcY_\lambda|BA_\lambda cY|BA_\lambda cY_\lambda|B_\lambda AcY|B_\lambda AcY_\lambda|B_\lambda A_\lambda cY|B_\lambda A_\lambda cY_\lambda|\lambda$$

$$B \to \lambda$$

P_4:(reduced),

$$S \to AdX|A_\lambda dX|AdX_\lambda|A_\lambda dX_\lambda|Y$$

$$A \to aA|aA_\lambda$$

$$A_\lambda \to \lambda$$

$$X \to AbX|A_\lambda bX|AbX_\lambda|A_\lambda bX_\lambda$$

$$X_\lambda \to \lambda$$

$$Y \to B_\lambda AcY|B_\lambda AcY_\lambda|B_\lambda A_\lambda cY|B_\lambda A_\lambda cY_\lambda$$

$$Y_\lambda \to \lambda$$

$$B_\lambda \to \lambda$$

P_5: $S \to AdX \mid [A_\lambda d]X \mid Ad \mid [A_\lambda d] \mid Y$

$$A \to aA|a$$

$$X \to AbX \mid [A_\lambda b]X \mid Ab \mid [A_\lambda b]$$

$$Y \to [B_\lambda A]cY \mid [B_\lambda A]c \mid [A_\lambda c]Y \mid [A_\lambda c]$$

$$[B_\lambda A] \to [B_\lambda a]A \mid [B_\lambda a]$$

$$[B_\lambda a] \to a$$

$$[A_\lambda b] \to b$$

$$[A_\lambda c] \to c$$

$$[A_\lambda d] \to d$$

As was the case in § 3, the transformation can be further modified so that it produces smaller grammars. (For example, none of the bracketed context in the previous example is needed for the (1, 1) BRC property.)

The preservation of structure is somewhat less than in § 3. As can be seen from Facts 7 and 8, in going from G_4 to G_5, arbitrarily long sequences of λ-rule steps, and consequently, arbitrarily large subtrees of syntax trees, are lost. For instance, with reference to the previous example, syntax trees for ac are:

5. Comparison with Floyd's definition. Floyd's original formulation of bounded right context grammars [2] is somewhat different from ours. In this section we show that the two formulations are essentially equivalent, differing only in the use of the initial symbol and the occurrence of λ-rules in the grammars and in the inclusion of λ in the generated languages.

The following is the definition of bounded right context given by Floyd [2].

DEFINITION. Let $G = (V, V_T, P, S)$ be a context-free grammar with no λ-rules and no right parts containing S. Let $\not\subset$, \$ be symbols not in V. For any production $U \to u$ in P, $U \to u$ is an (m, n) FBRC (*Floyd bounded right context*) *production* $(m, n \geqq 0)$ if for any $t, t_1 \in \{\not\subset\}^* V^*$, $t_2, u, u_1, u_2, y \in V^+$, $v_1 \in V_T^+$, $v_2, v \in V_T^*\{\$\}^*$ such that $t = t_1 t_2$, $u = u_1 u_2$, $v = v_1 v_2$, $|t| = m$, $|v| = n$ and $\not\subset^m S\$^n \overset{*}{\Rightarrow} \cdots tUv \cdots$[14] is a rightmost derivation, the following 12 cases are unsatisfiable for any rightmost derivations and productions in G and any $X \in V$:

(BR1) $\not\subset^m S\$^n \overset{*}{\Rightarrow} \cdots Xv \cdots$, $X \to \cdots tu$,

(BR2) $\not\subset^m S\$^n \overset{*}{\Rightarrow} \cdots Xv_2 \cdots$, $X \to \cdots tuy, y \overset{*}{\Rightarrow} v_1$,

(BR3) $\not\subset^m S\$^n \overset{*}{\Rightarrow} \cdots X \cdots$, $X \to \cdots tuy, y \overset{*}{\Rightarrow} v \cdots$,

(BR4) $\not\subset^m S\$^n \overset{*}{\Rightarrow} \cdots t_1 Xv \cdots$, $X \to t_2 u$,

(BR5) $\not\subset^m S\$^n \overset{*}{\Rightarrow} \cdots t_1 Xv_2 \cdots$, $X \to t_2 uy, y \overset{*}{\Rightarrow} v_1$,

(BR6) $\not\subset^m S\$^n \overset{*}{\Rightarrow} \cdots t_1 X \cdots$, $X \to t_2 uy, y \overset{*}{\Rightarrow} v \cdots$,

(BR7) $\not\subset^m S\$^n \overset{*}{\Rightarrow} \cdots tXv \cdots$, $X \to u, X \neq U$,

(BR8) $\not\subset^m S\$^n \overset{*}{\Rightarrow} \cdots tXv_2 \cdots$, $X \to uy, y \overset{*}{\Rightarrow} v_1$,

(BR9) $\not\subset^m S\$^n \overset{*}{\Rightarrow} \cdots tX \cdots$, $X \to uy, y \overset{*}{\Rightarrow} v \cdots$,

(BR10) $\not\subset^m S\$^n \overset{*}{\Rightarrow} \cdots tu_1 Xv \cdots$, $X \to u_2$,

(BR11) $\not\subset^m S\$^n \overset{*}{\Rightarrow} \cdots tu_1 Xv_2 \cdots$, $X \to u_2 y, y \overset{*}{\Rightarrow} v_1$,

(BR12) $\not\subset^m S\$^n \overset{*}{\Rightarrow} \cdots tu_1 X \cdots$, $X \to u_2 y, y \overset{*}{\Rightarrow} v \cdots$.

G is an (m, n) FBRC *grammar* $(m, n \geqq 0)$ if every production $U \to u$ in P is an (m', n') FBRC production for some $m' \leqq m$, $n' \leqq n$.

G is an FBRC *grammar* if, for some $m, n \geqq 0$, G is an (m, n) FBRC grammar. We designate by (m, n) *FBRC* the class of languages generated by (m, n) FBRC grammars, and by *FBRC* the class of languages generated by FBRC grammars.

In § 4, we considered the differences in the two definitions of bounded right context stemming from occurrences of the initial symbol and inclusion of λ-rules. By comparing the two definitions in the absence of λ-rules and occurrences of the initial symbol in right parts, we get the following theorem.

THEOREM 5.1. (a) *Every* (m, n) BRC *grammar with no* λ*-rules and no right parts containing the initial symbol is an* (m, n) FBRC *grammar* $(m, n \geqq 0)$.

(b) *Every* (m, n) FBRC *grammar is an* (m, n) BRC *grammar* $(m, n \geqq 0)$.

(c) (m, n) *FBRC* $= \{L - \{\lambda\} | L \in (m, n)BRC\}$ $(m, n > 0)$.

[14] Here $\cdots$ denotes any element of $\{\not\subset\}^* V^* \{\$\}^*$.

Proof. (a) Let $G = (V, V_T, P, S)$ be an (m, n) BRC grammar with no λ-rules and no right parts containing S. For every rule $U \to u$ and every t, v such that $|t| = m$, $|v| = n$ and $\not\in^m S\$^n \overset{*}{\Rightarrow} \tau_1 tUvv_1 \Rightarrow \tau_1 tuvv_1$ is a rightmost derivation in G, each of the FBRC conditions can be expressed as a rightmost derivation

$$\not\in^m S\$^n \overset{*}{\Rightarrow} \sigma_1 X_1 \omega_1 \Rightarrow \tau_2 tuy\pi \overset{*}{\Rightarrow} \tau_2 tuvv_2,$$

where $\pi \in V_T^*\{\$\}^*$, $y\pi \overset{*}{\Rightarrow} vv_2$ and $|\omega_1| \leqq |y\pi| \leqq |vv_2|$ (since G has no λ-rules). If $y\pi = vv_2$, then this derivation has the form

$$\not\in^m S\$^n \overset{*}{\Rightarrow} \sigma_1 X_1 \omega_1 \Rightarrow \tau_2 tuvv_2, \quad \text{where } |\omega_1| \leqq |vv_2|. \tag{5.1}$$

If $y\pi \overset{+}{\Rightarrow} vv_2$, then this derivation has the form

$$\not\in^m S\$^n \overset{*}{\Rightarrow} \sigma_1 X_1 \omega_1 \Rightarrow \tau_2 tuy\pi \overset{*}{\Rightarrow} \tau_2 tu\sigma_2 X_2 \omega_2 \pi \Rightarrow \tau_2 tuvv_2, \tag{5.2}$$

where $|\sigma_2 X_2 \omega_2 \pi| \leqq |vv_2|$ (since G has no λ-rules) and therefore $|\omega_2 \pi| \leqq |vv_2|$. Letting $\sigma = \sigma_1, \omega = \omega_1, x = x_1$ in (5.1) and letting $\sigma = \tau_2 tu\sigma_2, \omega = \omega_2 \pi, X = X_2$ in (5.2), it follows from the fact that G is an (m, n) BRC grammar that in any such derivation, $\sigma = \tau_2 t$, $\omega = vv_2$ and $X = U$. Since $\sigma = \tau_2 t$, (BR1)–(BR6) and (BR10)–(BR12) are unsatisfiable; since $\omega = vv_2$, (BR2), (BR3), (BR5), (BR6), (BR8), (BR9), (BR11) and (BR12) are unsatisfiable, and since $X = U$ (where $\sigma = \tau_2 t$ and $\omega = vv_2$), (BR7) is unsatisfiable. Therefore G is an (m, n) FBRC grammar.

(b) Let $G = (V, V_T, P, S)$ be an (m, n) FBRC grammar. Suppose G is not an (m, n) BRC grammar. Since G has no right parts containing S, there is no derivation $S \overset{+}{\Rightarrow} S$ in G. Hence for some rules $U \to u$, $X \to x$, and appropriate strings $t, v, \sigma, \omega, \tau_1, \tau_2, v_1, v_2$, $\not\in^m S\$^n \overset{*}{\Rightarrow} \tau_1 tUvv_1 \Rightarrow \tau_1 tuvv_1$ and $\not\in^m S\$^n \overset{*}{\Rightarrow} \sigma X\omega \Rightarrow \tau_2 tuvv_2 = \sigma x\omega$ are rightmost derivations in G, where $|\omega| \leqq |vv_2|$, but either (i) $\sigma \neq \tau_2 t$, (ii) $|\omega| < |vv_2|$, or (iii) $\sigma = \tau_2 t$, $\omega = vv_2$ and $X \neq U$.

We enumerate the possible partitionings of $\tau_2 tuvv_2$ into σ, x, ω by listing the possible strings equal to x, where $u = u_1 u_2$, $t = t_1 t_2$, $v = v_1 v_2$. For each x, we indicate the corresponding satisfiable FBRC condition.

1. Suppose $\omega = vv_2$.

x	satisfiable condition
$\cdots tu$	(BR1)
$t_2 u$	(BR4)
u and $X \neq U$	(BR7)
u_2	(BR10)

2. Alternatively, suppose $|\omega| < |vv_2|$.

x	satisfiable condition	
$\cdots tuv_1$	(BR2)	$y = v_1$
$\cdots tuv \cdots$	(BR3)	$y = v$
$t_2 uv_1$	(BR5)	$y = v_1$
$t_2 uv \cdots$	(BR6)	$y = v$
uv_1	(BR8)	$y = v_1$
$uv \cdots$	(BR9)	$y = v$
$u_2 v_1$	(BR11)	$y = v_1$
$u_2 v \cdots$	(BR12)	$y = v$

The final possibility if $|\omega| < |vv_2|$ is that x is some string to the right of u, that is, $\sigma = \tau_2 tu$ or $\sigma = \tau_2 tuv_1$ or $\sigma = \tau_2 tuv \cdots$.

In that case, $\tau_2 tu$ was generated by previous steps in the derivation. In particular, we can write the derivation as

$$\not{c}^m S\$^n \overset{*}{\Rightarrow} \delta Z\gamma \Rightarrow \delta z\gamma \overset{*}{\Rightarrow} \sigma X\omega \Rightarrow \tau_2 tuvv_2,$$

where $\gamma \in V_T^*\{\$\}^*$ and the step $\delta Z\gamma \Rightarrow \delta z\gamma$ generates the rightmost symbol of u. Since $\gamma \in V_T^*\{\$\}^*$, $\delta z \overset{*}{\Rightarrow} \sigma X \cdots$. Since $\sigma = \tau_2 tu \cdots$ and since the derivation is rightmost, $\delta z = \tau_2 tuy$ for some y such that $y\gamma \overset{*}{\Rightarrow} vv_2$.

$$\begin{aligned}
&\text{If } \sigma = \tau_2 tu, && \text{then } y\gamma \overset{*}{\Rightarrow} X\omega \Rightarrow vv_2;\\
&\text{If } \sigma = \tau_2 tuv_1, && \text{then } y\gamma \overset{*}{\Rightarrow} v_1 X\omega \Rightarrow vv_2;\\
&\text{If } \sigma = \tau_2 tuv \cdots, && \text{then } y\gamma \overset{*}{\Rightarrow} v \cdots X\omega \Rightarrow vv_2.
\end{aligned}$$

In each case, either $y \overset{+}{\Rightarrow} v \cdots$ or $y \overset{+}{\Rightarrow} v_1$, where $v = v_1 \cdots$. We enumerate the possible values for δ and z, and the corresponding satisfiable FBRC conditions.

δ	z	satisfiable condition
τ_{21}	$\tau_{22}tuy$, where $\tau_2 = \tau_{21}\tau_{22}$, $\tau_{22} \in \{\not{c}\}^* V^*$	(BR2) if $y \overset{+}{\Rightarrow} v_1$ (BR3) if $y \overset{+}{\Rightarrow} v \cdots$
$\tau_2 t_1$	$t_2 uy$	(BR5) if $y \overset{+}{\Rightarrow} v_1$ (BR6) if $y \overset{+}{\Rightarrow} v \cdots$
$\tau_2 t$	uy	(BR8) if $y \overset{+}{\Rightarrow} v_1$ (BR9) if $y \overset{+}{\Rightarrow} v \cdots$
$\tau_2 tu_1$	$u_2 y$	(BR11) if $y \overset{+}{\Rightarrow} v_1$ (BR12) if $y \overset{+}{\Rightarrow} v \cdots$

We have shown that for every way in which G could fail to be an (m, n) BRC grammar, some corresponding (m, n) FBRC condition would be satisfiable. Therefore, since G is an (m, n) FBRC grammar, G is an (m, n) BRC grammar.

(c) (m, n) *FBRC* $\subseteq$ (m, n) *BRC*, $(m, n \geqq 0)$ follows from part (b). From Lemma 4.1, Theorem 4.3, and part (a), it follows that for any (m, n) BRC grammar G, $(m \geqq 0, n > 0)$ $L(G) - \{\lambda\}$ is generated by an (m, n) FBRC grammar. This completes the assertion. □

6. Concluding remarks. The results we have presented (and especially the transformations) are particularly relevant for comparisons of methods of parsing deterministic context-free languages. A variety of arguments have been advanced for and against various parsing methods on the grounds that the methods do or do not admit λ-rules or do or do not work for a sufficiently large class of grammars, or do or do not require large amounts of storage for the parsing tables. If transformations between grammar classes which make minimal modifications and which preserve the desired grammatical properties are regarded as additional steps in constructing the parsers, then many of these arguments become vacuous. Furthermore, in comparing parsing methods, one must be careful which parameters

are being measured. An optimized version of the transformation of §4 to eliminate λ-rules may still yield a transformed grammar that is substantially larger than the original (primarily because of the G_1 to G_4 transformations). This may suggest that λ-rules are a good thing. However, the derivations in the transformed grammar may be substantially shorter because the steps involving λ-rules have been eliminated.

It should be clear that in any computer implementation of transformations such as those presented in this paper, computational efficiency can be achieved in a variety of ways. For example, incremental versions of the transformations can be designed to produce only reduced grammars. However, one of the observations to be made from the two proofs of Lemma 3.3 is that it is possible to devise transformations which are too incremental and that a careful analysis of the underlying reasons for changing a grammar may yield techniques which are faster to carry out and which yield smaller grammars.

Acknowledgment. The author is very grateful to Michael Hammer of MIT for his helpful comments, which contributed significantly to the clarity of the presentation.

REFERENCES

[1] A. V. AHO AND J. D. ULLMAN, *The Theory of Parsing, Translation and Compiling*, vols. I and II, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[2] R. W. FLOYD, *Bounded context syntactic analysis*, Comm. ACM, 7 (1964), pp. 62–66.

[3] S. L. GRAHAM, *Precedence languages and bounded right context languages*, CS 223, Computer Science Dept., Stanford Univ., Stanford, Calif., 1971.

[4] ———, *Extended precedence languages and grammars*, in preparation.

[5] ———, *Grammatical transformations for bottom-up parsing*, in preparation.

[6] M. A. HARRISON AND I. M. HAVEL, *On the parsing of deterministic languages*, J. Assoc. Comput. Mach., 21 (1974).

[7] D. E. KNUTH, *On the translation of languages from left to right*, Information and Control, 8 (1965), pp. 607–639.

[8] D. LEHMANN, *LR(k) grammars and deterministic languages*, Israel J. Math., 10 (1971), pp. 526–530.

[9] D. J. ROSENKRANTZ AND R. E. STEARNS, *Properties of deterministic top-down grammars*, Information and Control, (1970), pp. 226–256.

[10] NIKLAUS WIRTH AND HELMUT WEBER, *EULER: A generalization of ALGOL and its formal definition: Parts I and II*, Comm. ACM, 9 (1966), pp. 13–23, pp. 89–99.

SIAM J. COMPUT.
Vol. 3, No. 4, December 1974

THE NUMBER OF 1'S IN BINARY INTEGERS: BOUNDS AND EXTREMAL PROPERTIES*

M. D. McILROY†

Abstract. Closed formulas provide tight bounds for $G(n)$, the total number of 1's in the binary representations of integers less than n. This function satisfies an extremal recurrence, which gives the maximum cost of a process that creates a set of n objects by repeatedly merging pairs of smaller sets, starting from n singletons, incurring a cost equal to the size of the smaller set at each merger:

$$G(n) = \max_{1 \leqq i \leqq n/2} [i + G(i) + G(n - i)],$$

where $G(1) = 0$. The set of pairs $(i, n - i)$ at which the maximum is attained has an interesting structure.

Key words. binary numbers, extremal recurrences, set merging

1. Basic recurrences. Let $G(n)$ be the total number of 1's in the list of integers $0, 1, 2, \cdots, n - 1$ expressed in binary notation. Evidently

$$G(2^m) = \tfrac{1}{2}m2^m, \qquad m = 0, 1, \cdots, \tag{1}$$

since the list then consists of all 2^m m-bit patterns of 0's and 1's, among which half the bits are 1's. Given

$$G(1) = 0, \tag{2}$$

the definition of $G(n)$ may be extended to $n = 0, 1, \cdots$ by any of these recurrences.[1]

$$G(2^m + i) = G(2^m) + G(i) + i, \qquad 0 \leqq i \leqq 2^m, \quad m = 0, 1, \cdots, \tag{3}$$

$$G(2n) = n + 2G(n), \qquad n = 0, 1, \cdots, \tag{4a}$$

$$G(2n + 1) = n + G(n) + G(n + 1), \qquad n = 0, 1, \cdots, \tag{4b}$$

$$G(n) = \lfloor n/2 \rfloor + G(\lfloor n/2 \rfloor) + G(\lceil n/2 \rceil), \qquad n = 0, 1, \cdots, \tag{5}$$

$$G(2^m + i) = i(m + 1) + G(2^m - i), \quad 0 \leqq i \leqq 2^m, \quad m = 0, 1, \cdots. \tag{6}$$

(Some cases of (3) or (4) involving $G(0)$ are redundant; they are so written to simplify later calculations.) The recurrence (3) derives from the fact that the binary representation of $2^m + i$, $0 \leqq i < 2^m$, is just the m-bit representation of i prefixed by a 1.

To derive (4b), split the set $\{i | 0 \leqq i < 2n + 1\}$ into an odd part $\{2i + 1 | 0 \leqq i < n\}$ and an even part $\{2i | 0 \leqq i \leqq n\}$. The odd part has n final 1's plus the number of 1's in $\{2i | 0 \leqq i < n\}$, which is just $G(n)$. Similarly, the even part contains $G(n + 1)$ 1's, whence the total number of 1's is $n + G(n) + G(n + 1)$. Recursion (4a) may be derived similarly; (5) merely combines (4a) and (4b).

* Received by the editor September 27, 1973, and in revised form April 2, 1974.

† Bell Laboratories, Murray Hill, New Jersey 07974.

[1] Except for a factor of 2, (4) occurs in another context in [1].

Recursion (6) was pointed out by a referee. It follows from observing that the numbers $2^m + i - 1$ and $2^m - i$ are $(m + 1)$-bit 1's complements and that exactly i such complementary pairs make up the set $\{n|2^m - i \leqq n \leqq 2^m + i - 1\}$.

2. Bounds. The following theorem lends precision to the result $G(n) = \frac{1}{2}n \log_2 n + O(n)$, which was announced by Bellman and Shapiro [2]. The deviation of $G(n)$ from these bounds is shown in Fig. 1.

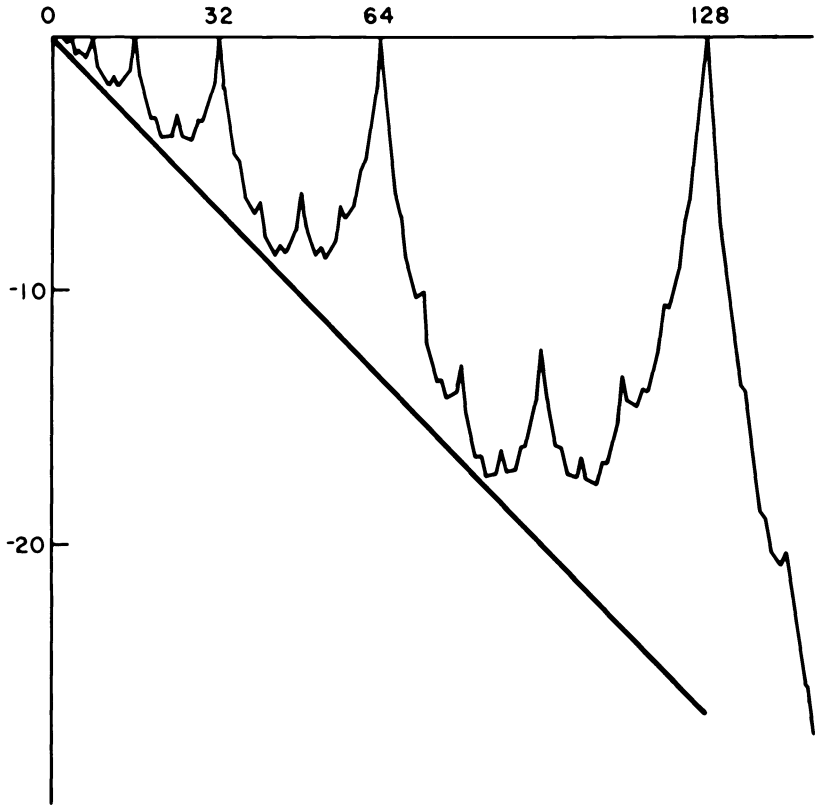


FIG. 1. *The deviation of $G(n)$ from its bounds. The curve represents $G(n) - \frac{1}{2}n \log_2 n$; the straight line is $\frac{1}{2}n \log_2 (\frac{3}{4}n) - \frac{1}{2}n \log_2 n$.*

THEOREM 1. *The function $G(n)$ defined above satisfies*

$$\lceil \tfrac{1}{2}n \log_2 (\tfrac{3}{4}n) \rceil \leqq G(n) \leqq \lfloor \tfrac{1}{2}n \log_2 n \rfloor, \qquad n = 0, 1, \cdots, \tag{7}$$

and each bound is tight for infinitely many values of n.[2]

The upper bound in (7) is evidently true for $n = 0$. Relation (1), $G(2^m) = \frac{1}{2}m2^m$, shows that the bound is valid and tight when n is a power of 2. Assuming the upper bound holds up to $n = 2^m$, we have by (3) and (1),

[2] We take $0 \log_2 0$ to be 0.

$$\begin{aligned} G(n+i) &= G(2^m) + G(i) + i && (0 \leqq i \leqq n = 2^m) \\ &\leqq \tfrac{1}{2}m2^m + \tfrac{1}{2}i \log_2 i + i \\ &= \tfrac{1}{2}n \log_2 n + \tfrac{1}{2}i \log_2 i + i \\ &\leqq \tfrac{1}{2}(n+i) \log_2 (n+i) && (0 \leqq i \leqq n = 2^m). \end{aligned}$$

The last inequality follows by considering the function

$$\tfrac{1}{2}(n+x) \log_2 (n+x) - (\tfrac{1}{2}n \log_2 n + \tfrac{1}{2}x \log_2 x + x),$$

which takes on the value 0 at $x = 0$ and $x = n$, and has a negative second derivative with respect to x throughout the open interval $(0, n)$. Thus $\tfrac{1}{2}(n+x) \log_2 (n+x)$ exceeds $\tfrac{1}{2}n \log_2 n + \tfrac{1}{2}x \log_2 x + x$ throughout that interval. The upper bound in (7) follows by induction on powers of 2.

The lower bound in (7) is evidently true for $n = 0$ and $n = 1$. Suppose that a bound of the form

$$G(n) \geqq \tfrac{1}{2}n \log_2 n - cn \tag{8}$$

is true for some $c > 0$ for all $n \leqq 2^m$. Then by (3) and (1),

$$\begin{aligned} G(n+i) &= G(2^m) + G(i) + i && (0 \leqq i \leqq n = 2^m) \\ &\geqq \tfrac{1}{2}m2^m + \tfrac{1}{2}i \log_2 i + i - ci \\ &= \tfrac{1}{2}n \log_2 n + \tfrac{1}{2}i \log_2 i + i - ci \end{aligned}$$

The desired result (8) follows by induction on m, provided that the following inequality holds for $0 \leqq i \leqq 2^m$:

$$\tfrac{1}{2}n \log_2 n + \tfrac{1}{2}i \log_2 i + i - ci \geqq \tfrac{1}{2}(n+i) \log_2 (n+i) - c(n+i).$$

Replace i by xn and simplify to obtain another relation whose truth would imply the result:

$$\tfrac{1}{2}x \log_2 x + x \geqq \tfrac{1}{2}(1+x) \log_2 (1+x) - c, \qquad 0 \leqq x \leqq 1.$$

Since both sides are continuous in x on $[0, 1]$, c can be chosen sufficiently large that the inequality is satisfied throughout the interval. By elementary calculus we find the smallest such c to be $\frac{1}{2} \log_2 \frac{4}{3}$, whence

$$G(n) \geqq \tfrac{1}{2}n \log_2 n - \tfrac{1}{2}n \log_2 \tfrac{4}{3}.$$

Since $G(n)$ takes on only integer values, we may round the right side up to the nearest integer, thus establishing the lower bound in (7).

The lower bound in (7) is tight for all n such that $|3n - 2^k| = 1$, $k = 0, 1, \cdots$, as may be verified directly for $k = 0$ and proved as follows for positive even k. A similar argument holds for odd k. Let $k = 2m + 2$, so that $v_m = \frac{1}{3}(2^{2m+2} - 1) = 2^{2m} + 2^{2m-2} + \cdots + 1$ is an integer satisfying $3v_m - 2^k = -1$. Since $v_m = 2^m + v_{m-1}$, we have by induction on (3) together with (1),

$$\begin{aligned} G(v_m) &= G(2^{2m}) + G(v_{m-1}) + v_{m-1} \\ &= \sum_{k=1}^{m} \frac{1}{2} \cdot 2k2^{2k} + G(v_0) + \sum_{k=0}^{m-1} v_k = \frac{m}{3}(2^{2m+2} - 1). \end{aligned}$$

For comparison, calculate

$$\begin{aligned}\tfrac{1}{2}v_m \log_2(\tfrac{3}{4}v_m) &= \tfrac{1}{2}\cdot\tfrac{1}{3}(2^{2m+2}-1)\log_2(\tfrac{1}{4}(2^{2m+2}-1))\\ &= \tfrac{1}{2}\cdot\tfrac{1}{3}(2^{2m+2}-1)[2m+\log_2(1-2^{-2m-2})]\\ &= \frac{m}{3}(2^{2m+2}-1)+\frac{1-2^{-2m-2}}{6\log 2}\sum_{j=1}^{\infty}\frac{(-1)^j}{j}2^{-2(m+1)(j-1)}\\ &= G(v_m)+\frac{1-2^{-2m-2}}{6\log 2}S,\end{aligned}$$

where S is the sum of the alternating series. For all $m \geqq 0$, $-1 < S \leqq 0$. It follows that

$$0 \leqq G(v_m) - \tfrac{1}{2}v_m \log_2(\tfrac{3}{4}v_m) \leqq \frac{1}{6\log 2}.$$

Since $1/(6\log 2) = .2404\cdots$ is less than 1, and $G(v_m)$ is an integer, $G(v_m)$ must in fact equal $\lceil \tfrac{1}{2}v_m \log_2(\tfrac{3}{4}v_m)\rceil$.

3. A set merging process. In various graph-theoretic algorithms, the following merging process occurs [3]. Start with n sets, each containing exactly one member. At step i, $i = 1, 2, \cdots, n-1$, merge any two sets. A cost equal to the size of the smaller set is incurred at each step. The maximum cost $\bar{G}(n)$ that can be incurred in the whole process is defined by

$$\bar{G}(1) = 0, \qquad \bar{G}(n) = \max_{1 \leqq i \leqq n/2}[i + \bar{G}(i) + \bar{G}(n-i)], \quad n = 2, 3, \cdots. \tag{9}$$

It turns out that $\bar{G}(n)$ is the same as $G(n)$ for $n = 1, 2, \cdots$. Given this fact, and noting that recurrences (3) and (4) both look like (9) with the max operation removed, we can read off two different cost-maximizing policies:

(a) Merge a set whose size is a power of 2 with a set of equal or smaller size.

(b) Merge two sets whose size differs by at most one.

Each of these policies is a special case of the general policy set forth in Theorem 2 below.

4. Proof of extremality. To prove that the extremal property (9) is possessed by $G(n)$, consider the function

$$F(p, q) = G(p+q) - [p + G(p) + G(q)], \qquad 0 \leqq p \leqq q. \tag{10}$$

$F(p, q)$ is the "deficiency" by which the cost of a set of $p + q$ elements would fall short of $G(p+q)$ if that set were created by merging sets of size p and q created by extremal routes whose costs were $G(p)$ and $G(q)$. A recurrence for $F(p, q)$ follows from substituting (5) into (10).

$$F(p,q) = \left\lfloor \frac{p+q}{2} \right\rfloor + G\left(\left\lfloor \frac{p+q}{2} \right\rfloor\right) + G\left(\left\lceil \frac{p+q}{2} \right\rceil\right)$$
$$- \{p + \lfloor p/2 \rfloor + G(\lfloor p/2 \rfloor) + G(\lceil p/2 \rceil) + \lfloor q/2 \rfloor + G(\lfloor q/2 \rfloor) + G(\lceil q/2 \rceil)\}$$
$$= G\left(\left\lfloor \frac{p+q}{2} \right\rfloor\right) - \{\lfloor p/2 \rfloor + G(\lfloor p/2 \rfloor) + G(\lceil q/2 \rceil)\}$$
$$+ G\left(\left\lceil \frac{p+q}{2} \right\rceil\right) - \{\lceil p/2 \rceil + G(\lceil p/2 \rceil) + G(\lfloor q/2 \rfloor)\}$$
$$+ \left\{\left\lfloor \frac{p+q}{2} \right\rfloor + \lceil p/2 \rceil - p - \lfloor q/2 \rfloor\right\}.$$

The last bracketed quantity is 1 when both p and q are odd and is 0 otherwise, so can be more compactly written as $pq \pmod 2$. Unless p is even and q is odd, the middle line is exactly $F(\lfloor p/2 \rfloor, \lceil q/2 \rceil)$ and the next is $F(\lceil p/2 \rceil, \lfloor q/2 \rfloor)$. But in that special case exchange the places of

$$G\left(\left\lfloor \frac{p+q}{2} \right\rfloor\right) \quad \text{and} \quad G\left(\left\lceil \frac{p+q}{2} \right\rceil\right)$$

to see the same thing. Thus

$$F(p,q) = F(\lfloor p/2 \rfloor, \lceil q/2 \rceil) + F(\lceil p/2 \rceil, \lfloor q/2 \rfloor) + (pq \pmod 2), \quad 0 \leqq p < q. \tag{11a}$$

The domain must be restricted to $0 \leqq p < q$ lest $F(\lceil p/2 \rceil, \lfloor q/2 \rfloor)$ go outside the original domain of (10) when $p = q$. The boundary conditions[3]

$$F(p,p) = F(p,p+1) = F(0,q) = 0, \qquad p,q = 0,1,\cdots, \tag{11b}$$

follow from the facts that

$$G(2p+1) = p + G(p) + G(p+1), \quad G(2p) = p + 2G(p), \quad G(1) = G(0) = 0.$$

We are now in a position to justify the assertion that $G(n)$ is indeed the largest cost that can be incurred in the merging problem, or that $G(n)$ satisfies (9). The proof by induction on G assumes that G satisfies (9) up to $n-1$. Then

$$\max_{1 \leqq i \leqq n/2} [i + G(i) + G(n-i)] = G(n) - \min_i [G(n) - i - G(i) - G(n-i)] \tag{12}$$
$$= G(n) - \min_i F(i, n-i)$$

Now by (11), F is obviously nonnegative, and by (11b), F does take on the value zero at (i,i) or $(i,i+1)$, whence expression (12) is exactly $G(n)$.

5. Extremal policies. The set of maximum cost merges is the set of pairs (p,q) with $0 \leqq p \leqq q$ such that $G(p+q) = p + G(p) + G(q)$, or equivalently,

[3] These conditions are not mutually independent. An independent set would be $F(p,p) = F(0,1) = 0$.

such that $F(p, q) = 0$. When p and q are both odd and unequal, (11a) shows $F(p, q)$ to be nonzero. For other (p, q) we must chase the recurrence down to an unequal odd pair to show that $F(p, q) \neq 0$, or else chase all paths to the boundary (11b) without encountering such a pair to show that $F(p, q) = 0$.

It is easy to verify that as long as the first member of the pair remains even, n-fold application of (11a) visits only the points $(p/2^i, \lfloor q/2^i \rfloor)$ and $(p/2^i, \lceil q/2^i \rceil)$, $i = 1, 2, \cdots, n$. Now suppose that p and q may be represented by $p = 2^a p_0$ and $q = 2^b q_0$, where $a \geqq b$ and p_0 and q_0 are both odd. Then the recurrence cannot reach an odd pair until the ath iteration, where we will have typical terms $F(p_0, \lfloor q/2^a \rfloor)$ and $F(p_0, \lceil q/2^a \rceil)$. If $\lfloor q/2^a \rfloor$ is odd, then for $F(p, q)$ to be zero we must have $\lfloor q/2^a \rfloor = p_0$. If $\lfloor q/2^a \rfloor$ is even, then $\lceil q/2^a \rceil$ is odd, since otherwise q_0 would have been even, and hence $F(p, q)$ is nonzero. Similarly if $a < b$, we find that $F(p, q)$ is nonzero unless $\lfloor p/2^b \rfloor = q_0 - 1$. In other words, $F(p, q)$ is zero if and only if

$$p = 2^a p_0 \quad \text{and} \quad q \in \{p, p + 1, \cdots, p + 2^a - 1\}, \qquad a = 0, 1, 2, \cdots, \quad p_0 \text{ odd},$$

or

$$q = 2^b q_0 \quad \text{and} \quad p \in \{q - 2^b + 1, \cdots, q - 1, q\}, \qquad b = 0, 1, 2, \cdots, \quad q_0 \text{ odd},$$

or

$$p = 0.$$

Stated still another way for nonzero p and q, this criterion is Theorem 2.

THEOREM 2. *Let a set of n objects be created by $n - 1$ merges of pairs of sets starting from n singletons. If a cost equal to the size of the smaller of the pair is incurred at each step, then the maximal total cost for the process is $G(n)$ and is achieved if and only if for every merged pair, the sizes of the two sets differ by less than the largest power of 2 that divides one of the two sizes.*

The locus of maximal cost merges makes the interesting recursive pattern shown in Fig. 2.

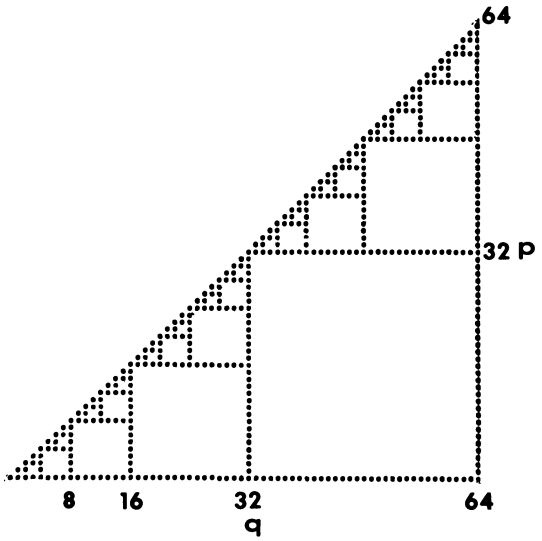


FIG. 2. *Maximum cost merging strategies. Dots mark all places where the inequality $G(p + q) \geqq p + G(p) + G(q)$ is tight.*

6. Related work. Hopcroft and Ullman have a "practically linear" algorithm for performing the bookkeeping connected with the set-merging process [4]. Functions or extremal recurrences similar to $G(n)$ have been studied in [5]–[8].

REFERENCES

[1] E. N. GILBERT, *Games of identification and convergence*, SIAM Rev., 4 (1962), pp. 16–24.
[2] R. BELLMAN AND H. N. SHAPIRO, *On a problem in additive number theory*, Ann. of Math., 49 (1948), pp. 333–340.
[3] D. E. KNUTH, *Complexity analysis of equivalence algorithms*, Unpublished notes, Mathematisk Institutt, Blindern, Norway, 1972.
[4] J. E. HOPCROFT AND J. D. ULLMAN, *Set merging algorithms*, this Journal, 2 (1973), pp. 294–303.
[5] M. L. FREDMAN AND D. E. KNUTH, *Recurrences based on minimization*, CS–72–248, Dept. of Computer Sci., Stanford Univ., Stanford, Calif., 1971.
[6] E. WONG, *A linear search problem*, SIAM Rev., 6 (1964), pp. 168–174.
[7] R. MORRIS, *Some theorems on sorting*, SIAM J. Appl. Math., 17 (1967), pp. 1–6.
[8] L. CARLITZ, *A sorting function*, Duke Math. J., 38 (1971), pp. 561–568.

SIAM J. COMPUT.
Vol. 3, No. 4, December 1974

COMPUTATIONALLY RELATED PROBLEMS*

SARTAJ SAHNI†

Abstract. We look at several problems from areas such as network flows, game theory, artificial intelligence, graph theory, integer programming and nonlinear programming and show that they are related in that any one of these problems is solvable in polynomial time iff all the others are, too. At present, no polynomial time algorithm for these problems is known. These problems extend the equivalence class of problems known as P-Complete. The problem of deciding whether the class of languages accepted by polynomial time nondeterministic Turing machines is the same as that accepted by polynomial time deterministic Turing machines is related to P-Complete problems in that these two classes of languages are the same iff each P-Complete problem has a polynomial deterministic solution. In view of this, it appears very likely that this equivalence class defines a class of problems that cannot be solved in deterministic polynomial time.

Key words. complexity, polynomial reducibility, deterministic and nondeterministic algorithms, network flows, game theory, optimization, AND/OR graphs

1. Introduction. Cook [3] showed that determining whether the class of languages accepted by nondeterministic Turing machines operating in polynomial time was the same as that accepted by deterministic polynomial time bounded Turing machines was as hard as deciding if there was a deterministic polynomial algorithm for the satisfiability problem of propositional calculas (actually, Cook showed that there was a polynomial algorithm for satisfiability iff the deterministic and nondeterministic polynomial time languages were the same). This problem about equivalence of the two classes of languages is a long-standing open problem from complexity theory. Intuitively, it seems that the two classes are not the same. Consequently there may be no polynomial algorithm for the satisfiability problem. Further empirical evidence that the two classes may not be the same was provided by Karp in [5], where he showed that many other problems like the traveling salesman problem, finding the maximum clique of a graph, minimal colorings of graphs, minimal set covers, etc., had polynomial algorithms iff the two classes of languages were the same. In view of this relationship amongst all these problems, we can say that there is strong evidence to believe that there is no polynomial algorithm for any of the problems given in Karp [5]. However, no formal proof of this (if this is true) is available at this time.

The equivalence class of problems having the property that each member of the class has a polynomial algorithm iff nondeterministic and deterministic polynomial languages are the same is known as P-Complete. In [5], Karp presents 21 members of this class. The purpose of this paper is to extend the class of known P-Complete problems. Specifically, we show that several important problems from

* Received by the editors July 18, 1973, and in revised form April 6, 1974. The research reported here is part of the author's Ph.D. dissertation, Cornell University. An earlier version of these results was presented at the 1972 IEEE Annual Conference on Switching and Automata Theory. This research was supported in part by the National Science Foundation under Grant GJ-33169.

† Department of Computer Science, Cornell University, Ithaca, New York. Now at Department of Computer, Information, and Control Sciences, University of Minnesota, Minneapolis, Minnesota 55455.

areas such as artificial intelligence, game theory, graph theory, network flows and integer optimization are P-Complete. We also introduce the concept of P-Hard.

The rest of this section will be devoted to definitions and establishing our notation. In §2 the new members of the classes P-Complete and P-Hard are presented.

1.1. Definitions. As our computational model we shall use Turing machines. (See Hopcroft and Ullman [4] for a standard treatment of this model.) The reader unfamiliar with deterministic and nondeterministic polynomial time computations should see Karp [5].

DEFINITION 1. P (NP) is the class of languages recognizable by deterministic (nondeterministic) polynomial time bounded one-tape Turing machines.

Open problem. "Is P = NP?" We may rephrase this as, "Is there a deterministic polynomial algorithm for all languages in NP?" Call this *Problem* P1.

DEFINITION 2.[1] A *problem* is a total function $f: \Sigma^* \to 2^{\Sigma^*}$, which takes each finite string to a nonempty subset of strings. (Informally, the finite string represents an encoding of the input or data and f maps this onto a solution set. Thus, for language recognition problems, $f: \Sigma^* \to (0, 1)$, where $f = 0$ iff the input string is not in the language.)

We consider algorithms which, given $x \in \Sigma^*$, produce some $y \in f(x)$. The computing time of the algorithm will be measured as a function of the length of x ($|x|$). (All algorithms will be deterministic unless otherwise stated.)

DEFINITION 3. A problem L will be said to be P-Reducible to a problem M (written $L \propto M$) iff a polynomial algorithm for M implies a polynomial algorithm for L. That is, from a deterministic polynomial algorithm for M we can construct a deterministic polynomial algorithm for L.

DEFINITION 4. A problem L is P-Hard iff a polynomial algorithm for L implies P = NP.

DEFINITION 5. Two problems L and M are P-Equivalent iff $L \propto M$ and $M \propto L$.

Clearly, P-Reducible is a transitive relation and P-Equivalent is an equivalence relation.

DEFINITION 6. P-Complete (PC) is the equivalence class of P-Equivalent problems having a polynomial algorithm iff P = NP.

Our definition of P-Complete differs from that used by Karp [5]. However, it can easily be shown that any problem which is polynomial-Complete under his definition is P-Complete. The reverse, however, may not be true. (No proof of the equivalence or nonequivalence of the two definitions is known.) Note that all P-Complete problems are also P-Hard. In some cases, we may only be able to show the relation P-Hard rather than the stronger P-Complete relation. We shall often write $L \propto$ P1 when we mean "if P = NP, then L is polynomial solvable" and P1 $\propto L$ when we mean "if L is polynomial solvable, then P = NP". No ambiguity should arise from this double use of the symbol $\propto$.

---

[1] The author is grateful to an anonymous referee for suggesting this definition of a problem which encompasses both language recognition and optimization problems. Σ is the tape alphabet of the Turing machine and Σ^* is the set of all finite length strings or words from the alphabet Σ.

There are several ways to show that a problem L is P-Complete. For instance, one could show L to be P-Equivalent to M, where M is a problem already known to be P-Complete, or show that L has a polynomial algorithm iff P = NP, etc. Most of the proofs in the next section will adopt the following approach: (i) show that "if P = NP, then L" is polynomial solvable, i.e., $L \propto (\mathrm{P} = \mathrm{NP})$, and (ii) show $M \propto L$, where M is a problem known to be P-Complete. M will usually be the satisfiability problem of propositional calculus (see Karp [5] for a formal definition of this problem).

2. P-Complete and P-Hard problems. In this section we shall show that several frequently encountered problems in various areas such as network flows, game theory, graph theory, nonlinear and linear optimization are either P-Complete or at least P-Hard. The reductions are easily seen to be effective. The polynomial factors involved in the reduction are small (usually a constant or a polynomial of degree 1).

2.1. Some known P-Complete problems. To prove some of the reductions, we shall make use of some known members of PC. A brief description of these members is given below. (A more exhaustive list may be found in Karp [5].)

(i) *Propositional calculus.*

(a) *Satisfiability.* Given a formula from the propositional calculus, in conjunctive normal form (CNF), is there an assignment of truth values for which it is "true"?

(b) *Satisfiability with exactly* 3 *literals per clause.* This is the same as (a), except that each clause of the formula now has exactly 3 literals.

(c) *Tautology.* Given a formula, from the propositional calculus, in disjunctive normal form (DNF), does it have the value "true" for all possible assignments of truth values.

(ii) *Sum of subsets of integers.* Given a multiset $S = (s_1, \cdots, s_r)$ of positive integers and a positive integer M, does there exist a submultiset of S that sums to M? (This problem is called the Knapsack problem in [5]. However, here we shall denote by "Knapsack problem" a similar integer optimization problem.) Note that a multiset is a collection of elements that may not necessarily be distinct.

(iii) *Maximum independent set.* Let G be a graph with vertices $v_1, v_2, \cdots, v_n$. A set of vertices is *independent* if no two members of the set are adjacent in G. A *maximum* independent set is an independent set that has a maximum number of vertices.

(iv) *Directed Hamiltonian cycle.* Given a directed graph G, does it have a cycle that includes each vertex exactly once?

THEOREM 2.1. *The following problems are in* PC:

(i) *Satisfiability, satisfiability with exactly three literals per clause, tautology*;

(ii) *Sum of subsets of integers*;

(iii) *Maximum independent set of a graph*;

(iv) *Directed Hamiltonian cycle.*

Proof. (i) is proved in Cook [3]. The rest are proved in Karp [5].

Cook [3] actually shows that satisfiability with at most three literals per clause is P-Complete. From this result one may trivially show that satisfiability with exactly three literals per clause is P-Complete. We show how to convert a

two-literal clause into an equivalent pair of three-literal clauses. Let $(x_1 + x_2)$ be the clause and y a variable not occurring in the formula. Then $(x_1 + x_2 + y) \wedge (x_1 + x_2 + \bar{y})$ is satisfiable iff the two-literal clause is. All two-literal clauses may be replaced by pairs of three-literal clauses as above. This at most doubles the number of clauses. Clauses with only one literal can be deleted, the literal determining the truth assignment to that variable.

2.2. Integer network flows. We define the following network problems.

Problem N(i). *Network flows with multipliers.* Let G be a directed graph with vertices $\hat{s}_1, \hat{s}_2, v_1, \cdots, v_n$ and edges (arcs) $e_1, e_2, \cdots, e_m$. Let $w^-(v)$ be the set of arcs directed into vertex v and $w^+(v)$ those arcs directed away from v.

G will be said to denote a *network with multipliers* if:

(a) the source $\hat{s}_1$ of the network has no incoming arcs, i.e., $w^-(\hat{s}_1) = \varnothing$;

(b) the sink $\hat{s}_2$ has no outgoing arcs, i.e., $w^+(\hat{s}_2) = \varnothing$;

(c) to every vertex v_i (excluding the source and sink) there corresponds an integer $h_i > 0$, called its *multiplier.*

(d) to each edge e_i there corresponds an interval $[a_i, b_i]$;

Conditions (a)–(d) are said to define a *transportation* network.

We are required to find a flow vector, with integer entries, $\Phi = (\phi_1, \phi_2, \cdots, \phi_m)$ such that the following conditions hold.

Condition 1. $a_i \leqq \phi_i \leqq b_i$;

Condition 2. $h(v) \sum_{i \in w^-(v)} \phi_i = \sum_{i \in w^+(v)} \phi_i$ for all $v \in V(G)$, $v \neq \hat{s}_1$ $v \neq \hat{s}_2$;

Condition 3. $\sum_{i \in w^-(s_2)} \phi_i$ is maximized.

In what follows, we assume $a_i = 0$.

Problem N(ii). *Multicommodity network flows.* The transportation network is as above, but now $h(v) = 1$ for all v in $V(G)$. We have, however, several different commodities $c_1, c_2, \cdots, c_n$, and some arcs may be labeled, i.e., they can carry only certain commodities. Each arc is assigned a capacity, and we wish to know whether a flow $R = (r_1, r_2, \cdots, r_n)$, where r_i is the quantity of the ith commodity, is feasible in the network.

Problem N(iii). *Integer flows with homologous arcs.* The transportation network remains the same. Also, $h(v) = 1$ and there is only one commodity. Certain arcs are paired, and we require that if arcs i, j are paired, then $\phi_i = \phi_j$. We wish to know if a flow of at least F is feasible in the network.

Problem N(iv). *Integer flows with bundles.* The arcs in the network are divided into sets $I_1, \cdots, I_k$ (the sets may overlap). Each set is called a *bundle*, and with each bundle is associated a capacity C_i. We wish to know if a flow $\geqq F$ is feasible in the network:

$$\sum_{i \in I_j} \phi_i \leqq C_j, \qquad 1 \leqq j \leqq k$$

and

$$h(v) = 1 \quad \forall\, v \in V(G).$$

THEOREM 2.2. *Problems* N(i)–N(iv) *are* in PC.

Proof. (a) N(i), N(ii), N(iii), N(iv) $\propto$ P1. The nondeterministic turing machine (NDTM) just guesses the flows in each arc and then verifies Conditions 1 and 2. In addition, it does the following:

(i) for N(ii) it verifies that the resultant flow is $\geqq R$;
(ii) for N(iii) the "homologous conditions" are checked and $\sum_{i \in w^-(\hat{s}_2)} \phi_i \geqq F$ verified;
(iii) for N(iv) the bundle restrictions are checked and $\sum_{i \in w^-(\hat{s}_2)} \phi_i \geqq F$ verified.

If in N(i) we replace the max $\sum_{i \in w^-(\hat{s}_2)} \phi_i$ requirement to:

$$T: \sum_{i \in w^-(\hat{s}_2)} \phi_i \geqq F, \tag{2.2.1}$$

then from the above it follows that $T \propto$ P1.[2] To see N(i) $\propto T$, we note that if the length of the input on a Turing machine's tape is n, then the largest number it can represent is c^n, for some constant c which depends only on the Turing machine. Hence the maximum capacity of an arc is bounded by c^n and so max $\sum_{i \in w^-(\hat{s}_2)} \phi_i \leqq k^n$, for some constant k. Now, assume there is a polynomial $[p(n)]$ algorithm for T. Then, using the method of bisection, we can determine max $\sum_{i \in w^-(\hat{s}_2)} \phi_i$ in at most $\log_2 k^n = n \log_2 k$ applications of T. This, therefore, gives a polynomial algorithm for N(i). Therefore N(i) $\propto T \propto$ P1, and from the transitivity of $\propto$ we conclude N(i) $\propto$ P1. Clearly, this proof technique can be used to show N(iii) and N(iv) to be complete when they are changed to maximization problems.

(b) We now show the reduction for N(i)–N(iv), in the other direction.

(i) Sum of subsets of integers $\propto$ N(i). We construct a network flow problem of type N(i) such that max $\sum_{i \in w^-(\hat{s}_2)} \phi_i = M$ iff there is a submultiset of $S = \{s_1, \cdots, s_r\}$ that sums to M.

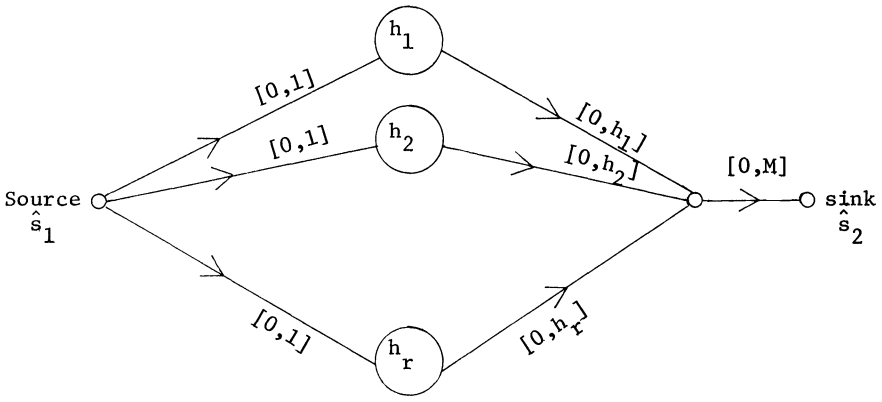


FIG. 2.2.1. *Construction for sum of subsets* $\propto$ N(i)

Consider the construction of Fig. 2.2.1 with $h_i = s_i$, $1 \leqq i \leqq r$. Clearly

$$\max \sum_{i \in w^-(\hat{s}_2)} \phi_i = M$$

iff some submultiset of S sums to M.

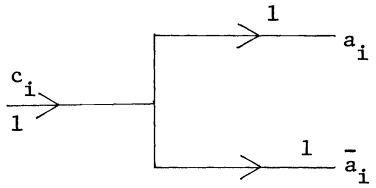
(ii) Tautology $\propto$ N(ii). Suppose that the formula P in DNF has n variables $a_1, a_2, \cdots, a_n$. We shall construct a multicommodity network with n commodities

[2] Recall that P1 was defined in §1.2 to be the decision problem: is NP = P?

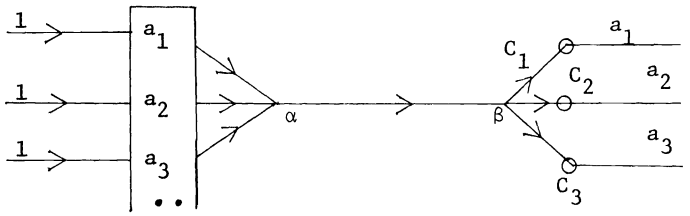
$c_1, c_2, \cdots, c_n$ such that the flow $R(1, \cdots, 1)$ is feasible iff P is not a tautology. The network of Fig. 2.2.2 realizes this.

Discussion.

[A] This section of the network ensures that there is a flow through only one of the nodes a_i or $\bar{a}_i$. In terms of the formula A, a flow through a_i means a truth assignment of 1 to a_i while a flow through $\bar{a}_i$ means an assignment of 0 to a_i.



[B] For each clause (K_i) in P we have a section of the form



If there are j literals in the clause, then arc (α, β) is assigned a capacity of $j - 1$. This requires that the truth assignments be such that clause k_i is false (as at least one term in it is false). Node β is where the "multicommodity" property of the network is used. Here the flow through α is correctly separated into its components, i.e., we are able to get back the truth values of the variables. The components for each flow are connected in series as in Fig. 2.2.2.

We now want to know if a flow $R = (1, 1, \cdots, 1)$ is feasible. It is easy to see that such a flow is possible iff there is a truth assignment to $a_1, \cdots, a_n$ for which each clause is false, i.e., iff P is not a tautology.

(iii) Tautology $\propto$ N(iii). The construction is very similar to that for multicommodity network flows. The network is as in Fig. 2.2.3. Homologous arcs are marked with the same subscripted Greek letter.

The arcs (α, β) have a capacity that is one less than the number of terms in the clause, thereby ensuring that truth assignments that would make the preceding clause "true" cannot occur. The "homologous conditions" permit the separation of the flow at β into the original "truth assignments".

The maximum capacity of the sink is n. Hence there is a flow $\geqq n$ iff there is a consistent assignment of truth values to $a_1, \cdots, a_n$ such that no clause is "true", and hence P is not a tautology.

(iv) Maximum independent set $\propto$ N(iv).[3] Let $G(V, E)$ be an undirected graph for which we want to determine the maximum independent set.

[3] The author is grateful to S. Even for pointing out an error in the original proof and for suggesting the correction.

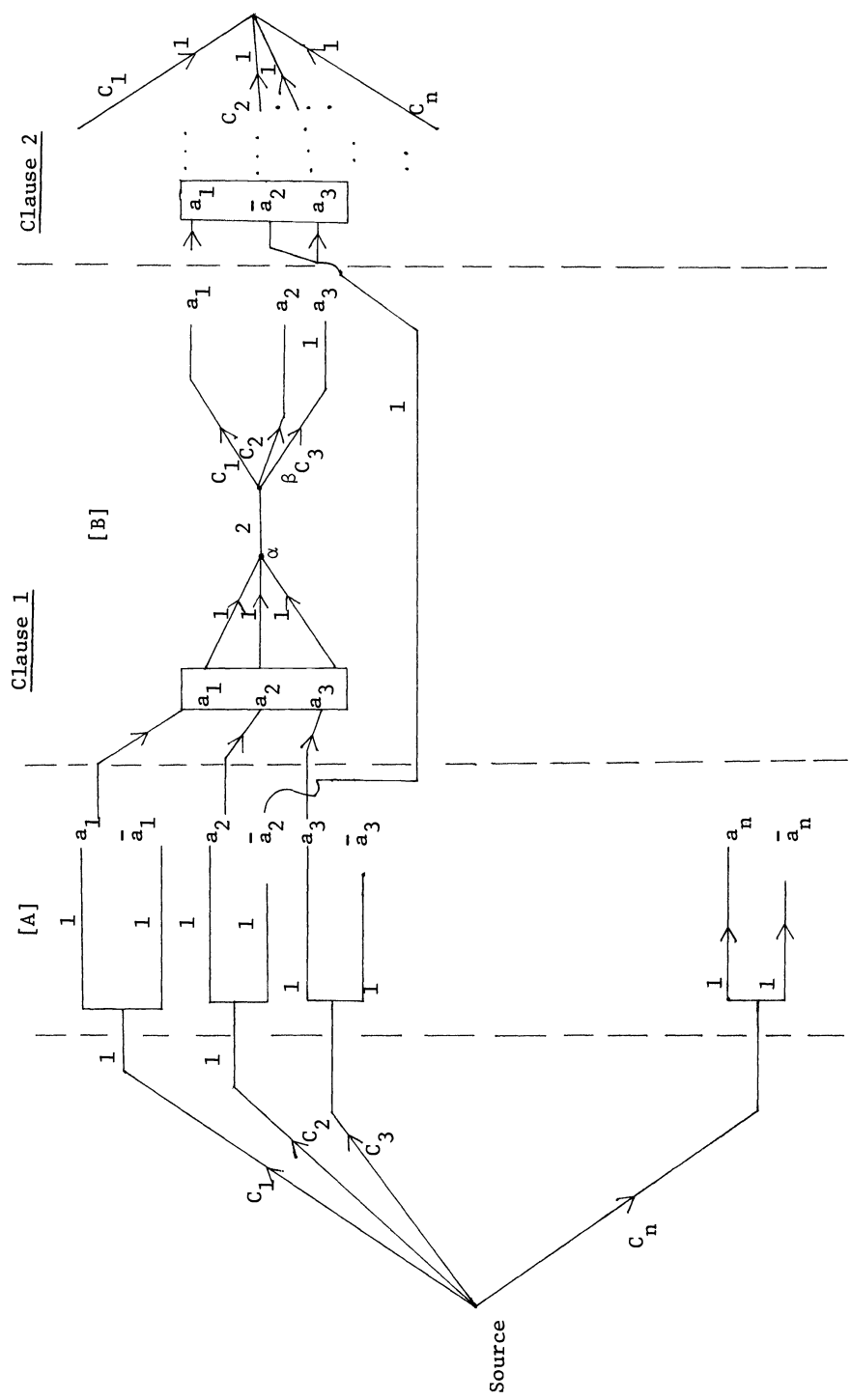


FIG. 2.2.2. *Tautology α multicommodity network flows*

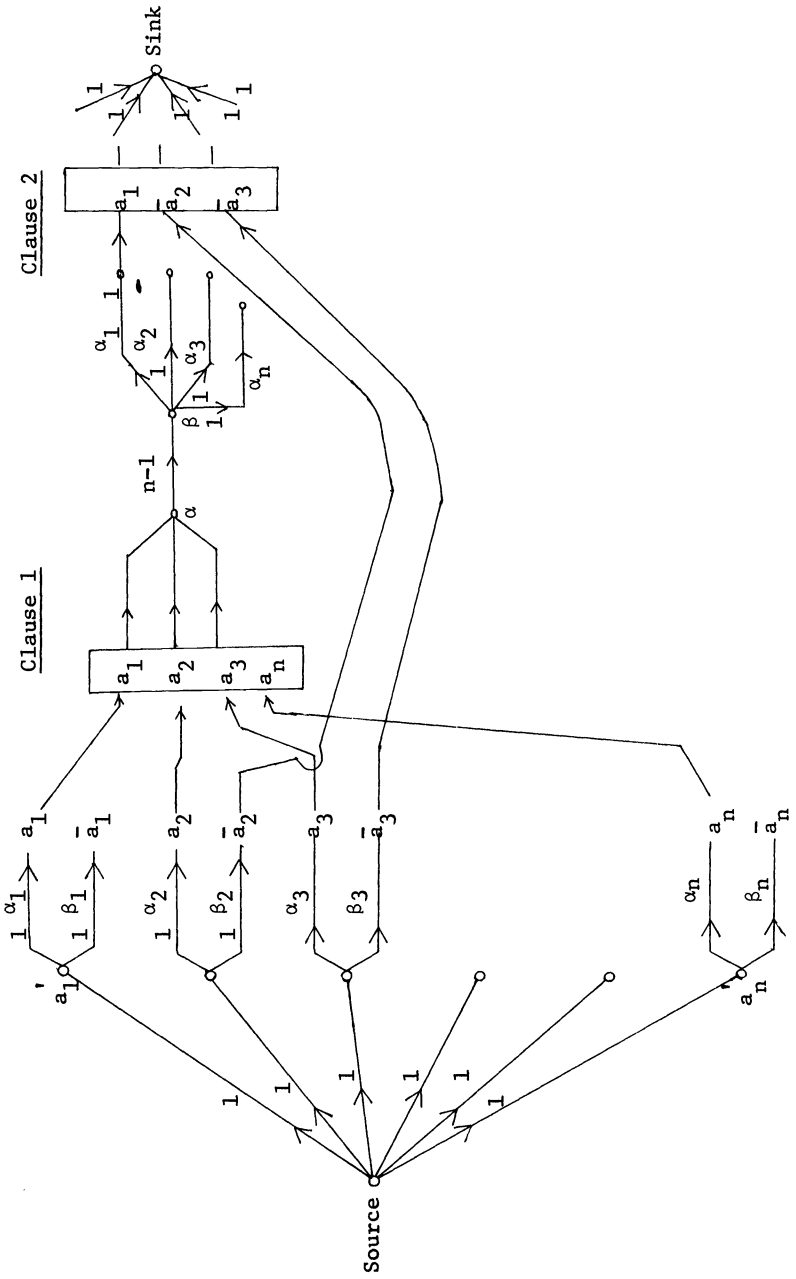


FIG. 2.2.3. *Network with homologous arcs*

Construct a network as below:

Let $\hat{s}_1, v_1, \cdots, v_n, \hat{s}_2$ be the nodes of the network $n = |V|$. From the source node, draw an arc of capacity 1 to each of the nodes v_i, $1 \leqq i \leqq n$. From each node v_i, draw an arc a_i to the sink node $\hat{s}_2$. For each edge in G, define a bundle (a_i, a_j) if this edge joins vertices v_i and v_j in G. These are the only bundles in the network. Each bundle is assigned a capacity 1. This ensures that if vertex v_i is chosen in the maximum independent set (i.e., if there is a nonzero flow through it), then there is no flow through vertices adjacent to v_i (i.e., adjacent vertices are not chosen).

Now there is a flow $\geqq F$ iff there is an independent set of cardinality $\geqq F$. We solve the flow problem for $F = n, n-1, \cdots, 1$, and the first F for which we get a feasible flow defines a maximum independent set.

Example 2.2.1.

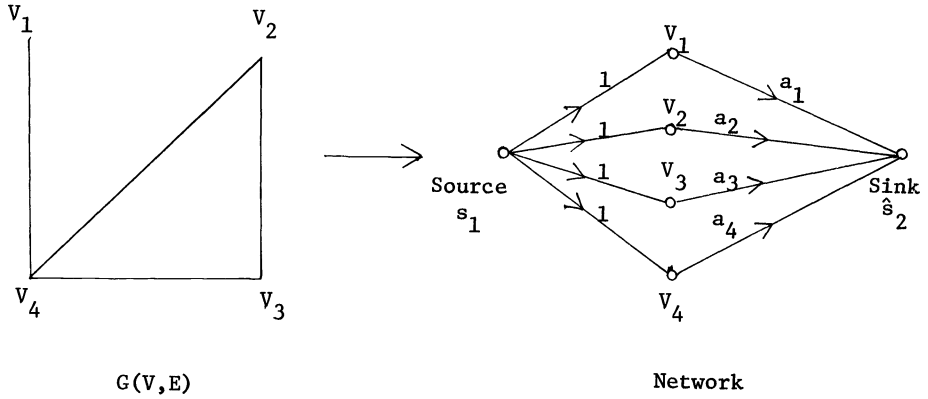


FIG. 2.2.4. *Example for maximum independent set* α N(iv)

The largest k for which there is a feasible flow is $k = 2$, through vertices V_1 and V_2. Thus the maximum independent set of G is of size 2, and one such set is $\{V_1, V_2\}$. The bundles are: (a_1, a_4), (a_2, a_3), (a_2, a_4) and (a_3, a_4).

It is interesting to note that all these problems are related to a similar, polynomial time, flow problem (see [1]).

2.3. Graph theory.

Problem G1. *Minimal equivalent graph of a digraph.* Given a directed graph $G(V, E)$, we wish to remove as many edges from G as possible, getting a graph G_1 such that:

(2.3.1a) In G, there is a path from v_i to v_j iff there is a path in G_1 from v_i to v_j;

(2.3.1b) $E(G_1) \subseteq E(G)$ ($E(G)$ is the set of edges of G), i.e., we want the smallest subset of $E(G)$ such that the transitive closure of G_1 = transitive closure of G.

THEOREM 2.3.1. G1 *is in* PC.

Proof. (a) G1 α P1, Let n = number of vertices in $G = |V(G)|$; then

$$|E(G)| \leqq n(n-1) < n^2.$$

We can easily construct an NDTM, T, which given G and an integer k, determines if there is a subset of k edges satisfying (2.3.1a,b). T can be constructed so as to work in $O(n^3)$ time. If NP = P, then there is a deterministic algorithm that does

this in $p(n)$ time. We find the smallest $k \leqq n^2$ for which such a subset exists. After determining k, the k edges can be determined as below.

Define a sequence $\bar{E}$ of maximum length $|E(G)|$. Set $\bar{e}_i = 1$ if edge i is among the k edges and $\bar{e}_i = 0$ otherwise.

Suppose it is already known that $\bar{E} = (i_1, \cdots, i_j)$ is a correct "partial" choice; then we ask if $\bar{E}(i_{j+1} = 1)$ is.

If yes, then set $\bar{E} = (i_1, i_2, \cdots, i_j, 1)$.

If no, then set $\bar{E} = (i_i, i_2, \cdots, i_j, 0)$.

Do this for $j = 0, 1, 2, \cdots, |E| - 1$.

(b) Directed Hamilton cycle α G1.

Note. (i) If the directed graph G has a Hamilton cycle, then its transitive closure is the "complete directed graph" on $|V(G)|$ points. The smallest graph with this transitive closure is the cycle on $|V(G)|$ points. Thus if there is a Hamilton cycle, then this cycle forms the minimal equivalent graph of G.

(ii) Conversely, if the minimal equivalent graph is a cycle on $|V(G)|$ points, then G has a Hamilton cycle.

Therefore G has a Hamiltonian cycle iff the minimal equivalent graph of G is a Hamiltonian cycle.

Problem G2. *Optimal solution to* AND/OR *graphs.* This is a problem frequently encountered in artificial intelligence; see [2], [9] and [10]. We are given a directed graph $G(V, E)$. Each node of G represents a subproblem. In order to solve this subproblem, one might have to solve either all of its successors or only one of them. In the former case the node will be denoted an AND node, while in the latter case it is an OR node. The arcs are weighted, and the weights represent the cost associated with solving the parent node given that the successor (or son) node has been solved. There is one special node, S, which has no incoming arcs. This node represents the total problem being solved. The problem then is to find a minimum solution to S.

As an example, consider the directed graph of Fig. 2.3.1. The problem to be solved is P_1. To do this, one may solve either nodes P_2, P_3 or P_7, as P_1 is an OR node. The cost incurred is then either 2, 2 or 8 (i.e., cost in addition to that of solving one of P_2, P_3 or P_7). To solve P_2, both P_4 and P_5 have to be solved, as P_2 is an AND node. The total cost to do this is 2. To solve P_3, we may solve either P_5 or P_6. The minimum cost to do this is 1. P_7 is free. In this example, then, the optimal

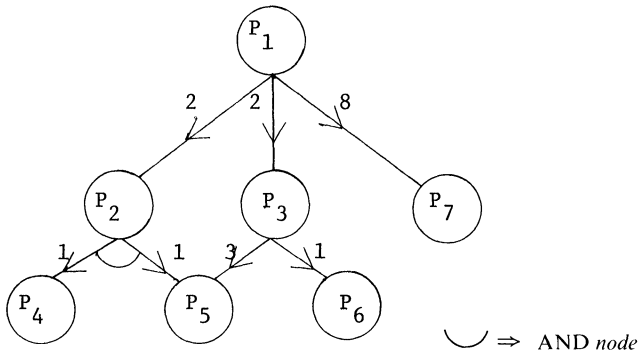


FIG. 2.3.1. AND/OR *graph*

way to solve P_1 is first solve P_6, then P_3 and finally P_1. The total cost for this solution is 3.

THEOREM 2.3.2. G2 $\in$ PC.

Proof. (a) G2 $\propto$ (P = NP). The proof for this part is very similar to the part (a) of the proofs of each of Theorems 2.3.1 and 2.5.1 (see §2.5).

(b) Satisfiability $\propto$ G2. We show how to transform a formula P in CNF into an AND/OR graph such that the AND/OR graph so obtained has a certain minimum cost solution iff P is satisfiable.

Let $$P = \bigwedge_{i=1}^{k} C_i, \qquad C_i = \bigvee_{j=1}^{3} l_j,$$

where the l_j's are literals and the variables of P, $V(P)$ are $x_1, x_2, \cdots, x_n$. The AND/OR graph will then have nodes as follows:

1. There is a special node, S, with no incoming arcs. This node represents the problem to be solved.

2. S is an AND node with descendent nodes $P, x_1, x_2, \cdots, x_n$.

3. Each node x_i represents the corresponding variable x_i in the formula P. Each x_i is an OR node with two descendents denoted Tx_i and Fx_i, respectively. If Tx_i is solved, then this will correspond to assigning a truth value of "true" to the variable x_i. Solving node Fx_i will then correspond to assigning a truth value of "false" to x_i.

4. The node P represents the formula P, and is an AND node. It has k descendents $C_1, C_2, \cdots, C_k$. Node C_i corresponds to the clause C_i in the formula P. The nodes C_i are OR nodes.

5. Each node of type Tx_i or Fx_i has exactly one descendent node which is terminal (i.e., has no edges leaving it). These terminal nodes shall be denoted $v_1, v_2, \cdots, v_{2n}$.

To complete the construction of the AND/OR, graph the following edges and costs are added:

1. From each node C_i an edge (C_i, Tx_j) is added if x_j occurs in clause C_i. An edge (C_i, Fx_j) is added if $\bar{x}_j$ occurs in the clause C_i. This is done for all variables x_j appearing in the clause C_i. C_i is designated an OR node.

2. Edges from nodes of type Tx_i or Fx_i to their respective terminal nodes are assigned a weight or cost 1.

3. All other edges have a cost 0.

In order to solve S, each of the nodes $P, x_1, x_2, \cdots, x_n$ must be solved. Solving nodes $x_1, x_2, \cdots, x_n$ costs n. To solve P, we must solve all the nodes $C_1, C_2, \cdots, C_k$. The cost of a node C_i is at most 1. However, if one of its descendent nodes was solved while solving the nodes $x_1, x_2, \cdots, x_n$, then the additional cost to solve C_i is 0, as the edges to its descendent nodes have cost 0 and one of its descendents has already been solved. That is, a node C_i can be solved at no cost if one of the literals occurring in the clause C_i has been assigned a value "true." From this it follows that the entire graph (i.e., node S) can be solved at a cost n if there is some assignment of truth values to the x_i's such that at least one literal in each clause is true under that assignment, i.e, if the formula P is satisfiable. If P is not satisfiable, then the cost is $>n$.

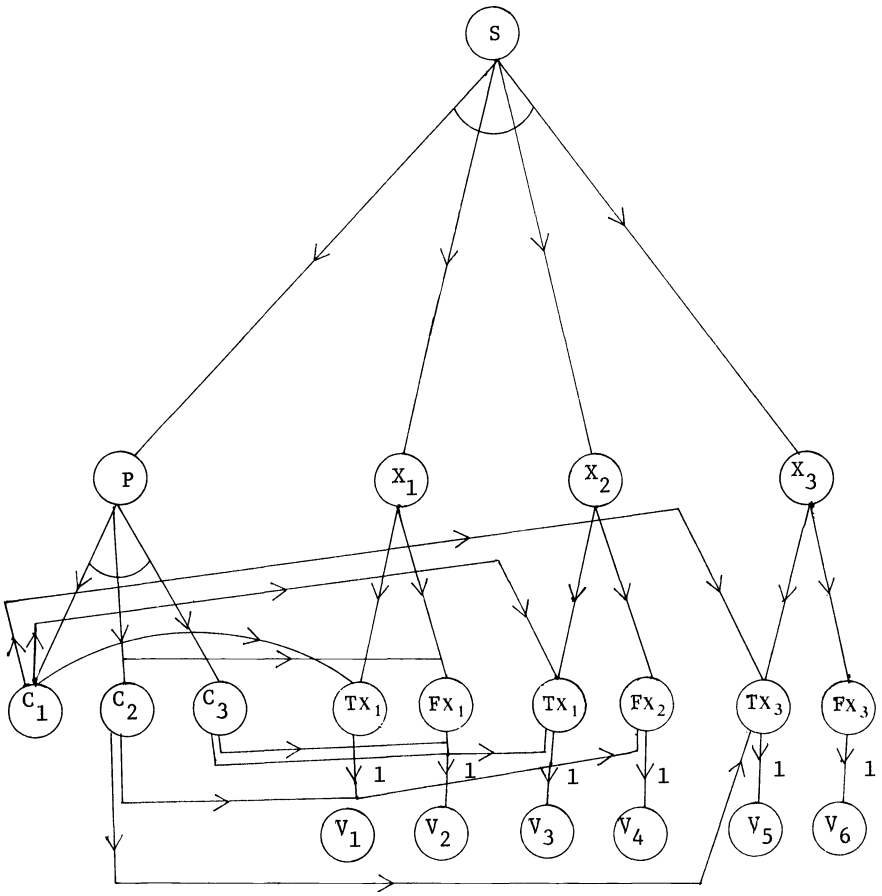
We have now shown how to construct an AND/OR graph from a formula P such that the AND/OR graph so constructed has a solution of cost n iff P is satisfiable. Otherwise the cost is $>n$. Hence from the minimum solution to the AND/OR graph, one can determine if P is satisfiable. The construction clearly takes only polynomial time. This completes the proof.

Example 2.3.1. Consider

$$P = (x_1 + x_2 + x_3)(\bar{x}_1 + \bar{x}_2 + \bar{x}_3)(\bar{x}_1 + x_2), \quad V(P) = x_1, x_2, x_3, \quad n = 3.$$

Figure 2.3.2 shows the AND/OR graph obtained by applying the transformation of Theorem 2.3.2.

The nodes Tx_1, Tx_2, Tx_3 can be solved at a total cost of 3. The node P then costs nothing extra. The node S can then be solved by solving all its descendent nodes and the nodes Tx_1, Tx_2 and Tx_3. The total cost for this solution is 3 (which is n). Assigning the truth value "true" to the variables of P results in P being "true."



AND *nodes marked* ⌣
All other nodes are OR

FIG. 2.3.2. AND/OR *graph for Example* 2.3.1

2.4. *n*-person game theory. Following Lucas [7], we have:

An n-person noncooperative game in normal form consists of a set N of n players denoted $1, 2, \cdots, n$, a finite set $N_i = 0, 1, \cdots, n_i$ of $n_i + 1$ pure strategies for each player $i \in N$, and a payoff function F from $N_1 \times \cdots \times N_n$ to R^n.

A *strategy* n-tuple $(S_1^*, \cdots, S_n^*)$ is said to be an *equilibrium* n-tuple iff for all i, $i \in N$ and $S_i \in N_i$,

$$F_i(S_1^*, \cdots, S_n^*) \geqq F_i(S_1^*, \cdots, S_{i-1}^*, S_i, S_{i+1}^*, \cdots, S_n^*), \tag{2.4.1}$$

where F_i is the ith component of F. That is, there is no advantage for a player to unilaterally deviate from an equilibrium point.

Problem GT1. Given a game $G = (F, n, \bar{N})$, does it have an equilibrium point?

THEOREM 2.4.1. GT1 $\in$ PC.

Proof. (a) GT1 $\propto$ P1. The nondeterministic Turing machine just guesses an equilibrium point and verifies that the equilibrium condition (2.4.1) is satisfied.

(b) Satisfiability (3 literals/clause) $\propto$ GT1. Let P be the formula in CNF in n variables. Define an n-person game as below:

Each player has two strategies 0 and 1. Strategy 0 corresponds to assigning a truth value "false" to the corresponding variable and strategy 1 to a "true" assignment.

$$\text{Let} \qquad P = C_1 \wedge C_2 \wedge \cdots \wedge C_k, \qquad C_i = C_{i_1} \vee C_{i_2} \vee C_{i_3},$$

where the variables are $x_1, x_2, \cdots, x_n$. Replace each variable in the clause C_i by x_i if $x_i \in C_i$ and by $(1 - x_i)$ if $\bar{x}_i \in C_i$

Replace "$\vee$" by "$+$", getting C_i'.

Example. $C_i = x_i \vee x_2 \vee \bar{x}_3 \Rightarrow C_i' = x_1 + x_2 + (1 - x_3) = x_1' + x_2' + x_3'$. In order that C_i' has a (0, 1) value, replace $x_1' + x_2' + x_3'$ by

$$f_i(\mathbf{x}') = x_1' + x_2'(1 + x_1') + x_3'(1 - x_1')(1 - x_2').$$

Clearly, $f_i(\mathbf{x}') = 1$ iff $C_i(x)$ is "true". Define

$$h_1(x') = 2 \prod_{i=1}^{k} f_i(\mathbf{x}') \quad \text{and} \quad F_1(\mathbf{x}') = \begin{bmatrix} h_1(x') \\ \vdots \\ h_1(x') \end{bmatrix}.$$

From the above definition of $F_1(\mathbf{x}')$, it follows that

$$\max F_1(\mathbf{x}') = \begin{cases} \begin{bmatrix} 2 \\ 2 \\ \vdots \\ 2 \end{bmatrix} & \text{if } P(\mathbf{x}) \text{ is satisfiable,} \\ \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} & \text{otherwise.} \end{cases}$$

Let $G_2(x_1, x_2)$ be a 2-person game with 2 strategies per player and with no equilibrium point:

$$G_2(\mathbf{x}) = \begin{bmatrix} g_1(\mathbf{x}) \\ g_2(\mathbf{x}) \end{bmatrix}, \qquad \begin{bmatrix} g_1 \\ g_2 \end{bmatrix} \leqq \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}.$$

Define

$$F_2(\mathbf{x}) = \begin{bmatrix} g_1(\mathbf{x}) \\ g_2(\mathbf{x}) \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

Then $F_2(\mathbf{x})$ defines an n-person game with no equilibrium point. Set

$$F(\mathbf{x}) = F_1(\mathbf{x}) + F_2(\mathbf{x}) \begin{bmatrix} 2 \\ 2 \\ \vdots \\ 2 \end{bmatrix} - F_1(\mathbf{x}).$$

Then $F(\mathbf{x})$ defines an n-person game in which each player has 2 strategies.

For any choice of strategy vector $\mathbf{x}$, we have either (i) or (ii) below.

(i) $$F_1(\mathbf{x}) = 0, \qquad F(\mathbf{x}) = 2F_2(\mathbf{x}) \leqq \begin{bmatrix} 1 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

By changing the strategies for either x_1 or x_2, we can increase the payoff to x_1 or x_2, respectively, as $F_2(\mathbf{x})$ defines a game with no equilibrium point. If such a change results in

$$F_1(\mathbf{x}) = \begin{bmatrix} 2 \\ 2 \\ \vdots \\ 2 \end{bmatrix},$$

then everyone's payoff increases. In any case, such an $\mathbf{x}$ cannot be an equilibrium point.

(ii) $$F_1(\mathbf{x}) = \begin{bmatrix} 2 \\ 2 \\ \vdots \\ 2 \end{bmatrix}.$$

Such a point is an equilibrium point, as now

$$F(x) = F_1(\mathbf{x}) = \begin{bmatrix} 2 \\ 2 \\ \vdots \\ 2 \end{bmatrix}.$$

and 2 is the maximum payoff any player can get. So no change from this point, unilateral or otherwise, would be advantageous to any player. Therefore the n-person game defined above has an equilibrium point iff $P(\mathbf{x})$ is satisfiable.

As an example for $G_2(x_1, x_2)$, consider:

Strategy	Payoff
(0, 0)	[0, 1]
(1, 0)	[1, 0]
(1, 1)	[0, 1]
(0, 1)	[1, 0]

$$g_1(\mathbf{x}) = (2 - x_1 - x_2)(x_1 + x_2),$$
$$g_2(\mathbf{x}) = (1 - x_1 - x_2)^2.$$

Clearly, no $\mathbf{x}$ is a stable (equilibrium) point. Set

$$G_2(\mathbf{x}) = \begin{bmatrix} g_1(\mathbf{x})/2 \\ g_2(\mathbf{x})/2 \end{bmatrix}.$$

2.5. Optimization.

Problem K1. *One-dimensional* 0-1 *Knapsack problem.* The problem is:

$$\text{(i) maximize} \quad \sum_{i=1}^{n} x_i p_i,$$

$$\text{subject to} \quad \sum_{i=1}^{n} x_i w_i \leqq M$$

$$x_i = 0, 1, \qquad 1 \leqq i \leqq n,$$

$$p_i > 0, \qquad w_i > 0$$

THEOREM 2.5.1. K1 $\in$ PC.

Proof. (a) K1 $\propto$ P1. Clearly, the problem is reducible to P1 if (i) is replaced by (i') $\sum x_i p_i \geqq Z$. Now if the length of the input is n then each $p_i < k^n$ for some k. So using the method of bisection, we can find the optimal Z in $\log_2 k^n = n \log_2 k$ query steps of (i') for some k, $k \leqq |\Sigma|$ (here $|\Sigma|$ = number of letters in the alphabet for the NDTM above).

(b) Sum of subsets of integers $\propto$ K1. Let $S = (s_1, \cdots, s_n)$ be the multiset of integers. We want to find a subset (if one exists) that sums to M. This may be stated in the form of a K1 problem as below:

$$\text{maximize} \quad \sum x_i s_i,$$

$$\text{subject to} \quad \sum x_i s_i \leqq M,$$

$$x_i = 0, 1.$$

From this we trivially conclude that the general 0-1 integer programming problem with nonnegative coefficients is complete. The 0-1 constraint may be replaced by the inequalities $x_i \leqq 1, 1 \leqq i \leqq n$.

The remarks of the last paragraph naturally lead us to the question of the status of the general integer programming problem (i.e., with both negative and positive coefficients). Here again, we are interested in only nonnegative solutions.

Problem I1. Determining if $Cx = b$ has a nonnegative solution is P-Hard. (Note the entries of C are integer. If C has all entries of the same sign, then the problem is P-Complete.)

To see this, consider the following formulation of the sum of subsets problem:

$$\sum_{i=1}^{n} w_i x_i = M,$$

$$w_i + y_i = 1, \qquad 1 \leqq i \leqq n.$$

Problem I2. Determining if $Cx \geqq 0$ has any integer solution (i.e., the x_i's are not constrained to be nonnegative) is P-Hard.

Application of Knuths' algorithm [6, vol. 2, p. 303] for obtaining integer solutions to $Cx = b$ yields a set of inequalities of the form $Dy \geqq w$. Setting $w = 0$ restricts the x to be $\geqq 0$. Hence $Dy \geqq 0$ has an integer solution iff $Cx = b$ has a nonnegative integer solution. Knuths' algorithm takes only polynomial time, so this problem is P-Hard. If the sign restriction on x is removed, then Knuths' algorithm solves $Cx = b$ in polynomial time. (This result was obtained together with H. B. Hunt III.)

Problem PF. *Permutation functions.* We are given a function $F(\mathbf{i})$ which is defined over all permutations of the elements of the vector $\mathbf{i} = (1, 2, \cdots, n)$. We wish to determine that permutation which minimizes F over all permutations. F is assumed to be polynomially computable.

THEOREM 2.5.2. PF $\in$ PC.

Proof. (a) PF α (P = NP). This part of the proof is very similar to that used in Theorem 2.5.1.

(b) Sum of subsets α PF. Define

$$F_k(\mathbf{i}) = -\left(\sum_{i=1}^{k} w(x_i)\right)\left(2M - \sum_{i=1}^{k} w(x_i)\right),$$

where x_i is the ith element of i.

We compute min F_k over all permutations of $\mathbf{i}$ for $k = 1, 2, \cdots, n$. If there is a subset that sums to M, then it has j elements in it, and min F_j is $-M$. If, on the other hand, for some $k = l$, min F_l is $-M$, then $\sum_{i=1}^{l} w(x_i) = M$. This defines an algorithm to solve the sum of subsets problem in polynomial time if we have a polynomial algorithm for PF.

Problem LB. *Assembly line balancing.* In this problem we are given n jobs $1, 2, \cdots, n$. Each job i requires a certain amount of processing time t_i. We have available machines, each having an available process time T. We want to determine the minimum number of machines needed to process all the jobs (the processing of a job cannot be split up among several machines).

THEOREM 2.5.3. LB $\in$ PC.

Proof. (a) LB $\propto$ (P = NP). This part of the proof is similar to Theorem 2.5.1.

(b) The following known member of PC shall be used (Karp [5]). Given a set of positive integers $s_1, s_2, \cdots, s_n$, is there a partition I such that

$$\sum_{i \in I} s_i = \sum_{i=1}^{n} s_i/2.$$

We show how this problem may be formulated as a line balancing problem. Let

$$t_i = s_i \quad \text{and} \quad T = \sum_{i=1}^{n} s_i/2;$$

then the jobs $1, 2, \cdots, n$ can be processed on 2 machines iff there is a partition I of the jobs such that

$$\sum_{i \in I} t_i = T = \sum_{i=1}^{n} s_i/2.$$

This is the minimum number of machines on which the jobs can be processed as $\sum_{i=1}^{n} t_i = 2T$.

Problem PI. *Quadratic programming.* Here, the constraints are linear while the optimization function is quadratic.

THEOREM 2.5.4. PI *is P-Hard.*

Proof. Sum of subsets of integers $\propto$ PI.

$$\begin{aligned} & \text{maximize} && \sum_i x_i(x_i - 1) + \sum_i x_i s_i = f(x), \\ \text{(i)} \qquad & \text{subject to} && \sum_i x_i s_i \leqq M, \\ & && 0 \leqq x_i \leqq 1. \end{aligned}$$

For $0 < x_i < 1$, $x_i(x_i - 1) < 0$. This, together with (i), implies $f(x) < M$ if for some i, $0 < x_i < 1$. Thus max $f(x) = M$ iff S has a subset that sums to M.

The following variation of this problem may also be shown to be P-Hard: linear programming with one nonlinear constraint. Call this problem PI(b). To show that sum of subsets $\propto$ PI(b), just consider the formulation:

$$\begin{aligned} & \text{maximize} && \sum x_i s_i, \\ & \text{subject to} && \sum_i x_i s_i \leqq M, \\ & && \sum_i x_i(x_i - 1) \geqq 0, \\ & && 0 \leqq x_i \leqq 1. \end{aligned}$$

2.6. Minimal equivalent Boolean form.

Problem B1. Given a formula B from the propositional calculus, we wish to find the shortest formula equivalent to it.

THEOREM 2.6.1. B1 $\in$ PC.

Proof. (a) B1 α P1. Define B1k to be the problem: is there a Boolean form of length k equivalent to B? We first show that a polynomial algorithm for P1 implies a polynomial algorithm for B1k. For this, we construct a nondeterministic Turing machine that guesses the Boolean form of length k and then uses the "tautology algorithm" to check that it is equivalent to B. If P1 works in $p(n)$ time, then the "tautology algorithm" works in $p_2(n)$ time (as tautology α P1), and so the Turing machine constructed above works in $p_2(n)$ time. Hence B1k α P1. The proof for B1 α B1k is similar to part (a) of the proof of Theorem 2.3.1. We note that this proof relies heavily on our informal notion of P-Reducibility. The proof does not show that B1 is polynomially related to the other problems in PC. If the time complexity of the tautology problem is $f_1(n)$ and that of P1 if $f_2(n)$, then this reduction gives a $f_2(f_1(n))$ algorithm for B1. If f_1 (and consequently f_2) is exponential, then $f_2(f_1(n))$ is of the form 2^{2^n}. All our other reductions have been of the form $p(n) \cdot f_2(n)$ or $f_2(p(n))$ for some polynomial p.[4]

(b) tautology α B1. A formula P is a tautology iff its minimal form is "1".

3. Conclusions. We have extended the class of known P-Complete problems to include some important applications from network flows, game theory, artificial intelligence and integer optimization. We have also introduced the notion of P-Hard. The results indicate that many of the problems for which no polynomial time bounded algorithm is known are related in terms of time complexity. Indeed, all the evidence to date suggests that there is no polynomial algorithm for any of these problems.

Acknowledgments. I am grateful to Professor Ellis Horowitz for many stimulating discussions on this subject. This work was motivated by the work of Cook [3] and Karp [5].

REFERENCES

[1] C. BERGE AND GHOUILA-HOWRI, *Programming, Games and Transportation Networks*, John Wiley, New York, 1964.

[2] C. L. CHANG AND J. R. SLAGLE, *An admissable and optimal algorithm for searching AND/OR graphs*, Artificial Intelligence, 2 (1971), pp. 117–128.

[3] S. A. COOK, *The complexity of theorem proving procedures*, Conference Record of Third ACM Symposium on Theory of Computing, 1971, pp. 151–158.

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

[5] 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.

[6] D. E. KNUTH, *Art of Computer Programming*, vols. 1 and 2, Addison-Wesley, Reading, Mass., 1969.

[7] W. F. LUCAS, *Some recent development in n-person game theory*, SIAM Rev., 13 (1971), pp. 491–523.

[8] D. M. MOYLES AND G. L. THOMSON, *An algorithm for finding a minimum equivalent graph of a digraph*, J. Assoc. Comput. Mach., 16 (1969), pp. 455–460.

[9] N. J. NILSSON, *Problem Solving Methods in Artificial Intelligence*, McGraw-Hill, New York, 1971.

[10] R. SIMON AND R. C. T. LEE, *On the optimal solution of AND/OR series-parallel graphs*, J. Assoc. Comput. Mach., 18 (1971), pp. 354–372.

[4] Note that here we are not saying that the best way to solve this problem takes 2^{2^n} time on a deterministic machine. In fact one can easily solve it in time bounded by 2^{cn}. We are just making the point that this particular reduction does not show that the two problems are polynomially related.

SIAM J. COMPUT.
Vol. 3, No. 4, December 1974

A NOTE ON PERFECT ELIMINATION DIGRAPHS*

D. J. KLEITMAN†

Abstract. The purpose of this note is to settle a conjecture on perfect elimination digraphs raised by Haskins and Rose in this journal, and to make some remarks that illuminate the nature of digraphs satisfying the conditions defined by them.

Key words. digraphs, sparse matrices, perfect elimination digraphs

The purpose of this note is to settle a conjecture on perfect elimination digraphs raised by Haskins and Rose [1] in this journal and to make some remarks that illuminate the nature of digraphs satisfying the conditions defined by them.

Haskins and Rose are concerned with characterizing "perfect elimination digraphs". The class P of perfect elimination digraphs, may be defined recursively, according to: G is in P if G contains a vertex x such that $G - \{x\}$ is in P, and if, for any distinct vertices y and z of G, (y, x) and (x, z) are arcs of G, then (y, z) is an arc of G. Here $G - \{x\}$ is the digraph obtained by omitting the vertex x and all arcs containing x from G.

Perhaps the clearest statement of the perfect elimination property is as follows. We say that x *shorts* y *and* z *in* G if the arcs (y, x) and (x, z) are arcs of G but (y, z) is not. Then G is *not* a perfect elimination digraph when it contains a set Q consisting of three or more vertices, every one of which shorts two others.

We say that x *separates* y *and* z (for x, y, z distinct) *in* G if x lies on a chordless path in G from y to z, or z to y; and x separates y and y if x lies on a chordless path in G from y to itself that contains at least one other vertex. Haskins and Rose introduce two conditions, the "antisymmetric separation condition" and the "chorded path condition". They note that each of these conditions must be satisfied by a digraph possessing the perfect elimination property, and they conjecture that the conditions are each sufficient for strongly connected digraphs to possess that property. Below we make several observations that lead to the conclusion that neither these conditions nor any other conditions that seek to characterize strongly connected perfect elimination digraphs or even strongly connected "minimally imperfect" elimination digraphs by behavior on a finite set of paths can succeed.

The "antisymmetric separation condition" states that there are no vertices x, y, u, v, w, t of G satisfying u, v, w, t separate (x, y) and x separates (u, v), y separates (w, t) (u, v, w, t need not be distinct here).

The "chorded path condition" is that no x, y, u, v, w, t satisfy u, v, w, t separate (x, y), and there is a path P_1 from u to v in G passing through x whose length cannot be diminished by one by using a chord, and likewise a path P_2 from w to t in G passing through y that cannot be similarly shortened.

* Received by the editors January 22, 1974, and in revised form June 5, 1974.

† Mathematics Department, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. Supported in part by the Office of Naval Research under contract ONR N00014-67-A-0204-0063.

Haskins and Rose show that the second of these conditions implies the first, and both must be satisfied by any perfect elimination digraph. They suggest the question: "Are there digraphs that satisfy either or both of these properties without being perfect elimination digraphs?" They provide an example of such a digraph that is not strongly connected.

We make the following remarks.

Remark 1. Every vertex other than x and y on a chordless (x, y) path (having at least 3 vertices) shorts other vertices on the path. Also every non-end vertex on paths like P_1 and P_2 in the "chorded path condition" shorts other vertices on these paths.

Remark 2. In consequence, if the antisymmetric separation condition fails, so that there exist x, y, u, v, w, t satisfying

(i) u, v, w, t lie on chordless (x, y) paths,

(ii) x lies on a chordless (u, v) path,

(iii) y lies on a chordless (w, t) path.

Every vertex on the at most 6 paths referred to here shorts two other vertices on these paths. Thus, G restricted to the paths mentioned here will fail to be a perfect elimination digraph.

Remark 3. Likewise, if the "chorded path condition" fails, G restricted to the union of the chordless (x, y) paths containing u, v, w and t and P_1 and P_2 will fail to be a perfect elimination graph.

Remark 4. Given a digraph G, let the digraph $G \oplus x_0$ be the digraph formed from G by adding one new vertex x_0 and arcs joining x_0 to and from every vertex of G. Then $G \oplus x_0$ is a perfect elimination digraph whenever G is (since no vertex of G shorts x_0, y for any y in $G \oplus x_0$, and any set of vertices Q, $Q \subset G \oplus x_0$, such that every element of Q shorts two others in Q will be such a Q in G, or the addition of x_0 to such a Q in G).

Remark 5. $G \oplus x_0$ satisfies Haskins and Rose's two conditions mentioned above whenever G does. (As there can be no chordless (x_0, y) or (y, x_0) path having 3 or more vertices, x_0 cannot be x, y, u, v, w or t satisfying the conditions in Remark 2 above. Since no chordless path in $G \oplus x_0$ containing x_0 can have any other interior vertices, x_0 cannot even lie on the 6 paths mentioned in Remark 2, so that if the antisymmetric separation condition fails for $G \oplus x_0$, it fails for G. The converse is trivial. For the "chorded path condition", one can observe that x_0 cannot lie on paths having the properties of P_1 and P_2 and the union of paths in Remark 3; therefore if they exist in $G \oplus x_0$, they exist in G and vice versa.)

Remark 6. Since $G \oplus x_0$ is strongly connected for any G, any example G of a nonstrongly connected digraph satisfying the two Haskins–Rose conditions not being a perfect elimination digraph immediately gives rise to a strongly connected example, $G \oplus x_0$.

Remark 7. Consider the digraph G_k consisting of the integers $1, 2, \cdots, k$ as vertices with (i, j) in G_k if $i + 1 = j$ and $i \neq 3 \pmod 4$ or if $i = j + 1$ and $j \neq 1 \pmod 4$. G_k is clearly a perfect elimination digraph, and yet no path in G_k contains 4 or more arcs.

Let $\bar{G}_k$ differ from G_k only in containing the arcs $(1, k)$ and $(k, 1)$ in addition to the arcs of G_k. G_k is a peculiarly oriented "chain"; $\bar{G}_k$ a similarly oriented polygon. It is obvious that $\bar{G}_k$ for $k \geqq 4$ is not a perfect elimination digraph, yet becomes one

if any one of its vertices is omitted. On the other hand, the six paths of Remark 2 or Remark 3 above cannot possibly include more than 27 vertices in $\bar{G}_k$, since only the one containing 1 and k can include more than 4 vertices, while that one can have 7. Thus if for $k \geqq 28$, $\bar{G}_k$ were to fail to satisfy the Haskins–Rose conditions, by Remarks 2 and 3, a subgraph of $\bar{G}_k$ having 27 or fewer vertices would fail to satisfy these conditions, and therefore not be a perfect elimination digraph. This cannot be. In fact, the six paths of Remarks 2 and 3 must overlap, and $\bar{G}_8$ satisfies the two conditions and is not a perfect elimination digraph. In light of Remarks 4, 5 and 6, the strongly connected "wheel graph" H_k for $k \geqq 8$ consisting of $\bar{G}_k \oplus x_0$ similarly satisfies these conditions without being a perfect elimination digraph.

Remark 8. In light of the previous remarks and the above example, it is clear that no property that seeks to characterize perfect elimination digraphs in terms of their behavior on a finite set of chordless paths can do so, in either the strongly connected or general case.

Remark 9. In the construction above, though x_0 supplies strong connectedness to the graph H_k, x_0 is irrelevant to the fact that H_k satisfies the given conditions without being a perfect elimination digraph. In fact, this irrelevance is the content of Remarks 4 and 5. One is tempted to ask, "Is there a similar example in which the strongly connectedness is more essentially interwoven into the digraph?" One could, in other words, formulate the following conjecture: in any strongly connected digraph G satisfying the Haskins–Rose conditions without being a perfect elimination digraph, there is a set of vertices Z such that the graph induced by G on $G - Z$ is a nonstrongly connected digraph satisfying the Haskins–Rose conditions without being a perfect elimination digraph. In our examples above, Z would consist of the vertex x_0. Even this conjecture fails to the same extent as the former one. A simple example is the "biwheel". Take two copies H_k, H_k' of H_k for $k \geqq 7$, identify x_0 with $1'$ and x_0' with 1 and remove the arcs $(k, 1)$ and $(k', 1')$ (leaving $(1, k)$ and $(1', k')$). The resulting graph is strongly connected, does not satisfy perfect elimination, does satisfy the Haskins–Rose conditions, and removal of any vertex leaves a perfect elimination digraph.

Acknowledgment. The author wishes to thank the referees for their helpful commentary.

REFERENCE

[1] L. Haskins and D. J. Rose, *Toward characterization of perfect elimination digraphs*, this Journal, 2 (1973), pp. 217–224.

SIAM J. COMPUT.
Vol. 3, No. 4, December 1974

REVERSAL-BOUNDED ACCEPTORS AND INTERSECTIONS OF LINEAR LANGUAGES*

RONALD BOOK†, MAURICE NIVAT‡ AND MICHAEL PATERSON§

Abstract. A Turing machine whose behavior is restricted so that each read-write head can change its direction only a bounded number of times is *reversal-bounded.* Here we consider nondeterministic multitape acceptors which are both reversal-bounded and also operate in linear time. Our main result shows that such an acceptor need have only three pushdown stores as auxiliary storage, each pushdown store need make only one reversal, and the acceptor can operate in real time.

Key words. nondeterministic multitape Turing acceptors, reversal-bounded, real time, linear context-free languages, language hierarchies, resource bounded acceptors

Introduction. In 1963, Rabin [10] showed that for deterministic Turing acceptors which operate in real time, two storage tapes yield more computational power than one, i.e., there is a set which is accepted in real time by a deterministic Turing acceptor with two storage tapes but which is not accepted in real time by any deterministic Turing acceptor with only one storage tape. The question of whether for every k there exists a set accepted in real time by a $k+1$ storage tape acceptor but not by any k tape acceptor became known as "Rabin's problem". In 1970, Book and Greibach [3] solved the nondeterministic version of this problem, showing that for every k, every language accepted in real time by a nondeterministic Turing acceptor with k storage tapes can be accepted in real time by a nondeterministic Turing acceptor with just two storage tapes. Recently, Aanderaa [1] solved Rabin's problem, showing that for every k, $k+1$ tapes are more powerful than k tapes. The purpose of the present paper is to establish results similar to those of Book and Greibach in a more restricted setting.

A Turing machine whose behavior is restricted so that each read-write head can change its direction only a bounded number of times is *reversal-bounded.* Reversal-bounded multitape acceptors which operate without further restrictions have been studied in [2]. Here we consider nondeterministic multitape acceptors which are both reversal-bounded and also operate in linear time Our main result, Theorem 3.1, shows that such an acceptor need have only three pushdown stores as auxiliary storage, each pushdown store need make only one reversal, and the acceptor can operate in real time. Hence, for every k, a language can be expressed as the nonerasing homomorphic image of the intersection of k linear context-free

* Received by the editors January 25, 1974, and in revised form June 3, 1974. These results were announced at the 6th Annual ACM Symposium on Theory of Computing held in May 1974, at Seattle, Washington. An extended abstract appears in the Proceedings of that Symposium. The research was performed while the second author visited the Center for Research in Computing Technology at Harvard University, and Project MAC at MIT, and while the third author visited Project MAC. The work was supported in part by the National Science Foundation under Grants GJ-30409 and GJ-34671.

† Yale University, New Haven, Connecticut 06520.

‡ University of Paris VII, Paris, France.

§ University of Warwick, Coventry, Warwickshire, England.

language if and only if it can be expressed as the length-preserving homomorphic image of the intersection of three linear context-free languages.

Let us consider this result from a different vantage point. Liu and Weiner [9] have shown that for any k, there is a language which can be expressed as the intersection of $k + 1$ context-free languages but which cannot be expressed as the intersection of k context-free languages. The argument in [9] shows more than is stated: for any k, there is a language which can be expressed as the intersection of $k + 1$ *linear* context-free languages (which happen to be deterministic counter languages) but not as the intersection of k context-free languages. Hence one obtains an infinite hierarchy of classes of languages by taking intersections of linear context-free languages. On the other hand, Baker and Book [2] have shown that if one considers the image under arbitrary homomorphic mappings of intersections of just two linear context-free languages, then one obtains the entire class of recursively enumerable sets. Hence there is only a trivial finite hierarchy obtained by taking images under arbitrary homomorphic mappings of intersections of linear context-free languages.

The situation studied here lies between the results of Liu and Weiner and those of Baker and Book. If one considers the images under *nonerasing* homomorphic mappings of intersections of linear context-free languages, then there is only a finite hierarchy: it is enough to consider images of intersections of just three linear context-free languages. (It is not known whether it is sufficient to consider intersections of only two such languages.) Thus only a finite hierarchy exists. This is similar to results in [3] which show that if one considers the images under nonerasing homomorphic mappings of intersections of arbitrary context-free languages, then there is only a finite hierarchy.

In order to prepare for our main results (§ 3), we set forth some facts about reversal-bounded acceptors and multitape acceptors in § 1. In § 2 we show that reversal-bounded computation in linear time is no more powerful than reversal-bounded computation in real time.

In this paper we assume a familiarity with concepts from automata and formal language theory. We do not define specific models in detail because the results are independent of the many minor variations in the definition of a Turing machine which abound in the literature. When certain conventions regarding a machine's operation are useful, we state them in as much detail as is necessary.

1. Preliminaries. In this section we present background material needed in later sections. The results in this section are not new, but some may not have been stated in the way they are presented here, and some are not well known.

Unless otherwise noted, we consider only multitape Turing acceptors which have a one-way read-only input tape and some number of Turing machine tapes as auxiliary storage. The set of input strings accepted by a machine M is denoted by $L(M)$. An acceptor M operates in *real time* if for each input string w accepted by M, there is an accepting computation with precisely $|w|$ steps;[1] alternatively, M reads a new input symbol at every step of the computation. An acceptor M operates in *linear time* if there is a constant $t \geqq 1$ such that for each input string w accepted by M, there is an accepting computation with at most $t|w|$ steps.

[1] For a string w, $|w|$ is the length of w.

Now let us consider multitape acceptors which operate with a bound on the number of "reversals" that each auxiliary storage tape can make in any computation. A *reversal* of a Turing machine tape is a step in a computation which causes the read-write head to change direction, e.g., after a sequence of transitions in which the head moves only to the right, a transition occurs in which the head moves to the left. See [2], [4] and [7] for results on reversal-bounded computations.

It is well known that the computation of a single Turing machine tape can be imitated by two pushdown stores without loss of time. Further, if the Turing machine tape makes r reversals, then each pushdown store will make at most r reversals (again, without loss of time).[2] Thus we shall restrict attention to multitape Turing acceptors with pushdown stores as auxiliary storage.

Suppose we restrict the operation of a pushdown store so that there is some fixed bound on the number of reversals that it can make in any computation. We claim that such a pushdown store can be imitated without loss of time by a number of pushdown stores such that each makes at most one reversal in any computation. If one pushdown store makes at most $2r - 1$ reversals in a computation, then consider r pushdown stores which operate as follows. For each $i = 1, \cdots, r$, pushdown store i is "pushed" when the original pushdown store is "pushed" between reversals $2i - 2$ and $2i - 1$. When the original pushdown store is "popped", then the jth pushdown store is "popped", where j is the largest index of a pushdown store which is not empty.

We summarize the above remarks in the following lemma.

LEMMA 1.1. *If M_1 is a multipushdown acceptor with k pushdown stores which operate in such a way that in every computation each pushdown store makes at most $2r - 1$ reversals, then from M_1 one can construct a multipushdown acceptor M_2 with kr pushdown stores such that $L(M_2) = L(M_1)$ and in every computation of M_2, each pushdown store makes at most one reversal. Further, M_2 is deterministic if M_1 is, and if $t(n)$ is a function which bounds M_1's running time (work space), then $t(n)$ also bounds M_2's running time (work space).*[3]

Consider a nondeterministic multipushdown acceptor M which operates in real time. If M has only one pushdown store, then the language accepted by M is a context-free language. However, even if M has more than one pushdown store as auxiliary storage, then the language $L(M)$ accepted by M can still be represented in terms of context-free languages.

LEMMA 1.2. *For every $k \geqq 1$, a language L is accepted by a nondeterministic Turing acceptor M with k pushdown stores as auxiliary storage which operates in real time if and only if there exist k deterministic context-free languages $L_1, \cdots, L_k$ and a length-preserving homomorphism*[4] *h such that L is the image of the intersection of $L_1, \cdots, L_k$ under h, i.e., $L = h(L_1 \cap \cdots \cap L_k)$. Further, if the i-th pushdown store of M, $1 \leqq i \leqq k$, is restricted in its operation (e.g., is reversal bounded or is a counter), then the language L_i can be accepted by a deterministic pushdown store*

[2] Note that a reversal of a pushdown store is a change from "pushing" to "popping" or vice versa.

[3] A bound on the work space is a bound on the number of tape squares on each storage tape used in the computation. It is a bound which applies to each storage tape, not to the sum taken over all tapes.

[4] Here we consider only homorphisms between free monoids. A homorphism $h: \sum^* \to \Delta^*$ is *nonerasing* if for all $w \in \sum^*$, $h(w) = e$ implies $w = e$, where e is the empty word (the identity of $\sum^*$), and is *length-preserving* if for all $w \in \sum^*$, $h(w)$ has the same length as w.

acceptor N_i which operates in real time and whose pushdown store is restricted in the same way as the i-th pushdown store of M.

The first sentence of Lemma 1.2 is Lemma 3.2 of [6].[5] The second part is an observation based on the proof given in [6]. Since this result is very useful (and does not appear to be well known), we present a brief sketch of the proof.

First recall that every context-free language can be accepted by a nondeterministic pushdown store acceptor which operates in real time, and that the class of languages accepted by those nondeterministic Turing acceptors with which we deal is closed under length-preserving homomorphism. Thus, the "if" part is straightforward.

Suppose M is a nondeterministic Turing acceptor with k pushdown stores as auxiliary storage and M operates in real time. At any one step of a computation of M, the transition depends upon (i) the input symbol currently read, (ii) the state of the finite-state control, (iii) for each of the k pushdown stores, the symbol being scanned on the *top* of the pushdown store. (Note that the total contents of a pushdown store do not affect a particular transition—only the symbol on the top of the store affects the transition.) A "history" of an accepting computation of M must record for each step the input symbol read, the current state of M, the symbols being scanned on the top of the k pushdown stores, and the transition to be made.

Let Q be the set of states of M, let Σ be the input alphabet of M, and for each $i = 1, \cdots, k$, let Γ_i be the alphabet for the ith pushdown store. If δ is the transition function of M, then for each $q \in Q$, $a \in \Sigma$, $\gamma_i \in \Gamma_i$, $1 \leqq i \leqq k$, label each transition in the set $\delta(q, a, \gamma_1, \cdots, \gamma_k)$ (recall: M is nondeterministic). Now for each $i = 1, \cdots, k$, construct a deterministic pushdown store acceptor M_i with $Q \cup \{D\}$, $D \notin Q$, as its set of states, $Q \times \Sigma \times \Gamma_1 \times \cdots \times \Gamma_k \times \Pi$ as its input alphabet, where Π is a list of all transitions given by δ, with pushdown store alphabet Γ_i, and with initial and accepting states just as in M. The transition function δ_i is defined as follows:

(i) for each $q \in Q$, $a \in \Sigma$, $\gamma_i \in \Gamma_i$, $1 \leqq i \leqq k$, and each transition π, if π is $(p, z_1, \cdots, z_k) \in \delta(q, a, \gamma_1, \cdots, \gamma_k)$, $p \in Q$, $z_j \in \Gamma_j^*$, $1 \leqq j \leqq k$, then

$$\delta_i(q, [q, a, \gamma_1, \cdots, \gamma_k, \pi], \gamma_i) = (p, z_i);$$

(ii) for each $q \in Q$, $y \in Q \times \Sigma \times \Gamma_i \times \cdots \times \Gamma_k \times \Pi$, $\gamma_i \in \Gamma_i$ such that $\delta_i(q, y, \gamma_i)$ is not defined by (i), $\delta_i(q, y, \gamma_i) = (D, \gamma_i)$;

(iii) for each $y \in Q \times \Sigma \times \Gamma_1 \times \cdots \times \Gamma_k \times \Pi$, $\gamma_i \in \Gamma_i$, $\delta_i(D, y, \gamma_i) = (D, \gamma_i)$.

What language does the pushdown store acceptor M_i accept? In an accepting computation, the pushdown store of M_i imitates the action of the ith pushdown store of M during an accepting computation, accepting if M would accept its input in a computation where the various choices of symbols from $\Gamma_1 \times \cdots \times \Gamma_k$ being input to M_i appear on the top of M's k pushdown stores. Thus, a string in $L(M_1) \cap \cdots \cap L(M_k)$ is a "history" of an accepting computation of M. Let $h: (Q \times \Sigma \times \Gamma_1 \times \cdots \times \Gamma_k \times \Pi)^* \to \Sigma^*$ be the homomorphism determined by defining $h(q, a, \gamma_1, \cdots, \gamma_k, \pi) = a$. If $L_i = L(M_i)$ for $i = 1, \cdots, k$, then clearly $L = h(L_1 \cap \cdots \cap L_k)$.

[5] A more abstract version of this result (stated in terms of AFL and AFA) is the "multitape representation" theorem of [5].

Notice that each M_i reads a new input symbol at every step, that is, each M_i operates in real time.

Notice that if some Γ_i is an alphabet with only one symbol, then M_i is a counter. If the ith pushdown store of M is reversal-bounded, then there is a bound on the number of reversals M_i makes in its accepting computations. Altering M_i's finite-state control to count the number of reversals and to halt and reject if M_i attempts to accede this bound yields the reversal-bounded acceptor N_i.

This concludes our sketch of the proof of Lemma 1.2.

Recall that a language is linear context-free if and only if it is accepted by a nondeterministic pushdown store acceptor which operates in such a way that in every computation the pushdown store makes at most one reversal (see [4]).[6] Combining this fact with Lemmas 1.1 and 1.2, we note the following fact.

LEMMA 1.3. *If M is a nondeterministic Turing acceptor which has k pushdown stores, each of which makes at most $2r - 1$ reversals in any computation and which operates in real time, then the language accepted by M is the length-preserving homomorphic image of the intersection of kr linear context-free languages, i.e., there exist linear context-free languages $L_1, L_2, \cdots, L_{kr}$ and a length-preserving homomorphism h such that $L(M) = h(L_1 \cap \cdots \cap L_{kr})$. Further, each of the linear context-free languages is accepted by a deterministic pushdown store acceptor which runs in real time and operates in such a way that in every computation the pushdown store makes at most one reversal.*

In Lemma 1.2, the restriction that h be length-preserving is of importance since it is shown in [2] that every recursively enumerable set is the homomorphic image of the intersection of two linear context-free languages.

2. Linear time is no more powerful than real time. In [3] it is shown that for nondeterministic multitape Turing acceptors, linear time is no more powerful than real time. That is, if M_1 is a nondeterministic multitape Turing acceptor which operates in linear time, then there is a nondeterministic multitape Turing acceptor M_2 such that M_2 operates in real time and $L(M_2) = L(M_1)$. In this section we give a new proof of this result, a proof that shows that one need add only reversal-bounded pushdown stores to M_1.

LEMMA 2.1. *Let M_1 be a multipushdown acceptor with p pushdown stores as auxiliary storage. If M_1 operates in linear time, then there is a nondeterministic acceptor M_2 with $p + 3$ pushdown stores which* (i) *operates in real time,* (ii) *accepts precisely $L(M_1)$ (i.e., $L(M_2) = L(M_1)$), and* (iii) *operates in such a way that in every computation three of the pushdown stores each make at most one reversal.*

Proof. Let $t \geqq 1$ be a constant such that for every input string accepted by M_1, there exists an accepting computation of M_1 on w with at most $t|w|$ steps. From M_1 we construct a nondeterministic acceptor M_2 with $p + 3$ pushdown stores. The first three pushdown stores will be referred to as Tapes 1, 2, and 3. M_2 operates in three distinct phases as follows.

Phase 1. In this phase, Tapes 1 and 3 are active while Tape 2 is not. The computation begins with M_2 writing an arbitrary string $a_1 \cdots a_n$ on Tape 1 and simul-

[6] Note that one can modify the argument used in [4] to obtain a nondeterministic acceptor with both runs in real time and makes only one reversal.

taneously simulating a computation of M_1 on $a_1 \cdots a_n$, that is, M_2 nondeterministically "guesses" what the steps at which M_1 reads a new input symbol are, guesses what this symbol is, and stores this guess on Tape 1. To achieve the improvement in running time, M_2 will imitate $3t$ steps of M_1's computation during each step of M_2's computation and will store the string $a_1 \cdots a_n$ by storing $3t$ symbols in each tape square of Tape 1.[7]

Simultaneously, M_2 will read a new input symbol from its own input tape at each step of the computation, storing the real input on Tape 3 with $3t$ symbols on each tape square.

If the computation of M_1 being imitated by M_2 is an accepting computation, then M_2 transfers to Phase 2 (below) upon completing the imitated computation. Otherwise, M_2 halts.

If the computation of M_1 being imitated by M_2 is both an accepting computation and has at most tn steps, then M_2 takes at most $tn/3t = n/3$ steps to complete Phase 1.[8]

Phase 2. M_2 "pops" Tape 1, at the rate of one tape square (hence, $3t$ symbols) per step, storing the contents on Tape 2 until it guesses that only $a_1 \cdots a_{n/2}$ remains on Tape 1 and $a_n \cdots a_{n/2+1}$ has been written on Tape 2. During this process, M_2 continues to read a new symbol from its input tape at each step of the computation, storing the real input on Tape 3 with $3t$ symbols in each tape square. Phase 2 continues until M_2 guesses that a total of $n/2$ input symbols have been read.

Phase 3. Tapes 1 and 3 are now matched against each other to check that the first half of the guessed input string actually equals the first half of the real input. At the same time, M_2 reads the remaining input at the rate of one per step and checks these against Tape 2 to verify the second half of the guess. If the checks all succeed, then M_2 accepts the input at the step where it reads the last input symbol.

M_2 runs in real time, and clearly each of Tapes 1, 2, and 3 make at most one reversal. □

From the construction used in the proof of Lemma 2.1, we note that any restrictions on the behavior of the tapes of M_1 will be preserved by the corresponding tapes of M_2. Thus we have the following result.

COROLLARY. *Let M_1 be a multipushdown acceptor with k pushdown stores which operate in such a way that in every computation each pushdown store makes at most one reversal. If M_1 operates in linear time, then there is a nondeterministic multipushdown acceptor M_2 with $k+3$ pushdown stores such that* (i) $L(M_2) = L(M_1)$, (ii) *M_2 operates in real time, and* (iii) *in every computation, each pushdown store makes at most one reversal. Thus there exist deterministic linear context-free languages $L_1, \cdots, L_{k+3}$ and a length-preserving homomorphism h such that $L(M_1) = h(L_1 \cap L_2 \cap \cdots \cap L_{k+3})$.*

In the next section we show how the total number of tapes used may be reduced to just three. A technique which will be very useful in proving that result is illustrated in the proof of the following lemma.

[7] The imitation of $3t$ steps of M_1's computation by M_2 follows the standard "speed-up" techniques of [8].

[8] We leave to the reader the alterations needed when n is not divisible by $6t$.

LEMMA 2.2. *For any p and any finite alphabet Σ, there are linear context-free languages, L_1 and L_2, and a length-preserving homomorphism h such that $\{(wc)^p | w \in \Sigma^*\} = h(L_1 \cap L_2)$, where c is a symbol not in Σ.*

Proof. We assume that $p = 2q$, the proof for p odd being very similar, and we describe a nondeterministic online acceptor M such that $L(M) = \{(wc)^{2q} | w \in \Sigma^*\}$, M operates in real time, and M has two pushdown stores, each of which make only one reversal. The existence of L_1, L_2, and h follows from Lemma 1.3.

We refer to M's two pushdown stores as Tape 1 and Tape 2. Each tape has two tracks. On each tape M will write a sequence of "blocks". A *block* consists of two strings u, v of the same length terminated by endmarkers to the final symbol. One string is written on each track, say u on the top track and v on the bottom.

M begins its computation by reading its input at the rate of one symbol per step. It writes $q - 1$ blocks on both Tapes 1 and 2 as follows. On the top track of each tape, M records the input that it reads until the $(q - 1)$st c has been read. On the bottom track of each tape, M nondeterministically guesses and writes an arbitrary symbol from Σ at every step at which a symbol from Σ is read as input, and M writes a c at every step at which a c is read as input. Thus, each block consists of a pair $u_i c$, $v_i c$ of strings in $\Sigma^* c$ such that $|u_i| = |v_i|$ and $u_i c$ is part of the actual input read by M (the total input read by M up to this point is $u_1 c u_2 c \cdots u_{q-1} c$).

Now M reads the next portion x of input, copying it onto the top tracks of both tapes, and guesses a string y, $|y| = |x|$, which is written on the bottom tracks of both tapes. At some point in reading u_q, M starts to pop Tape 2, copying the symbols from x, y onto the bottom and top tracks, respectively, of Tape 1. Simultaneously, M matches the symbols popped from the bottom track of Tape 2 with the actual input read. If the next input symbol is c, then it is copied onto both tracks of Tape 1. After this operation, the contents of the tapes is as follows.

$$\begin{aligned} \text{Tape 1:} \quad & u_1 c\, u_2 c \cdots c u_{q-1}\, c x y^R c \\ & v_1 c\, v_2 c \cdots c v_{q-1}\, c y x^R c \\ \text{Tape 2:} \quad & u_1 c\, u_2 c \cdots c u_{q-1}\, c \\ & v_1 c\, v_2 c \cdots c v_{q-1}\, c \end{aligned}$$

where the rightmost c is being scanned.[9] If all the matched symbols have agreed, then the input read so far is recorded on the top track of Tape 1.

M's next operation is to pop both tapes simultaneously, comparing blocks symbol by symbol. If these comparisons all succeed, then $u_1 = u_2 = \cdots = u_{q-1} = (xy)^R = ((yx^R)^R = (v_{q-1})^R = \cdots = (v_2)^R = (v_1)^R$. Simultaneously, M compares the bottom track of Tape 1 with the remaining input, and continues this comparison either until the input is exhausted or until Tape 1 is empty. If all of these comparisons agree, then the input to M was $(wc)^{2q}$ for some $w \in \Sigma^*$, where $w = u_i = (v_i)^R = (xy)^R$ for each i, $1 \leqq i \leqq q - 1$, and M halts in an accepting state. Here, $|w|$ is even. The case for $|w|$ odd is dealt with in a similar way.

It is clear that M meets all the requirements of the lemma. □

[9] For a string w, w^R is the reversal of w.

3. Main results.

THEOREM 3.1. *Let L be a language. The following are equivalent:*

(i) *L is accepted by a nondeterministic multipushdown acceptor which operates in such a way that in every computation each pushdown store makes at most a bounded number of reversals and which runs in linear time;*

(ii) *L is accepted by a nondeterministic multipushdown acceptor which operates in such a way that in every accepting computation each pushdown store makes at most one reversal and which runs in real time;*

(iii) *L is the length-preserving homomorphic image of the intersection of some finite number of linear context-free languages;*

(iv) *L is accepted by a nondeterministic acceptor with three pushdown stores which operates in such a way that in every computation each pushdown store makes at most one reversal and which runs in real time;*

(v) *L is the length-preserving homomorphic image of the intersection of three linear context-free languages.*

From Lemma 1.1 and the corollary to Lemma 2.1, we see that (i) implies (ii). From Lemma 1.3, we see that (ii) implies (iii), and that (iv) implies (v). It is easy to see that the class of languages accepted by nondeterministic multipushdown reversal-bounded acceptors is closed under nonerasing homomorphism, and recalling that a language is linear context-free if and only if it is accepted by a nondeterministic pushdown store acceptor which makes at most one reversal on the pushdown store and which runs in real time, we see that (v) implies (iv). Obviously, (iv) implies (i). Thus we are left with the task of showing that (iii) implies (iv). To accomplish this, we rely on the fact that the class of languages specified by the acceptors in (iv) is closed under nonerasing homomorphism, so that it is sufficient to establish the following lemma.

LEMMA 3.2. *For any $k \geqq 1$ and any choice of k linear context-free languages, $L_1, \cdots, L_k$, there is a nondeterministic acceptor M with three pushdown stores as auxiliary storage such that* (i) $L(M) = L_1 \cap \cdots \cap L_k$, (ii) *$M$ operates in real time, and* (iii) *in every computation of M, each pushdown store makes at most one reversal.*

Proof. Before we embark on a detailed description of the construction of M, it may be helpful to present an informal outline showing how the various techniques, already introduced, are to be combined. For each $i = 1, \cdots, k$, let M_i be a nondeterministic one-reversal pushdown store acceptor which accepts the language L_i in real time. We must describe a nondeterministic acceptor M with three pushdown stores, referred to as Tapes 1, 2, and 3.

The principal problem—of how M can simulate all of the M_i while using only one reversal per tape—is solved by using Tape 3 first to simulate the "pushing" phases of the computations of $M_1, \cdots, M_k$ successively and then to simulate the "popping" phases of the computations of $M_k, \cdots, M_1$ successively. This is accomplished by using Tapes 1 and 2 to provide $2k$ copies of the same string for the $2k$ sections of the simulation. We need to ensure that the correct segments of these words are read during each part of the simulation, and for this purpose the "turning point" for each M_i is inscribed on each copy.

With all the simulations, M would run naturally in linear time rather than in real time. The method of the proof of Lemma 2.1 is employed to make M operate in real time. In the first portion of M's computation, all of the simulations are

completed on guessed input, while the real input is stored on another channel of Tape 1. Further, another copy of the guessed input is stored on Tape 3, with the first half in forward order on one channel and the second half in reverse order on another channel. The final phase involves checking both the real input stored on Tape 1 and the real input as it continues to be read against these two channels, just as in the proof of Lemma 2.1.

One task for Tapes 1 and 2 is to provide multiple copies of the guessed input for the simulation of the M_i's and also a "folded" copy of this guess which is preserved on Tape 2 for the checking phase just described. An examination of the construction used in proving Lemma 2.2 shows that these elements are already present as long as M guesses both parts of each block. The corresponding value there of q is $4k + 4$, and the folded string is found on the two channels of the first half block. Tape 1 will have three channels, while Tapes 2 and 3 each will have two channels.

Now we can proceed to a more detailed account. A *block* is to consist of two strings of symbols written with $4k + 5$ symbols to the tape square on each of two channels of a tape (so there are $2(4k + 5)$ symbols on each tape square). The top channel contains some string $a_1 \cdots a_n$, for some n, which is increased in length to the next multiple of $8k + 10$ by adding initial dummy symbols standing for blanks, i.e., $b \cdots ba_1 \cdots a_n$. The a_i's are taken from the input alphabet, except that k distinct special markers can be added to any symbol, and in $a_1 \cdots a_n$, each marker occurs exactly once. The ith marker ($1 \leqq i \leqq k$) will be intended to indicate the position in the string $a_1 \cdots a_n$ at which M_i reverses its pushdown store (from pushing to popping). The lower channel is of the same form but in reverse order.

An accepting computation of M can be described in three phases. Input symbols are read throughout at the rate of one per step, so M operates in real time.

Phase 1. For the first d steps, where M guesses d, the only activity is that the input symbols are recorded on the third channel of Tape 1, with one symbol per tape square. (The correct value for d to allow the computation to succeed will be defined later.) After d steps, M begins to guess a block B_0 and writes it on the first two channels of Tapes 1, 2 and 3. Throughout Phase 1 (and Phase 2) the input to M continues to be copied onto channel 3 of Tape 1 at the rate of one symbol per tape square.

Phase 2. M guesses a new block B_1 and writes it on the first two channels of Tape 1 and on the two channels of Tape 2. As this proceeds, M simulates M_1 on the string $a_1 \cdots a_n$ contained in this block, using a channel of Tape 3 to simulate M_1's pushdown store. When marker 1 in this string is reached, if M_1 is at that point of its computation where the pushdown store is to be reversed, then the state of M_1 is recorded on the other channel of Tape 3 and the simulation is suspended. If M_1 is not at that point, then this computation of M is unsuccessful. Since $4k + 5$ simulated input symbols must be read at each step, M's simulation of M_1 must be sped up by a factor of $4k + 5$, using the techniques of [8]. M proceeds in the same way with $k - 1$ more blocks $B_2, \cdots, B_k$ and simulates the "pushing" part of computations of $M_2, \cdots, M_k$ in turn. See Fig. 1 for a sketch of M's tape contents at this point.

Next, k more blocks are guessed and recorded on Tapes 1 and 2, while M attempts to simulate in order the "popping" part of computations by M_k,

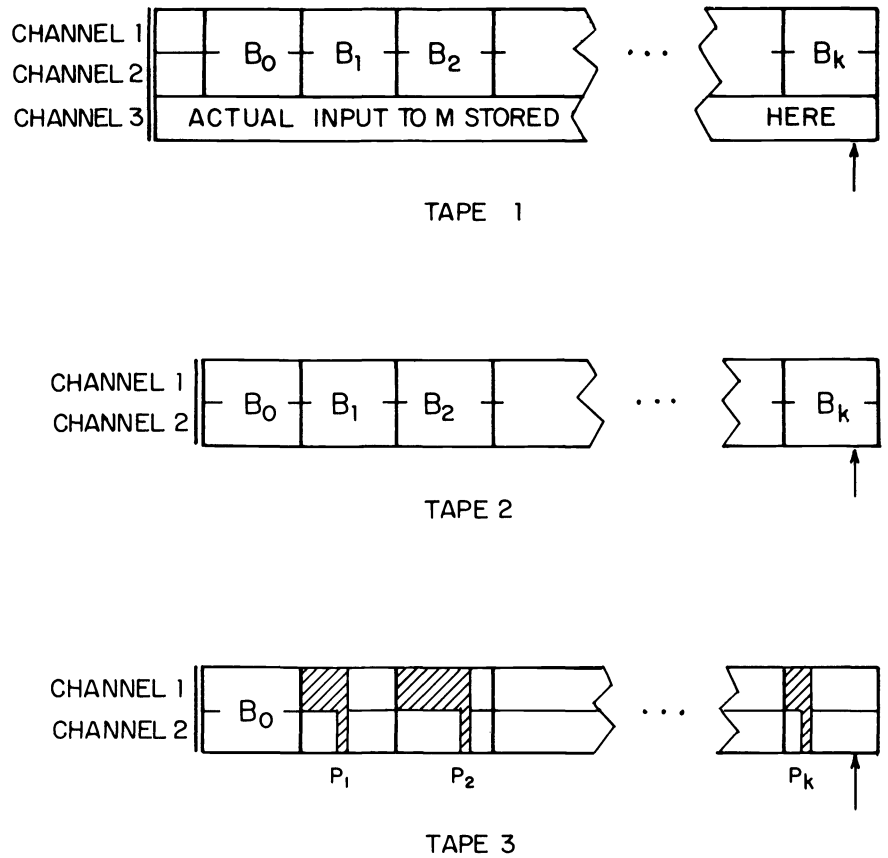


FIG. 1. *A sketch of the contents of Tapes* 1, 2 *and* 3 *after the first part of Phase* 2. *For each i,* P_i *is the configuration of* M_1 *as it reverses*

$M_{k-1}, \cdots, M_1$. The simulation of M_i is resumed in the state recorded on Tape 3 and at the point in the guessed block marked by the ith marker. The computation of M is unsuccessful unless all thc M_i are simulated to acceptance. See Fig. 2 for a sketch of M's tape contents at this point.

A *half-block* is now guessed and copied onto Tapes 1 and 2. Then this half-block is popped from Tape 2, and the reverse of channels 1 and 2 of Tape 2 is pushed onto channels 2 and 1, respectively, of Tape 1. As this half-block is popped from Tape 2, the same number of symbols are popped from Tape 3. The real input to M is still being recorded on the third channel of Tape 1.

Intermission (for the reader, not for M). Let us review the situation at this point. Tape 1 contains the real input read by M so far, with one symbol per tape square on the third channel. The other two channels of Tape 1 contain $2k + 2$ blocks, say $B_0, B_1, \cdots, B_{2k+1}$. The simulations of $M_1, \cdots, M_k$ have been completed successfully but not necessarily on the same input string. Tape 3 contains only part of the block written there in Phase 1, and so has the form

$$b \cdots b a_1 \cdots a_r$$

$$a_n \cdots a_{s+1}$$

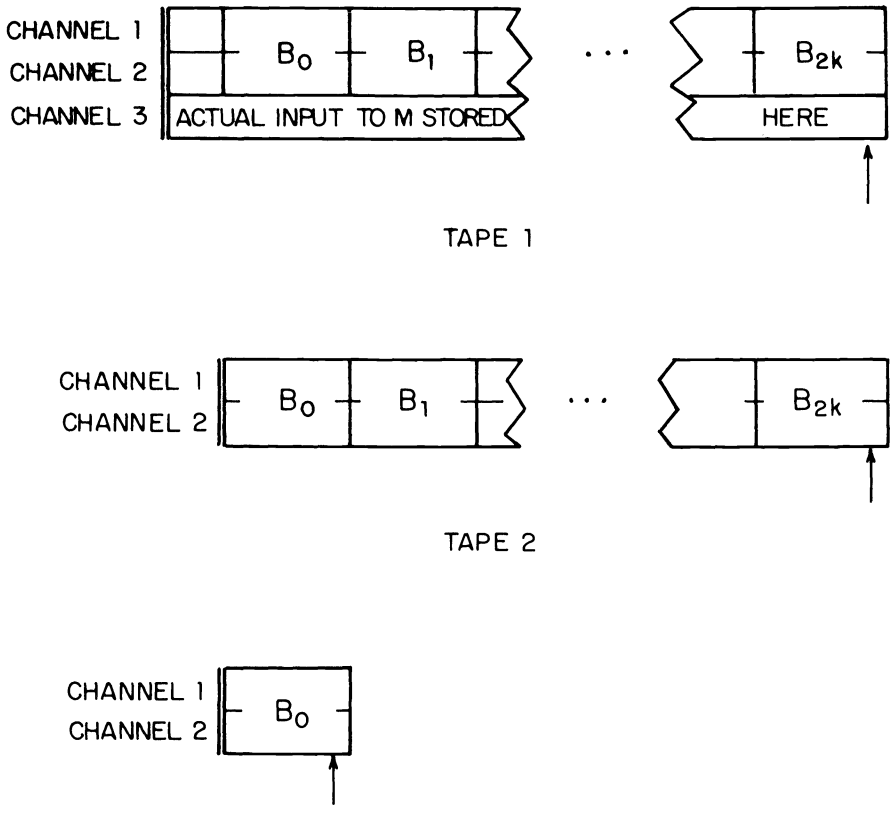


FIG. 2. *A sketch of the contents of Tapes* 1, 2 *and* 3 *after the second part of Phase* 2

for some r, s and n.

The parameter d guessed by M at the beginning of Phase 1 is to be such that the total number of steps taken in Phases 1 and 2 is precisely r. We now resume the development.

Phase 3. Tapes 1 and 2 are popped together and compared to verify that $B_{2k+1} = B_{2k}$ and then to verify that $B_{2k} = B_{2k-1}, \cdots, B_1 = B_0$. If all of these comparisons succeed, then $B_0 = B_1 = \cdots = B_{2k+1}$, and indeed all the simulations were appropriate and were accepting computations on the same guessed input string. Furthermore, the blocks are of even length, and it is precisely half of the block that remains on Tape 3 at the beginning of Phase 3. This has the form

$$b \cdots b a_1 \cdots a_r$$
$$a_n \cdot \cdot \cdot a_{r+1}$$

where the guessed string $a_1 \cdots a_n$ is in $L(M_i) = L_i$ for every $i = 1, \cdots, k$. If the guess of d was correct, then the first r symbols of the guessed input to M are stored in forward order on the first channel of Tape 3, the remaining $n - r$ are on

the second channel in reverse order, while the first r symbols of the real input are on the third channel of Tape 1 at the start of Phase 3. Therefore, as Tape 1 is popped during Phase 3 in order to match the blocks, Tape 3 can be popped at the rate of one square every $4k + 5$ steps, and the first r symbols of the real and guessed input can be verified to be identical. Simultaneously, the second channel of Tape 3 can be compared with the real input symbols which are being read at the rate of one per step. If these checks are all successful, then the computation is complete as the nth and final input symbol is read.

If the real input is of length n and the blocks written by M are of length $2m$, then in the event of a successful simulation:

$e + n = 2m(4k + 5)$ where e is the number of dummy symbols in a block;

$r = d + 2m(2k + 2)$ = maximum length of Tape 1;

$n - r = m(4k + 5)$ = number of packed symbols in half block on Tape 3.

Hence, $d = m - e$. Since $e < 8k + 10$ and $m = \lceil n/(8k + 10) \rceil$, d is nonnegative provided n is sufficiently large. Inputs of length less than this can be dealt with by the finite state control of M.

It is clear that M operates in real time and makes only one reversal on each tape, and that $L(M) = L_1 \cap L_2 \cap \cdots \cap L_k$. □

Suppose that $L_1, \cdots, L_k$ are arbitrary context-free languages. We do not know whether $L_1 \cap \cdots \cap L_k$ can be accepted in real time by an acceptor with pushdown stores that make a bounded number of reversals. However, by varying the construction used in the proof of Lemma 3.2, it is easy to see that such an intersection can be accepted in real time by an acceptor with three pushdown stores such that two of the pushdown stores make at most one reversal. This leads to the following result which strengthens the main result in [3].

THEOREM 3.3. *Let L be a language. The following are equivalent:*

(i) *L is accepted by a multitape nondeterministic Turing acceptor which operates in linear time;*

(ii) *L is the length-preserving homomorphic image of the intersection of some finite number of context-free languages;*

(iii) *L is accepted by a nondeterministic acceptor which operates in real time and which has three pushdown stores as auxiliary storage tapes, two of which make at most one reversal during any computation;*

(iv) *there exist a context-free language L_1, two linear context-free languages L_2 and L_3, and a length-preserving homomorphism h such that $L = h(L_1 \cap L_2 \cap L_3)$.*

Let f be a function acting as a time bound. A nondeterministic acceptor M operates within time bound f if for every input string w accepted by M, there is some accepting computation of M on w with no more than $f(|w|)$ steps. The proof of Theorem 3.1 can be extended to establish the following result.

THEOREM 3.4. *For any time bound f, if M_1 is a multitape Turing acceptor which operates within time bound f, then there is a nondeterministic Turing acceptor M_2 such that M_2 operates within time bound f, M_2 has three pushdown stores as auxiliary storage, two of M_2's pushdown stores make at most one reversal in any computation, and M_2 accepts just those input strings accepted by M_1 (i.e., $L(M_1) = L(M_2)$). Further, if M_1 operates in such a way that each of its storage tape heads makes at most a bounded number of reversals, then M_2's third pushdown store need make only one reversal.*

4. Concluding remarks. In Theorem 3.1 we specify nondeterministic reversal-bounded multitape acceptors that operate in linear time. We show that it is sufficient to consider those acceptors which have just three pushdown stores, where each pushdown store makes at most one reversal, and where the acceptor operates in real time. In particular, this shows that there is only a finite hierarchy of classes of languages based on the number of storage tapes a reversal-bounded acceptor needs. The fact that just three pushdown stores are used plays no particular role. However, it is an open question whether these acceptors are more powerful than those reversal-bounded acceptors with just two pushdown stores.

Recall from [3, Lemma 1.2] that a language is accepted in real time (in fact, linear time) by a nondeterministic multitape Turing acceptor if and only if it is the length-preserving homomorphic image of the intersection of some finite number of context-free languages. (By the present Theorem 3.3 or [3], it is sufficient to intersect three context-free languages.) Thus if every language accepted in real time by a nondeterministic multitape Turing acceptor is accepted in real time by a nondeterministic reversal-bounded multipushdown acceptor, then every context-free language is accepted in real time by a nondeterministic reversal-bounded multipushdown acceptor. Conversely, suppose that every context-free language is accepted by a nondeterministic reversal-bounded multipushdown acceptor. Clearly, the class of languages accepted in real time by reversal-bounded multipushdown acceptors is closed under intersection and length-preserving homomorphism. Hence if this class contains all the context-free languages, then it contains all the languages accepted in real time by multitape acceptors. We conjecture that this is not the case, that is, we conjecture that (i) the class of languages accepted in real time by nondeterministic multitape Turing acceptors properly contains the class of languages accepted in real time by nondeterministic reversal-bounded multipushdown acceptors, and equivalently that (ii) there exist context-free languages which are not accepted in real time by nondeterministic reversal-bounded multipushdown acceptors. Since there are languages in the latter class which are not context-free (e.g., $\{a^n b^n c^n | n > 0\}$), (ii) is equivalent to the conjecture that this class and the class of context-free languages are not comparable.

REFERENCES

[1] S. AANDERAA, *On k-tape versus (k + 1)-tape real time computation*, to appear.

[2] B. BAKER AND R. BOOK, *Reversal-bounded multi-pushdown machines*, J. Comput. System Sci., 8 (1974), pp. 315–332.

[3] R. BOOK AND S. GREIBACH, *Quasi-realtime languages*, Math. Systems Theory, 4 (1970), pp. 97–111.

[4] S. GINSBURG AND E. SPANIER, *Finite-turn pushdown automata*, SIAM J. Control, 4 (1966), pp. 429–453.

[5] S. GREIBACH AND S. GINSBURG, *Multi-tape AFA*, J. Assoc. Comput. Mach., 19 (1972), pp. 192–221.

[6] S. GREIBACH AND J. HOPCROFT, *Scattered context grammars*, J. Comput. Systems Sci., 3 (1969), pp. 233–247.

[7] J. HARTMANIS, *Tape-reversal bounded Turing machine computations*, Ibid., 2 (1968), pp. 117–135.

[8] J. HARTMANIS AND R. STEARNS, *On the computational complexity of algorithms*, Trans. American Math. Soc., 117 (1965), pp. 285–306.

[9] L. LIU AND P. WEINER, *An infinite hierarchy of intersections of context-free languages*, Math. Systems Theory, 7 (1973), pp. 185–192.

[10] M. RABIN, *Real-time computation*, Israel J. Math., 1 (1963), pp. 203–211.

SIAM J. COMPUT.
Vol. 3, No. 4, December 1974

EXISTENCE OF GRAPHS WITH THREE SPANNING TREES AND GIVEN DEGREE SEQUENCE*

SUKHAMAY KUNDU†

Abstract. A simple necessary and sufficient condition is given for a degree sequence to be realizable by a graph that contains three mutually edge-disjoint spanning trees.

Key words. degree sequence, spanning tree, realizability

The problem. In this note, we are concerned with a certain variation of the realizability problem of a degree sequence. A degree sequence $[d_i]$ consists of n positive integers d_i, which are to be regarded as the number of edges incident with vertices v_i ($1 \leqq i \leqq n$) of some n-point graph G. Problem: when does there exist a graph G whose degree sequence is $[d_i]$ and which contains three mutually edge-disjoint spanning trees?

For our purposes, a graph shall have no multiple edges and no loops, and shall have 6 or more vertices. (The complete graph K_6 on six vertices is the smallest graph that contains three edge-disjoint spanning trees.) Moreover, the sequence $[d_i]$ will be assumed to be nonincreasing, i.e., $d_i \geqq d_{i+1}$ for $1 \leqq i \leqq n-1$.

It is a simple matter to see that the following two conditions are necessary for the existence of a graph G containing 3 edge-disjoint spanning trees.

(a) $[d_i]$ is graphical, i.e., there exists a graph whose degree sequence is $[d_i]$.

(b) $d_i \geqq 3$ for all i, and the sum of d_i is at least $6(n-1)$.

The second condition holds because each tree has at least one edge incident with v_i, and the trees together contain $3(n-1)$ edges. It is the purpose of this note to show that the conditions (a) and (b) are also sufficient.

THEOREM. *A degree sequence $[d_i]$ is realizable by a graph G containing three mutually edge-disjoint spanning trees if and only if conditions* (a) *and* (b) *hold.*

A similar theorem was previously obtained for the case of two trees [2]. It would be interesting to know if conditions (a) and (b), suitably modified as $d_i \geqq t$ and $\sum d_i \geqq 2t(n-1)$, remain sufficient for the general case, namely, the existence of a graph G containing t ($\geqq 4$) mutually edge-disjoint spanning trees whose degree sequence is given by $[d_i]$. We conjecture that this is indeed the case.

Proof of the theorem. Relevant to our proof by induction are the following two lemmas (see [3]), whose proofs are omitted here. The first lemma allows us to construct the spanning trees of graph G from those in the realizing graph of certain reduced sequences, and Lemma 2 shows that such reduced sequences are graphical.

LEMMA 1. *Let T_i, $i = 1, 2, 3$, be three mutually edge-disjoint trees, all spanning the same vertex set. Then for every choice of a vertex v_i and two trees T_j, T_k, there exist two edges, one from each tree T_j and T_k, which are themselves nonadjacent and are not incident with vertex v_i.*

* Received by the editors July 24, 1973.

† IBM Thomas J. Watson Research Center, Yorktown Heights, New York. Now at Department of Computer Sciences, University of Texas at Austin, Austin, Texas 78712.

LEMMA 2. *Let* $\sum d_i = 6(n-1)$ *and* $d_i \geqq 4$ *for all* i. *Then there exists a graph* G *with degree sequence* $[d_i]$ *if and only if* $d_1 \leqq n-1$.

We now assume that $n \geqq 7$ and that the theorem is true for sequences of length $(n-1)$ or less. There are three cases to consider according as $d_n = 3$, 4, or 5.

Case 1. Suppose $d_n = 3$. By the induction hypothesis, there exists a graph H with degree sequence

$$[d_1 - 1, d_2 - 1, d_3 - 1, d_4, d_5, \cdots, d_{n-1}]$$

which contains three mutually edge-disjoint spanning trees. To graph H, we add the vertex v_n and the edges $\{(v_n, v_1), (v_n, v_2), v_n, v_3)\}$, and call the resulting graph G. The spanning trees of H together with one of the edges incident with v_n constitute the required trees in G.

A similar construction holds if $\sum d_i \geqq 6(n-1)+2$ and $d_n = 4$, or $\sum d_i \geqq 6(n-1)+4$ and $d_n = 5$, or $d_n \geqq 6$. Of course, only three of the d_n edges incident with vertex v_n are included in the spanning trees of G in each case.

Case 2. Let $d_n = 4$ and $\sum d_i = 6(n-1)$. The reduced sequence

$$[d_i'] = [d_1 - 1, d_2 - 1, d_3, \cdots, d_{n-1}]$$

is graphical unless $[d_i]$ equals one of the following sequences:

$$[8,8,8,4,4,4,4,4], \quad [7,7,7,5,4,4,4,4],$$
$$[6,6,6,6,4,4,4], \quad [6,6,6,5,5,4,4].$$

Figures 1(a)–(d) give realizing graphs of these sequences, each of which is a disjoint union of three spanning trees shown respectively in three different kinds of lines. If $[d_i']$ is graphical, let H be a realizing graph with disjoint spanning trees T_1', T_2', T_3'. One of the trees, say T_3', (by Lemma 1 or directly) contains an edge (v_p, v_q) which is not incident with v_1 or v_2. Now we let G denote the graph obtained by adding vertex v_n to H and making it adjacent to each of v_1, v_2, v_p, and v_q while the edge (v_p, v_q) is removed. Clearly, $G = T_1 \cup T_2 \cup T_3$, where

$$T_1 = T_1' + (v_n, v_1), \qquad T_2 = T_2' + (v_n, v_2),$$

and

$$T_3 = T_3' + (v_n, v_p) + (v_n, v_q) - (v_p, v_q)$$

gives a decomposition of G into three spanning trees, and G has degree sequence $[d_i]$. Finally we have the third case.

Case 3. Suppose $d_n = 5$ and $\sum d_i = 6(n-1)$. In this case, we start with a graph H which is the union of 3 disjoint spanning trees T_1', T_2', T_3' and whose degree sequence is given by

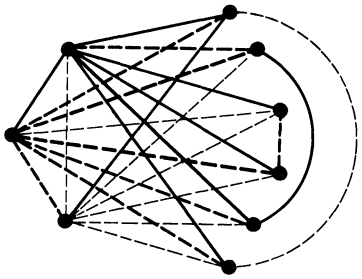
$$[d_i'] = [d_1 - 1, d_2, d_3, \cdots, d_{n-1}].$$

In view of Lemma 1, there exist edges (v_p, v_q) in T_2' and (v_r, v_s) in T_3' which are not incident with vertex v_1, and vertices v_p, v_q, v_r, v_s are distinct. Define

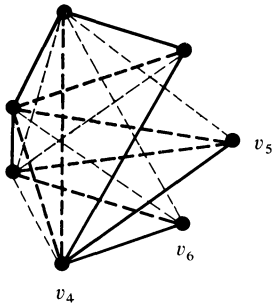
$$T_1 = T_1' + (v_n, v_1),$$
$$T_2 = T_2' + (v_n, v_p) + (v_n, v_q) - (v_p, v_q),$$

and

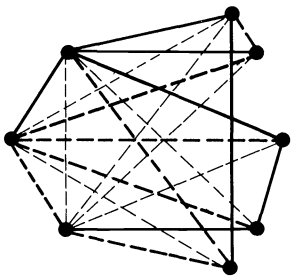
$$T_3 = T_3' + (v_n, v_r) + (v_n, v_s) - (v_r, v_s).$$



(a) [8, 8, 8, 4, 4, 4, 4, 4, 4]



(c) [6, 6, 6, 6, 4, 4, 4]



(b) [7, 7, 7, 5, 4, 4, 4, 4]

Replace the edge (v_4, v_6) in (c) by (v_5, v_6).

(d) [6, 6, 6, 5, 5, 4, 4]

FIG. 1

The trees T_i are edge-disjoint and span the vertices $v_1, v_2, \cdots, v_n$. It remains to put $G = T_1 \cup T_2 \cup T_3$. A similar argument holds for $\sum d_i = 6(n-1) + 2$ provided $[d_i]$ is not equal to [7, 7, 5, 5, 5, 5, 5] or [6, 6, 6, 5, 5, 5, 5] because then the sequence $[d_i']$ is not graphical. However, the theorem can be directly verified in these two cases. The proof is complete.

Remark. For $d_n \geqq 6$, our theorem can also be obtained by combining the results from [1] and [4]. Edmonds' theorem shows that $[d_i]$ has a 6-edge connected realizing graph G, while it follows from [4] that such a graph has at least three mutually disjoint spanning trees.

REFERENCES

[1] J. EDMONDS, *Existence of k-edge connected graphs with prescribed degrees*, J. Res. Nat. Bur. Standards Sect. B, 68 (1964), pp. 73–74.

[2] S. KUNDU, *Disjoint representation of tree realizable sequences*, SIAM J. Appl. Math., 26 (1974), pp. 103–107.

[3] ———, *Disjoint representation of three tree realizable sequences. I*, Ibid., 28 (1975), pp. 290–302.

[4] ———, *Bounds on the number of disjoint spanning trees*, to appear.

SIAM J. COMPUT.
Vol. 3, No. 4, December 1974

WORST-CASE PERFORMANCE BOUNDS FOR SIMPLE ONE-DIMENSIONAL PACKING ALGORITHMS*

D. S. JOHNSON†, A. DEMERS‡, J. D. ULLMAN§,
M. R. GAREY‖ AND R. L. GRAHAM‖

Abstract. The following abstract problem models several practical problems in computer science and operations research: given a list L of real numbers between 0 and 1, place the elements of L into a minimum number L^* of "bins" so that no bin contains numbers whose sum exceeds 1. Motivated by the likelihood that an excessive amount of computation will be required by any algorithm which actually determines an optimal placement, we examine the performance of a number of simple algorithms which obtain "good" placements. The first-fit algorithm places each number, in succession, into the first bin in which it fits. The best-fit algorithm places each number, in succession, into the most nearly full bin in which it fits. We show that neither the first-fit nor the best-fit algorithm will ever use more than $\frac{17}{10}L^* + 2$ bins. Furthermore, we outline a proof that, if L is in decreasing order, then neither algorithm will use more than $\frac{11}{9}L^* + 4$ bins. Examples are given to show that both upper bounds are essentially the best possible. Similar results are obtained when the list L contains no numbers larger than $\alpha < 1$.

1. Introduction. Recent results in complexity theory [3], [10] indicate that many combinatorial optimization problems may be effectively impossible to solve, in the sense that a prohibitive amount of computation is required to construct optimal solutions for all but very small cases. In order to solve such problems in practice, one is forced to use approximate, heuristic algorithms which hopefully compute "good" solutions in an acceptable amount of computing time. Thus, instead of seeking the fastest algorithm from the set of exact optimization algorithms, one seeks the best approximation algorithm from the set of "sufficiently fast" algorithms. Unfortunately it is usually difficult to evaluate and compare the performance of heuristic algorithms, other than by running them on large problem sets with known optimal solutions. A more rigorous approach is to mathematically analyze the performance of such algorithms to determine how closely the constructed solutions approximate optimal solutions. In this paper, we consider a number of heuristic algorithms for an important one-dimensional packing problem and determine worst-case performance bounds, relative to the optimal solution for each.

We base our theoretical performance analyses on worst-case, rather than average, behavior. The analysis of expected performance for a realistic probability distribution on the problem domain (which is, in itself, usually difficult to

* Received by the editors December 3, 1973, and in revised form April 8, 1974.

† Department of Mathematics, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. The research reported here was supported in part by Project MAC, an MIT research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract N00014-70-A-3062-0006, and by the National Science Foundation under Contract GJ00-4327. Now at Bell Laboratories, Murray Hill, New Jersey 07974.

‡ Department of Electrical Engineering, Princeton University, Princeton, New Jersey 08540. This work was supported by the National Science Foundation under Grant GJ-35570.

§ Department of Electrical Engineering, Princeton University, Princeton, New Jersey 08540. This work was supported by the National Science Foundation under Grant GJ-1052.

‖ Bell Laboratories, Murray Hill, New Jersey 07974.

determine) appears at present to be considerably more difficult. Worst-case results are easier—though decidedly nontrivial—to obtain and are quite useful, especially because they enable one to *guarantee* that a particular algorithm will never exceed the optimal solution by more than a known, hopefully small, percentage. Intuitively, one might expect that a "mechanism" which causes a particular algorithm to have a certain worst-case behavior might also be expected to manifest itself to a certain extent in the "average" case. Some experiments [4], [8] with randomly generated data have tended to confirm the hypothesis that, for the algorithms considered here, worst-case analysis does provide valid comparisons.

The basic problem to be considered can be stated quite simply: given a list $L = (a_1, a_2, \cdots, a_n)$ of real numbers in $(0, 1]$, place the elements of L into a minimum number L^* of "bins" so that no bin contains numbers whose sum exceeds 1.

This problem,[1] which is a special case of the one-dimensional "cutting-stock" problem [7] and the "assembly-line balancing" problem [2], models several practical problems in computer science. Some examples are:

1. *Table formatting.* Let the "bins" be computer words of fixed size: k bits. Suppose there are items of data (e.g., bit string of length 6, character string of 3 bytes, half-word integer) requiring $ka_1, ka_2, \cdots, ka_n$ bits, respectively. It is desirable to place the data in as few words as possible. The minimum is given by L^*, where L is the list $(a_1, a_2, \cdots, a_n)$.

2. *Prepaging.* Here, the bins are pages and the numbers in the list L represent fractions of a page required by program segments which should appear on a single page, e.g., inner loops, arrays, etc.

3. *File allocation.* It is desired to place files of varying sizes on as few tracks of a disc as possible, where files may not be broken between tracks.

Brown [1] gives a number of additional applications in industry and business. Since the abstract "bin packing" problem is "NP-complete" in the sense of Cook [3] and Karp [10], we can expect the problem of finding a packing which uses exactly L^* bins to require in general a lengthy combinatorial search for its solution. Thus we feel justified in considering the performance of various heuristic algorithms for constructing packings. In particular we shall consider the following four placement algorithms.

ALGORITHM 1 (First-fit). Let the bins be indexed as $B_1, B_2, \cdots$, with each initially filled to level zero. The numbers $a_1, a_2, \cdots, a_n$ will be placed in that order. To place a_i, find the least j such that B_j is filled to level $\beta \leqq 1 - a_i$, and place a_i in B_j. B_j is now filled to level $\beta + a_i$.

ALGORITHM 2 (Best-fit). Let the bins be indexed as $B_1, B_2, \cdots$, with each initially filled to level zero. The numbers $a_1, a_2, \cdots, a_n$ will be placed in that order. To place a_i, find the least j such that B_j is filled to level $\beta \leqq 1 - a_i$ and β is as large as possible, and place a_i in B_j. B_j is now filled to level $\beta + a_i$.

ALGORITHM 3 (First-fit decreasing). Arrange $L = (a_1, a_2, \cdots, a_n)$ into nonincreasing order and apply Algorithm 1 to the derived list.

ALGORITHM 4 (Best-fit decreasing). Arrange $L = (a_1, a_2, \cdots, a_n)$ into nonincreasing order and apply Algorithm 2 to the derived list.

[1] First brought to the attention of the last-named author by E. Arthurs (via S. A. Burr).

We use FF(L), BF(L), FFD(L) and BFD(L) to denote the number of bins used in applying each of the four algorithms, respectively, to the list L. The performance measure in which we are interested is the ratio of the number of bins used by a particular algorithm executed on L to the optimum number of bins L^*. Accordingly we use $R_{\mathrm{FF}}(k)$ to denote the maximum value achieved by the ratio $\mathrm{FF}(L)/L^*$ over all lists with $L^* = k$, with $R_{\mathrm{BF}}(k)$, $R_{\mathrm{FFD}}(k)$ and $R_{\mathrm{BFD}}(k)$ being defined similarly. Our main results, the first two of which appeared in a preliminary version of this paper [6], can be summarized as follows:

$$\lim_{k\to\infty} R_{\mathrm{FF}}(k) = \tfrac{17}{10}, \tag{1}$$

$$\lim_{k\to\infty} R_{\mathrm{BF}}(k) = \tfrac{17}{10}, \tag{2}$$

$$\lim_{k\to\infty} R_{\mathrm{FFD}}(k) = \tfrac{11}{9}, \tag{3}$$

$$\lim_{k\to\infty} R_{\mathrm{BFD}}(k) = \tfrac{11}{9}. \tag{4}$$

All these ratios are achieved for small values of k, so that these asymptotic results actually reflect the performance for essentially all values of k. In addition, similar results are obtained for certain restricted lists L.

2. First-fit and best-fit. We begin with a simple example which illustrates the type of list for which these two algorithms behave poorly. For any n divisible by 18 and $0 < \delta < \frac{1}{84}$, let the list $L = (a_1, a_2, \cdots, a_n)$ be defined by

$$a_i = \begin{cases} \frac{1}{6} - 2\delta, & 1 \leqq i \leqq n/3, \\ \frac{1}{3} + \delta, & n/3 < i \leqq 2n/3, \\ \frac{1}{2} + \delta, & 2n/3 < i \leqq n. \end{cases}$$

Clearly $L^* = n/3$ since the elements can be packed perfectly by placing one element from each of the three regions in each bin. However, as the reader can easily verify, both first fit and best fit will obtain the packing which consists of $n/18$ bins, each containing six elements of size $\frac{1}{6} - 2\delta$, $n/6$ bins, each containing two elements of size $\frac{1}{3} + \delta$, and $n/3$ bins each containing a single element of size $\frac{1}{2} + \delta$. The two packings are illustrated in Fig. 1. Thus we have

$$\frac{\mathrm{FF}(L)}{L^*} = \frac{\mathrm{BF}(L)}{L^*} = \frac{(n/18) + (n/6) + (n/3)}{n/3} = \frac{5}{3}.$$

By slightly modifying the list L given in the example, we can force even worse behavior and prove the following theorem.

THEOREM 2.1. *For every $k \geqq 1$, there exists a list L, with $L^* = k$, such that* $\mathrm{FF}(L) = \mathrm{BF}(L) > 1.7L^* - 8$.

Proof. As in the previous example, the elements of the list L will belong to three regions, with sizes nearly equal to $\frac{1}{6}$, $\frac{1}{3}$ and $\frac{1}{2}$, respectively. The number of elements belonging to each region will be the same, and those of the first region will precede those of the second region which precede those of the third region in the list L.

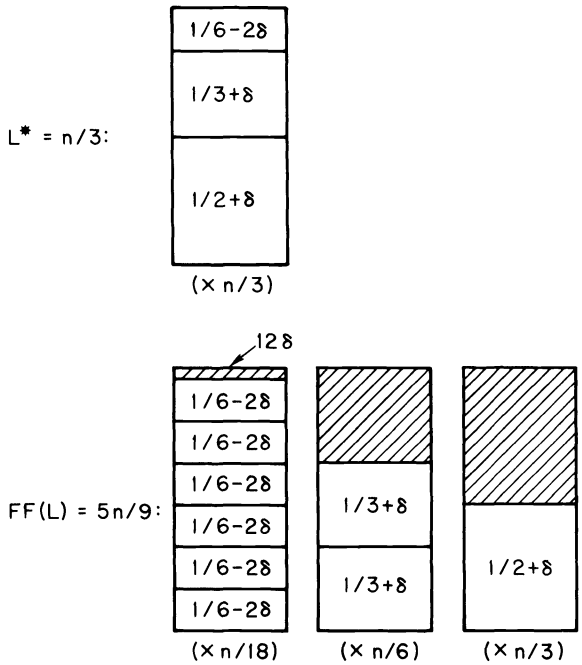


FIG. 1. *The 5/3 example*

Let N be a positive integer divisible by 17 and let δ be chosen so that $0 < \delta \ll 18^{-N/17}$. The first region will consist of $N/17$ *blocks* of ten numbers each. Let the numbers of the ith block of region 1 be denoted by $a_{0i}, a_{1i}, \cdots, a_{9i}$. These numbers are given by the following expressions, where $\delta_i = \delta \cdot 18^{(N/17)-i}$ for $1 \leqq i \leqq N/17$:

$$a_{0i} = \tfrac{1}{6} + 33\delta_i, \qquad a_{4i} = \tfrac{1}{6} - 13\delta_i,$$
$$a_{1i} = \tfrac{1}{6} - 3\delta_i, \qquad a_{5i} = \tfrac{1}{6} + 9\delta_i,$$
$$a_{2i} = a_{3i} = \tfrac{1}{6} - 7\delta_i, \qquad a_{6i} = a_{7i} = a_{8i} = a_{9i} = \tfrac{1}{6} - 2\delta_i.$$

Let the first $10N/17$ numbers in the list L be $a_{01}, a_{11}, \cdots, a_{91}, a_{02}, \cdots, a_{92}, \cdots, a_{0(N/17)}, \cdots, a_{9(N/17)}$. We notice that $a_{0i} + a_{1i} + \cdots + a_{4i} = \frac{5}{6} + 3\delta_i$, and $a_{5i} + a_{6i} + \cdots + a_{9i} = \frac{5}{6} + \delta_i$. Thus, for all i, the first five numbers of block i will fill up bin $2i - 1$, and the last five numbers of block i will fill up bin $2i$ when either the first-fit algorithm or the best-fit algorithm is applied to L. To see this we need only observe that a_{4i}, the smallest number in block i, will not fit in any of the previous bins, since the least filled of these, bin $2i - 2$, has contents totaling $\frac{5}{6} + \delta_{i-1} = \frac{5}{6} + 18\delta_i$. Also, the smallest of $a_{5i}, a_{6i}, \cdots, a_{9i}$, which is $\frac{1}{6} - 2\delta_i$, will not fit in bin $2i - 1$, which has contents totaling $\frac{5}{6} + 3\delta_i$. Thus the $N/17$ blocks in region 1 fill up $2N/17$ bins.

We now turn to region 2. Here the numbers are all about $\frac{1}{3}$, and they are again divided into $N/17$ blocks of ten numbers each. Let the ith block of region 2 be $b_{0i}, b_{1i}, \cdots, b_{9i}$. The numbers $b_{01}, b_{11}, \cdots, b_{91}, b_{02}, \cdots, b_{92}, \cdots, b_{0(N/17)}, \cdots, b_{9(N/17)}$ follow those of region 1 in the list L. The values of the numbers in block i

are given by:

$$b_{0i} = \tfrac{1}{3} + 46\delta_i, \qquad b_{4i} = \tfrac{1}{3} + 12\delta_i,$$
$$b_{1i} = \tfrac{1}{3} - 34\delta_i, \qquad b_{5i} = \tfrac{1}{3} - 10\delta_i,$$
$$b_{2i} = b_{3i} = \tfrac{1}{3} + 6\delta_i, \qquad b_{6i} = b_{7i} = b_{8i} = b_{9i} = \tfrac{1}{3} + \delta_i.$$

The numbers of block i fill bins $(2N/17) + 5i - 4$ through $(2N/17) + 5i$. These are filled with b_{0i} and b_{1i}, b_{2i} and b_{3i}, etc. To see this, we observe that the contents of the five bins filled by block i sum, respectively, to

$$\tfrac{2}{3} + 12\delta_i, \quad \tfrac{2}{3} + 12\delta_i, \quad \tfrac{2}{3} + 2\delta_i, \quad \tfrac{2}{3} + 2\delta_i, \quad \tfrac{2}{3} + 2\delta_i.$$

Thus $b_{5i} = \frac{1}{3} - 10\delta_i$ cannot fall into either of the first two bins, and $b_{1i} = \frac{1}{3} - 34\delta_i$ cannot fall into any of the bins for previous blocks since these are all filled to at least level $\frac{2}{3} + 2\delta_{i-1} = \frac{2}{3} + 36\delta_i$. Thus the $N/17$ blocks in region 2 fill up $5N/17$ bins.

The third region consists of $10N/17$ numbers, each equal to $\frac{1}{2} + \delta$. These complete the list L and fill one bin each. The total number of bins filled by either the first-fit algorithm or the best-fit algorithm applied to list L is thus $2N/17$ from region 1, $5N/17$ from region 2, and $10N/17$ from region 3, for a total of N bins.

However, the numbers on the list L can be packed into $(10N/17) + 1$ bins as follows. All but two of these bins contain one of the numbers $\frac{1}{2} + \delta$. The remaining space in each of these bins is filled with one of the following combinations:

(i) $a_{ji} + b_{ji}$ for some $2 \leqq j \leqq 9$ and $1 \leqq i \leqq N/17$,
(ii) $a_{0i} + b_{1i}$ for some $1 \leqq i \leqq N/17$,
(iii) $a_{1i} + b_{0(i+1)}$ for some $1 \leqq i \leqq N/17$.

This leaves b_{01}, $a_{1(N/17)}$, and one number $\frac{1}{2} + \delta$ which may be packed easily into the remaining two bins. We have thus shown that $L^* \leqq 1 + 10N/17$, so

$$\mathrm{FF}(L)/L^* \geqq 17N/(10N + 17) > 1.7 - 2/L^*$$

and, similarly

$$\mathrm{BF}(L)/L^* \geqq 17N/(10N + 17) > 1.7 - 2/L^*.$$

To obtain values of L^* not congruent to 1 (mod 10), we can form the list L' by adjoining to L m elements, each with size 1, where m is a fixed positive integer $\leqq 9$. The preceding arguments then show

$$\mathrm{FF}(L') = \mathrm{FF}(L) + m \quad \text{and} \quad L'^* = L^* + m$$

so that

$$\frac{\mathrm{FF}(L')}{L'^*} > \frac{17N + 17m}{10N + 17 + 17m} \geqq \frac{17}{10} - \frac{7m + 17}{10((10N/17) + m + 1)}$$
$$\geqq \frac{17}{10} - \frac{(7m + 17)/10}{L'^*} \geqq 1.7 - \frac{8}{L'^*},$$

since $m \leqq 9$. The same argument applies to $\mathrm{BF}(L)$. This proves Theorem 2.1.

We will now show that the examples constructed in the previous proof are essentially the worst possible, that is, 1.7 is the asymptotic least upper bound of the ratios $R_{\mathrm{FF}}(k)$ and $R_{\mathrm{BF}}(k)$.

THEOREM 2.2. *For every list* L, $\mathrm{FF}(L) \leqq 1.7L^* + 2$ *and* $\mathrm{BF}(L) \leqq 1.7L^* + 2$.

Proof. We use only the two following properties of the FF and BF algorithms.

(i) No element is placed in an empty bin unless it will not fit in *any* nonempty bin.

(ii) If there is a unique nonempty bin with lowest level, no element will be placed there unless it will *not* fit in any lower numbered bin.

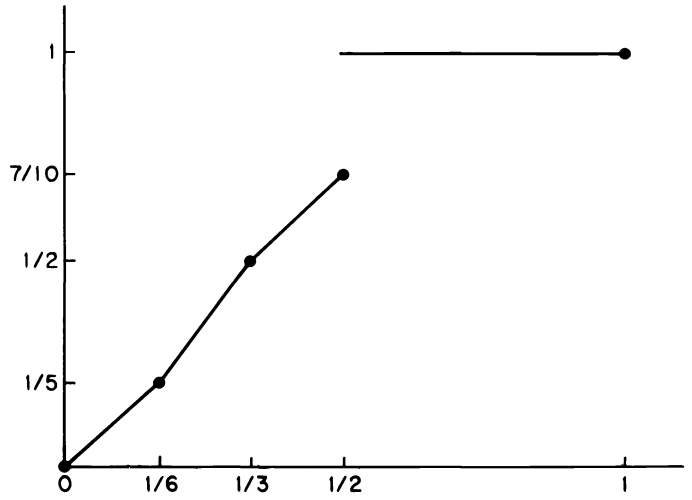


FIG. 2. *The function* $W(\alpha)$

Define W: $[0, 1] \to [0, 1]$ as follows (see Fig. 2):

$$W(\alpha) = \begin{cases} \frac{6}{5}\alpha & \text{for } 0 \leqq \alpha \leqq \frac{1}{6}, \\ \frac{9}{5}\alpha - \frac{1}{10} & \text{for } \frac{1}{6} < \alpha \leqq \frac{1}{3}, \\ \frac{6}{5}\alpha + \frac{1}{10} & \text{for } \frac{1}{3} < \alpha \leqq \frac{1}{2}, \\ 1 & \text{for } \frac{1}{2} < \alpha \leqq 1. \end{cases}$$

CLAIM 2.2.1. *Let some bin be filled with* $b_1, b_2, \cdots, b_m$. *Then* $\sum_{i=1}^{m} W(b_i) \leqq \frac{17}{10}$.

Proof. If $b \leqq \frac{1}{2}$, then $W(b)/b \leqq \frac{3}{2}$. The extreme ratio is reached only when $b = \frac{1}{3}$ and is less otherwise. Thus the lemma is immediate unless one b_i is greater than $\frac{1}{2}$. We may take this one to be b_1, and must show that if $\sum_{i=2}^{m} b_i < \frac{1}{2}$, then $\sum_{i=2}^{m} W(b_i) \leqq \frac{7}{10}$.

It should be noted that since the slope of $W(b)$ is the same in the regions $[0, \frac{1}{6}]$ and $[\frac{1}{3}, \frac{1}{2}]$, any b_i which is in the latter region can be replaced without loss of generality by two numbers of $\frac{1}{3}$ and $b_i - \frac{1}{3}$, respectively. We therefore assume that $b_i \leqq \frac{1}{3}$ for $2 \leqq i \leqq n$. Moreover, if b_j and b_k are both $\leqq \frac{1}{6}$, they can be combined into one and $\sum_i W(b_i)$ will not decrease; in fact it may increase. Thus we may assume that at most one of the b_i's, $i \geqq 2$, is in the range $(0, \frac{1}{6}]$, and the rest are in $(\frac{1}{6}, \frac{1}{3}]$.

We have consequently reduced the proof to the consideration of four cases:

Case 1. $m = 2, b_2 \leqq \frac{1}{3}$.

Case 2. $m = 3, \frac{1}{6} < b_2 \leqq b_3 \leqq \frac{1}{3}$.

Case 3. $m = 3, b_2 \leqq \frac{1}{6} < b_3 \leqq \frac{1}{3}$.

Case 4. $m = 4, b_2 \leqq \frac{1}{6} < b_3 \leqq b_4 \leqq \frac{1}{3}$.

Case 1 is immediate since $b_2 \leqq \frac{1}{2}$ implies $W(b_2) \leqq \frac{7}{10}$. In Case 2, $W(b_2) + W(b_3) = \frac{9}{5}(b_2 + b_3) - \frac{1}{5} \leqq \frac{9}{5} \cdot \frac{1}{2} - \frac{1}{5} = \frac{7}{10}$, since $b_2 + b_3 \leqq \frac{1}{2}$. For Case 3, $W(b_2) + W(b_3) = \frac{6}{5}b_2 + \frac{9}{5}b_3 - \frac{1}{10} \leqq \frac{1}{5} + \frac{3}{5} - \frac{1}{10} = \frac{7}{10}$. And finally, in Case 4, $W(b_2) + W(b_3) + W(b_4) \leqq \frac{6}{5}b_2 + \frac{9}{5}(b_3 + b_4) - \frac{1}{5} = \frac{9}{5}(b_2 + b_3 + b_4) - \frac{3}{5}b_2 - \frac{1}{5} \leqq \frac{9}{10} - \frac{1}{5} = \frac{7}{10}$, since $b_2 + b_3 + b_4 \leqq \frac{1}{2}$.

Let us define the *coarseness* of a bin to be the largest α such that some bin with smaller index is filled to level $1 - \alpha$. The coarseness of the first bin is 0.

CLAIM 2.2.2. *Suppose bins are filled according to either the* FF *or the* BF *algorithm, and some bin B has coarseness* α. *Then every member of B that was placed there before B was more than half full exceeds* α.

Proof. Until the bin has been filled to a level greater than $\frac{1}{2}$, it must be either empty or the unique nonempty bin of lowest level (by property (i) of the placement algorithm), so by constraints (i) and (ii), any element placed in the bin must not fit in any bin with lower index, and hence must exceed α.

CLAIM 2.2.3. *Let a bin of coarseness* $\alpha < \frac{1}{2}$ *be filled with numbers* $b_1 \geqq b_2 \geqq \cdots \geqq b_m$ *in the completed* FF*-packing* (BF*-packing*). *If* $\sum_{i=1}^{m} b_i > 1 - \alpha$, *then* $\sum_{i=1}^{m} W(b_i) \geqq 1$.

Proof. If $b_1 > \frac{1}{2}$, then the result is immediate since $W(b_1) = 1$. We therefore assume that $b_1 \leqq \frac{1}{2}$. If $m \geqq 2$, then the second element placed in the bin was placed before the bin was more than half full, so by Claim 2.2.2, at least two of the elements exceed α. In particular, we must have $b_1 \geqq b_2 \geqq \alpha$. We consider several cases depending on the range of α.

Case 1. $\alpha \leqq \frac{1}{6}$. Then $\sum_{i=1}^{m} b_i > 1 - \alpha \geqq \frac{5}{6}$. Since $W(\beta)/\beta \geqq \frac{6}{5}$ in the range $0 \leqq \beta \leqq \frac{1}{2}$, we immediately have $\sum_{i=1}^{m} W(b_i) \geqq \frac{6}{5} \cdot \frac{5}{6} = 1$.

Case 2. $\frac{1}{6} \leqq \alpha \leqq \frac{1}{3}$. We consider subcases (a)–(c), depending on the value of m.

(a) $m = 1$. Since $b_1 \leqq \frac{1}{2}$, we must have $1 - \alpha \leqq \frac{1}{2}$ or $\alpha \geqq \frac{1}{2}$, which contradicts our assumption that $\alpha \leqq \frac{1}{3}$.

(b) $m = 2$. If both b_1 and b_2 are $\geqq \frac{1}{3}$, then $W(b_1) + W(b_2) \geqq (\frac{6}{5} \cdot \frac{1}{3} + \frac{1}{10}) \cdot 2 = 1$. If both are $< \frac{1}{3}$, then $b_1 + b_2 < \frac{2}{3} < 1 - \alpha$, which contradicts our hypothesis. If $b_1 \geqq \frac{1}{3}$ and $b_2 < \frac{1}{3}$, then, since both must be greater than α, $\alpha < b_1 < \frac{1}{3} \leqq b_2 \leqq \frac{1}{2}$. Hence $W(b_1) + W(b_2) = \frac{9}{5}b_1 - \frac{1}{10} + \frac{6}{5}b_2 + \frac{1}{10} = \frac{6}{5}(b_1 + b_2) + \frac{3}{5}b_1$. Since $b_1 + b_2 \geqq 1 - \alpha$ and $b_1 > \alpha$, we thus have $W(b_1) + W(b_2) \geqq \frac{6}{5}(1 - \alpha) + \frac{3}{5}\alpha = 1 + \frac{1}{5} - \frac{3}{5}\alpha \geqq 1$, since $\alpha \leqq \frac{1}{3}$.

(c) $m \geqq 3$. As in the previous case, if two of the b_i are $\geqq \frac{1}{3}$, the result is immediate. If $b_1 \geqq \frac{1}{3} > b_2 \geqq \alpha$, then

$$W(b_1) + W(b_2) + \sum_{i=3}^{m} W(b_i)$$

$$\geqq \tfrac{6}{5}b_1 + \tfrac{1}{10} + \tfrac{9}{5}b_2 - \tfrac{1}{10} + \frac{6}{5}\sum_{i=3}^{m} b_i$$

$$= \frac{6}{5}\sum_{i=1}^{m} b_i + \tfrac{3}{5}b_2 \geqq \tfrac{6}{5}(1 - \alpha) - \tfrac{3}{5}\alpha = 1 + \tfrac{1}{5} - \tfrac{3}{5}\alpha \geqq 1.$$

If $\frac{1}{3} > b_1 \geqq b_2 > \alpha$, then

$$W(b_1) + W(b_2) + \sum_{i=3}^{m} W(b_i) \geqq \tfrac{9}{5}(b_1 + b_2) - \tfrac{1}{5} + \frac{6}{5}\sum_{i=3}^{m} b_i$$

$$\geqq \tfrac{6}{5}(1 - \alpha) + \tfrac{3}{5}(2\alpha) - \tfrac{1}{5} = 1 + \tfrac{6}{5}\alpha - \tfrac{6}{5}\alpha = 1.$$

Case 3. $\frac{1}{3} < \alpha < \frac{1}{2}$. If $m = 1$, we have $b_1 \geqq 1 - \alpha > \frac{1}{2}$, so $W(b_1) = 1$.

If $m \geqq 2$, then $b_1 \geqq b_2 > \frac{1}{3}$ and the result is immediate.

CLAIM 2.2.4. *If a bin of coarseness* $\alpha < \frac{1}{2}$ *is filled with* $b_1 \geqq \cdots \geqq b_m$ *and* $\sum_{i=1}^{m} W(b_i) = 1 - \beta$, *where* $\beta > 0$, *then either*

(i) $m = 1$ *and* $b_1 < \frac{1}{2}$, *or*

(ii) $\sum_{i=1}^{m} b_i \leqq 1 - \alpha - \frac{5}{9}\beta$.

Proof. If $m = 1$ and $b_1 > \frac{1}{2}$, it is impossible that $\beta > 0$. Therefore, if (i) does not hold, we may assume that $m \geqq 2$, and hence $b_1 \geqq b_2 \geqq \alpha$ by reasoning of the previous claim. Let $\sum_{i=1}^{m} b_i = 1 - \alpha - \gamma$. Then we may construct a bin filled with $b_3, b_4, \cdots, b_m$ and two other numbers δ_1 and δ_2, selected so that $\delta_1 + \delta_2 = b_1 + b_2 + \gamma$, $\delta_1 \geqq b_1$, $\delta_2 \geqq b_2$, and so that neither δ_1 nor δ_2 exceeds $\frac{1}{2}$. By the proof of Claim 2.2.3 and the fact that both δ_1 and δ_2 exceed α, $\sum_{i=3}^{m} W(b_i) + W(\delta_1) + W(\delta_2) \geqq 1$. But since the slope of W in the range $[0, \frac{1}{2}]$ does not exceed $\frac{9}{5}$, it follows that $W(\delta_1) + W(\delta_2) \leqq W(b_1) + W(b_2) + \frac{9}{5}\gamma$. Therefore $\gamma \geqq \frac{5}{9}\beta$, and (ii) holds.

We are now prepared to complete the proof. Let $L = (a_1, a_2, \cdots, a_n)$ and $\overline{W} = \sum_{i=1}^{n} W(a_i)$. By Claim 2.2.1, $L^* \cdot \frac{17}{10} \geqq \overline{W}$.

Suppose that in the FF (BF) algorithm bins $B'_1, B'_2, \cdots, B'_m$ are all the bins that receive at least one element and for which $\sum_j W(a_j) = 1 - \beta_i$ with $\beta_i > 0$, where j ranges over all elements in bin B'_i. We assume that $1 \leqq i < j \leqq m$ implies that B'_i had a smaller index than B'_j in the original indexing of all bins. Let α_i be the coarseness of B'_i. Since B'_i contains no element exceeding $\frac{1}{2}$, we must have each $\alpha_i < \frac{1}{2}$. By Claim 2.2.4 and the definition of coarseness,

$$\alpha_i \geqq \alpha_{i-1} + \tfrac{5}{9}\beta_{i-1} \quad \text{for } 1 < i \leqq m.$$

Thus

$$\sum_{i=1}^{m-1} \beta_i \leqq \frac{9}{5}\sum_{i=2}^{m} (\alpha_i - \alpha_{i-1}) = \tfrac{9}{5}(\alpha_m - \alpha_1) \leqq \tfrac{9}{5} \cdot \tfrac{1}{2} < 1.$$

Since β_m cannot exceed 1, we have $\sum_{i=1}^{m} \beta_i \leqq 2$. Applying Claim 2.2.3, we obtain

$$\mathrm{FF}(L) \leqq \overline{W} + 2 \leqq (1.7)L^* + 2 \quad \text{and} \quad \mathrm{BF}(L) \leqq \overline{W} + 2 \leqq (1.7)L^* + 2,$$

completing the proof.

As a consequence of Theorems 2.1 and 2.2, we have a corollary.

COROLLARY. (i) $\lim_{k\to\infty} R_{\mathrm{FF}}(k) = 1.7$,

(ii) $\lim_{k\to\infty} R_{\mathrm{BF}}(k) = 1.7$.

It is interesting to note that for several values of k, the ratio $\frac{17}{10}$ can actually be attained. In particular, there is a list L with $L^* = 10$ and $\mathrm{FF}(L) = \mathrm{BF}(L) = 17$. The two packings, with all quantities in units of $\frac{1}{101}$, are shown in Fig. 3. The list L is in nondecreasing order. There is also a list L having $L^* = 20$ and $\mathrm{FF}(L) = \mathrm{BF}(L) = 34$. It may be true, however, that $R_{\mathrm{FF}}(k) < 1.7$ and $R_{\mathrm{BF}}(k) < 1.7$ for $k > 20$.

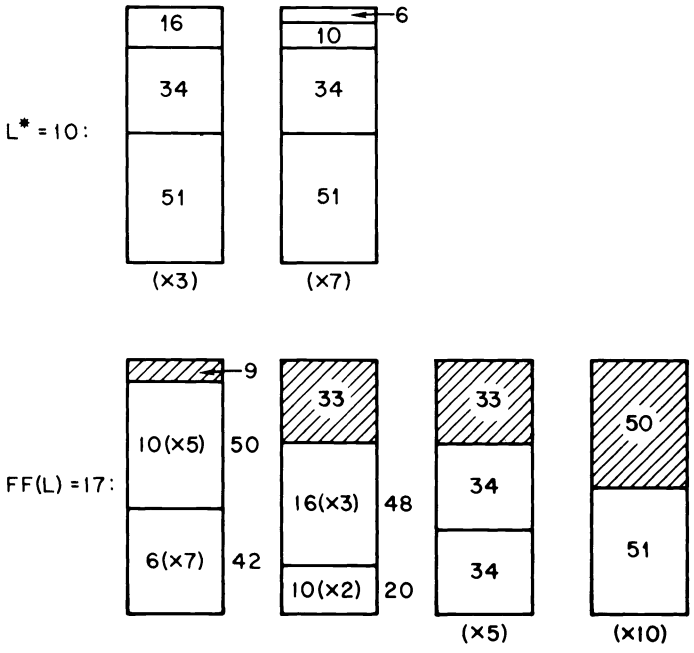


FIG. 3. *An example with* FF(L) = 17 *and* L^* = 10

If the list L is such that $a_i \leqq \alpha \leqq \frac{1}{2}$ for all i, the worst-case behavior of the two placement algorithms is not as extreme. We then have the following result.

THEOREM 2.3. *For any positive* $\alpha \leqq \frac{1}{2}$, *let* $m = \lfloor \alpha^{-1} \rfloor$. *Then we have*

(*i*) *for each* $k \geqq 1$, *there exists a list*[2] $L \subseteq (0, \alpha]$ *with* $L^* = k$ *such that* $\mathrm{FF}(L) \geqq [(m+1)/m]L^* - (1/m)$; *and*

(*ii*) *for any list* $L \subseteq (0, \alpha]$, $\mathrm{FF}(L) \leqq [(m+1)/m]L^* + 2$.

Both (*i*) *and* (*ii*) *hold with* FF *replaced by* BF.

Proof. We first describe how one constructs lists L, with no element exceeding α, for which

$$\frac{\mathrm{FF}(L)}{L^*} = \frac{\mathrm{BF}(L)}{L^*} = \frac{m+1}{m} - \frac{1}{mL^*}.$$

Let k be any positive integer. The list L is composed of elements which are all very close to $1/(m+1)$. The elements are of two types, described as follows:

$$b_j = 1/(m+1) - m^{2j+1}\delta, \qquad j = 1, 2, \cdots, k-1;$$

$$a_{1j} = a_{2j} = \cdots = a_{mj} = 1/(m+1) + m^{2j}\delta, \quad j = 1, 2, \cdots, k,$$

where $\delta > 0$ is chosen suitably small. The list L has the a-type elements occurring in nonincreasing order and the b-type elements occurring in strictly increasing order interspersed so that each successive pair, b_j and b_{j-1}, of b-type elements has

[2] Strictly speaking, L is not a set, but a sequence. However, the use of set terminology is convenient and should cause no confusion. Other instances will follow.

precisely m a-type elements occurring between them. The list L is then completely specified by the property that b_{k-1} occurs as the second element. We leave it for the reader to verify that

$$\mathrm{FF}(L) = \mathrm{BF}(L) = \left\lceil \frac{k(m+1)-1}{m} \right\rceil.$$

It is easy to see that the elements of L can be packed optimally by placing $b_j, a_{1j}, a_{2j}, \cdots, a_{mj}$ in a single bin for each $j = 1, \cdots, k-1$ and placing $a_{1k}, a_{2k}, \cdots, a_{mk}$ in one additional bin. This gives $L^* = k$. We then have

$$\frac{\mathrm{FF}(L)}{L^*} = \frac{\mathrm{BF}(L)}{L^*} \geqq \frac{k(m+1)-1}{mk} = \frac{m+1}{m} - \frac{1}{mL^*}.$$

The upper bound is also easily proved. Suppose that the list L contains no element exceeding $1/m$, m an integer.

Consider an FF packing of L. Every bin, except possibly the last bin, contains at least m elements. Disregarding the last bin, suppose two bins B_i and B_j, $i < j$, each contain elements totaling less than $m/(m+1)$. Then, since B_j contains m elements, B_j must contain an element with size less than $1/(m+1)$. But this element would have fit in B_i and thus could not have been placed in B_j by FF, a contradiction. Thus all but at most two bins must contain elements totaling at least $m/(m+1)$. Thus, letting $w(L)$ denote the sum of all elements on L, we have

$$L^* \geqq w(L) \geqq \frac{m}{m+1}(\mathrm{FF}(L) - 2),$$

so that

$$\mathrm{FF}(L) \leqq (m+1)L^*/m + 2.$$

A similar, but slightly more complicated, argument can be used to prove this for BF.

If we let $R^{\alpha}_{\mathrm{FF}}(k)$ and $R^{\alpha}_{\mathrm{BF}}(k)$ be defined analogously to $R_{\mathrm{FF}}(k)$ and $R_{\mathrm{BF}}(k)$ for lists $L \subseteq (0, \alpha]$, we have the following immediate corollary:

COROLLARY. $\lim_{k\to\infty} R^{\alpha}_{\mathrm{FF}}(k) = \lim_{k\to\infty} R^{\alpha}_{\mathrm{BF}}(k) = 1 + \lfloor \alpha^{-1} \rfloor^{-1}$.

3. First-fit decreasing and best-fit decreasing. The main results about FFD and BFD are the following.

THEOREM 3.1. *For each $k \geqq 1$, there exists a list L with $L^* = k$ such that* $\mathrm{FFD}(L) = \mathrm{BFD}(L) > \frac{11}{9}L^* - 2$.

THEOREM 3.2. *For all lists L,* $\mathrm{FFD}(L) \leqq \frac{11}{9}L^* + 4$, *and* $\mathrm{BFD}(L) \leqq \frac{11}{9}L^* + 4$. From these it follows that $\lim_{k\to\infty} R_{\mathrm{FFD}}(k) = \lim_{k\to\infty} R_{\mathrm{BFD}}(k) = \frac{11}{9}$.

The proof of Theorem 3.1 consists of a simple construction. Let ε satisfy $0 < \varepsilon < \frac{1}{12}$, $N = \lfloor k/9 \rfloor$, $\bar{k} \equiv k \pmod 9$ with $0 \leqq \bar{k} < 9$, $n = 30N + \bar{k}$, and consider the list $L = (a_1, \cdots, a_n)$ formed as follows:

$$a_i = \begin{cases} .5 + \varepsilon & \text{for } 1 \leqq i \leqq 6N, \\ .25 + 2\varepsilon & \text{for } 6N < i \leqq 12N, \\ .25 + \varepsilon & \text{for } 12N < i \leqq 18N, \\ .25 - 2\varepsilon & \text{for } 18N < i \leqq 30N, \\ 1.0 & \text{for } 30N < i \leqq n. \end{cases}$$

When L is put in decreasing order, the elements of size 1 will head the list, and BFD and FFD will yield the same packing. Figure 4 shows both this and the optimal packings. We have $L^* = 9N + \bar{k} = k$, and $\mathrm{FFD}(L) = \mathrm{BFD}(L) = 11N + \bar{k} > \frac{11}{9}L^* - 2$.

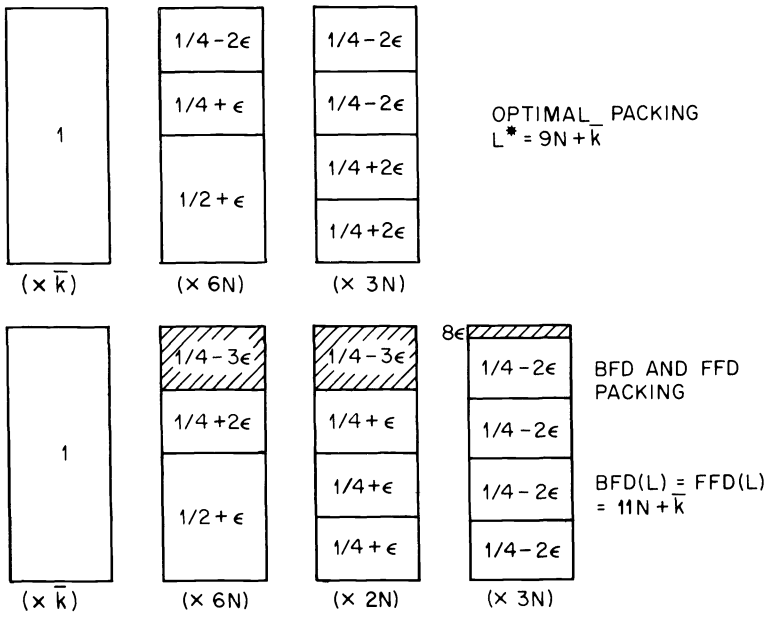


FIG. 4. *An 11/9 example*

The proof of Theorem 3.2 is considerably more complicated than the upper bound proofs in § 2, although some of the same ideas are involved. In this section we will show that we need only prove the result for the algorithm FFD and a restricted class of lists. In § 4 we will indicate how we go about proving the simplified, though still very difficult, result.

LEMMA 3.3. *Suppose L is a list such that* $\mathrm{FFD}(L) > rL^* + d$, *with* $r, d \geqq 1$. *Then the list L' obtained from L by deleting all elements not exceeding $(r-1)/r$ also has* $\mathrm{FFD}(L') > rL'^* + d$. *The same holds if* FFD *is replaced by* BFD.

Proof. Let P be the packing of L and P' the packing of L', using K and K' bins, respectively. If $K > K'$, then no element in the last bin of P can be larger than $(r-1)/r$, and hence all but the last must have levels exceeding $1/r$, since neither BF nor FF will start a new bin with an element which would fit in a previous bin. Thus $L^* \geqq \sum a_i > (1/r)(K-1)$ and so $K < rL^* + 1$, contrary to hypothesis. Hence $K' \geqq K$ and the lemma is proved.

Thus to prove Theorem 3.2 we need only consider lists $L \subset (\frac{2}{11}, 1]$. We would also like to use a single proof that would simultaneously yield the desired result for both FFD and BFD. For instance, in § 2 we simultaneously proved Theorem 2.2 for both FF and BF by only using properties the two algorithms have in common. Although *this* approach has not been successful for the current theorem, we could still use just one proof for both BFD and FFD if it could be shown that the result for one were just a simple corollary of the result for the other.

For instance, if for all lists L, $\mathrm{FFD}(L) \leqq \mathrm{BFD}(L)$, the result for FFD would follow immediately from that for BFD, or vice versa. Unfortunately, as the examples in Figs. 5 and 6 show, there are both lists L with $\mathrm{FFD}(L) < \mathrm{BFD}(L)$ and ones with $\mathrm{BFD}(L) < \mathrm{FFD}(L)$. Figure 5 presents packings of lists L with $\mathrm{BFD}(L) = \frac{10}{9}\mathrm{FFD}(L)$, and Fig. 6 presents L with $\mathrm{FFD}(L) = \frac{11}{10}\mathrm{BFD}(L)$. However, observe that the first example contains numbers less than $\frac{1}{6}$, whereas any list $L \subset (\frac{2}{11}, 1]$ contains no such numbers. Thus if these numbers less than $\frac{1}{6}$ are *essential* for examples like those in Fig. 5, we will still have $\mathrm{BFD}(L) \leqq \mathrm{FFD}(L)$ for all lists L we need to consider for Theorem 3.2, and so the result for BFD would follow from that for FFD. This is indeed the case, and we devote the remainder of this section to the details of the proof.

THEOREM 3.4. *Suppose* $L \subseteq [\frac{1}{6}, 1]$. *Then* $\mathrm{BFD}(L) \leqq \mathrm{FFD}(L)$.

Proof. Let $L = (a_1, \cdots, a_n)$ be ordered so that $a_i \geqq a_{i+1}$, $1 \leqq i < n$, with $a_n \geqq \frac{1}{6}$. We assume we have a copy of the FFD packing of L, denoted by PF, and are now proceeding to construct the BFD packing, element by element. At each step we will show that there is a way to extend the current packing to a packing of all of L using no more than $\mathrm{FFD}(L)$ bins.

For each i, $0 \leqq i \leqq n$, let L_i be the final segment of L consisting of all elements with index exceeding i. Thus $L_0 = L$. Let P_0 be the empty packing of $L - L_0 = \varnothing$, with which we begin the generation of the BFD packing, and let $f_0 : L_0 \to N \times N$ be defined as follows: if $a_i \in L$ is the kth largest element in bin j in PF (with ties broken according to the ordering of L), then $f_0(a_i) = (j, k)$.

The ordered pair (j, k) may be thought of as representing the kth *position* in bin j. We let k_j denote the number of elements in bin j in PF. Then we can define

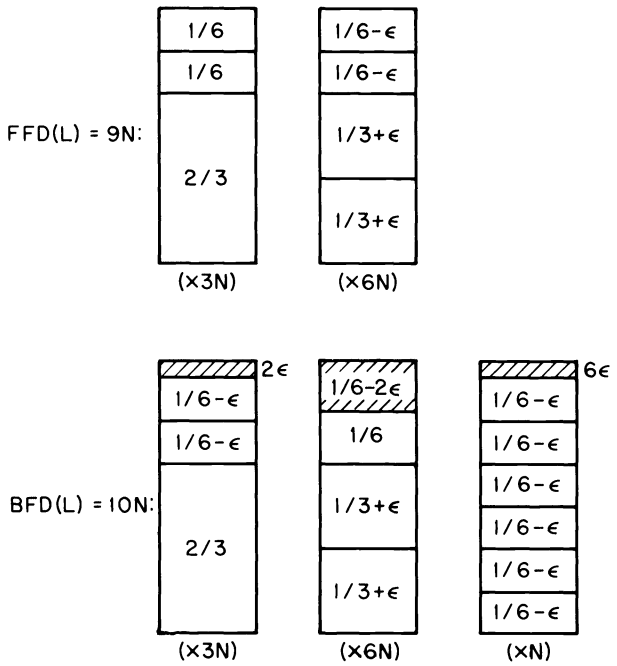


FIG. 5. *An example with* L^* *large and* $\mathrm{BFD}(L)/\mathrm{FFD}(L) = 10/9$

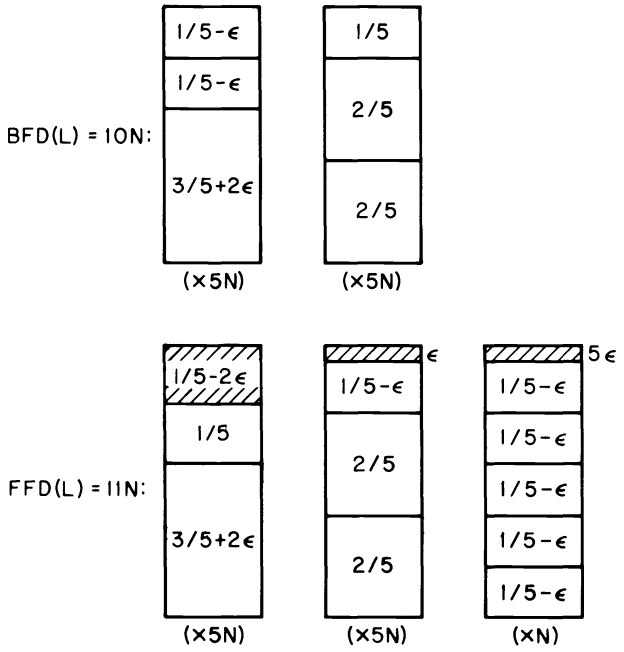


FIG. 6. *An example with L* large and* FFD(*L*)/BFD(*L*) = 11/10

the *empty positions of* P_0 to be

$$S_0 = \{(j,k): 1 \leqq j \leqq \mathrm{FFD}(L),\ 1 \leqq k \leqq k_j\},$$

that is, the positions which are *filled* in PF but not in P_0. This makes f_0 a 1–1 map from L_0, the elements of L remaining to be packed, to S_0, the positions remaining to be filled in P_0. Thus, in essence, f_0 shows us how to extend P_0 to a packing of all of L, by placing each remaining a_i in position $f_0(a_i)$, with none of the elements going in bins which were not used in PF.

The properties of f_0, P_0, and S_0 which allow this to happen can be summarized as follows, with $i = 0$:

(A) $f_i: L_i \to S_i$ is 1–1.

(B) $S_i \subseteq \{(j,h): (j,h)$ is filled in PF but empty in $P_i\}$.

(C) For each $j \geqq 1$, the sum of the elements in bin j in P_i, plus the sum of the elements which *map* to bin j under f_i, does not exceed 1.

As the generation of the BF packing proceeds, we construct P_i, S_i and f_i: $L_i \to S_i$ for $1 \leqq i \leqq n$ as follows: the packing P_i is obtained by adding element a_i to packing P_{i-1}. The bin it goes in is chosen according to the BFD placement rule. If that bin is the jth, the *position* a_i fills is that (j,k) which was unfilled in P_{i-1} and has minimal k. (It is possible that we may have to take $k > k_j$, if the bin already contains k_j elements.) We then set $S_i = S_{i-1} - \{(j,k)\}$. f_i is identical with f_{i-1} restricted to L_i (that is, with a_i deleted from its domain), with one possible exception. If $a_{i'}$, with $i' > i$, has $f_{i-1}(a_{i'}) = (j,k)$, we set $f_i(a_{i'}) = f_{i-1}(a_i)$. This insures that $f_i(a_{i'}) \in S_i$ and that f_i remains 1–1. In fact, a very elementary induction will establish

CLAIM 3.4.1. (A) *and* (B) *hold for all* i, $0 \leqq i \leqq n$.

Thus each f_i will provide us with a way to extend P_i to a packing of the entire list L with each element of L_i going in a bin which was used in PF, if only we can show that (C) also holds. What is more, this will prove the theorem, for we have the following claim.

CLAIM 3.4.2. *If for all* $i, 0 \leqq i \leqq n$, (A), (B) *and* (C) *hold, then* $\mathrm{BFD}(L) \leqq \mathrm{FFD}(L)$.

Proof. Suppose $\mathrm{BFD}(L) > \mathrm{FFD}(L)$. Let a_{m+1} be the first element assigned to bin $\mathrm{FFD}(L) + 1$ under BFD. By (A) and (B), $f_m(a_{m+1})$ is a position in one of the first $\mathrm{FFD}(L)$ bins of P_m, and by (C), a_{m+1} will fit in that bin. Thus a_{m+1} could not have gone in an empty bin (bin $\mathrm{FFD}(L) + 1$) to the right of that bin without violating the BFD placement rule. Thus bin $\mathrm{FFD}(L) + 1$ can never become nonempty, and so $\mathrm{BFD}(L) \leqq \mathrm{FFD}(L)$.

The proof of Theorem 3.4 is thus reduced to showing that (C) holds for $1 \leqq i \leqq n$. Rather than prove this directly by induction, we shall use two slightly more technical induction hypotheses which, together with (A) and (B), imply (C).

(D) If $(j, k) \in S_i \cap \mathrm{Range}(f_i)$, then $\mathrm{index}[f_i^{-1}(j, k)] \geqq \mathrm{index}[f_0^{-1}(j, k)]$.

(E) If a_r fills position (j, k), $1 \leqq k < k_j$, in P_i, then $r \geqq \mathrm{index}[f_0^{-1}(j, k)]$.

CLAIM 3.4.3. *If* (A), (B), (D) *and* (E) *hold for* i, *then* (C) *also holds for* i.

Proof. Let us consider an arbitrary bin j. We wish to show that the sum of the elements in bin j in P_i, plus the sum of the elements mapping to bin j under f_i, does not exceed 1. If no element of L_i maps to bin j, the result is immediate. If any element *does*, then by (A) and (B), position (j, k_j) must be empty in P_i, so that (E) will apply to all elements in the bin. If we were to place all elements that map to bin j in the positions to which they map, there would still be at most one element per position, because by (A) f_i is 1–1, and by (A) and (B) no element is mapped to a position which is already filled in P_i. By (D) and (E), each element would have index no smaller, and hence size no larger, than the element which filled its position in PF. Since the sum of the elements in any bin in PF does not exceed 1, the claim is proved.

CLAIM 3.4.4. (D) *holds for all* $i, 0 \leqq i \leqq n$.

Proof. (D) holds trivially for $i = 0$. If (D) holds for $i - 1$, the only opportunity for it to fail for i would be an $a_{i'}$, $i' > i$, for which $f_i(a_{i'}) \neq f_{i-1}(a_{i'})$. This can only happen if $f_i(a_i') = f_{i-1}(a_i)$. However, by (D) for $i - 1$, $\mathrm{index}[f_0^{-1}(f_{i-1}(a_i))] \leqq i < i'$, so (D) continues to hold for i. The claim follows by induction.

The proof of Theorem 3.4 is thus reduced to showing that (E) holds for all i, $0 \leqq i \leqq n$. We do this in two steps. First let $h = \max\{i : a_i > \frac{1}{3}\}$, where h is taken to be 0 if the set is empty.

CLAIM 3.4.5. *Each* a_i, $1 \leqq i \leqq h$, *is placed by* BFD *in position* $f_0(a_i)$.

Proof. Since the first elements placed in each of the bins under BFD form a nonincreasing sequence from left to right, the first time that the BFD choice could differ from the FFD choice can only have occurred when some bin B to the right of the FFD choice already contained two or more elements. But if this happens before a_h is assigned, the right-hand bin B would have had a level exceeding $\frac{2}{3}$ and would not have room for any additional elements exceeding $\frac{1}{3}$. Thus, until a_h is assigned, each a_i goes under BFD into the position it filled in PF, that is, position $f_0(a_i)$.

COROLLARY. (E) *holds for all* $i, 0 \leqq i \leqq h$.

Now let us order the positions by letting $(j, k) \leqq (j', k')$ mean that either $j < j'$ or $j = j'$ and $k \leqq k'$. The following fact about f_0 will be useful in showing that (E) holds for $i > h$.

CLAIM 3.4.6. *If* $(j, k), (j', k') \in S_h, k < k_j$, *and* $(j, k) \leqq (j', k')$, *then* $\text{index}[f_0^{-1}(j, k)] \leqq \text{index}[f_0^{-1}(j', k')]$.

Proof. Let a_i be the element in position (j, k) in PF, $a_{i'}$ the element in (j', k'). Since the positions are empty in P_h, we must have $\frac{1}{6} \leqq a_i, a_{i'} \leqq \frac{1}{3}$. If $j = j'$, the result is immediate, as position (j, k'), $k' \geqq k$, cannot have been filled before (j, k) under FF and so we must have $i' \geqq i$ as desired. So assume $j < j'$. Since $k < k_j$, position (j, k_j) must have been unfilled when a_i was to be assigned under FF, and so until a_i was assigned, the gap in bin j, was at least $2 \cdot \frac{1}{6} = \frac{1}{3}$. If $i' < i$, then $a_{i'}$ was assigned before a_i was, and so would have fit in bin j, contradicting our assumption that the FF rule assigned $a_{i'}$ to bin j', which is to the right of bin j. Thus we must have $i' \geqq i$ in this case also, and the claim is proved.

We are now ready to conclude the proof of Theorem 3.4 with the following claim.

CLAIM 3.4.7. (E) *holds for all* i, $h \leqq i \leqq n$.

Proof. By Claim 3.4.5, we know that (E) holds for $i = h$. Suppose it holds for $i - 1$. We shall show it holds for i, and the claim will follow by induction. Consider element a_i. Let (j, k) be the position it fills in P_i, and let $(j', k') = f_{i-1}(a_i)$. If $k \geqq k_j$, then (E) does not apply to the position filled by a_i, and so automatically continues to hold. So we may assume $k < k_j$. If $(j, k) \leqq (j', k')$, then by Claim 3.4.6 and (D),

$$\text{index}[f_0^{-1}(j, k)] \leqq \text{index}[f_0^{-1}(j', k')] \leqq \text{index}[f_{i-1}^{-1}(j', k')] = i,$$

and (E) would not be violated.

The only other possibility is $(j, k) > (j', k')$ and $k < k_j$. We shall show that in fact this cannot happen. Since both positions (j, k) and (j', k') must be empty in P_{i-1} and a_i goes in the bottom-most unfilled position in bin j, $(j, k) > (j', k')$ implies $j' < j$, and so bin j is to the right of bin j'. By (D) and (E) for $i - 1$ and Claim 3.4.3, a_i would have fit in bin j' of P_{i-1}. Since it went to the right of bin j' under the BFD rule, the level of bin j must have exceeded that of bin j' in P_{i-1}. However, since $k < k_j$, (E) applies to the elements in bin j in P_{i-1}, and so they take up no more space than the corresponding elements in PF. Consequently the gap in bin j, which we shall write gap(j), is at least as large as $f_0^{-1}(j, k) + f_0^{-1}(j, k_j) \geqq \frac{1}{6} + \frac{1}{6} = \frac{1}{3}$. Thus gap($j'$), the gap in bin j' in P_{i-1}, must *exceed* $\frac{1}{3}$. Now we cannot have $k = 1$, as then gap(j) $= 1 \geqq$ gap(j'), and we must have gap(j') $>$ gap(j). Thus $k > 1$, and consequently bin j must contain a bottom element b_1 with $b_1 >$ gap(j') $> \frac{1}{3}$ (see Fig. 7). Moreover, since list L is in decreasing order, the bottom elements in the bins form a nonincreasing sequence from left to right, and so if $k = 2$ we would again have gap(j) $\geqq$ gap(j'). Hence $k > 2$ and there is a second element in bin j in P_{i-1} (call it b_2) with $b_2 >$ gap(j') $> \frac{1}{3}$. But by (E) for $i - 1$, this means that the sum of the bottom two elements in bin j in PF also exceeds $\frac{2}{3}$, and hence the bin could have contained at most one additional element greater than or equal to $\frac{1}{6}$, so that $k_j \leqq 3$. Since $k > 2$ we thus have $k \geqq k_j$, the desired contradiction. So this case is impossible, (E) cannot be violated, and the claim is proved.

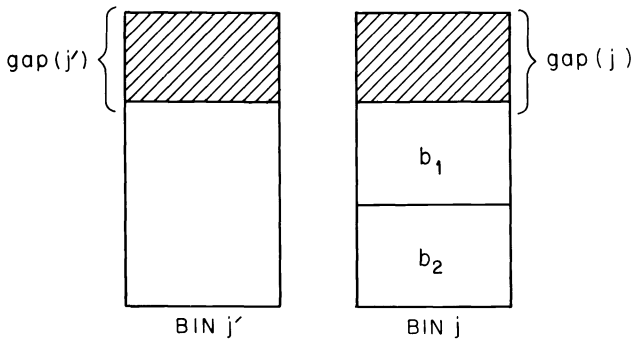


FIG. 7

Thus by Claims 3.4.5 and 3.4.7, (E) holds for all i, $0 \leqq i \leqq n$. Claim 3.4.1 and 3.4.4 tell us that (A), (B) and (D) also hold for such i. Thus (C) holds by Claim 3.4.3, and the theorem follows by Claim 3.4.2. Q.E.D.

A similar argument [6], [8] can be used to show the following result (which, however, will not be needed in our main task of completing the proof of Theorem 3.2).

THEOREM 3.5. *If* $L \subseteq [\frac{1}{5}, 1]$, *then* $\mathrm{BFD}(L) = \mathrm{FFD}(L)$.

Notice that the examples given in Figs. 5 and 6 show that the lower bounds of $\frac{1}{6}$ and $\frac{1}{5}$ in Theorems 3.4 and 3.5 are best possible.

4. First-fit decreasing upper bounds. In the previous section we reduced the proof of Theorem 3.2 to the task of showing that if $L \subset (\frac{2}{11}, 1]$, then $\mathrm{FFD}(L) \leqq \frac{11}{9}L^* + 4$. In this section we shall indicate how this can be done, and prove a simpler upper bound for the case when $L \subset (0, \frac{1}{2}]$.

The strategy behind such proofs is basically the same as the one used in § 2 for FF and BF upper bounds. Essentially, a "weighting function" is defined which assigns real number values or "weights" to the elements of L, depending on their size, in such a way that

(i) The total "weight" of all the elements in the list L is no less than a fixed constant c short of the number of bins used in the particular packing under consideration (e.g., FF or FFD).

(ii) The total weight of any legally packed bin must be less than some fixed constant r.

For FF we had $r = \frac{17}{10}$ and $c = 2$; for FFD we shall have $r = \frac{11}{9}$ and $c = 4$.

However, the actual details of the proof for FFD are considerably more complex than for FF and BF, requiring the introduction of a number of new concepts. Rather than burden the reader with this long[3] and detailed proof, we shall attempt to illustrate the basic ideas involved by describing in detail the major techniques used in the proof of a slightly simpler result, followed by an indication of the method for extending that proof to a proof of Theorem 3.2. Complete details can be found in Johnson [8].

THEOREM 4.1. *For all lists* $L \subset (0, \frac{1}{2}]$, $\mathrm{FFD}(L) \leqq \frac{71}{60}L^* + 5$.

[3] Exceeding 75 pages.

Remark. Theorem 4.1 gives the best bound possible, as can be seen from Fig. 8, which gives optimal and FFD packings of lists L for which $\mathrm{FFD}(L) = \frac{71}{60}L^*$, even though all elements in L are less than $\frac{1}{2}$, in fact, less than $\frac{1}{3}$. The ε in the figure must satisfy $0 < \varepsilon \leqq \frac{5}{87}$. We thus will be able to conclude that

$$\lim_{k\to\infty} R_{\mathrm{FFD}}^{1/2}(k) = \lim_{k\to\infty} R_{\mathrm{FFD}}^{1/3}(k) = \frac{71}{60},$$

in the terminology of § 2.

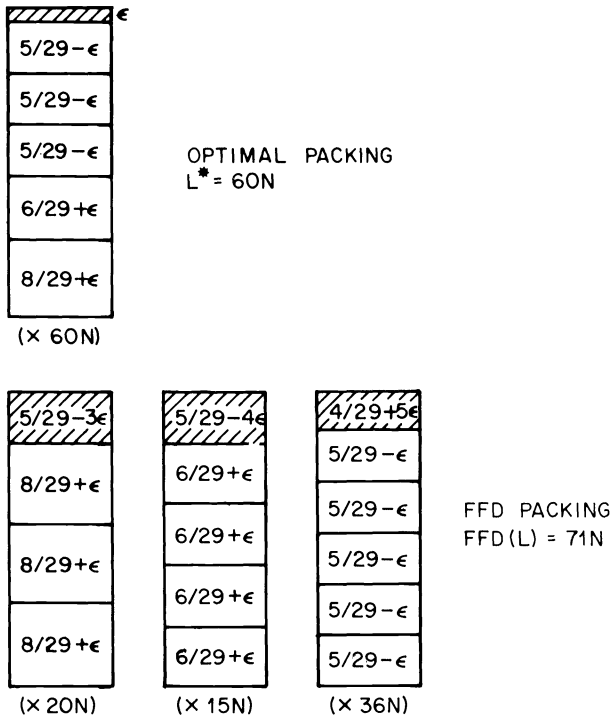


FIG. 8. *A 71/60 example*

We first give an overview of the proof, and then proceed to fill in many of the details. To begin with, the weighting function W which will be used is not really a function of *elements*, but rather a function of *sets* of elements,

$$W: 2^L \to Q \text{ (the rational numbers)}.$$

W will be defined in terms of two auxiliary functions.

$$w_1: L \to Q \quad \text{and} \quad w_2: L \times L \to Q.$$

The definitions of w_1 and w_2 will be made precise when we give the details of the proof.

Given a set of elements $X \subseteq L$, we obtain the weight $W(X)$ as follows: for any partition π of X into one- and two-element sets, let

$\pi(1) = \{x : \{x\} \in \pi\}$, (the set of elements which are in one-element sets in the partition π)

$\pi(2) = \{(x, y) : \text{index}(x) < \text{index}(y)$ and $\{x, y\} \in \pi\}$, (a set of ordered pairs corresponding to the two-element sets in π).

Then let

$$w_{12}(\pi) = \sum_{x \in \pi(1)} w_1(x) + \sum_{(x,y) \in \pi(2)} w_2(x, y).$$

Finally we define $W(X) = \min_\pi \{w_{12}(\pi)\}$, where π ranges over all partitions of X into one- and two-element sets. An elementary consequence of the definition as given so far is that W is subadditive, i.e.,

$$W\left(\bigcup_{i=1}^{k} X_i\right) \leqq \sum_{i=1}^{k} W(X_i).$$

Having defined the weighting function W, we can then divide the remainder of the proof of Theorem 4.1 into the two parts indicated at the beginning of the section. We shall state each part as a lemma, with the first given in sufficient generality so that it can also be used in the proof of the $\frac{11}{9}$ result.

LEMMA 4.2. *For any integer* $N \geqq 4$ *and* $L \subset (1/N, \frac{1}{2}]$,

$$W(L) \geqq \text{FFD}(L) - N + 2.$$

LEMMA 4.3. *If* $X \subseteq (\frac{1}{7}, \frac{1}{2}]$ *is any set of elements whose sum does not exceed* 1, *then* $W(X) \leqq \frac{71}{60}$.

By combining these two lemmas with the subadditivity of W, we can conclude that for any $L \subset (\frac{1}{7}, \frac{1}{2}]$, and X_i the set of elements in the ith bin of a given optimal packing of L, we have

$$\text{FFD}(L) - 5 \leqq W(L) \leqq \sum_{i=1}^{L^*} W(X_i) \leqq \tfrac{71}{60} L^*.$$

Thus Theorem 4.1 will be proved, since by Lemma 3.3 we *can* restrict our attention to lists $L \subseteq (\frac{1}{7}, \frac{1}{2}]$ when proving this upper bound.

We now begin a proof of Lemma 4.2, during which we will provide the remaining details of the definition of W. Let us call $x \in L$ a *k-piece* if $x \in (1/(k+1), 1/k]$. We shall also refer to 2-pieces as B-pieces, 3-pieces as C-pieces, etc. By a *k-bin* we mean a bin whose largest element is a k-piece.

Define $w_1(x) = \lfloor 1/x \rfloor^{-1}$. Thus if x is a k-piece, $w_1(x) = 1/k$. Let BASIC denote the set $\{x \in L$: for some k, x is a k-piece and is in a k-bin in the FFD packing of $L\}$.

CLAIM 4.2.1. *If* $L \subset (1/N, \frac{1}{2}]$, *then*

$$\sum_{x \in \text{BASIC}} w_1(x) \geqq \text{FFD}(L) - \sum_{j=2}^{N-1} \frac{j-1}{j}.$$

Proof. Note that for each k, $2 \leqq k < N$, all k-bins, except possibly for the last (rightmost) k-bin, must contain k k-pieces. For instance, every B-bin, that is, every 2-bin, must contain 2 B-pieces, except possibly for the rightmost one, which may contain only one B-piece. Thus, with the possible exception of the rightmost k-bin, all k-bins must contain elements of BASIC whose total w_1-weight is at least $k(1/k) = 1$. Since even the last k-bin must contain one k-piece belonging to BASIC, its deficiency can be at most $(k-1)/k$. This proves the claim.

As a consequence of Claim 4.2.1, we could satisfy Lemma 4.2 by merely defining $W(X) = w_1(X) = \sum_{x \in X} w_1(x)$. The reason we do *not* do this, but instead introduce w_2, is in order that we can prove Lemma 4.3. There are many sets of elements X whose sum does not exceed 1 and yet for which $w_1(x) > \frac{71}{60}$. This is not surprising in light of the fact that there are probably many elements from L which are not in BASIC, so that in fact $w_1(L)$ is probably much larger than w_1(BASIC) and hence much larger than FFD(L). Thus w_1 is in a sense "overcharging" the bins of the FFD packing.

The elements of SURPLUS = L − BASIC can thus be considered excess baggage in the weight calculated by w_1. The purpose of w_2 is to enable us to avoid counting this unneeded contribution to the total weight by SURPLUS. Given a pair of elements, w_2 will do this by "discounting" the w_1-weight of the second element by an appropriate amount if certain *discounting relations* are satisfied by the members of the pair. The relations can be generally described as follows: (x, y) is said to obey *relation* k if x is a k-piece and $kx + y \leqq 1$. We define w_2 by

$$w_2(x, y) = \begin{cases} w_1(x) + [(k-1)/k]w_1(y) & \text{if } (x, y) \text{ obeys relation } k, \\ w_1(x) + w_1(y) & \text{otherwise.} \end{cases}$$

Another way of looking at w_2 is to note that if (x, y) obeys relation k, then $w_1(\{x, y\}) - w_2(x, y) = w_1(y)/k$, and y has been discounted by a factor of $1/k$.

As a concrete example, suppose x is a B-piece (2-piece), y a C-piece, and $2x + y \leqq 1$. Then $w_1(\{x, y\}) = \frac{1}{2} + \frac{1}{3} = \frac{5}{6}$ and $w_2(x, y) = \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{3} = \frac{2}{3} = \frac{5}{6} - \frac{1}{2} \cdot \frac{1}{3}$. If $2x + y > 1$, then $w_2(x, y) = w_1(\{x, y\}) = \frac{5}{6}$.

We are now going to show that the "discount" due to using w_2 instead of w_1 on a pair of elements actually corresponds to an element in SURPLUS, or at least a portion of one, so that, modulo certain edge effects, the lower bound proven for w_1(BASIC) also holds for $w_{12}(\pi)$, where π is *any* partition of L into one- and two-element sets. Formally, we have Claim 4.2.2.

CLAIM 4.2.2. *If* $N \geqq 4$ *and* π *is a partition of* $L \subseteq (1/N, \frac{1}{2}]$ *into one- and two-element sets, then*

$$w_{12}(\pi) \geqq w_1(\text{BASIC}) - \sum_{j=3}^{N-1} \frac{1}{j}.$$

Lemma 4.2 will follow from Claims 4.2.1 and 4.2.2, since together they tell us that for all such π,

$$w_{12}(\pi) \geqq \text{FFD}(L) - \sum_{j=2}^{N-1} \frac{j-1}{j} - \sum_{j=3}^{N-1} \frac{1}{j} \geqq \text{FFD}(L) - N + 2,$$

and so $W(L) = \min_\pi w_{12}(\pi)$ must also exceed that lower bound.

Proof of Claim 4.2.2. Let R_k, $2 \leqq k \leqq N - 2$, be the set of all pairs in the partition π which obey relation k. Then, since no pairs in π can obey any relation $k' > N - 2$, $R = \bigcup_{k=2}^{N-2} R_k$ is the set of all pairs in π obeying discounting relations. If we let DISCOUNT$(x, y) = w_1(x) + w_1(y) - w_2(x, y)$, and DISCOUNT$(X)$ $= \sum_{(x,y) \in X}$ DISCOUNT(x, y) for any set of pairs X, we then have

$$w_{12}(\pi) = w_1(\text{BASIC}) + w_1(\text{SURPLUS}) - \text{DISCOUNT}(R),$$

and consequently all we need prove is that

$$w_1(\text{SURPLUS}) \geqq \text{DISCOUNT}(R) - \sum_{j=3}^{N-1} \frac{1}{j}. \tag{4.2.2a}$$

To show that (4.2.2a) holds, we construct a system of "billing" so that for each $(x, y) \in R$, DISCOUNT(x, y) is billed to some specific element of SURPLUS (or perhaps divided among a number of such elements), and no element of SURPLUS is billed for a total of more than its w_1-weight. A small number of DISCOUNTS will go unbilled and these will account for the $\sum(1/j)$ term. More formally, we construct a billing map BILL$: R \times \text{SURPLUS} \to (0, \frac{1}{6}]$ and a set UNBILLED such that

$$\text{DISCOUNT(UNBILLED)} \leqq \sum_{j=2}^{N-2} 1/(j+1). \tag{4.2.2b}$$

For all $(x, y) \in R - \text{UNBILLED}$,

$$\sum_{z \in \text{SURPLUS}} \text{BILL}((x, y), z) \geqq \text{DISCOUNT}(x, y). \tag{4.2.2c}$$

For all $z \in$ SURPLUS,

$$\sum_{(x,y) \in R} \text{BILL}((x, y), z) \leqq w_1(z). \tag{4.2.2d}$$

Inequality (4.2.2a), and hence Claim 4.2.2 will then follow from (4.2.2b)–(4.2.2d).

To keep this paper to a reasonable length, we shall not present the billing procedure in all its intricacies [8]. However, we will present the basic idea behind it and an indication of why additional intricacies are necessary.

Initially we set BILL$((x, y), z) = 0$ for all $(x, y) \in R$, $z \in$ SURPLUS. As we proceed, some of these values will be reset. At any given point in time, $z \in$ SURPLUS will have been *charged* the current value of $\sum_{(x,y) \in R}$ BILL$((x, y), z)$. For $(x, y) \in R$, we will say that DISCOUNT(x, y) has been *billed* if

$$\sum_{z \in \text{SURPLUS}} \text{BILL}((x, y), z) \geqq \text{DISCOUNT}(x, y).$$

We shall treat each R_k in turn, defining a set UNBILLED$_k \subseteq R_k$ and then billing the DISCOUNT for each pair in $R_k -$ UNBILLED$_k$ in such a way that no element of SURPLUS will have been charged more than its w_1-weight. UNBILLED will be defined as $\bigcup_{k=2}^{N-2}$ UNBILLED$_k$. And we will have DISCOUNT (UNBILLED$_k$) $\leqq 1/k + 1$, $2 \leqq k \leqq N - 2$. The basic idea involved in the processing of R_k can be explained as follows: Assume that

(G1) no $z \in$ SURPLUS which is in a k'-bin, $k' \geqq k$, in the FFD-packing has yet been charged more than 0,

(G2) no member of any pair in R_k is in a k'-bin, $k' < k$.

All the billing we shall do in this case will be to elements of SURPLUS$_k$, the members of SURPLUS which are in k-bins.

First take all pairs in R_k and relabel them (x_i, y_i) in order of increasing index (with respect to the original list) of their second components. We will thus have index$(y_1) <$ index$(y_2) < \cdots <$ index(y_m), where $m = |R_k|$, and hence $y_1 \geqq$

$y_2 \geqq \cdots \geqq y_m$. Note that the x_i's and y_i's are all distinct since the $\{x_i, y_i\}$'s form a partition of R_k and hence are disjoint.

Suppose we can construct a 1–1 map

$$g:\{y_{kj}:1 \leqq j \leqq \lceil m/k \rceil\} \to \text{SURPLUS}_k$$

such that for all $y_i \in \text{Domain}(g)$,

(G3) $\text{index}(g(y_i)) \leqq \text{index}(y_i)$.

We can then use g to define BILL for elements of R_k. For $1 \leqq j \leqq \lceil m/k \rceil$ and $0 \leqq i \leqq k-1$, let

$$\text{BILL}((x_{kj+i}, y_{kj+i}), g(y_{kj})) = \text{DISCOUNT}(x_{kj+i}, y_{kj+i}).$$

(For $j = \lceil m/k \rceil$ and $i > m - k\lceil m/k \rceil$ the definition will be vacuous.)

If we let $\text{UNBILLED}_k = \{(x_i, y_i): 1 \leqq i < k\}$ we thus have for all $(x, y) \in R_k - \text{UNBILLED}_k$ that $\text{DISCOUNT}(x, y)$ has been billed. Moreover,

$$\text{DISCOUNT}(\text{UNBILLED}_k) = \sum_{i=1}^{k-1} \text{DISCOUNT}(x_i, y_i) \leqq (k-1)\frac{1}{k} \cdot \frac{1}{k+1} < \frac{1}{k+1}.$$

Finally, the only elements charged are $g(y_{kj})$, and since g is 1–1, the most $g(y_{kj})$ is charged is

$$\begin{aligned} \sum_{(x,y)\in R_k} \text{BILL}((x, y), g(y_{kj})) &= \sum_{i=0}^{k-1} \text{DISCOUNT}(x_{kj+i}, y_{kj+i}) \\ &= \frac{1}{k} \sum_{i=0}^{k-1} w_1(y_{kj+i}) \\ &\leqq \frac{1}{k}[k \cdot w_1(y_{kj})] = w_1(y_{kj}) \leqq w_1(g(y_{kj})) \end{aligned}$$

by (G3) because $\text{index}(y_{kj+k-1}) > \cdots > \text{index}(y_{kj}) \geqq \text{index}(g(y_{kj}))$ and L is in decreasing order.

If the above held for all k, $2 \leqq k \leqq N-2$, and if g were 1–1 throughout the composite range $[y_{kj}: 1 \leqq j \leqq \lceil m/k \rceil, 2 \leqq k \leqq N-2\}$, we would have properties (4.2.2b) through (4.2.2d), and hence Claim 4.2.2 would be proved.

How might we define g so that the above *does* hold? In the case when both (G1) and (G2) hold, as they must trivially for $k = 2$, the process is fairly straightforward.

Observe that since all $(x_i, y_i) \in R_k$ obey relation k, we have $y_i + kx_i \leqq 1$, $1 \leqq i \leqq m$, and hence, by the indexing of the pairs, $y_i + kx_j \leqq 1$ for $1 \leqq j \leqq i$.

Let us now look at the FFD-packing again, in particular the k-bins. (See Fig. 9.) There must be at least $\lceil m/k \rceil$ k-bins, since by assumption (G2) there are at least $|R_k| = m$ k-pieces in k'-bins for $k' \geqq k$ and hence in k-bins. We label the bottom k elements in each k-bin from top to bottom and right to left, as shown in Fig. 9. Then b_k must be a k-piece, as are all the labeled elements with higher index. The remaining elements in the bin containing b_k (the b_k-*bin*) need not all be k-pieces. Indeed, some may not even exist, if this is the last bin in the packing, in

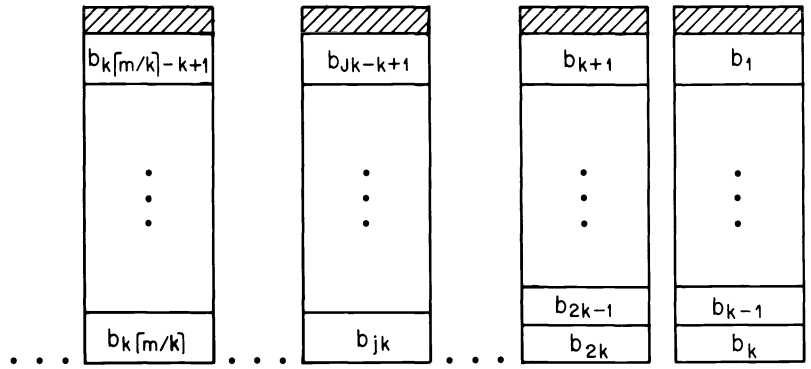


FIG. 9. *K-bins of* FFD *packing labeled for Lemma* 4.2.2

which case we set the corresponding $b_i = 0$. By the way FFD is executed, we must have

$$b_{k\lceil m/k\rceil} \geqq b_{k\lceil m/k\rceil - 1} \geqq \cdots \geqq b_3 \geqq b_2 \geqq b_1.$$

Now all the x_i's occur in this list; thus for no $j \leqq k\lceil m/k\rceil$ can we have $b_j > x_i$, for all i, $1 \leqq i \leqq j$, as that would put one too many x_i's to the right of b_j in the sequence. Thus for each j there exists an $i(j) \leqq j$ such that $x_{i(j)} \geqq b_j$. We can thus conclude that for all h, j, $1 \leqq j \leqq h \leqq \lceil m/k\rceil$,

$$y_{kh} + \sum_{i=1}^{k} b_{kj-k+i} \leqq y_{kh} + kb_{kj} \leqq y_{kh} + kx_{i(kj)} \leqq 1.$$

Consequently y_{kh} would fit as the $(k+1)$st element in any of the bins to the right of and including the b_{kh}-bin. Using this fact we can define our function g as follows.

If y_k is in a k-bin in the FFD-packing, then we must have $y_k \in \text{SURPLUS}_k$. Let $g(y_k) = y_k$. If not, then y_k must by assumption be in some bin to the right of the b_k-bin. Since y_k would have fit in that bin unless it already contained $k+1$ elements, the b_k-bin must have contained that many elements when y_k was assigned, one of which must be an element of SURPLUS_k and have lower index than y_k. Let $g(y_k)$ be the largest such element.

Note that in both cases, $g(y_k) \in \text{SURPLUS}_k$ and obeys $(G3)$ for $i = k$, i.e., $\text{index}(g(y_k)) \leqq \text{index}(y_k)$.

Continuing by induction, assume that values for $g(y_{kh})$, $1 \leqq h \leqq j \leqq [m/k]$, have been assigned and are all distinct elements of SURPLUS_k obeying (G3). If y_{kj} is in a k-bin, it cannot be $g(y_{kh})$ for any $h < j$, since by (G3) and the labeling of the y_i's we have $\text{index}(g(y_{kh})) \leqq \text{index}(y_{kh}) < \text{index}(y_{kj})$. So in this case we again define $g(y_{kj}) = y_{kj}$. If y_{kj} is not in a k-bin, then it must have gone to the right of the b_k- through b_{kj}-bins, into each of which it would have fit as the $(k+1)$st element. Hence all j bins must contain elements of SURPLUS with index lower than that of y_{jk}. Since at most $j-1$ of them can have yet been assigned to the range of g, there is at least one still unassigned and we can let such an element of SURPLUS be $g(y_{kj})$. This maintains the 1–1 property of g and insures that (G3) will hold for $i = kj$.

Thus by induction we have defined our map

$$g:\{y_{kj}:1 \leqq j \leqq \lceil m/k \rceil\} \to \text{SURPLUS}$$

obeying property (G3) throughout its domain.

The above analysis depended on assumptions (G1) and (G2), which, as we have said, clearly hold for $k = 2$. We might thus hope to proceed by induction. In our billing procedure, only elements of SURPLUS_k received new charges, so (G1) will continue to hold when we begin to process R_{k+1}.

However, there is no guarantee that (G2) will hold for any $k > 2$. This is what gives rise to complications. If $(x, y) \in R_k$ and x is not in a k-bin, then it must be in a k'-bin, $k' < k$, and hence a member of SURPLUS. If x has not yet been charged, we can bill DISCOUNT(x, y) to x. If it has been charged more than 0, then x must be $g(z)$ for some z, with $z \leqq x$ by (G1), and z may be a k-piece in a k-bin. The more intricate argument here omitted shows how to modify our billing procedure to take advantage of such possibilities and still guarantee that (4.2.2b)–(4.2.2d) hold, and hence Claim 4.2.2, will hold. Q.E.D.

We have already seen that Lemma 4.2 follows from Claim 4.2.1 and 4.2.2. To complete the proof of Theorem 4.1, we must now turn our attention to Lemma 4.3, which says that for any set $X \subseteq (\frac{1}{7}, \frac{1}{2}]$ whose sum does not exceed 1, $W(X) \leqq \frac{71}{60}$. Since $W(X)$ depends just on the types of the elements in X (e.g., B, C, $\cdots$, etc.) and which discounting relations are satisfied (not on the precise *values* of the elements), it is easy to see that there are a relatively small finite number of possible configurations to consider. We shall illustrate the type of calculation necessary by treating several typical cases, leaving the remaining 70-odd, more or less routine, cases to the ambitious reader.[4]

(i) $X = \{B_1, C_2, C_3\}$, i.e., X consists of one B-piece and two C-pieces with $C_2 \geqq C_3$. Then

$$W(X) \leqq w_1(X) = w_1(B_1) + w_1(C_2) + w_1(C_3) = \tfrac{1}{2} + \tfrac{1}{3} + \tfrac{1}{3} = \tfrac{7}{6} < \tfrac{71}{60}$$

(ii) $X = \{B_1, B_2, E_3, F_4\}$. Here $w_1(X) = \frac{1}{2} + \frac{1}{2} + \frac{1}{5} + \frac{1}{6} = \frac{41}{30} > \frac{71}{60}$ so the partition of X into 1-element sets is not adequate for determining W. Note, however, that

$$B_1 + E_3 \leqq 1 - B_2 - F_4 < 1 - \tfrac{1}{3} - \tfrac{1}{7} = \tfrac{11}{21},$$

which implies

$$2B_1 + E_3 \leqq \tfrac{22}{21} - \tfrac{1}{6} < 1,$$

so that (B_1, E_3) obeys relation 2. Hence we get a discount of $\frac{1}{2} \cdot \frac{1}{5} = \frac{1}{10}$ here. Similarly (B_2, F_4) obeys relation 2 and we get an additional discount of $\frac{1}{12}$. Thus

$$W(X) \leqq \tfrac{41}{30} - \tfrac{1}{10} - \tfrac{1}{12} = \tfrac{71}{60},$$

as required.

(iii) $X = \{C_1, C_2, E_3, E_4, E_5\}$. This configuration is impossible since $C_i \in (\frac{1}{4}, \frac{1}{3}]$ and $E_j \in (\frac{1}{6}, \frac{1}{5}]$ imply $C_1 + C_2 + E_3 + E_4 + E_5 > 1$.

(iv) $X = \{C_1, D_2, E_3, E_4, E_5\}$. Then $W(X) \leqq w_1(X) = \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{5} + \frac{1}{5} = \frac{71}{60}$. Note that this is the configuration which occurs in the construction used to prove the lower bound for Theorem 4.1.

[4] Complete details may be found in [8].

When all possible cases are finally checked (there are only two which yield a bound of $\frac{71}{60}$; all other cases are bounded by $\frac{7}{6}$), the proof of Theorem 5 follows at once from the preceding remarks.

It is not difficult to show that if no element of L exceeds $\frac{8}{29}$, then the preceding analysis proves $W(X) \leqq \frac{7}{6}$ for all sets X of elements in a legally packed bin. (Case (ii) cannot occur since there are now no B-pieces; in case (iv), additional discounting now becomes possible.) This observation leads us to the following corollary.

COROLLARY. (a) *For* $\alpha \in (\frac{8}{29}, \frac{1}{2}]$,

$$\lim_{k \to \infty} R^{\alpha}_{\mathrm{FFD}}(k) = \tfrac{71}{60};$$

(b) *for* $\alpha \in (\frac{1}{4}, \frac{8}{29}]$,

$$\lim_{k \to \infty} R^{\alpha}_{\mathrm{FFD}}(k) = \tfrac{7}{6}.$$

The lower bound for (b) is proved with a construction similar to those used earlier, with the list $L = (a_1, a_2, \cdots, a_n)$ composed of $24N$ elements, half equal to $.25 + \varepsilon$ and half equal to $.25 - \varepsilon$, where $0 < \varepsilon < \frac{1}{12}$.

A similar (but more complicated) analysis can be given to establish Theorem 4.4.

THEOREM 4.4. *For* $\alpha \in (\frac{1}{5}, \frac{1}{4}]$,

$$\lim_{k \to \infty} R^{\alpha}_{\mathrm{FFD}}(k) = \tfrac{23}{20}.$$

The reader is referred to Johnson [8] for a much more complete discussion of the details needed to complete the proofs of these and other similar results.

As proposed at the beginning of this section, the techniques used in the proof of Theorem 4.1 form a significant part of the proof of Theorem 3.2, whose crucial assertion is that $\mathrm{FFD}(L) \leqq \frac{11}{9}L^* + 4$ for all lists $L \subseteq (\frac{2}{11}, 1]$.

The inclusion of A-pieces, that is, elements which exceed $\frac{1}{2}$, causes rather severe problems for the relatively simple calculations we were able to perform in proving Theorem 4.1. We are now forced to resort to somewhat more subtle ideas. The basic strategy is as follows.

(a) We assume we have a list $L = (a_1 \geqq a_2 \geqq \cdots \geqq a_n)$, consisting of A-, B-, C-, D- and E-pieces, where the E-pieces lie in $(\frac{2}{11}, \frac{1}{5}]$. Let P^* denote some optimal packing of L and let P_{FFD} denote the FFD-packing of L. Define $\mathscr{B}$ to be the set of non-A-pieces in L and

$$\mathscr{O} \equiv \{a_i \in \mathscr{B} : a_i \text{ is not in an A-bin in } P^*\},$$

$$\mathscr{F} \equiv \{a_i \in \mathscr{B} : a_i \text{ is not in an A-bin in } P_{\mathrm{FFD}}\}.$$

(b) A key idea now is to note that a packing of the non-A-bins in P_{FFD} is the same as if we had applied FFD directly to $\mathscr{F}$ alone. And since $\mathscr{F}$ does not contain any A-pieces, all the facts we proved before about W will hold for that packing. In particular, from Lemma 4.2 with $N = 6$ we have

$$(*) \qquad W(\mathscr{F}) \geqq \mathrm{FFD}(L) - |\mathrm{A}| - 4$$

where $|\mathrm{A}|$ denotes the number of A-pieces in L.

(c) However, we still face the problem that $\mathcal{O}$ need not contain the same elements as $\mathcal{F}$. Some elements that are not in A-bins in one packing may be in A-bins in the other. In addition, there is the fact that the number of A-bins is the same in both packings and must somehow be counted when we try to put a bound on FFD(L). To take care of these two problems, we introduce two functions:

$$f: L \to 2^{\mathcal{F}} \quad \text{and} \quad g: L \to Q.$$

They satisfy the two properties: (i) $\mathcal{F} = \bigcup_{i=1}^{n} f(a_i)$ and (ii) $|A| \geqq \sum_{i=1}^{n} g(a_i)$.

(d) f and g can be extended to set functions on subsets $X \subseteq L$ by

$$f(X) = \bigcup_{a \in X} f(a), \qquad g(X) = \sum_{a \in X} g(a).$$

We can then use a case analysis to establish the following critical inequality for the set of pieces X in any legally filled bin:

$$(**) \qquad W(f(X)) + g(X) \leqq \tfrac{11}{9}(y(X) + g(X)),$$

where

$$y(X) = \begin{cases} 1 & \text{if } X \text{ contains no A-pieces,} \\ 0 & \text{otherwise.} \end{cases}$$

We can say that the left-hand side of the inequality counts bins in P_{FFD} and the right-hand side does the same for P^* (and multiplies the result by $\frac{11}{9}$), since W counts non-A-bins in P_{FFD}, y counts them in P^*, and g counts the A-bins in both packings.

(e) Given this intuitive way of looking at (**), we observe that, if f and g satisfy (i) and (ii), respectively, and if property (**) holds for all possible X_i (the set of elements in the ith bin of P^*), Theorem 3.2 then follows by summation. For in this case we would have

$$\begin{aligned} W(\mathcal{F}) + g(L) &\leqq \bigcup_{i=1}^{L^*} W(f(X_i)) + \sum_{i=1}^{L^*} g(X_i) && \text{by subadditivity of } W \text{ and (i) and (ii)} \\ &\leqq \sum_{i=1}^{L^*} \tfrac{11}{9} y(X_i) + \tfrac{11}{9} g(L) && \text{by (**)} \\ &= \tfrac{11}{9}(L^* - |A| + g(L)) && \text{by definition of } y. \end{aligned}$$

Thus by (*),

$$\text{FFD}(L) - |A| - 4 + g(L) \leqq \tfrac{11}{9}(L^* - |A| + g(L)),$$

implying that

$$\begin{aligned} \text{FFD}(L) &\leqq \tfrac{11}{9} L^* - \tfrac{2}{9}(|A| - g(L)) + 4 \\ &\leqq \tfrac{11}{9} L^* + 4 \qquad \text{by (ii)}, \end{aligned}$$

which is just Theorem 3.2.

Unfortunately, the amount of space required to present the details necessary to establish the preceding remarks prohibit us from giving them here. The definitions of f and g are also somewhat complicated, although intuitively f assigns to

each piece a_i in L a subset of $\mathscr{F}$ for which a_i is, in a rough sense, "uniquely responsible", while g serves to count those A-bins which "collaborated" in this "responsibility". Needless to say, the actual arguments are considerably more complex than those given for Theorem 4.1. The interested reader is referred to Johnson [8] in which the complete details of these proof techniques may be found.

5. Concluding remarks. The four bin-packing algorithms studied in this paper are actually special cases of more general classes of algorithms which have been considered in some detail by Johnson [8], [9]. We mention here several relatively unexplored directions in this area which seem to be of some interest.

(i) What is the worst-case behavior for BFD and FFD when the lists L are restricted from above and below, i.e., $L \subseteq (a, \beta)$ for fixed $0 < \alpha \leqq \beta < 1$. The corresponding results for FF and BF are known and can be found in Johnson [8]. It appears that the precise bounds on this behavior will probably be rather complicated functions of α and β, depending on certain of their number-theoretic properties.

(ii) How do these various algorithms compare among themselves? For example, we have seen that $\mathrm{BFD}(L) \leqq \mathrm{FFD}(L)$ for $L \subseteq [\frac{1}{6}, 1]$. On the other hand, $\mathrm{BFD}(L)/\mathrm{FFD}(L) \geqq \frac{10}{9}$ can occur for lists L with arbitrarily large L^*. How large can this ratio be for large L^*? How small can it be? The same questions can be asked for other pairs of algorithms.

(iii) What is the trade-off between the effectiveness of an algorithm and the efficiency of implementation of the algorithm? For example, for a list L with n elements $\mathrm{FFD}(L)$ and $\mathrm{BFD}(L)$ are both bounded above by $\frac{11}{9}L^*$ and both can be implemented using $O(n \log n)$ operations. How well can an $O(n)$ algorithm perform? If we are willing to use an $O(n^2)$ algorithm, how close to L^* can we be guaranteed of coming?

(iv) It is possible to consider bins with differing capacities. How does the ordering of the bin sizes affect the number of bins required by the various algorithms under consideration? For example, by how much can ordering the bins *largest first* differ from the optimal ordering when FFD is applied?

(v) All the questions raised so far also apply (with suitable modifications) to *two-dimensional* bin packing. In view of potential applications, this direction would seem to warrant further investigation.

(vi) What is the *expected* behavior of these algorithms? For example, if the elements of L are chosen uniformly from $[0, 1]$, what is the expected value of $\mathrm{FF}(L)/L^*$? $\mathrm{FFD}(L)/L^*$? Simulation results on FF, BF, FFD, BFD [4], [8] indicate that $\mathrm{FFD}(L)$ and $\mathrm{BFD}(L)$ are almost always better than $\mathrm{FF}(L)$ and $\mathrm{BF}(L)$ for a random L, with BF occasionally slightly better than FF.

REFERENCES

[1] A. R. BROWN, *Optimum Packing and Depletion*, American Elsevier, New York, 1971.

[2] R. W. CONWAY, W. L. MAXWELL AND L. W. MILLER, *Theory of Scheduling*, Addison-Wesley, Reading, Mass., 1967.

[3] S. A. COOK, *The complexity of theorem-proving procedures*, Proc. 3rd Annual ACM Symp. on the Theory of Computing, 1971, pp. 151–158.

[4] J. CUNY, Private communication.

[5] S. EILON AND N. CHRISTOFIDES, *The loading problem*, Management Sci., 17 (1971), pp. 259–268.

[6] M. R. GAREY, R. L. GRAHAM AND J. D. ULLMAN, *Worst-case analysis of memory allocation algorithms*, Proc. 4th Annual ACM Symp. on the Theory of Computing, 1972, pp. 143–150.

[7] P. C. GILMORE AND R. E. GOMORY, *A linear programming approach to the cutting stock problem II*, Operations Res., 11 (1963), pp. 863–888.

[8] D. S. JOHNSON, Near-optimal bin packing algorithms, Doctoral thesis, Mass. Inst. of Tech., Cambridge, Mass., 1973.

[9] ———, *Fast algorithms for bin packing*, J. Comput. Systems Sci., 8 (1974), pp. 272–314.

[10] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.

SIAM J. COMPUT.
Vol. 3, No. 4, December 1974

ERRATUM: GENETIC ALGORITHMS AND THE OPTIMAL ALLOCATION OF TRIALS*

JOHN H. HOLLAND†

The statement of Theorem 1 (l. 5, p. 92) contains an important misprint. For "ξ_2" one should read "$\xi_{(2)}$". ($\xi_{(2)}$ is the schema with the lowest *observed* payoff rate after N trials.)

* This Journal, 2 (1973), pp. 88–102. Received by the editors October 29, 1973.

† Department of Computer and Communication Sciences, University of Michigan, Ann Arbor, Michigan 48104.