SIAM J. COMPUT.
Vol. 11, No. 1, February 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1101-0001 $01.00/0

STRAIGHT-LINE PROGRAMS WITH ONE INPUT VARIABLE*

OSCAR H. IBARRA† AND BRIAN S. LEININGER†

Abstract. Let $\mathbb{C}$ be the set of all straight-line programs with *one* input variable, x, using the following instruction set: $y \leftarrow 0$, $y \leftarrow 1$, $y \leftarrow y + w$, $y \leftarrow y - w$, $y \leftarrow y * w$, and $y \leftarrow \lfloor y/w \rfloor$. We show that two programs in $\mathbb{C}$ are equivalent over integer inputs if and only if they are equivalent on all inputs x such that $|x| \leqq 2^{2^{\lambda r^2}}$ (λ is a fixed positive constant and r is the maximum of the lengths of the programs). In contrast, we prove that the zero-equivalence problem (deciding whether a program outputs 0 for all inputs) is undecidable for programs with *two* input variables. An interesting corollary is the following: Let $\mathbb{N}$ be the set of natural numbers and f be any total one-to-one function from $\mathbb{N}$ onto $\mathbb{N} \times \mathbb{N}$ (f is called a pair generator. Such functions are useful in recursive function theory and computability theory.) Then f cannot be computed by any program in $\mathbb{C}$.

Key words. straight-line program, equivalence, zero-equivalence, decidable, undecidable, Hilbert's tenth problem, pair generator

1. Introduction. In this paper, we study the problem of deciding the equivalence of straight-line programs over integer inputs using the operations $+$, $-$, $*$, and $/$, where division is $\lfloor x/y \rfloor$ = the greatest integer $\leqq x/y$ (e.g., $\lfloor 5/4 \rfloor = 1$, $\lfloor -4/3 \rfloor = -2$, etc.).[1] Two programs are equivalent if they are defined at the same points and equal wherever they are defined. Our main result is that equivalence is decidable for straight-line programs with *one* input variable. (There is no restriction on the number of auxiliary and output variables.) More precisely, we show that two programs with one input variable over the instruction set $\{y \leftarrow 0, y \leftarrow 1, y \leftarrow y + w, y \leftarrow y - w, y \leftarrow y * w, y \leftarrow \lfloor y/w \rfloor\}$ are equivalent if and only if they are equivalent on all inputs x such that $|x| \leqq 2^{2^{\lambda r^2}}$, where λ is a fixed positive constant and r is the maximum of the lengths of the programs. (The length of a program is the number of instructions in it.) The double exponential bound cannot be reduced substantially since we can show that for infinitely many r's there are nonequivalent programs with at most r instructions that are equivalent on all inputs x such that $|x| \leqq 2^{2^{\lambda' r}}$ (λ' is a fixed positive constant). In contrast, we can show that the zero-equivalence problem (deciding whether a program outputs 0 for all inputs) is undecidable for programs with *two* input variables. An interesting corollary is that no pair generator can be computed by a program using the instruction set above. A pair generator is any one-to-one function from $\mathbb{N}$ (set of natural numbers) onto $\mathbb{N} \times \mathbb{N}$. Such functions are useful in recursive function theory and computability theory. The undecidability of the zero-equivalence problem for programs with *two* input variables should be contrasted with a recent result in [6]. It was shown in [6] that the zero-equivalence problem for $\{y \leftarrow 0, y \leftarrow 1, y \leftarrow y + w, y \leftarrow \lfloor y/w \rfloor\}$-programs[2] with *ten* input variables is undecidable. This result does not use the operations $-$ and $*$.

There are other types of division: $\lceil x/y \rceil$ and $\langle x/y \rangle$. $\lceil x/y \rceil$ is the least integer $\geqq x/y$ (e.g. $\lceil 5/4 \rceil = 2$, $\lceil -4/3 \rceil = -1$, etc.), $\langle x/y \rangle$ is the integral part of x/y (e.g. $\langle 5/4 \rangle = 1$, $\langle -4/3 \rangle = -1$, etc.). Clearly, $\lfloor x/y \rfloor$ and $\langle x/y \rangle$ are identical when $xy \geqq 0$, but may differ when $xy < 0$. The following propositions whose proofs are given in the Appendix show that $\lfloor x/y \rfloor$, $\lceil x/y \rceil$, and $\langle x/y \rangle$ are not independent operations.

* Received by the editors November 28, 1979, and in final form March 31, 1981. This research was supported by the National Science Foundation under grant MCS78-01736.

† Department of Computer Science, University of Minnesota, Minneapolis, Minnesota 55455.

[1] We assume that division by 0 is undefined. If an instruction attempts to divide by 0, the program goes into an infinite loop, and its output is undefined.

[2] $\{i_1, \cdots, i_k\}$-programs denotes the class of programs using the instruction set $\{i_1, \cdots, i_k\}$.

PROPOSITION 1. *The instruction* $x \leftarrow \langle x/y \rangle$ *can be computed by a fixed program using only instructions of the form* $y \leftarrow 0$, $y \leftarrow 1$, $y \leftarrow y + w$, $y \leftarrow y - w$, $y \leftarrow y * w$, $y \leftarrow \lfloor y/w \rfloor$.

PROPOSITION 2. *The instructions* $x \leftarrow \lfloor x/y \rfloor$ *and* $x \leftarrow \lceil x/y \rceil$ *can be computed by fixed programs using only instructions of the form* $y \leftarrow 0$, $y \leftarrow 1$, $y \leftarrow y + w$, $y \leftarrow y - w$, $y \leftarrow y * w$, $y \leftarrow \langle y/w \rangle$.

For notational convenience, most of the results in the paper are stated using only the division $\lfloor x/y \rfloor$. However, it is obvious from Propositions 1 and 2 that the results remain valid when all types of division are used.

Remark. We could include the construct $y \leftarrow c$ (c is any positive integer) in our instruction set. However, it is clear that $y \leftarrow c$ can be computed by a $\{y \leftarrow 0, y \leftarrow 1, y \leftarrow y + w\}$-program with at most $O(\log c)$ instructions. Thus, inclusion of the construct $y \leftarrow c$ in the instruction set is not necessary.

2. The main result. Our main theorem is a generalization of the fact that two nth degree polynomials (outputs of programs using only $+$, $-$, $*$) are identical if they agree on $\{0, 1, \cdots, n\}$. Division introduces several complications:

(a) division by 0 is possible (so functions become nontotal);
(b) rational functions can be computed instead of polynomials;
(c) truncation permits selective forward conditional branching, by evaluating different functions for different residue classes.

Nevertheless, it is possible to associate a rational function R of the input variable with each program statement, such that $R(x)(\lfloor R(x) \rfloor)$ will be the value computed by the statement. We show in this paper that two such expressions are equivalent if they are identical for some computable initial segment of the integers. This is so because rational functions behave asymptotically as polynomials; in particular, they are nonzero for large values of the input, unless they are identically zero. Thus, (a) and (b) can be handled. Truncation may introduce branching, but the degree of branching is bounded by the number of residue classes of the greatest value computed by the program (relative to all possible moduli, i.e., relative to all possible moduli smaller than this value). Thus the number of cases is finite for each program, and can be tested. Our task then is to formally prove that the strategy above can be carried out. For convenience, we introduce the following notation.

Notation. Let $c = m/n$ be a rational number (positive, negative, or zero), where m and n are integers with $\gcd(m, n) = 1$. If $c = 0$, take $m = 0$ and $n = 1$. We use the following notation: Num $(c) = |m|$ = absolute value of m and Denom $(c) = |n|$.

Our first lemma concerns polynomial division. It says that, for sufficiently large values of x, a rational form $r(x)$ behaves like a polynomial. Lemma 1 gives a sufficient condition on x for a good approximation.

LEMMA 1. *Let* $r(x) = s(x)/z(x) = p(x) + q(x)/z(x)$, *where* $s(x)$, $z(x)$, $p(x)$ *and* $q(x)$ *are polynomials with rational coefficients such that* $z(x) \neq 0$ ($p(x)$ *and* $q(x)$ *are the polynomials given by the division algorithm*). *Let*

$$b = \max\{2, \text{Num}(c), \text{Denom}(c) \mid c \text{ is a rational coefficient in } s(x) \text{ or } z(x)\},$$

$$d = \max\{\text{degree}(s(x)), \text{degree}(z(x))\},$$

$$\alpha = b^{2(d+2)^3}.$$

Then for all $x \geqq \alpha$

$$\lfloor r(x) \rfloor = \lfloor s(x)/z(x) \rfloor = \begin{cases} \lfloor p(x) \rfloor & \text{if } p(x) \text{ is not an integer or if} \\ & \text{sign } (q(\alpha)/z(\alpha)) \text{ is nonnegative}, \\ \lfloor p(x) - 1 \rfloor & \text{otherwise.} \end{cases}$$

Moreover, if c is a coefficient of $p(x)$, *then* Num (c), Denom $(c) \leqq b^{2(d+1)^2}$.

Proof. Let h be the least positive integer such that $h \cdot s(x)$ and $h \cdot z(x)$ are polynomials with integer coefficients. Clearly, the absolute values of the coefficients of $h \cdot s(x)$ and $h \cdot z(x)$ do not exceed the bound b^{2d+2}. So without loss of generality we may assume that the coefficients of $s(x)$ and $z(x)$ are integers and their absolute values are bounded by b^{2d+2}.

Let $s(x) = s_1x^k + \cdots + s_kx + s_{k+1}$ and $z(x) = z_1x^l + \cdots + z_lx + z_{l+1}$ for some $k, l \geqq 0$. Assume similar notation for the other polynomials in this proof. Also, by multiplying $s(x)$ and $z(x)$ by -1 if necessary, we can assume $z_1 > 0$. For convenience, define $z_j = 0$ for $j \geqq l+2$. Then by the division algorithm

$$r(x) = \frac{s(x)}{z(x)} = p(x) + \frac{q(x)}{z(x)}.$$

Since we would like to derive the worst possible bound on c, we assume the following:

(i) $k \geqq l$ since, otherwise, $p(x) = 0$ and $q(x) = s(x)$.

(ii) $l \geqq 1$ (i.e., degree $(z(x)) \geqq 1$). Note that this implies that degree $(q(x)) < l$. One can easily check that these assumptions given rise to a worst-case bound on c.

We can find the coefficients of $p(x)$ by the division algorithm. We have

$$p(x) = p_1x^{k-l} + \cdots + p_{k-l}x + p_{k-l+1}, \quad \text{where}$$

$$p_1 = \frac{s_1}{z_1}, \tag{1}$$

$$p_i = \frac{s_i - \sum_{j=1}^{i-1} p_j z_{i+1-j}}{z_1} = \frac{s_i - \sum_{j=1}^{i-1} p_j z_{i+1-j}}{z_1} \cdot \frac{z_1^{i-1}}{z_1^{i-1}} \quad \text{for } 2 \leqq i \leqq k-l+1. \tag{2}$$

Also, by induction on i,

$$p_i z_1^i \text{ is an integer for } 1 \leqq i \leqq k-l+1. \tag{3}$$

From (1)–(3) and the fact that $s_1, \cdots, s_{k+1}, z_1, \cdots, z_{l+1}$ are integers with absolute values bounded by b^{2d+2}, we have

$$\begin{aligned} &\text{Num } (p_i) \leqq 2^{i-1} b^{(2d+2)i} \quad \text{and} \\ &\text{Denom } (p_i) \leqq b^{(2d+2)i} \quad \text{for } 1 \leqq i \leqq k-l+1. \end{aligned} \tag{4}$$

Since $b \geqq 2$ and $d \geqq k \geqq l \geqq 1$, (4) becomes

$$\begin{aligned} &\text{Num } (p_i) \leqq 2^{d-1} b^{(2d+2)d} \leqq b^{2(d+1)^2} \quad \text{and} \\ &\text{Denom } (p_i) \leqq b^{(2d+2)d} \leqq b^{2(d+1)^2} \quad \text{for } 1 \leqq i \leqq k-l+1. \end{aligned} \tag{5}$$

Now $q(x) = s(x) - p(x)z(x)$. Then

$$\begin{aligned} q_1x^{l-1} + \cdots + q_{l-1}x + q_l = {} & (s_1x^k + \cdots + s_kx + s_{k+1}) \\ & - (p_1x^{k-l} + \cdots + p_{k-l}x + p_{k-l+1})(z_1x^l + \cdots + z_lx + z_{l+1}). \end{aligned}$$

So

$$q_{l-r} = s_{k-r+1} - \sum_{j=0}^{r} (p_{k-l-j+1})(z_{l+j-r+1}) \quad \text{for } r = 0, 1, \cdots, l-1.$$

Letting $r = l - i$, we have

(6) $$q_i = s_{k-l+i+1} - \sum_{j=0}^{l-i} (p_{k-l-j+1})(z_{i+j+1}) \quad \text{for } i = 1, 2, \cdots, l.$$

From (5) and (6), we easily obtain upper and lower bounds on the absolute values of the *nonzero* q_i's:

(7) $$|q_i| \leqq b^{2d+2} + lb^{2(d+1)^2} b^{(2d+2)} \leqq b^{2d+2} + db^{2(d+1)^2+(2d+2)} \leqq b^{2(d+2)^2},$$

(8) $$|q_i| \geqq \frac{1}{(b^{2(d+1)^2})^l} \geqq \frac{1}{b^{2(d+1)^2 d}} \geqq \frac{1}{b^{2(d+1)^3}}.$$

We can write $p(x) = u(x)/m$, where m is the least positive integer such that $u(x)$ is a polynomial with integer coefficients. Now choose the least nonnegative integer β such that

$$\left|\frac{q(x)}{z(x)}\right| < \frac{1}{m} \quad \text{for all } x \geqq \beta.$$

The bounds for m and β are found as follows:

From (5),

(9) $$m \leqq (b^{2(d+1)^2})^{\text{degree}(p(x))+1} \leqq b^{2(d+1)^3}.$$

Now, for all x, $|q(x)/z(x)| < 1/m$ if and only if $|z(x)| - m|q(x)| > 0$. Hence from (7) and (9), for sufficiently large x,

$$\begin{aligned} |z(x)| - m|q(x)| &\geqq x^l - l(b^{2d+2} + mb^{2(d+2)^2})x^{l-1} \\ &\geqq (x - d(b^{2d+2} + b^{2(d+1)^3} b^{2(d+2)^2}))x^{l-1} \\ &= (x - d(b^{2d+2} + b^{2(d+1)^3+2(d+2)^2}))x^{l-1} > 0. \end{aligned}$$

Let $\beta = b^{2(d+2)^3}$. Then $\beta > d(b^{2d+2} + b^{2(d+1)^3+2(d+2)^2})$ and, for all $x \geqq \beta$, $|q(x)/z(x)| < 1/m$.

Let α be the least integer such that $\alpha \geqq \beta$ and sign $(q(x)/z(x)) = $ sign $(q(\alpha)/z(\alpha))$ for all $x \geqq \alpha$. We consider two cases:

Case 1. sign $(q(\alpha)/z(\alpha))$ is nonnegative or $p(x)$ is not an integer. Then clearly $\lfloor r(x) \rfloor = \lfloor s(x)/z(x) \rfloor = \lfloor p(x) \rfloor$.

Case 2. sign $(q(\alpha)/z(\alpha))$ is negative and $p(x)$ is an integer. Then $\lfloor r(x) \rfloor = \lfloor s(x)/z(x) \rfloor = \lfloor p(x) - 1 \rfloor = p(x) - 1$.

The bound on α is derived as follows: Let the degree of $q(x)$ be $l - i$ for some $1 \leqq i \leqq l - 1$. Then from (7) and (8), for sufficiently large x,

$$\begin{aligned} |q(x)| &\geqq \frac{x^{l-i}}{b^{2(d+1)^3}} - (l-i)b^{2(d+2)^2} x^{l-i-1} \\ &\geqq \left(\frac{x - (l-i)b^{2(d+1)^3+2(d+2)^2}}{b^{2(d+1)^3}}\right) x^{l-i-1} \\ &\geqq \left(\frac{x - db^{2(d+1)^3+2(d+2)^2}}{b^{2(d+1)^3}}\right) x^{l-i-1} > \left(\frac{x - b^{2(d+2)^3}}{b^{2(d+1)^3}}\right) x^{l-i-1} \end{aligned}$$

and

$$|z(x)| \geqq x^l - lb^{2d+2} x^{l-1} \geqq (x - db^{2d+2})x^{l-1} > (x - b^{3(d+1)})x^{l-1}.$$

Let $\alpha = \max\{\beta, b^{2(d+2)^3}, b^{3(d+1)}\} = b^{2(d+2)^3}$. Then for all $x \geqq \alpha$, $\operatorname{sign}(q(x)/z(x)) = \operatorname{sign}(q(\alpha)/z(\alpha))$. □

It is well known that every nonzero polynomial has a finite number of zeros. The next proposition bounds the absolute value of a zero.

PROPOSITION 3. *Let $p(x)$ be a nonzero polynomial with rational coefficients. Let $d = \text{degree}(p(x))$ and $b = \max\{\text{Num}(c), \text{Denom}(c) | c$ is a rational coefficient in $p(x)\}$. Let $\beta = db^2 + 1$. Then $p(x) \neq 0$ for all $x \geqq \beta$.*

Proof. Clearly, $p(x) \neq 0$ for all x such that $x^d/b - dbx^{d-1} > 0$. □

We will also need the following proposition which is easily verified.

PROPOSITION 4. *Let $(m_1, n_1), \cdots, (m_k, n_k)$ be pairs of integers such that $0 \leqq m_i < n_i$ for $1 \leqq i \leqq k$. Let l be a positive integer. If there exists an integer x_0 such that $x_0 > l + n_1 n_2 \cdots n_k$ and $x_0 \bmod n_i = m_i$ for[3] $1 \leqq i \leqq k$, then there exists another integer x_0' such that $l \leqq x_0' \leqq l + n_1 \cdots n_k$ and $x_0' \bmod n_i = m_i$ for $1 \leqq i \leqq k$.*

Proof. Let $x_0' = x_0 - rn_1 \cdots n_k$, where r is the largest positive integer such that $x_0 - rn_1 \cdots n_k \geqq l$. □

Let P be a $\{y \leftarrow 0, y \leftarrow 1, y \leftarrow y + w, y \leftarrow y - w, y \leftarrow y * w, y \leftarrow \lfloor y/w \rfloor\}$-program with one input variable x. The next lemma shows that the value of any variable y at the end of r instructions can be described by a finite nonempty set $S(r, y)$ of congruence classes of the input. Each congruence class is a pair $(p(x), T)$, where $p(x)$ is a polynomial in x with rational coefficients and T is a finite nonempty set of pairs of integers. $(p(x), T)$ in $S(r, y)$ means that the value of y on large enough input x_0 at the end of r instructions (if defined) is equal to $p(x_0)$ if and only if $x_0 \bmod n = m$ for each (m, n) in T.

LEMMA 2. *Let P be a $\{y \leftarrow 0, y \leftarrow 1, y \leftarrow y + w, y \leftarrow y - w, y \leftarrow y * w, y \leftarrow \lfloor y/w \rfloor\}$-program with one input variable x (but with an arbitrary number of auxiliary and output variables). Assume without loss of generality that x does not appear on the left-hand side* (LHS) *of any instruction in P. Let y be any variable in P (possibly x), and $r \geqq 1$. Then there is a finite nonempty set $S(r, y)$ with elements of the form $(p(x), T)$, where $p(x)$ is a polynomial in x with rational coefficients and T is a finite nonempty set of pairs of integers. $S(r, y)$ has the following properties:*

(1) *Let x_0 be an input such that $x_0 \geqq 2^{2^{3r^2}}$. If the value of y is defined at the end of r instructions then there is a unique element $(p(x), T)$ in $S(r, y)$ such that $x_0 \bmod n = m$ for all (m, n) in T, and the value of y on input x_0 (at then end of r instructions) is given by $p(x_0)$. (We say in this case that $(p(x), T)$ uniquely defines y on input x_0.) Moreover, if x_0' is such that $x_0' \geqq 2^{2^{3r^2}}$ and $x_0' \bmod n = m$ for all (m, n) in T, then $(p(x), T)$ also uniquely defines y on input x_0'.*

(2) *Let $x_0 \geqq 2^{2^{3r^2}}$. Suppose y on input x_0 is uniquely defined by $(p(x), T)$ in $S(r, y)$ (at the end of r instructions). If $p(x)$ is not the zero polynomial, then the value of $y = p(x_0) \neq 0$.*

(3) *$|S(r, y)| =$ cardinality of $S(r, y) \leqq 2^{2^{5(r+1)^2}}$.*

(4) *If $(p(x), T)$ is in $S(r, y)$, then* degree $(p(x)) \leqq 2^r$.

(5) *If c is a rational coefficient in $p(x)$, then* Num (c), Denom $(c) \leqq 2^{2^{3r^2}}$.

(6) *If (m, n) is in T, then $0 \leqq m < n \leqq 2^{2^{3r^2}}$.*

(7) *$|T| = \leqq 2^{r^2}$.*

Proof. The proof is an induction on r. At the start, all variables except x have the value 0. Let y be the variable on the LHS of the first instruction. By assumption,

[3] For integers u and v with $v > 0$, let $r =$ remainder of $|u|/v$. Then

$$u \bmod v = \begin{cases} r & \text{if } r = 0 \text{ or } u \geqq 0, \\ v - r & \text{if } r > 0 \text{ and } u < 0. \end{cases}$$

$y \neq x$. Then

$$S(1, x) = \{(x, \{(0, 1)\})\} \quad \text{and}$$
$$S(1, w) = \{(0, \{(0, 1)\})\} \quad \text{for all } w \neq x, w \neq y.$$

There are four cases to consider for $S(1, y)$:

(i) If the first instruction is $y \leftarrow c (c = 0$ or $1)$, then let $S(1, y) = \{(c, \{(0, 1)\})\}$.

(ii) If the first instruction is $y \leftarrow y + w$ or $y \leftarrow y - w$ and $w \neq x$, then let $S(1, y) = \{(0, \{(0, 1)\})\}$.

(iii) If the first instruction is $y \leftarrow y + x$ or $y \leftarrow y - x$, then let $S(1, y) = \{(x, \{(0, 1)\})\}$ or $S(1, y) = \{(-x, \{(0, 1)\})\}$, respectively.

(iv) If the first instruction is $y \leftarrow y * w$ (w can be x) or $y \leftarrow \lfloor y/x \rfloor$ then let $S(1, y) = \{(0, \{(0, 1)\})\}$.

Clearly, properties (1)–(7) hold. Assume now that the lemma holds for sequences of $r \geqq 1$ instructions. We show that it also holds for sequences of $r+1$ instructions. Let y be the variable on the LHS of the $(r+1)$st instruction (note that $y \neq x$). Then, for each variable $w \neq y$, define $S(r+1, w) = S(r, w)$. Obviously, properties (1)–(7) hold for $S(r+1, w)$. If $w = y$, we consider 3 cases.

Case 1. The $(r+1)$st instruction is $y \leftarrow c$, where $c = 0$ or 1. Then let $S(r+1, y) = \{(c, \{(0, 1)\})\}$. Clearly, properties (1)–(7) hold.

Case 2. The $(r+1)$st instruction is $y \leftarrow y$ op w, where op is $+$, $-$ or $*$. Let x_0 be an input such that $x_0 \geqq 2^{2^{3(r+1)^2}}$. Suppose that on input x_0, y and w are defined at the end of r instructions and their values are uniquely defined by $(s(x), T_1)$ in $S(r, y)$ and $(z(x), T_2)$ in $S(r, w)$, respectively. Then $(s(x) \text{ op } z(x), T_1 \cup T_2)$ uniquely defines y on input x_0 at the end of $r+1$ instructions, and $(s(x) \text{ op } z(x), T_1 \cup T_2)$ should be in the set $S(r+1, y)$. It is straightforward to verify that properties (1) and (4)–(7) hold for $(s(x) \text{ op } z(x), T_1 \cup T_2)$. That property (2) is satisfied follows from Proposition 3. The elements of $S(r+1, y)$ are obtained by varying the value of $x_0 \geqq 2^{2^{3(r+1)^2}}$. Now $|T_1 \cup T_2| \leqq 2^{r^2} + 2^{r^2} \leqq 2^{(r+1)^2}$, and if (m, n) is in $T_1 \cup T_2$, then $0 \leqq m < n \leqq 2^{2^{3r^2}} \leqq 2^{2^{3(r+1)^2}}$. It follows from Proposition 4 that there are at most $2^{2^{5(r+1)^2}}$ values of x_0 giving rise to distinct elements of $S(r+1, y)$. Hence, $|S(r+1, y)| \leqq 2^{2^{5(r+1)^2}}$, showing property (3).

Case 3. The $(r+1)$st instruction is $y \leftarrow \lfloor y/w \rfloor$. Let x_0 be an input such that $x_0 \geqq 2^{2^{3(r+1)^2}}$. Suppose that on input x_0, y and w are defined at the end of r instructions and their values are uniquely defined by $(s(x), T_1)$ in $S(r, y)$ and $(z(x), T_2)$ in $S(r, w)$, respectively. If $z(x)$ is not the zero polynomial, then the value of y at the end of $r+1$ instructions is given by $\lfloor s(x_0)/z(x_0) \rfloor$. Note that by induction hypothesis, $z(x_0) \neq 0$.

We show how to construct an element $(u(x), T)$ in $S(r+1, y)$ uniquely defining y on input x_0. Now,

$$\left\lfloor \frac{s(x)}{z(x)} \right\rfloor = \left\lfloor p(x) + \frac{q(x)}{z(x)} \right\rfloor,$$

where $p(x)$ and $q(x)$ are obtained from $s(x)$ and $z(x)$ by the division algorithm. By the induction hypothesis, degree $(s(x))$, degree $(z(x)) \leqq 2^r = d$. Moreover, if c is a coefficient in $s(x)$ or $z(x)$ then Num (c), Denom $(c) \leqq 2^{2^{3r^2}} = b$. Let

$$p(x) = c_1 x^k + \cdots + c_k x + c_{k+1}, \quad \text{where } k \leqq d.$$

For $1 \leqq i \leqq k$, let $c_i = v_i / n_i$, where v_i and n_i are integers such that $n_i > 0$ and gcd $(v_i, n_i) = 1$. (If $c_i = 0$, take $v_i = 0$ and $n_i = 1$). Then, for $1 \leqq i \leqq k$,

$$c_i x_0^{k-i+1} = \lfloor c_i x_0^{k-i+1} \rfloor + \frac{l_i}{n_i},$$

where $l_i = (v_i x_0^{k-i+1}) \bmod n_i$. Then

$$\left\lfloor \frac{s(x_0)}{z(x_0)} \right\rfloor = \left\lfloor p(x_0) + \frac{q(x_0)}{z(x_0)} \right\rfloor$$

$$= \left\lfloor c_1 x_0^k + \cdots + c_k x_0 + c_{k+1} + \frac{q(x_0)}{z(x_0)} - \sum_{i=1}^{k} \frac{l_i}{n_i} + \sum_{i=1}^{k} \frac{l_i}{n_i} \right\rfloor$$

$$= \left\lfloor \left(c_1 x_0^k - \frac{l_1}{n_1} \right) + \cdots + \left(c_k x_0 - \frac{l_k}{n_k} \right) + c_{k+1} + \frac{q(x_0)}{z(x_0)} + \sum_{i=1}^{k} \frac{l_i}{n_i} \right\rfloor$$

$$= c_1 x_0^k + \cdots + c_k x_0 - \sum_{i=1}^{k} \frac{l_i}{n_i} + \left\lfloor c_{k+1} + \frac{q(x_0)}{z(x_0)} + \sum_{i=1}^{k} \frac{l_i}{n_i} \right\rfloor.$$

Now $c_1 x_0^k + \cdots + c_k x_0 - \sum_{i=1}^k l_i/n_i$ is an integer. Then by Lemma 1, since $x_0 \geqq 2^{2^{3(r+1)^2}} \geqq (2^{2^{3r^2}})^{2(2^r+2)^3} = b^{2(d+2)^3} = \alpha$,

$$\left\lfloor c_{k+1} + \frac{q(x_0)}{z(x_0)} + \sum_{i=1}^{k} \frac{l_i}{n_i} \right\rfloor = \begin{cases} \left\lfloor c_{k+1} + \sum_{i=1}^{k} \frac{l_i}{n_i} \right\rfloor & \text{if } c_{k+1} + \sum_{i=1}^k l_i/n_i \text{ is not an integer or if sign } (q(\alpha)/z(\alpha)) \text{ is nonnegative,} \\ \left\lfloor c_{k+1} + \left(\sum_{i=1}^{k} \frac{l_i}{n_i} \right) - 1 \right\rfloor & \text{otherwise.} \end{cases}$$

Let $u(x) = c_1 x^k + \cdots + c_k x + c'_{k+1}$, where

$$c'_{k+1} = \begin{cases} -\sum_{i=1}^{k} \frac{l_i}{n_i} + \left\lfloor c_{k+1} + \sum_{i=1}^{k} \frac{l_i}{n_i} \right\rfloor & \text{if } c_{k+1} + \sum_{i=1}^k l_i/n_i \text{ is not an integer or if sign } (q(\alpha)/z(\alpha)) \text{ is nonnegative,} \\ -\sum_{i=1}^{k} \frac{l_i}{n_i} + \left\lfloor c_{k+1} + \left(\sum_{i=1}^{k} \frac{l_i}{n_i} \right) - 1 \right\rfloor & \text{otherwise.} \end{cases}$$

The bounds on the coefficients can be obtained using Lemma 1: For $1 \leqq i \leqq k+1$,

$$\text{Num}(c_i), \text{Denom}(c_i) \leqq b^{2(d+1)^2} \leqq (2^{2^{3r^2}})^{2(2^r+1)^2} \leqq 2^{2^{3(r+1)^2}}.$$

It follows that for $1 \leqq i \leqq k$, $0 \leqq l_i < n_i \leqq b^{2(d+1)^2}$. Hence,

$$\text{Denom}(c'_{k+1}) \leqq (b^{2(d+1)^2})^k \leqq b^{2(d+1)^3} \leqq 2^{2^{3(r+1)^2}}$$

and

$$\begin{aligned} \text{Num}(c'_{k+1}) &\leqq k b^{2(d+1)^2 k} + b^{2(d+1)^2 k}(b^{2(d+1)^2} + k + 1) \\ &\leqq d b^{2(d+1)^2 d} + b^{2(d+1)^2 d}(b^{2(d+1)^2} + d + 1) \leqq b^{2(d+2)^3} \leqq 2^{2^{3(r+1)^2}}. \end{aligned}$$

Now let $m_i = x_0 \bmod n_i$ for $1 \leqq i \leqq k$. Then l_i depends only on m_i. Define $T = T_1 \cup T_2 \cup \{(m_1, n_1), \cdots, (m_k, n_k)\}$. Clearly, properties (1), (4), (5) and (6) hold. Also $|T| = |T_1| + |T_2| + k \leqq 2^{r^2} + 2^{r^2} + 2^r \leqq 2^{(r+1)^2}$. Hence property (7) is satisfied. The proof that property (3) holds is similar to that of case 2. Now $u(x)$ has degree at most $d = 2^r$, and if c is a coefficient in $u(x)$, Num (c), Denom $(c) \leqq b^{2(d+2)^3}$, where $b = 2^{2^{3r^2}}$. Hence, by Proposition 3, unless $u(x) = 0$, $u(x_0) \neq 0$ for all $x_0 \geqq 2^{2^{3(r+1)^2}} > 2^r((2^{2^{3r^2}})^{2(2^r+2)^3})^2 = d(b^{2(d+2)^3})^2$. Thus, (2) holds. $\square$

To handle inputs which cause division by 0, we need the next lemma.

LEMMA 3. *Let P and r be as in Lemma 2. There is a (possibly empty) set $Z(r)$ with elements that are finite nonempty sets of pairs of integers. $Z(r)$ has the following*

properties:

(1) $|Z(r)| \leqq 2^{2^{5(r+1)^2}}$.

(2) *If* T *is in* $Z(r)$, *then* $|T| \leqq 2^{r^2}$.

(3) *If* (m, n) *is in* T, *then* $0 \leqq m < n \leqq 2^{2^{3r^2}}$.

(4) *A division by* 0 *on input* $x_0 \geqq 2^{2^{3r^2}}$ *occurs during the first* r *instructions if and only if there is a* T *in* $Z(r)$ *such that* $x_0 \bmod n = m$ *for all* (m, n) *in* T.

Proof. We describe the construction of $Z(r)$. For convenience, define $Z(0) = \emptyset$. Now assume that $Z(r)$ has been constructed for $r \geqq 0$ and it satisfies (1)–(4) of the lemma. If the $(r+1)$st instruction is not a division instruction, let $Z(r+1) = Z(r)$. Now suppose that $(r+1)$st instruction is $y \leftarrow \lfloor y/w \rfloor$. Let x_0 be an input such that $x_0 \geqq 2^{2^{3r^2}}$. The instruction $y \leftarrow \lfloor y/w \rfloor$ will contribute to a division by zero on input x_0 if and only if w is defined and is equal to 0 at the end of r instructions. By Lemma 2 (property (2)), the element $(p(x), T)$ in $S(r, w)$ uniquely defining w (at the end of r instructions) on input x_0 must have $p(x)$ identically equal to 0. Add T to $Z(r+1)$. Clearly, the number of T's added to $Z(r+1)$ is at most $|S(r, w)| \leqq 2^{2^{5(r+1)^2}}$, and hence, $|Z(r+1)| \leqq 2^{2^{5(r+2)^2}}$. □

We are now ready to prove our main theorem.

THEOREM 1. *Let* P_1 *and* P_2 *be two* $\{y \leftarrow 0,\ y \leftarrow 1,\ y \leftarrow y+w,\ y \leftarrow y-w,\ y \leftarrow y*w,\ y \leftarrow \lfloor y/w \rfloor\}$*-programs with one input variable* x (*but with an aribtrary number of auxiliary and output variables*). *Assume that* P_1 *and* P_2 *have the same number of output variables. Let* $r = \max\{r_1, r_2\}$, *where* r_i = *number of instructions in* P_i. *Then* P_1 *and* P_2 *are equivalent over nonnegative integer inputs if and only if they are equivalent on all inputs* $0 \leqq x_0 \leqq 2^{2^{\lambda r^2}}$, *where* λ *is some fixed constant.*

Proof. By Lemma 2, the value of any variable y of P_i on input $x_0 \geqq 2^{2^{3r_i^2}}$ at the end of r_i instructions (if defined) is uniquely determined by an element $(p(x), T)$ in $S(r_i, y)$ where $p(x)$ is a polynomial of degree at most 2^{r_i} with rational coefficients, and T is a set of integers of the form (m, n), $0 \leqq m < n \leqq 2^{2^{3r_i^2}}$, and $|T| \leqq 2^{r_i^2}$. Similarly, by Lemma 3, the values of $x_0 \geqq 2^{2^{3r_i^2}}$ which cause program P_i to divide by 0 are determined by the set $Z(r_i)$, where an element of $Z(r_i)$ is a nonempty set T of integers (m, n). (Again, $0 < m < n \leqq 2^{2^{3r_i^2}}$ and $|T| \leqq 2^{r_i^2}$.) Now two polynomials of degree at most 2^r are identical if and only if they agree on 2^r+1 points. It follows from Proposition 4 that P_1 and P_2 are equivalent if and only if they are equivalent on all inputs

$$x_0 \leqq (2^r+1)[2^{2^{3r^2}} + (2^{2^{3r^2}})^{2^{r^2}}] \leqq 2^{2^{5r^2}}. \qquad \square$$

The double exponential bound of Theorem 1 cannot substantially be reduced since we can prove the following proposition.

PROPOSITION 5. *There are nonequivalent programs* P_1 *and* P_2 *with at most* $r \geqq 5$ *instructions that are equivalent on all inputs* $\leqq 2^{2^{r-4}}$.

Proof. Let P_1 and P_2 be the following programs (x is the input/output variable):

P_1:		P_2:	
$y \leftarrow 1$		$y \leftarrow 1$	
$y \leftarrow y+y$		$y \leftarrow y+y$	
$y \leftarrow y*y$		$y \leftarrow y*y$	
⋮	$r-3$	⋮	$r-4$
$y \leftarrow y*y$		$y \leftarrow y*y$	
$x \leftarrow \lfloor x/y \rfloor$		$x \leftarrow \lfloor x/y \rfloor$	

Clearly, P_1 and P_2 agree on all inputs $x \leqq 2^{2^{r-4}}$. But P_1 and P_2 are not equivalent. □

Theorem 1 also holds when the input variable can assume negative values:

COROLLARY 1. *Let P_1, P_2, and r be as in Theorem 1. Then P_1 and P_2 are equivalent over integer inputs if and only if they are equivalent on all inputs x such that $|x| \leqq 2^{2^{\lambda r^2}}$ (λ is some fixed constant). Moreover, the upper bound cannot be reduced substantially, since for infinitely many r's there are nonequivalent programs with at most r instructions that are equivalent on all inputs x such that $|x| \leqq 2^{2^{\lambda' r}}$ (λ' is a fixed constant).*

Proof. Let P_1 and P_2 be two programs. For $i = 1, 2$, construct program P_i' by inserting the following code at the beginning of P_i (z is a new variable):

$$z \leftarrow 0$$

$$z \leftarrow z - x$$

$$x \leftarrow 0$$

$$x \leftarrow x + z.$$

Then P_1 and P_2 are equivalent over (positive, negative, or zero) integer inputs if and only if P_1 and P_2 are equivalent over nonnegative integer inputs and P_1' and P_2' are equivalent over nonnegative integer inputs. The result now follows from Theorem 1 and Proposition 5. □

Our next result shows that Theorem 1 does not hold for programs with two input variables.

THEOREM 2. *The zero-equivalence problem for $\{y \leftarrow 0, y \leftarrow 1, y \leftarrow y + w, y \leftarrow y - w, y \leftarrow y * w, y \leftarrow \lfloor y/w \rfloor\}$-programs with two input variables (over nonnegative integer inputs) is undecidable. The result holds for programs with no more than 6 program variables. Moreover, the programs are total in that no division by 0 occurs.*

Proof. The proof uses the undecidability of Hilbert's tenth problem [3]. Let F be a Diophantine polynomial with r variables. Let

$$F = F(x_1, \cdots, x_r) = \sum_{j=1}^{m} c_j x_1^{j_1} \cdots x_r^{j_r},$$

where $|c_j| > 0$ and $j_k \geqq 0$ for $1 \leqq j \leqq m$ and $1 \leqq k \leqq r$. We shall construct a program P_F with input variables x and y and output variable z such that P_F outputs 0 for all nonnegative integer values of x and y if and only if F has no nonnegative integer solution in $x_1, \cdots, x_r$. The result would then follow from the fact that it is undecidable to determine if an arbitrary Diophantine polynomial has a nonnegative integer solution [3].

Given a nonnegative integer x and a positive integer y, we can think of x as a number in base y,

$$x = x_0 + x_1 y^1 + x_2 y^2 + \cdots + x_r y^r + v y^{r+1},$$

where $x_0, x_1, \cdots, x_r$, and v are nonnegative integers with $0 \leqq x_i < y$. Clearly, $(x_1, \cdots, x_r)$ can be made to assume all possible r-tuples by varying x and y. The program P_F decodes $x_1, \cdots, x_r$ and computes $F(x_1, \cdots, x_r)$. P_F then outputs 0 if and only if $F(x_1, \cdots, x_r) \neq 0$. The program P_F is given by the following code, which is

easily translated to a program using the instruction set $\{y \leftarrow 0,\ y \leftarrow 1,\ y \leftarrow y+w,\ y \leftarrow y-w,\ y \leftarrow y * w,\ y \leftarrow \lfloor y/w \rfloor\}$:

$z \leftarrow 0$

$y \leftarrow 2y+1$ (makes y nonzero)

α_{10}

α_{11}

$\vdots$

$\alpha_{1(r+1)}$

α_{20}

α_{21}

$\vdots$

$\alpha_{2(r+1)}$

$\vdots$

α_{m0}

α_{m1}

$\vdots$

$\alpha_{m(r+1)}$

β

where

(1) For $1 \leqq j \leqq m$, α_{j0} is the code

$w \leftarrow 1$

(2) For $1 \leqq j \leqq m$ and $1 \leqq k \leqq r$, α_{jk} is the code

$v \leftarrow \lfloor x/y^k \rfloor$

$s \leftarrow \lfloor x/y^{k+1} \rfloor y$

$v \leftarrow v - s$

$w \leftarrow w * v^{i_k}$

At the end of α_{jk}, w will contain $x_1^{i_1} \cdots x_k^{i_k}$.

(3) For $1 \leqq j \leqq m$, $\alpha_{j(r+1)}$ is the code

$z \leftarrow z + c_j w$

Clearly, at the end of $\alpha_{m(r+1)}$, z will contain $F(x_1, \cdots, x_r) = \sum_{j=1}^{m} c_j x_1^{i_1} \cdots x_r^{i_r}$.

(4) The code for β is

$z \leftarrow \lfloor (z^2+1)/(2z^2+1) \rfloor$.

Then $z=0$ if and only if $F(x_1, \cdots, x_r) \neq 0$. It follows that P_F outputs 0 for all nonnegative integer values of x and y if and only if F has no solution. □

Theorem 2 remains valid when the input variables can assume negative values:

COROLLARY 2. *Same as Theorem* 2, *but now the input variables can assume all integer values.*

Proof. Modify P_F by using the following code for β:

$$x \leftarrow \lfloor x/(x^2+1) \rfloor$$
$$y \leftarrow \lfloor y/(y^2+1) \rfloor$$
$$z \leftarrow z^2+x^2+y^2$$
$$z \leftarrow \lfloor (z+1)/(2z+1) \rfloor.$$

Call the new program P'_F. Then P'_F outputs 0 for all integer values of x and y if and only if F has no nonnegative integer solution. □

Remark. By a slightly more complicated coding the number of program variables in Theorem 2 and Corollary 2 can be reduced to 5.

3. An application. Let $\mathbb{N}$ be the set of natural numbers. It is well known that there exist effectively computable total one-to-one functions from $\mathbb{N}$ onto $\mathbb{N} \times \mathbb{N}$. Such functions are called *pair generators* [5]. Pair generators are useful in recursive function theory and computability theory (see, e.g., [2], [4], [5], [7]). Our next theorem shows that pair generators are "not easy" to compute.

THEOREM 3. *No* $\{y \leftarrow 0,\ y \leftarrow 1,\ y \leftarrow y+w,\ y \leftarrow y-w,\ y \leftarrow y*w,\ y \leftarrow \lfloor y/w \rfloor\}$-*program (with one input variable and two output variables) can compute a pair generator.*

Proof. The proof is by contradiction. Suppose that $f: \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ is a pair generator which is computed by a program P. Let u be the input variable of P and x, y be its output variables. For each program P_F constructed in the proof of Theorem 2, we define a new program P'_F:

P
P_F

We assume that P_F has input variables x and y and these are the only variables that P_F has in common with P. Now, P'_F has one input variable u, and P'_F outputs 0 for all nonnegative integer values of u if and only if P_F outputs 0 for all nonnegative integer values of x and y. The result now follows from Theorems 1 and 2. □

Theorem 3 does not hold for inverses of pair generators. (The inverses are called *pairing functions* [4], [7].) There are pair generators with easily computable inverses. For example, consider the pair generator f shown below:

	0	1	2	3	4	5	⋯ x ⋯
0	0	2	5	9	14		
1	1	4	8	13			
2	3	7	12				
3	6	11					
4	10						
5							
⋮							
y							z
⋮							

$$f: \mathbb{N} \to \mathbb{N} \times \mathbb{N}$$
$$z \to (x, y).$$

$f^{-1}: \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is given by

$$f^{-1}(x, y) = z = \left\lfloor \frac{(x+y)^2 + 3x + y}{2} \right\rfloor.$$

$f^{-1}(x, y)$ is computable by a $\{x \leftarrow 1, x \leftarrow x + y, x \leftarrow x * y, x \leftarrow \lfloor x/2 \rfloor\}$-program (with two input variables). The function f is defined by two functions g_1 and g_2 (see [2]):

$$x = g_1(z) = \left\lfloor \frac{Q_2(z) - Q_1(z)}{2} \right\rfloor,$$

$$y = g_2(z) = Q_1(z) - \left\lfloor \frac{Q_2(z) - Q_1(z)}{2} \right\rfloor,$$

where

$$Q_1(z) = \left\lfloor \frac{\lfloor \sqrt{8z+1} \rfloor + 1}{2} \right\rfloor - 1,$$

$$Q_2(z) = 2z - (Q_1(z))^2.$$

Hence, there are pair generators that are computable by $\{y \leftarrow 0, y \leftarrow 1, y \leftarrow y + w, y \leftarrow y - w, y \leftarrow y * w, y \leftarrow \lfloor y/w \rfloor, y \leftarrow \lfloor \sqrt{y} \rfloor\}$-programs, and from Theorem 2 we have

COROLLARY 3. *The zero-equivalence problem for* $\{y \leftarrow 0, y \leftarrow 1, y \leftarrow y + w, y \leftarrow y - w, y \leftarrow y * w, y \leftarrow \lfloor y/w \rfloor, y \leftarrow \lfloor \sqrt{y} \rfloor\}$*-programs with one input variable is undecidable.*

4. Extension. We can use "forward" **if** statements in our straight-line programs and the results of §§ 2 and 3 still apply. Specifically, we can add the following constructs: **skip** l, **if** $p(y)$ **then skip** l (where l is a nonnegative integer), and **halt.** $p(y)$ is a predicate of the form $y > 0$, $y \geqq 0$, or $y = 0$, and **skip** l causes the $(l+1)$st instruction following the current instruction to be executed next. A program can terminate a computation in three ways: by executing a **halt** instruction, by executing a transfer to a nonexistent instruction, or by executing the last statement of the program.[4]

The following proposition shows that the **if** constructs are not independent. (The proof is given in the Appendix.)

PROPOSITION 6. *The instructions* **if** $y > 0$ **then skip** l *and* **if** $y \geqq 0$ **then skip** l *can be expressed in terms of the instruction* **if** $y = 0$ **then skip** l.

Notation. Let L be the instruction set $\{y \leftarrow 0, y \leftarrow 1, y \leftarrow y + w, y \leftarrow y - w, y \leftarrow y * w, y \leftarrow \lfloor y/w \rfloor$, **skip** l, **if** $p(y)$ **then skip** l, **halt**$\}$.

Referring now to the proof of Lemma 2, we see that in order to extend the proof to L-programs, we need only consider (by Proposition 6) two other cases.

Case 4. The $(r+1)$st instruction is **skip** l. Let $S(r+1+l, w) = S(r, w)$ for each variable w. Then continue the construction with instruction $r + l + 2$.

Case 5. *The* $(r+1)$st instruction is **if** $y = 0$ **then skip** l. Let x_0 be an input such that $x_0 \geqq 2^{2^{3(r+1)^2}}$. Suppose that on input x_0, y is defined at the end of r instructions and its value is uniquely defined by $(s(x), T)$ in $S(r, y)$. If $s(x)$ is not the zero polynomial, then let $S(r+1, w) = S(r, w)$ for each variable w and continue the construction with instruction $r+2$. If $s(x)$ is the zero polynomial, then let $S(r+1+l, w) = S(r, w)$ for each variable w and continue the construction with instruction $r + l + 2$. The construction is completed when a **halt** instruction is encountered, or when a transfer to a nonexistent instruction is executed, or when the last instruction of the program has been considered.

---

[4] By convention, the program goes into an infinite loop when a division by 0 occurs.

Thus, Lemma 2 holds for the extended language, and all the results of §§ 2 and 3 apply. In particular, we have

THEOREM 4. *Let P_1 and P_2 be two L-programs with one input variable. Then P_1 and P_2 are equivalent if and only if they are equivalent on all inputs x such that $|x| \leqq 2^{2^{\lambda r^2}}$ (r is the maximum of the lengths of P_1 and P_2, and λ is a fixed positive constant).*

THEOREM 5. *No L-program can compute a pair generator.*

Appendix. Proofs of Propositions 1, 2, and 6.

Proof of Proposition 1. The following program which can easily be translated to a program over the instruction set $\{y \leftarrow 0,\ y \leftarrow 1,\ y \leftarrow y+w,\ y \leftarrow y-w,\ y \leftarrow y^*w,\ y \leftarrow \lfloor y/w \rfloor\}$ computes $\langle x/y \rangle$:

$$w \leftarrow \lfloor 1/(4xy+2) \rfloor \qquad w = \begin{cases} 0 & \text{if } xy \geqq 0, \\ -1 & \text{if } xy < 0 \end{cases}$$

$$v \leftarrow \lfloor (x^2+w)/y^2 \rfloor - \lfloor x^2/y^2 \rfloor - w \qquad v = \begin{cases} 0 & \text{if } y \neq 0 \text{ and } w = 0, \\ 0 & \text{if } y \neq 0,\ w = -1, \text{ and } x \text{ is a multiple of } y \\ 1 & \text{if } y \neq 0,\ w = -1, \text{ and } x \text{ is not a multiple of } y \\ \text{undefined} & \text{if } y = 0 \end{cases}$$

$$x \leftarrow \lfloor x/y \rfloor$$

$$x \leftarrow x + v$$

□

Proof of Proposition 2. Programs P_1 and P_2 below (which can easily be transformed to programs over the instruction set $\{y \leftarrow 0,\ y \leftarrow 1,\ y \leftarrow y+w,\ y \leftarrow y-w,\ y \leftarrow y*w,\ y \leftarrow \langle y/w \rangle\}$) compute $\lfloor x/y \rfloor$ and $\lceil x/y \rceil$, respectively.

Program P_1

$$w \leftarrow \langle 3xy/(3xy+1) \rangle \qquad w = \begin{cases} 0 & \text{if } xy \geqq 0, \\ 1 & \text{if } xy < 0 \end{cases}$$

$$v \leftarrow \langle (x^2-w)/y^2 \rangle - \langle x^2/y^2 \rangle + w \qquad v = \begin{cases} 0 & \text{if } y \neq 0 \text{ and } w = 0, \\ 0 & \text{if } y \neq 0,\ w = 1, \text{ and } x \text{ is a multiple of } y, \\ 1 & \text{if } y \neq 0,\ w = 1, \text{ and } x \text{ is not a multiple of } y, \\ \text{undefined} & \text{if } y = 0. \end{cases}$$

$$x \leftarrow \langle x/y \rangle$$

$$x \leftarrow x - v$$

Program P_2

$$w \leftarrow \langle 5x/(4x+1) \rangle \qquad w = \begin{cases} 0 & \text{if } x = 0, \\ 1 & \text{if } x \neq 0. \end{cases}$$

$$v \leftarrow \langle (x^2-w)/y^2 \rangle - \langle x^2/y^2 \rangle + w$$

$$x \leftarrow \lfloor x/y \rfloor$$

$$x \leftarrow x + v$$

□

Proof of Proposition 6. The constructions are straightforward. For example, **if** $y > 0$ **then skip** l can be coded as

$u \leftarrow 1$	
$v \leftarrow 1$	
$v \leftarrow v + u$	$v = 2$
$z \leftarrow 0$	
$z \leftarrow z + y$	$z = y$
$z \leftarrow z * v$	$z = 2y$
$z \leftarrow z - u$	$z = 2y - 1$
$w \leftarrow 0$	
$w \leftarrow w + z$	$w = 2y - 1$
$w \leftarrow \langle w/v \rangle$	
$w \leftarrow w * v$	
$z \leftarrow z - w$	$z = \begin{cases} 1 & \text{if } y > 0, \\ -1 & \text{if } y \leqq 0 \end{cases}$
$z \leftarrow z - u$	$z = \begin{cases} 0 & \text{if } y > 0, \\ -2 & \text{if } y \leqq 0 \end{cases}$
if $z = 0$ **then skip** l	

where u, v, w, and z are new variables. □

Acknowledgment. We would like to thank the referees for their suggestions and detailed comments which improved the presentation of our results.

REFERENCES

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

[2] M. DAVIS, *Computability and Unsolvability*, McGraw-Hill, New York, 1958.

[3] M. DAVIS, Y. MATIJASEVIČ, AND J. ROBINSON, *Hilbert's tenth problem. Diophantine equations: Positive aspects of a negative solution*, Proc. Symp. Pure Mathematics, 28 (1976), pp. 323–378.

[4] F. HENNIE, *Introduction to Computability*, Addison-Wesley, Reading, MA, 1977.

[5] J. E. HOPCROFT AND J. D. ULLMAN, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.

[6] O. H. IBARRA AND B. S. LEININGER, *The complexity of the equivalence problem for straight-line programs*, Proc. 12th Annual ACM Symposium on Theory of Computing, 1980, pp. 273–280.

[7] H. ROGERS, *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York, 1967.

SIAM J. COMPUT.
Vol. 11, No. 1, February 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1101-0002 $01.00/0

THE COMPLEXITY OF THE EQUIVALENCE PROBLEM FOR SIMPLE LOOP-FREE PROGRAMS*

OSCAR H. IBARRA† AND BRIAN S. LEININGER†

Abstract. We consider a simple class of loop-free programs whose instruction repertoire consists of $x \leftarrow 0$, $x \leftarrow c$, $x \leftarrow cx$, $x \leftarrow x/c$, $x \leftarrow x+y$, $x \leftarrow x-y$, **skip** l, **if** $p(x, y)$ **then skip** l, and **halt**. (x and y are integer variables, c is a positive integer, x/c is integer division, l is a nonnegative integer, and $p(x, y)$ is a predicate of the form $x>y$, $x \geqq y$, $x=y$, $x \neq y$, $x \leqq y$, or $x<y$; **skip** l causes the $(l+1)$st instruction following the current instruction to be executed next.) We show that the equivalence problem for this class is decidable in $2^{\lambda N^2}$ time (N = sum of the sizes of the programs and λ is a fixed positive constant). The bound cannot be reduced to a polynomial in N unless P = NP. In fact, we have the following rather surprising result: The equivalence problem for programs with one input variable (which also serves as the output variable) and one auxiliary variable using only instructions $x \leftarrow 2x$, $x \leftarrow x/2$, and $x \leftarrow x+y$ is NP-hard.

Key words. Complexity, equivalence, zero-equivalence, loop-free programs, NP-hard

1. Introduction. In an earlier paper [6], we showed that the equivalence problem[1] for several classes of straight-line programs (over positive, negative, or zero integer inputs) using only arithmetic operations is undecidable. In particular, we showed the following:

(a) The one-equivalence problem[2] is undecidable for $\{x \leftarrow 1, x \leftarrow 2x, x \leftarrow x+y, x \leftarrow x/y\}$-programs[3].

(b) The one-equivalence problem is undecidable for $\{x \leftarrow 1, x \leftarrow x/2, x \leftarrow x-y, x \leftarrow x^*y\}$-programs.

In this paper, we study a simple class of straight-line programs with a decidable equivalence problem. Specifically, we consider the class of loop-free programs whose instruction repertoire is $R=\{x \leftarrow 0, x \leftarrow c, x \leftarrow cx, x \leftarrow x/c, x \leftarrow x+y, x \leftarrow x-y,$ **skip** l, **if** $p(x, y)$ **then skip** l, **halt**$\}$. x and y are distinct integer variables, c is any positive integer, l is any nonnegative integer, and $p(x, y)$ is a predicate of the form $x>y$, $x \geqq y$, $x=y$, $x \neq y$, $x \leqq y$, or $x<y$. **skip** l causes the $(l+1)$st instruction following the current instruction to be executed next. A program (which need not contain a **halt** instruction) can terminate its computation in three ways: by executing a **halt** instruction, by executing a transfer to a nonexistent instruction (via **skip** l or **if** $p(x, y)$ **then skip** l), or by executing the last statement of the program. Two distinguished (not necessarily disjoint) sets of variables are designated input variables and output variables, respectively. We assume that all noninput variables are initialized to 0.

The main results of this paper are the following:

(1) The equivalence problem for R-programs is decidable in $2^{\lambda N^2}$ time (λ is a fixed positive constant and N is the sum of the sizes of the programs). For programs with a *fixed* number of input variables, the bound is $2^{\lambda N}$.

(2) The inequivalence problem for R-programs is in NP (= the class of languages accepted by nondeterministic polynomial-time bounded Turing machines [3]).

* Received by the editors December 19, 1979. This research was supported by the National Science Foundation under grant MCS-78-01736.

† Computer Science Department, Institute of Technology, University of Minnesota, Minneapolis, Minnesota 55455.

[1] Given two programs, are they defined at the same points and equal wherever they are defined?

[2] Given a program, does it output 1 for all inputs?

[3] $\{i_1, \cdots, i_k\}$-programs denotes the class of programs using only instructions of the form $i_1, \cdots, i_k$. x/y is integer division. (Thus, 4/3 is 1 and −4/3 is −1.)

(3) The equivalence problem for $\{x \leftarrow 2x,\ x \leftarrow x/2,\ x \leftarrow x+y\}$-programs with one input/output variable (i.e., the input variable is also the output variable) and one auxiliary variable is NP-hard. (The result also holds when $x \leftarrow x+y$ is replaced by $x \leftarrow x-y$.)

(4) The zero-equivalence problem (= does a program output 0 for all inputs?) for each of the following classes is NP-hard:

(i) $\{x \leftarrow 0,\ x \leftarrow 2x,\ x \leftarrow x/2,\ x \leftarrow x+y,\ x \leftarrow x-y\}$-programs with one input/output variable and one auxiliary variable.

(ii) $\{x \leftarrow 2x,\ x \leftarrow x/2,\ x \leftarrow x-y,\ x \leftarrow y\}$-programs with one input/output variable and one auxiliary variable.

(iii) $\{x \leftarrow 0,\ x \leftarrow x/2,\ x \leftarrow x-y\}$-programs with one input/output variable and two auxiliary variables.

(5) The zero-equivalence problem for each of the following classes is decidable in polynomial time.

(i) $\{x \leftarrow 0,\ x \leftarrow c,\ x \leftarrow -c,\ x \leftarrow cx,\ x \leftarrow x/c,\ x \leftarrow x+c,\ x \leftarrow x-c,\ x \leftarrow x-y\}$-programs with at most two variables. (This shows that (4) (ii)–(iii) may be the best possible results.)

(ii) $\{x \leftarrow 0,\ x \leftarrow c,\ x \leftarrow -c,\ x \leftarrow cx,\ x \leftarrow x/c,\ x \leftarrow x+c,\ x \leftarrow x-c,\ x \leftarrow x+y,\ x \leftarrow y\}$-programs (with no restriction on the number of input and auxiliary variables). This contrasts (3) and (4) (ii)–(iii).

2. An upper bound on the complexity of the equivalence problem for *R*-programs. In this section we show that the equivalence problem for R-programs is decidable in $2^{\lambda N^2}$ time (N = sum of the sizes of the programs and λ is a fixed positive constant). For programs with a *fixed* number of input variables (but no restriction on the number of output and auxiliary variables), the bound is $2^{\lambda N}$. We begin with the following lemma.

LEMMA 1. *Let P be an R-program. Assume that P has m input variables $x_1, \cdots, x_m$ and one output variable x_1. Let the other variables be $x_{m+1}, \cdots, x_n$. Let r be the number of instructions in P and K = product of all positive integer constants (i.e., c's) appearing in instructions of P. Then we can construct a collection D of systems of linear Diophantine equations*[4] *with the following properties:*

(1) *D has at most $2^n \cdot 6^r$ systems of equations.*

(2) *Let S be any system in D. Then*

(i) *S has at most $2n+5r$ equations in at most $3n+5r+1$ variables.*

(ii) *Each equation in S has at most 3 variables.*

(iii) *The maximum of the absolute values of all subdeterminants of the augmented matrix*[5] *of S is $K^2 4^{2n+5r}$.*

(iv) *S has $2m$ distinguished variables $x_1^0, \cdots, x_m^0, s_1^0, \cdots, s_m^0$, where the pair (x_i^0, s_i^0) is associated with the input variable x_i of P. s_i^0 will always have value 0 or 1. $s_i^0 = 1$ is interpreted as x_i^0 being actually negative.*

(3) *P computes a nonzero function (i.e., a function which is not zero on all inputs) if and only if one of the systems in D has a nonnegative integer solution. Moreover, if a system S in D has a nonnegative integer solution, then the values of $x_1^0, \cdots, x_m^0$ with appropriate signs attached (as given by the values of $s_1^0, \cdots, s_m^0$) when input to P will make P output a nonzero value.*

[4] A linear Diophantine equation is an equation of the form $a_1v_1 + \cdots + a_kv_k = b$, where $a_1, \cdots, a_k, b$ are (positive, negative, or zero) integer constants and $v_1, \cdots, v_k$ are integer variables.

[5] The augmented matrix of a system of equations $A\bar{y} = \bar{b}$ is A augmented by column vector $\bar{b}$.

Proof. We describe the construction of a system S in D. In the construction, we will be introducing new variables: For each $1 \leqq i \leqq n$, $x_i^0, x_i^1, x_i^2, \cdots$ and $s_i^0, s_i^1, s_i^2, \cdots$ are distinct variables associated with P's input variable x_i. For $t_i \geqq 0$, $s_i^{t_i}$ denotes the sign of the value of variable $x_i^{t_i}$, where $s_i^{t_i} = 0(1)$ denotes nonnegative (negative). Similarly, new variables $u^0, u^1, u^2, \cdots$ will be used. The construction of S involves defining for each instruction in P one or more equations describing the change that occurs in the variable that is modified by the instruction. The algorithm below forms the system S as a set of equations. The algorithm is nondeterministic with each choice giving rise to a new system.

ALGORITHM CONSTRUCT

$S \leftarrow \varnothing$
$q \leftarrow 0$

Add $(s_{m+1}^0 = 0, x_{m+1}^0 = 0, \cdots, s_n^0 = 0, x_n^0 = 0)$	Add to S the equations which initialize the noninput variables to 0 and their signs to 0 (i.e., nonnegative)
Add $(s_1^0 = 0)$ **or** [Add $(s_1^0 = 1, x_1^0 = u^q + 1)$; $q \leftarrow q + 1$] $\vdots$ Add $(s_m^0 = 0)$ **or** [Add $(s_m^0 = 1, x_m^0 = u^q + 1)$; $q \leftarrow q + 1$]	Nondeterministically choose the signs of $x_1^0, \cdots, x_m^0$. If s_i^0 is chosen to be 1, check that $x_i^0 > 0$

for $i \leftarrow 1$ **to** n **do**
  $t_i \leftarrow 0$

end
$p \leftarrow 1$

while $p \leqq r$ **do**
  case if pth instruction is
    $:x_i \leftarrow 0$: Add $(s_i^{t_i+1} = 0, x_i^{t_i+1} = 0)$; $t_i \leftarrow t_i + 1$; $p \leftarrow p + 1$
    $:x_i \leftarrow c$: Add $(s_i^{t_i+1} = 0, x_i^{t_i+1} = c)$; $t_i \leftarrow t_i + 1$; $p \leftarrow p + 1$
    $:x_i \leftarrow cx_i$: Add $(s_i^{t_i+1} = s_i^{t_i}, x_i^{t_i+1} = cx_i^{t_i})$; $t_i \leftarrow t_i + 1$; $p \leftarrow p + 1$
    $:x_i \leftarrow x_i + x_j$: **do** (1) **or** (2) **or** (3) **or** (4) **or** (5)
      (1) Add $(s_i^{t_i} = s_j^{t_j}, s_i^{t_i+1} = s_i^{t_i}, x_i^{t_i+1} = x_i^{t_i} + x_j^{t_j})$; $t_i \leftarrow t_i + 1$; $p \leftarrow p + 1$
      (2) Add $(s_j^{t_j} - s_i^{t_i} = 1, s_i^{t_i+1} = 0, x_i^{t_i} = x_j^{t_j} + u^q, x_i^{t_i+1} = x_i^{t_i} - x_j^{t_j})$; $t_i \leftarrow t_i + 1$; $q \leftarrow q + 1$; $p \leftarrow p + 1$
      (3) Add $(s_j^{t_j} - s_i^{t_i} = 1, s_i^{t_i+1} = 1, x_i^{t_i} + u^q + 1 = x_j^{t_j}, x_i^{t_i+1} = x_j^{t_j} - x_i^{t_i})$; $t_i \leftarrow t_i + 1$; $q \leftarrow q + 1$; $p \leftarrow p + 1$
      (4) Add $(s_i^{t_i} - s_j^{t_j} = 1, s_i^{t_i+1} = 0, x_i^{t_i} + u^q = x_j^{t_j}, x_i^{t_i+1} = x_j^{t_j} - x_i^{t_i})$; $t_i \leftarrow t_i + 1$; $q \leftarrow q + 1$; $p \leftarrow p + 1$
      (5) Add $(s_i^{t_i} - s_j^{t_j} = 1, s_i^{t_i+1} = 1, x_i^{t_i} = x_j^{t_j} + u^q + 1, x_i^{t_i+1} = x_i^{t_i} - x_j^{t_j})$; $t_i \leftarrow t_i + 1$; $q \leftarrow q + 1$; $p \leftarrow p + 1$

:$x_i \leftarrow x_i - x_j$: Same as for $x_i \leftarrow x_i + x_j$ except that in (1), (2), (3), (4), (5) the first equations are replaced, respectively, by:

$s_i^{t_i} + s_j^{t_j} = 1$

$s_i^{t_i} + s_j^{t_j} = 0$

$s_i^{t_i} + s_j^{t_j} = 0$

$s_i^{t_i} = 1,\ s_j^{t_j} = 1$

$s_i^{t_i} = 1,\ s_j^{t_j} = 1$

:$x_i \leftarrow x_i/c$: **do** (6) **or** (7) **or** (8)

(6) Add $(s_i^{t_i} = 0,\ s_i^{t_i+1} = 0,\ cx_i^{t_i+1} + u^q = x_i^{t_i},\ u^q + u^{q+1} = c-1)$; $t_i \leftarrow t_i + 1$; $q \leftarrow q+2$; $p \leftarrow p+1$

(7) Add $(s_i^{t_i} = 1,\ s_i^{t_i+1} = 1,\ cx_i^{t_i+1} + u^q = x_i^{t_i},\ u^q + u^{q+1} = c-1,\ x_i^{t_i+1} = u^{q+2} + 1)$; $t_i \leftarrow t_i + 1$; $q \leftarrow q+3$; $p \leftarrow p+1$

(8) Add $(s_i^{t_i} = 1,\ s_i^{t_i+1} = 0,\ x_i^{t_i} + u^q = c-1,\ x_i^{t_i+1} = 0)$; $t_i \leftarrow t_i + 1$; $q \leftarrow q+1$; $p \leftarrow p+1$

:**skip** l: $p \leftarrow p + l + 1$

:**if** $x_i > x_j$ **then skip** l: **do** (9) **or** (10) **or** (11) **or** (12) **or** (13) **or** (14)

(9) Add $(s_i^{t_i} = 0,\ s_j^{t_j} = 0,\ x_i^{t_i} = x_j^{t_j} + u^q + 1)$; $q \leftarrow q+1$; $p \leftarrow p+l+1$

(10) Add $(s_i^{t_i} = 0,\ s_j^{t_j} = 0,\ x_i^{t_i} + u^q = x_j^{t_j})$; $q \leftarrow q+1$; $p \leftarrow p+1$

(11) Add $(s_i^{t_i} = 0,\ s_j^{t_j} = 1)$; $p \leftarrow p+l+1$

(12) Add $(s_i^{t_i} = 1,\ s_j^{t_j} = 0)$; $p \leftarrow p+1$

(13) Add $(s_i^{t_i} = 1,\ s_j^{t_j} = 1,\ x_i^{t_i} + u^q + 1 = x_j^{t_j})$; $q \leftarrow q+1$; $p \leftarrow p+l+1$

(14) Add $(s_i^{t_i} = 1,\ s_j^{t_j} = 1,\ x_i^{t_i} = x_j^{t_j} + u^q)$; $q \leftarrow q+1$; $p \leftarrow p+1$

:**if** $x_i \geqq x_j$ **then skip** l:
:**if** $x_i < x_j$ **then skip** l:
:**if** $x_i \leqq x_j$ **then skip** l:
:**if** $x_i = x_j$ **then skip** l:
:**if** $x_i \neq x_j$ **then skip** l:
} Handled in a similar way as in **if** $x_i > x_j$ **then skip** l

:**halt**: $p \leftarrow r + 1$

end

end

Add $(x_1^{t_1} = u^q + 1)$ Insures that the final value of $x_1 \neq 0$

end

The algorithm above is nondeterministic. Every choice gives rise to a different system S. Clearly, there are at most $2^m \cdot 6^r \leqq 2^n \cdot 6^r$ systems S. This proves property (1). That properties (2) and (3) hold is easily verified. □

Next, we state a lemma concerning "small" nonnegative integer solutions to linear Diophantine equations. The proof of the lemma can be found in [5] (see also [1]).

LEMMA 2. *Let S: $A\bar{y} = \bar{b}$ be a system of linear Diophantine equations, where A is an $m \times n$ integer matrix, $\bar{y} = (y_1, \cdots, y_n)$ is a column vector of variables, and $\bar{b} = (b_1, \cdots, b_m)$ is a column vector of integer constants. If S has a nonnegative integer solution then it has a nonnegative integer solution $\hat{y}_1, \cdots, \hat{y}_n$ such that each $\hat{y}_i \leqq 3n\Delta^2$, where Δ is the maximum of the absolute values of all subdeterminants of the augmented matrix $A\bar{b}$.*

LEMMA 3. *Let P, n, r and K be as in Lemma 1. Then P computes a nonzero function if and only if it outputs a nonzero value for some input $(x_1, \cdots, x_m)$ in which each $|x_i| \leqq 2^{\lambda N}$, where N is the size of the program and λ is a fixed positive constant.*[6]

[6] $|x_i|$ = absolute value of x_i. The size of a program is the length of its representation.

Proof. From Lemmas 1 and 2, P computes a nonzero function if and only if it outputs a nonzero value for some input $(x_1, \cdots, x_m)$ in which each $|x_i| \leqq 3(3n+5r+1)(K^2 4^{2n+5r})^2$. Now if N is the size of P then $N \geqq r$, $N \geqq n \log n$, $N \geqq \log K$. It follows that $|x_i| \leqq 2^{\lambda N}$, where λ is a fixed positive constant. □

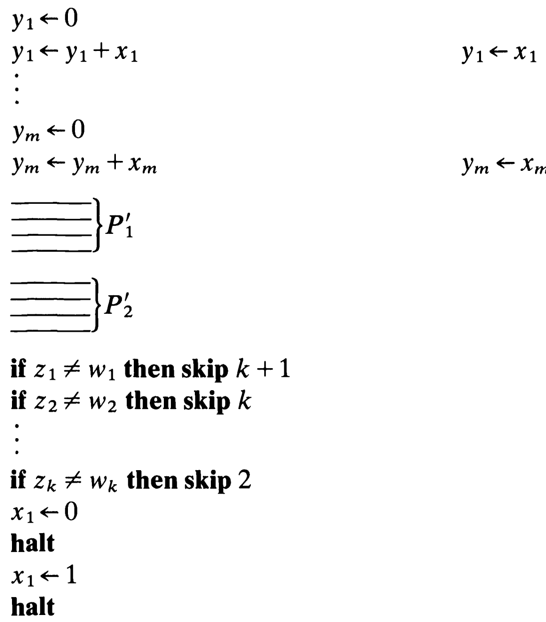
COROLLARY 1. $2^{\lambda Nm}$ *time is sufficient to decide if an arbitrary R-program computes a nonzero function.*

COROLLARY 2. *Deciding if an arbitrary R-program computes a nonzero function can be done in nondeterministic polynomial time* (NP).

We are now ready to prove the main result of this section.

THEOREM 1. *The equivalence problem for R-programs with m input variables is decidable in* $2^{\lambda Nm}$ *time* (N = *sum of the sizes of the* 2 *programs under consideration and* λ *is a fixed positive constant*).

Proof. Let P_1 and P_2 be two R-programs. Assume that they have disjoint sets of variables. Let their input variables be $x_1, \cdots, x_m$ and $y_1, \cdots, y_m$, respectively, and their output variables be $z_1, \cdots, z_k$ and $w_1, \cdots, w_k$, respectively. Define a new program P with input variables $x_1, \cdots, x_m$ and output variable x_1 as follows:



$y_1 \leftarrow 0$
$y_1 \leftarrow y_1 + x_1$ $\qquad$ $y_1 \leftarrow x_1$
⋮
$y_m \leftarrow 0$
$y_m \leftarrow y_m + x_m$ $\qquad$ $y_m \leftarrow x_m$
$\}P_1'$
$\}P_2'$
if $z_1 \neq w_1$ **then skip** $k+1$
if $z_2 \neq w_2$ **then skip** k
⋮
if $z_k \neq w_k$ **then skip** 2
$x_1 \leftarrow 0$
halt
$x_1 \leftarrow 1$
halt

For $i = 1, 2$, P_i' is P_i with each **halt** instruction replaced by **skip** l, where l is the number of instructions after the **halt** to the end of P_i. Then P computes a nonzero function if and only if P_1 is not equivalent to P_2. The result follows from Corollary 1. □

COROLLARY 3. *Equivalence of R-programs is decidable in* $2^{\lambda N^2}$ *time. For programs with a* **fixed** *number of input variables, the bound is* $2^{\lambda N}$.

From Corollary 2, we also have

COROLLARY 4. *The inequivalence problem for R-programs is in* NP.

In § 3 we will see that (in)equivalence of R-programs is NP-hard. In fact, the NP-hardness result holds for a very simple subset of R-programs.

For completeness, we mention the following result in [6] which contrasts with Corollary 3. (The input variables can assume positive, negative, or zero integer values. However, the result also applies to the case when the inputs are restricted to nonnegative integers.)

THEOREM 2.

(i) *The zero-equivalence problem for* $\{x \leftarrow 1, x \leftarrow 2x, x \leftarrow x+y, x \leftarrow x/y\}$*-programs is undecidable. The result holds even if we consider only programs that compute total functions with range* $\{0, 1\}$.

(ii) *The zero-equivalence problem for* $\{x \leftarrow 1, x \leftarrow x/2, x \leftarrow x-y, x \leftarrow x*y\}$*-programs is undecidable.*

Remark. The proof of Theorem 2 in [6] was for the one-equivalence problem (deciding if a program outputs 1 for all inputs). However, the proof can trivially be modified to apply to the zero-equivalence problem.

When there is no division, we have the following proposition.

PROPOSITION 1. *The equivalence problem for* $\{x \leftarrow 0, x \leftarrow c, x \leftarrow cx, x \leftarrow x+y, x \leftarrow x-y, x \leftarrow x*y\}$*-programs (with no restriction on the number of input, output, and auxiliary variables) is decidable.*

Proof. Let P be a program with input variables $x_1, \cdots, x_n$. Without loss of generality assume that the input variables do not appear on the left-hand sides of the instructions in P. Then the value of each output variable y at the end of the program can be represented by a polynomial $p(x_1, \cdots, x_n)$ in standard form (i.e., sum of products). Moreover, $p(x_1, \cdots, x_n)$ can be found effectively. Thus, to decide if two programs are equivalent, we find the polynomials representing their outputs. Then the programs are equivalent if and only if the polynomials representing their respective outputs are identical. (Note that this process will, in general, take exponential time since the sizes of the polynomials may grow exponentially with respect to the lengths of the programs.) □

Remark. One can easily check that all the results and proofs in this section remain valid when the inputs are restricted to nonnegative integers.

3. Two-variable $\{x \leftarrow 2x, x \leftarrow x/2, x \leftarrow x+y\}$-programs. It is very unlikely that equivalence of R-programs can be decided in polynomial time since we can show that the problem is NP-hard (see [3], [4], [7] for definitions and motivations of the terms NP-hard, NP-complete, etc.). In fact, we can show something quite surprising: The equivalence problem for $\{x \leftarrow 2x, x \leftarrow x/2, x \leftarrow x+y\}$-programs with one input variable (which is also the output variable) and one auxiliary variable is NP-hard. This result is interesting (and counterintuitive) for the following reasons:

(1) The proofs of most NP-hard results concerning equivalence of programs (see, e.g., [2]) actually show the NP-hardness of the zero-equivalence problem. Thus, for such proofs only one program is constructed. In the case of $\{x \leftarrow 2x, x \leftarrow x/2, x \leftarrow x+y\}$-programs, zero-equivalence is clearly decidable in polynomial time. Hence, the proof that equivalence is NP-hard involves the construction of two programs.

(2) There is no instruction that can set a variable to 0 or 1. Hence, there is no way to take complements, and a reduction to the satisfiability problem for Boolean formulas cannot be done directly.

(3) Only two variables (one of which is used for input/output) are needed to show NP-hardness.

(4) The variables can assume positive, negative, or zero integer values. This makes the proof harder. Note that there are some number-theoretic problems that are NP-hard when the variables are restricted to be nonnegative but become polynomial-time solvable when there is no such restriction. For example, deciding if a system of linear Diophantine equations has a nonnegative integer solution is NP-hard [9]. However, if we are interested only in any integer solution, the problem is solvable in polynomial time [8].

THEOREM 3. *The equivalence problem for* $\{x \leftarrow 2x,\ x \leftarrow x/2,\ x \leftarrow x+y\}$*-programs with one input variable (which is also the output variable) and one auxiliary variable is* NP-*hard*.

Proof. The proof uses a well-known result that the satisfiability problem for Boolean formulas in conjunctive normal form (CNF) with at most three literals per clause is NP-hard [3]. Let $F = C_1C_2 \cdots C_m$ be a Boolean formula over variables $x_1, \cdots, x_n$, where each C_i is a disjunction (i.e., sum) of at most 3 literals. (A literal is a variable or a negation of a variable.) We shall construct two programs P_F and P'_F such that they are equivalent if and only if F is not satisfiable. P_F has input/output variable x and auxiliary variable y, and it has the following form:

Initialization
P_1
$\vdots$
P_n
Adjustment
Q_1
$\vdots$
Q_m
Finalization

The input x, which can be positive, negative, or zero, is viewed as a binary number $x = \# x_n \cdots x_1 b$, where $b, x_1, \cdots, x_n$ are binary digits and $\#$ is some (possibly negative) finite string of binary digits. (*Convention*: in this proof, $\#$ represents any (possibly negative) finite string of binary digits whose exact composition is not important.) The construction is such that P_F and P'_F agree on all inputs with $b = 0$. They disagree on some input with $b = 1$ if and only if F is satisfiable. Thus, programs P_F and P'_F will not be equivalent if and only if F is satisfiable.

Before we write the codes for the different parts of P_F, we describe a routine $Z(k, l)$ which will be used many times. The parameters k and l are positive integers.

Code for $Z(k, l)$. Let $x = \# s_{k+l} \cdots s_{k+1} s_k \cdots s_1$, where $s_1, \cdots, s_{k+l}$ are binary digits. The code $Z(k, l)$ sets $s_{k+1}, \cdots, s_{k+l}$ to 0's without changing $s_1, \cdots, s_k$. We assume that $y = 0$ or it has the same sign as x at the beginning of $Z(k, l)$.

$y \leftarrow 2^{k+l} y$	coded: $y \leftarrow 2y; \cdots; y \leftarrow 2y$ $(k+l)$ times
	$y = \# \overbrace{0 \cdots 0}^{k+l}$
$y \leftarrow y + x$	$y = \# s_{k+l} \cdots s_{k+1} s_k \cdots s_1$
$y \leftarrow y/2^k$	coded: $y \leftarrow y/2; \cdots; y \leftarrow y/2$ (k times)
$y \leftarrow 2^k y$	$y = \# s_{k+l} \cdots s_{k+1} \overbrace{0 \cdots 0}^{k}$
$x \leftarrow x + y$	$x = \# \overbrace{s_{k+l-1} \cdots s_{k+1}}^{l-1} 0 s_k \cdots s_1$

$y \leftarrow 2y$ $\qquad y = \# \overbrace{s_{k+l-1} \cdots s_{k+1}}^{l-1} \overbrace{0 \cdots 0}^{k+1}$

$x \leftarrow x + y$ $\qquad x = \# \overbrace{s_{k+l-2} \cdots s_{k+1}}^{l-2} 00 s_k \cdots s_1$

$\vdots$

$y \leftarrow 2y$ $\qquad y = \# s_{k+1} \overbrace{0 \cdots 0}^{k+l-1}$

$x \leftarrow x + y$ $\qquad y = \# \overbrace{0 \cdots 0}^{l} s_k \cdots s_1$

We are now ready to write the codes for the different parts of P_F.

Code for Initialization.

$x \leftarrow 2^{3m} x$

At the end of Initialization,

$$x = \# x_n \cdots x_1 b d_m^3 d_m^2 d_m^1 \cdots d_1^3 d_1^2 d_1^1 = \# x_n \cdots x_1 b \overbrace{0 \cdots 0}^{3m}.$$

It will always be the case that at the beginning of code $P_k (1 \leqq k \leqq n)$, x has the form $x = \# x_{n-k+1} \cdots x_1 b d_m^3 d_m^2 d_m^1 \cdots d_1^3 d_1^2 d_1^1$ and y is either 0 or a number with the same sign as x. (Note that by convention, y is 0 at the beginning of the program since y is not an input variable.)

Code for P_k, $1 \leqq k \leqq n$.

$Z(3m + n - k + 2, 3m + n - k + 2)$ $\qquad x = \# \overbrace{0 \cdots 0}^{3m+n-k+2} x_{n-k+1} \cdots x_1 b d_m^3 \cdots d_1^1$

$y \leftarrow 2^{6m+2n+2} y$ $\qquad y = \# \overbrace{0 \cdots 0}^{6m+2n+2}$

$y \leftarrow y + x$

$y \leftarrow y / 2^{3m+n-k+1}$ $\qquad y = \# \overbrace{0 \cdots 0}^{3m+n-k+2} x_{n-k+1}$

(Let $C_{k_1}, \cdots, C_{k_r}$ be the clauses in which x_{n-k+1} appears, and assume $k_1 < \cdots < k_r$.)

$y \leftarrow 2^{3(k_1-1)} y$ $\qquad y = \# \overbrace{0 \cdots 0}^{3m+n-k+2} x_{n-k+1} \overbrace{0 \cdots 0}^{3(k_1-1)}$

$x \leftarrow x + y$ $\qquad d_{k_1}^2 d_{k_1}^1 \leftarrow d_{k_1}^2 d_{k_1}^1 + x_{n-k+1}$ with $x_{n-k+1}, \cdots, x_1, b, d_m^3, d_m^2, d_m^1, \cdots, d_{k_1}^3, d_{k_1-1}^3, d_{k_1-1}^2, d_{k_1-1}^1, \cdots, d_1^3, d_1^2, d_1^1$ unchanged.

$y \leftarrow 2^{3(k_2-k_1)}y$

$x \leftarrow x+y$

$\vdots$

$y \leftarrow 2^{3(k_r-k_{r-1})}y$

$x \leftarrow x+y$

$y \leftarrow 2^{3(m-k_r)+4}y$

$y \leftarrow y+x$ $\qquad y = \# b d_m^3 d_m^2 d_m^1 \cdots d_1^3 d_1^2 d_1^1$

$y \leftarrow y/2^{3m}$ $\qquad y = \# b$

$y \leftarrow 2^{3m+n-k+1}$ $\qquad y = \# b\overbrace{0 \cdots 0}^{3m+n-k+1}$

$x \leftarrow x+y$ $\qquad x = \# \underbrace{(x_{n-k+1}+b)}_{\bar{x}_{n-k+1} \text{ if } b=1} x_{n-k} \cdots x_1 b\, d_m^3 \cdots d_1^1$

$Z(3m+n-k+2, 3m+n-k+2)$

$y \leftarrow 2^{6m+2n+2}y$

$y \leftarrow y+x$

$y \leftarrow y/2^{3m+n-k+1}$ $\qquad y = \# \overbrace{0 \cdots 0}^{3m+n-k+2} \bar{x}_{n-k+1}$

(Let $C_{\bar{k}_1}, \cdots, C_{\bar{k}_s}$ be the clauses in which $\bar{x}_{n-k+1}$ appears, and assume $\bar{k}_1 < \cdots < \bar{k}_s$.)

$y \leftarrow 2^{3(\bar{k}_1-1)}y$

$x \leftarrow x+y$

$y \leftarrow 2^{3(\bar{k}_2-\bar{k}_1)}y$

$x \leftarrow x+y$

$\vdots$

$y \leftarrow 2^{3(\bar{k}_s-\bar{k}_{s-1})}y$

$x \leftarrow x+y$

Clearly, at the end of P_n, $x = \# b\, d_m^3\, d_m^2\, d_m^1 \cdots d_1^3\, d_1^2\, d_1^1$ where $d_k^3 = 0$ for $1 \leqq k \leqq m$. Moreover, if $b=1$ then $d_k^2\, d_k^1 > 0$ if and only if C_k is satisfied.

Code for Adjustment.

$Z(3m+1, 3m)$ $\qquad x = \# \overbrace{0 \cdots 0}^{3m} b\, d_m^3 \cdots d_1^1$

$y \leftarrow 2^{6m+1}y$

$y \leftarrow y+x$

$y \leftarrow y/2^{3m}$ $\qquad y = \# \overbrace{0 \cdots 0}^{3m} b$

$$\left.\begin{array}{l} x \leftarrow x+y \\ y \leftarrow 2y \\ x \leftarrow x+y \end{array}\right\} d_1^3\, d_1^2\, d_1^1 \leftarrow d_1^3\, d_1^2\, d_1^1 + bb$$

$$y \leftarrow 2^2 y$$

$$\left.\begin{array}{l} x \leftarrow x+y \\ y \leftarrow 2y \\ x \leftarrow x+y \end{array}\right\} d_2^3\, d_2^2\, d_2^1 \leftarrow d_2^3\, d_2^2\, d_2^1 + bb$$

$$y \leftarrow 2^2 y$$

$$\vdots$$

$$\left.\begin{array}{l} x \leftarrow x+y \\ y \leftarrow 2y \\ x \leftarrow x+y \end{array}\right\} d_m^3\, d_m^2\, d_m^1 \leftarrow d_m^3\, d_m^2\, d_m^1 + bb$$

$$x \leftarrow 2^m x$$

At the end of Adjustment,

$$x = \# d_m^3\, d_m^2\, d_m^1 \cdots d_1^3\, d_1^2\, d_1^1 \overbrace{0\cdots 0}^{m}$$

Moreover, $d_1^3 = d_2^3 = \cdots = d_m^3 = 1$ if and only if $b = 1$ and $F = C_1 C_2 \cdots C_m$ is satisfied.

Code for Q_k, $1 \leqq k \leqq m$. When Q_k is entered, x always has the form

$$x = \# d_{m-k+1}^3\, d_{m-k+1}^2\, d_{m-k+1}^1 \cdots d_1^3\, d_1^2\, d_1^1\, d_m^3\, d_{m-1}^3 \cdots d_{m-k+2}^3 \overbrace{0\cdots 0}^{m-k+1}$$

$$Z(m+3(m-k+1),\, m+3(m-k+1)) \qquad x = \# \overbrace{0\cdots 0}^{m+3(m-k+1)} d_{m-k+1}^3 \cdots d_1^1 d_m^3 \cdots d_{m-k+2}^3 \overbrace{0\cdots 0}^{m-k+1}$$

$$y \leftarrow 2^{2m+6(m-k+1)} y \qquad y = \# \overbrace{0\cdots 0}^{m+3(m-k+1)} d_{m-k+1}^3 \cdots d_1^1 d_m^3 \cdots d_{m-k+2}^3 \overbrace{0\cdots 0}^{m-k+1}$$

$$y \leftarrow y + x$$

$$y \leftarrow y/2^{m+3(m-k)+2} \qquad y = \# \overbrace{0\cdots 0}^{m+3(m-k+1)} d_{m-k+1}^3$$

$$x \leftarrow 2^{m-k} y$$

$$x \leftarrow x + y \qquad x = \# d_{m-k}^3 \cdots d_1^1\, d_m^3 \cdots d_{m-k+1}^3 \overbrace{0\cdots 0}^{m-k}$$

At the end of Q_m, $x = \# d_m^3\, d_{m-1}^3 \cdots d_1^3$ and $d_1^3 = d_2^3 = \cdots = d_m^3 = 1$ if and only if $b = 1$ and F is satisfied.

Code for Finalization.

$Z(m, m)$ $\qquad x = \#\overbrace{0 \cdots 0}^{m} d_m^3 d_{m-1}^3 \cdots d_1^3$

$y \leftarrow 2^{2m} y$

$y \leftarrow y + x$

$y \leftarrow y/2^{m-1}$ $\qquad y = \#\overbrace{0 \cdots 0}^{m} d_m^3$

$x \leftarrow x + y$

$x \leftarrow x/2^m$ $\qquad x = \# h$

At the end of Finalization, $x = \# h$, where $h = 0$ or 1. $h = 1$ if and only if $d_1^3 = d_2^3 = \cdots = d_m^3 = 1$, i.e., if and only if $b = 1$ and F is satisfied. It follows that P_F outputs an odd number for some input if and only if F is satisfiable.

Now let P'_F be the program obtained from P_F by adding the following instructions at the end of P_F:

$x \leftarrow x/2$

$x \leftarrow 2x$

Then P'_F is equivalent to P_F if and only if F is not satisfiable. One can easily check that the sum of the sizes (total number of instructions) of P_F and P'_F is some fixed polynomial in m and n (and, therefore, in the size of F). Hence, the construction of P_F and P'_F takes polynomial time in the size of F. Since the satisfiability problem for Boolean formulas in CNF with at most 3 literals per clause is NP-hard, the result follows. □

In Theorem 3, the instruction $x \leftarrow x + y$ can be replaced by $x \leftarrow x - y$:

COROLLARY 5. *The equivalence problem for* $\{x \leftarrow 2x, x \leftarrow x/2, x \leftarrow x - y\}$*-programs with one input variable (which is also the output variable) and one auxiliary variable is* NP*-hard.*

Proof. Replace the occurrences of instructions $x \leftarrow x + y$ and $y \leftarrow y + x$ in P_F by $x \leftarrow x - y$ and $y \leftarrow y - x$, respectively. □

When $x \leftarrow 0$, $x \leftarrow x + y$, and $x \leftarrow x - y$ are in the instruction set, we have

THEOREM 4. *The zero-equivalence problem for* $\{x \leftarrow 0, x \leftarrow 2x, x \leftarrow x/2, x \leftarrow x + y, x \leftarrow x - y\}$*-programs with one input variable and one auxiliary variable is* NP*-hard. The result holds even if the instructions* $x \leftarrow 0$ *and* $x \leftarrow x - y$ *are used exactly once in the programs.*

Proof. Let $\hat{P}_F$ be the program obtained from P_F (of Theorem 3) by adding the following instructions at the end:

$y \leftarrow 0$

$y \leftarrow y + x$ $\qquad y = x$

$y \leftarrow y/2$

$y \leftarrow 2y$

$x \leftarrow x - y$

Then $\hat{P}_F$ outputs 0 for all inputs if and only if F is not satisfiable. □

COROLLARY 6. *The zero-equivalence problem for* $\{x \leftarrow 2x,\ x \leftarrow x/2,\ x \leftarrow x-y,\ x \leftarrow y\}$-*programs with one input variable and one auxiliary variable is* NP-*hard. The result holds even if the instruction* $x \leftarrow y$ *is used exactly once in the programs.*

Proof. Replace the instructions $y \leftarrow 0$; $y \leftarrow y+x$ in the proof of Theorem 4 by $y \leftarrow x$. Then the results follows from Corollary 5. □

COROLLARY 7. *The zero-equivalence problem for* $\{x \leftarrow 0,\ x \leftarrow x/2,\ x \leftarrow x-y\}$-*programs with one input variable and two auxiliary variables is* NP-*hard.*

Proof. This follows from Theorem 4 and the observation that $x \leftarrow 2x$ and $x \leftarrow x+y$ can be coded as $z \leftarrow 0$; $z \leftarrow z-x$; $x \leftarrow x-z$ and $z \leftarrow 0$; $z \leftarrow z-y$; $x \leftarrow x-z$, respectively, z a new variable. □

Corollaries 6 and 7 may be the best possible results since we can prove the following theorem.

THEOREM 5. *The zero-equivalence problem for* $\{x \leftarrow 0,\ x \leftarrow c,\ x \leftarrow -c,\ x \leftarrow cx,\ x \leftarrow x/c,\ x \leftarrow x+c,\ x \leftarrow x-c,\ x \leftarrow x-y\}$-*programs with at most two variables* (*both may be input variables*) *is decidable in polynomial time* (*c is any positive integer constant*).

Proof. Let P be a program with r instructions, and let $d = \max\{c\text{'s appearing in } P\}+1$. Let x and y be the variables of P. We consider two cases.

Case 1. P has one input variable. Let d^{3r} be the input. Let $0 \leqq k \leqq r$. Then it is easy to show (by induction on k) that the following are true at the end of k instructions:

(1) Exactly one of (a) or (b) below holds for variable z (z is either x or y):

(a) $|\text{value}(z)| \leqq d^k$ and value (z) is independent of the input.

(b) $|\text{value}(z)| \geqq d^{3r-k}$.

(2) If $|\text{value}(x)| \geqq d^{3r-k}$ and $|\text{value}(y)| \geqq d^{3r-k}$, then value (x) and value (y) have *opposite* signs.

It follows from (1) and (2) that P computes the zero-function if and only if P outputs 0 on input d^{3r}.

Case 2. P has two input variables. As in Case 1, P computes the zero-function if and only if P outputs 0 on inputs $(0, d^{3r})$ and $(d^{3r}, 0)$. □

If the instruction $x \leftarrow x-y$ is replaced by $x \leftarrow x+y$ in Theorem 5, we can prove a stronger result.

THEOREM 6. *The zero-equivalence problem for* $\{x \leftarrow 0,\ x \leftarrow c,\ x \leftarrow -c,\ x \leftarrow cx,\ x \leftarrow x/c,\ x \leftarrow x+c,\ x \leftarrow x-c,\ x \leftarrow x+y,\ x \leftarrow y\}$-*programs* (*with no restriction on the number of input and auxiliary variables*) *is decidable in polynomial time.*

Proof. Let P be a program with r instructions, and let $d = \max\{c\text{'s appearing in } P\}+1$. Then P computes the zero-function if and only if P outputs 0 when all the input variables are set to d^r. □

For one-variable programs containing only instructions of the form $x \leftarrow 0$, $x \leftarrow 1$, $x \leftarrow 2x$ and $x \leftarrow x/2$, equivalence is decidable in polynomial time:

PROPOSITION 2. *The equivalence problem for one-variable* $\{x \leftarrow 0,\ x \leftarrow 1,\ x \leftarrow 2x,\ x \leftarrow x/2\}$-*programs is decidable in polynomial time.*

Proof. This is obvious since any program P can be reduced (in polynomial time) to one of the following forms (a, k and m are some nonnegative integer constants):

(1) $x \leftarrow a$

(2) $x \leftarrow 2^k x$

(3) $x \leftarrow x/2^k$

(4) $x \leftarrow x/2^k$; $x \leftarrow 2^m x$ □

When x is restricted to nonnegative integer inputs, we can prove a stronger result:

THEOREM 7. *The equivalence problem for one-variable* $\{x \leftarrow 0, x \leftarrow x+1, x \leftarrow 2x, x \leftarrow x/2\}$*-programs over nonnegative integer inputs is decidable in polynomial time.*

Proof. Any program P containing only instructions $x \leftarrow 0$, $x \leftarrow x+1$, $x \leftarrow 2x$, $x \leftarrow x/2$ can be reduced (in polynomial time) to one of the following forms (a, b, k, and m are nonnegative integers):

(1) $x \leftarrow a$
(2) $x \leftarrow 2^k x + a$
(3) $x \leftarrow x/2^k$
(4) $x \leftarrow x + a$; $x \leftarrow x/2^k$
(5) $x \leftarrow x/2^k$; $x \leftarrow 2^m x + b$
(6) $x \leftarrow x + a$; $x \leftarrow x/2^k$; $x \leftarrow 2^m x + b$

The reduction can be accomplished using the following transformations:

(a) $x \leftarrow 2^k x + a$; $x \leftarrow 2^m x + b$ reduces to $x \leftarrow 2^{k+m} x + (2^m a + b)$
(b) $x \leftarrow x/2^k$; $x \leftarrow x/2^m$ reduces to $x \leftarrow x/2^{k+m}$
(c) $x \leftarrow 2^k x + a$; $x \leftarrow x/2^m$ reduces to $x \leftarrow 2^{k-m} x + a/2^m$ if $k \geqq m$
(d) $x \leftarrow 2^k x + a$; $x \leftarrow x/2^m$ reduces to $x \leftarrow x + a/2^k$; $x \leftarrow x/2^{m-k}$ if $k < m$
(e) $x \leftarrow x/2^k$; $x \leftarrow x + a$ reduces to $x \leftarrow x + 2^k a$; $x \leftarrow x/2^k$ □

Remark. Again, all the results in this section remain valid when the inputs are restricted to nonnegative integers.

REFERENCES

[1] I. BOROSH AND L. B. TREYBIG, *Bounds on positive integral solutions of linear Diophantine equations*, Proc. Amer. Math. Soc., 55 (1976), pp. 299–304.

[2] R. L. CONSTABLE, H. B. HUNT AND S. SAHNI, *On the computational complexity of scheme equivalence*, Proc. 8th Annual Princeton Conference on Information Sciences and Systems, 1974, pp. 15–20.

[3] S. A. COOK, *The complexity of theorem-proving procedures*, Proc. 3rd Annual ACM Symposium on the Theory of Computing, 1971, pp. 151–158.

[4] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability—A Guide to the Theory of* NP-*Completeness*, W. H. Freeman, San Francisco, 1979.

[5] E. M. GURARI AND O. H. IBARRA, *An* NP-*complete number-theoretic problem*, J. Assoc. Comput. Mach. 26 (1979), pp. 567–581.

[6] O. H. IBARRA AND B. S. LEININGER, *On the simplification and equivalence problems for straight-line programs*, submitted to J. Assoc. Comput. Mach. (Available as Univ. of Minnesota, Dept. of Computer Science Technical Report 79-21, September, 1979.)

[7] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum, New York, pp. 85–104.

[8] D. E. KNUTH, *The Art of Computer Programming. Vol. 2—Seminumerical Algorithms*, Addison-Wesley, Reading, MA, 1969.

[9] S. SAHNI, *Computationally related problems*, this Journal, 3 (1974), pp. 262–279.

SIAM J. COMPUT.
Vol. 11, No. 1, February 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1101-0003 $01.00/0

A FAST ALGORITHM FOR THE EUCLIDEAN TRAVELING SALESMAN PROBLEM, OPTIMAL WITH PROBABILITY ONE*

J. H. HALTON† AND R. TERADA‡

Abstract. This paper presents an algorithm for the traveling salesman problem in k-dimensional Euclidean space. For n points independently uniformly distributed in a set $\mathbb{E}$, we show that, for any choice of a function σ of n increasing to infinity with n more slowly than n, we can adjust the algorithm so that, *in probability*, the time taken by the algorithm will be of order less than that of $n\sigma(n)$ as $n \to \infty$. The algorithm puts the n points in a cyclic order, and we also show that, *with probability one*, the length of the corresponding *tour* (that is, the sum of the n distances between adjacent points in the order given) will be asymptotic to the *minimal tour length* as $n \to \infty$. The latter is known (also with probability one) to be asymptotic to $\beta_k v(\mathbb{E})^p n^q$, where β_k is a constant depending only on the dimension k, $v(\mathbb{E})$ is the volume of the set $\mathbb{E}$, $p = 1/k$, and $q = 1 - p$. Our result is *stronger*, and the algorithm is *faster*, than any other we have been able to find in the literature.

Key words. traveling salesman, probabilistic algorithm, operations research, optimization

1. Introduction. Consider a set $\mathbb{A}$ of n points in the k-dimensional Euclidean space $\mathbb{R}^k$ (with the usual topology). A *tour* of $\mathbb{A}$ is defined to be a *cyclically ordered set containing* $\mathbb{A}$ (that is, a set $\mathbb{T}$ such that $\mathbb{A} \subseteq \mathbb{T} \subseteq \mathbb{R}^k$, with an *ordering relation* τ such that for any finite subset of $\mathbb{T}$, e.g., $\{A, B, C, D, E, F\}$, a unique, complete cyclic order exists, e.g., $\{A\tau C, C\tau B, B\tau F, F\tau D, D\tau E, E\tau A\}$, which we shall abbreviate to $A\tau C\tau B\tau F\tau D\tau E\tau A$, or just to the string of point-symbols $ACBFDE$). Note that a path, which may be intuitively viewed as a tour which crosses itself, can always be described as a cyclically ordered set by removing the single point of intersection from one of the branches. Similarly, a path which is traced more than once may be cyclically ordered by suitably interlacing the points of each passage. If a *metric* d is defined in $\mathbb{R}^k$ (not necessarily consistent with the topology of $\mathbb{R}^k$), such a tour will have a (possibly infinite) *length* $l(\mathbb{T}, \tau)$ (defined as the supremum of the sum of the metric distances between successive points in any finite subcycle in the tour, e.g., $d(A, C) + d(C, B) + d(B, F) + d(F, D) + d(D, E) + d(E, A)$). Since all tour-lengths are nonnegative, they are bounded below by zero; so that there will be an infimum for the lengths of all tours of a given set $\mathbb{A}$. We denote this by $l(\mathbb{A})$.

Given a tour $(\mathbb{T}, \tau)$ of $\mathbb{A}$, it will uniquely determine a cyclic ordering of $\mathbb{A}$ (since $\mathbb{A}$ is a finite subset of $\mathbb{T}$), so that $(\mathbb{A}, \tau)$ is itself a tour of $\mathbb{A}$. If we label the points of $\mathbb{A}$ in such a manner that the tour $(\mathbb{T}, \tau)$ imposes the cyclic order $A_0 \tau A_1 \tau A_2 \tau \cdots \tau A_n = A_0$, then the triangle inequality for the metric d ensures that the length $l(\mathbb{A}, \tau) = \sum_{i=1}^n d(A_{i-1}, A_i)$, and it is clear that this cannot exceed the length $l(\mathbb{T}, \tau)$. It follows that the infimum of the lengths of all tours of $\mathbb{A}$ is the same as the infimum of the lengths of all tours $(\mathbb{A}, \tau)$: this is the infimum of $l(\mathbb{A}, \tau)$ over all $(n-1)!$ cyclic orderings of $\mathbb{A}$. Since this last infimum is taken over a *finite* collection of lengths, it is certainly *attained*. We thus see that there will always exist at least one cyclic ordering of $\mathbb{A}$, which we may denote by π, such that $l(\mathbb{A}, \pi) = \inf_\tau l(\mathbb{A}, \tau) = l(\mathbb{A})$. Such a tour will be termed a *minimal tour* of $\mathbb{A}$. The search for minimal tour-lengths and for minimal tours in $\mathbb{R}^k$ is called the *traveling salesman problem* (k-TSP).

---

* Received by the editors January 23, 1979, and in revised form December 12, 1980.
† Computer Sciences Department, University of Wisconsin, Madison, Wisconsin 53706.
‡ Instituto de Matemática e Estatística, Universidade de São Paulo, CEP 05508, São Paulo, Brazil.

In this paper, we shall limit ourselves to the problems in which the metric d is the *Euclidean* (or *Pythagorean* or l^2) metric, for which $d(\mathbf{x}, \mathbf{y}) = \|\mathbf{x}-\mathbf{y}\|_2 = (\sum_{i=1}^{k} (x_i - y_i)^2)^{1/2}$. This is called the *Euclidean traveling salesman problem* (k-ETSP).

The 2-TSP has been shown to be NP-hard (see Garey, Graham, and Johnson [1976], Papadimitriou [1977], Garey and Johnson [1979], and this strongly suggests that there is no polynomial-time algorithm for obtaining the exact solution of this problem. By natural extension, we believe that the same is true for the k-TSP with $k \geqq 3$. Certainly, no such algorithm has been found so far.

On the other hand, there has been some research on fast heuristic methods for the solution of the 2-TSP: for example, computer programs to find near-optimal solutions for sets of up to 300 points in an acceptable amount of time have been described by Krolak, Felts, and Marble [1970], and by Lin and Kernighan [1973]. Their programs seem to give satisfactory results; but no rigorous analyses of the algorithms are available.

Bellman [1962] and Held and Karp [1962] describe a dynamic programming algorithm for the k-TSP, which determines an exactly minimal tour of a set of s points in a time

$$t_s = 2A(s-1)[2^{s-3}(s-2)+1] \quad \text{for } s \geqq 1, \tag{1.1}$$

where A is a computer-dependent constant (roughly, half the time needed for an addition). We subsume the use of this algorithm, which we shall refer to as Algorithm C, in constructing our own, and the estimate (1.1) yields our timing estimate in Theorem 2. (Should a faster algorithm than the above become available, it will lead to an increase in the speed of ours also.)

Since many important computational problems are known to be soluble only by exponential-time algorithms, interest has recently shifted to *probabilistic algorithms*, which (with a high degree of probability) will yield accurate answers in acceptably short times, but for which (with very low probability) either (i) accurate answers may take very long times to obtain, or (ii) answers obtained may not be accurate.

Beardwood, Halton, and Hammersley [1959] studied the statistical properties of the solutions of k-ETSP. In particular, they showed that, if $\mathbb{E}$ is a bounded, Lebesgue-measurable subset of $\mathbb{R}^k$, with k-dimensional Lebesgue measure (or volume) $v(\mathbb{E}) > 0$, and if $\mathbf{P}$ is an infinite sequence of points independently uniformly distributed in $\mathbb{E}$, with $\mathbf{P}^n$ denoting the set consisting of the first n points of $\mathbf{P}$, then there exists a constant β_k, not dependent on $\mathbb{E}$ or $\mathbf{P}$, such that, with probability one,

$$l(\mathbf{P}^n) \sim \beta_k v(\mathbb{E})^p n^q \quad \text{as } n \to \infty, \tag{1.2}$$

where $p = 1/k$ and $q = 1-p$. They also showed that, if the points of $\mathbf{P}$ are instead independently distributed in $\mathbb{E}$ with any fixed probability distribution and if the absolutely continuous component of this distribution is represented by a probability-density function ρ (whatever the discrete and singular components of the distribution may be), then, again with probability one,

$$l(\mathbf{P}^n) \sim \beta_k n^q \int_{\mathbb{E}} \rho^q \, dv \quad \text{as } n \to \infty. \tag{1.3}$$

When the density is constant, $\rho = 1/v(\mathbb{E})$, and (1.3) reverts to (1.2). We take our point of departure in the above paper, which we shall refer to as BHH. In the course of reviewing the proofs of various results in BHH, we found that the proof of their Lemma 7 had to be modified somewhat (the statement of the lemma remains correct).

This is discussed in Appendix II of Halton and Terada [1978], hereinafter referred to as HT. The present paper is a revised version of HT.

Karp [1977] has described a probabilistic algorithm for the 2-TSP: it is a recursive algorithm, for which he claims an *expected* running-time of the order of $n(\log n)^2$ and an *expected* resulting tour-length asymptotic to $l(\mathbb{A})$ as $n \to \infty$. It will be seen below that the algorithm presented here is proved *in probability* to run in a time which is $o[n\sigma(n)]$ for an arbitrarily chosen function σ satisfying

$$\sigma(n) \to \infty \quad \text{and} \quad \frac{\sigma(n)}{n} \to 0 \quad \text{as } n \to \infty \tag{1.4}$$

(see Theorem 2), and it is also proved that the resulting tour-length is asymptotic to $l(\mathbb{A})$ *with probability one* (see Theorem 3). Some questions and discussion of Karp's paper are given in Appendix III of HT; in any case, our results are stronger. We are not aware of the existence of any other algorithm comparable to ours.

2. The main algorithm. Given a set $\mathbb{A}$ of n points in $\mathbb{R}^k$, our algorithm covers it with a cubic lattice of cells, solves the k-ETSP in each cell by Algorithm C, and prescribes how these partial tours should be connected cell-to-cell to form a tour of $\mathbb{A}$. The all-important lattice is defined in such a way that the tour generated has the desirable properties of speed and accuracy claimed in Theorems 2 and 3 below. These are both *statistical* and *asymptotic* properties, derived by embedding the given problem in a large class of similar problems in two ways. First, the set $\mathbb{A}$ is viewed as the first n points of an infinite sequence of points. Secondly, the points of the sequence are assumed to be independently uniformly distributed at random in a set $\mathbb{E}$ having the properties:

(a) $\mathbb{E}$ is a Lebesgue-measurable set in $\mathbb{R}^k$, with positive volume $v(\mathbb{E})$.

(b) $\mathbb{E}$ is bounded in $\mathbb{R}^k$: we can find a semi-open hypercube (more briefly, a *cube*)

$$\mathbb{C} = \{\mathbf{x} = (x_1, x_2, \cdots, x_k) \in \mathbb{R}^k : b_i \leqq x_i < b_i + \lambda \text{ for } i = 1, 2, \cdots, k\}, \tag{2.1}$$

such that $\mathbb{E} \subseteq \mathbb{C}$ and $\mathbb{C}$ has sides of length λ.

(c) If the cube $\mathbb{C}$ defined in (b) is divided into a cubic lattice of $M = m^k$ similarly semi-open hypercubic *cells* $\mathbb{C}_j$ $(j = 1, 2, \cdots, M)$, each with sides of length λ/m, and if N_2 of these cells contain points both of $\mathbb{E}$ and of its complement $\mathbb{E}^c$, then the boundary of $\mathbb{E}$ is such that, as $M \to \infty$, $N_2 = O(M^q)$, where $q = 1 - 1/k$; thus, in particular, $N_2/M \to 0$. (We see that this property holds whenever the $(k-1)$-dimensional Lebesgue measure of the boundary of $\mathbb{E}$ is finite.)

It is clear that the given set $\mathbb{A} \subseteq \mathbb{E} \subseteq \mathbb{C}$; but, beyond this, the choice of $\mathbb{E}$ and $\mathbb{C}$ is free and will depend on our knowledge (or hunch) of the class of problems of which $\mathbb{A}$ is considered to be a sample. In the absence of more precise information, we may take $\mathbb{E} = \mathbb{C}$ and $\mathbb{C}$ to be the smallest cube (2.1) containing $\mathbb{A}$. The determination of $\mathbb{C}$ requires time of the order of kn, which is negligible, in view of Theorem 2.

Underlying the specification of the algorithm is the choice of a function σ of n, satisfying (1.4) but otherwise at our disposal. Because of Theorem 2 and Karp's claim of an expected running time of $O[n(\log n)^2]$, we will focus our attention on $\sigma(n)$ increasing with n no faster than $(\log n)^2$. If $p = 1/k$ and $\lceil \cdot \rceil$ denotes the "roof" function (the least upper bound among the integers), we can define the even integer

$$m = 2\left\lceil \frac{\lambda}{2}\left(\frac{2n}{v(\mathbb{E}) \log \sigma(n)}\right)^p \right\rceil. \tag{2.2}$$

From this, we can derive that, by (1.4),

$$M = m^k \sim \frac{2n\lambda^k}{v(\mathbb{E}) \log \sigma(n)} \quad \text{as } n \to \infty. \tag{2.3}$$

We also observe that

$$m \to \infty, \quad M \to \infty, \quad \frac{M}{n} \to 0 \quad \text{as } n \to \infty. \tag{2.4}$$

ALGORITHM A. [A1] Given a set $\mathbb{A}$ of n points in $\mathbb{R}^k$, choose the semi-open hypercube $\mathbb{C}$ defined as in (2.1), the set $\mathbb{E}$ contained in $\mathbb{C}$ and containing $\mathbb{A}$, having properties (a), (b), and (c) above, and a function σ satisfying (1.4). Hence, determine the even integer m, by (2.2), and $M = m^k$.

[A2] Divide each side of $\mathbb{C}$ into m equal parts, thus creating a cubic lattice of M semi-open hypercubic cells $\mathbb{C}_j$, with $j = 1, 2, \cdots, M$.

[A3] In each cell $\mathbb{C}_j$, find by Algorithm C a minimal tour of $\mathbb{A}\mathbb{C}_j$ (the intersection of $\mathbb{A}$ and $\mathbb{C}_j$, i.e., the set of points of $\mathbb{A}$ falling in $\mathbb{C}_j$). The result is a cyclic ordering of the points of $\mathbb{A}\mathbb{C}_j$ which may be written as a string of point-symbols

$$\mathcal{S}_j = A_1^{(j)} A_2^{(j)} \cdots A_{n_j}^{(j)}, \quad \text{where } \mathbb{A}\mathbb{C}_j = \{A_1^{(j)}, A_2^{(j)}, \cdots, A_{n_j}^{(j)}\}, \tag{2.5}$$

and n_j is the number of points of $\mathbb{A}$ in $\mathbb{C}_j$. We note that

$$\sum_{j=1}^{M} n_j = n. \tag{2.6}$$

Of course, if $\mathbb{A}\mathbb{C}_j = \varnothing$ for some j the corresponding string $\mathcal{S}_j$ will be null.

[A4] Using Algorithm B (defined below), determine a cyclic ordering of the M cells, which may, by suitable renumbering, be written as

$$\mathcal{B} = \mathbb{C}_1 \mathbb{C}_2 \cdots \mathbb{C}_M. \tag{2.7}$$

[A5] Applying the ordering (2.7) to the strings $\mathcal{S}_j$, form a string

$$\mathcal{S} = \mathcal{S}_1 \mathcal{S}_2 \cdots \mathcal{S}_M. \tag{2.8}$$

This represents a cyclic ordering of all the points of $\mathbb{A}$ (see Theorem 1 below), to which corresponds a tour, $(\mathbb{A}, \omega)$, say, of length

$$l_0(\mathbb{A}) = \sum_{j=1}^{M} \sum_{i=1}^{n_j} d(A_{i-1}^{(j)}, A_i^{(j)}), \tag{2.9}$$

where $A_0^{(j)} = A_{n_{j-1}}^{(j-1)}$ and $A_{n_0}^{(0)} = A_{n_M}^{(M)}$.

3. The cell-tour algorithm. The following algorithm obtains the ordering (2.7) of the cells $\mathbb{C}_j$ in a time of the order of M. Denote the set $\{0, 1, 2, \cdots, m-1\}$ by $\mathbb{L}$ and define a lattice of vectors $\mathbf{a} = (a_1, a_2, \cdots, a_k)$ with each $a_i \in \mathbb{L}$. Then it is easily seen that there is a one-to-one correspondence between the M vectors $\mathbf{a}$ and the M cells $\mathbb{C}_j$, defined by

$$\mathbb{C}(\mathbf{a}) = \left\{ \mathbf{x} \in \mathbb{R}^k : b_i + \frac{\lambda}{m} a_i \leqq x_i < b_i + \frac{\lambda}{m}(a_i + 1) \text{ for } i = 1, 2, \cdots, k \right\}. \tag{3.1}$$

Thus, an ordering of the cells will correspond uniquely to an ordering of the lattice vectors $\mathbf{a}$. We write $\mathbf{e}_i$ for the unit vector in the ith coordinate direction, and we

associate with each $\mathbf{a}$ the numbers

$$r_i = r_i(\mathbf{a}) = (-1)^{1+a_1+a_2+\cdots+a_{i-1}} \tag{3.2}$$

for $i = 2, 3, \cdots, k$. We note that the r_i take the values ± 1 only, and that, for any $\mathbf{a}$, $a_i + r_i \in \mathbb{L}$, unless either $a_i = 0$ and $r_i = -1$ or $a_i = m - 1$ and $r_i = +1$. Therefore, for any $\mathbf{a}$, there is at most one value of t such that

$$\begin{gathered} a_i + r_i \notin \mathbb{L} \quad \text{for } i = k, k-1, \cdots, t+1, \\ a_t + r_t \in \mathbb{L}, \quad \text{and} \quad t \geqq 3. \end{gathered} \tag{3.3}$$

ALGORITHM B. [B1] If there exists an index t satisfying (3.3), then the algorithm identifies the successor of $\mathbf{a}$ as the vector

$$\mathbf{a}' = \mathbf{a} + r_t \mathbf{e}_t, \tag{3.4}$$

that is, the vector with $a_i' = a_i$ for all $i \neq t$ and with $a_t' = a_t + r_t$.

[B2] If (3.3) does not hold for any t, then the successor of $\mathbf{a}$ is determined as follows:

(i) If $a_1 = 1$ and $a_2 = 0$, or if $a_1 > 1$ and a_2 is even, $\mathbf{a}' = \mathbf{a} - \mathbf{e}_1$.
(ii) If $a_1 = 0$ and $a_2 = m - 1$, or if $0 < a_1 < m - 1$ and a_2 is odd, $\mathbf{a}' = \mathbf{a} + \mathbf{e}_1$.
(iii) If $a_1 = 1$, $a_2 \neq 0$, and a_2 is even, or if $a_1 = m - 1$ and a_2 is odd, $\mathbf{a}' = \mathbf{a} - \mathbf{e}_2$.
(iv) If $a_1 = 0$ and $a_2 < m - 1$, $\mathbf{a}' = \mathbf{a} + \mathbf{e}_2$.

In order to apply Algorithms A and B, we need to show that (1) the algorithms do indeed generate a uniquely-defined tour of $\mathbb{A}$, (2) the algorithms are fast, and (3) the tour produced is minimal, or nearly so. These assertions are the burden of Theorems 1, 2, and 3, respectively.

4. The algorithms yield a tour.

THEOREM 1. *Algorithms* A *and* B *define a tour of the set* $\mathbb{A}$. *The length of this tour is less than* $\sum_{j=1}^{M} l(\mathbb{A}\mathbb{C}_j) + \lambda M^a \sqrt{k+3}$.

Proof. (i) It is clear from (2.1) and (3.1) that

$$\mathbb{C} = \bigcup_{j=1}^{M} \mathbb{C}_j \quad \text{and} \quad \text{all } \mathbb{C}_j \text{ are disjoint.} \tag{4.1}$$

Since $\mathbb{A} \subseteq \mathbb{C}$, it follows that each point of $\mathbb{A}$ occurs in one and only one of the $\mathbb{C}_j$, and so is mentioned in exactly one of the strings $\mathcal{S}_j$ generated by Algorithm C, in step [A3]. Therefore, if Algorithm B does indeed yield a cyclic ordering of all M cells $\mathbb{C}_j$, as is asserted in step [A4], and if the corresponding strings $\mathcal{S}_j$ are combined, as in step [A5] and (2.8), into a final string $\mathcal{S}$; then this string will mention each point of $\mathbb{A}$ exactly once, and so will define a tour of $\mathbb{A}$.

(ii) In Algorithm B, either step [B1] or step [B2] will be executed in finding the successor of any vector in the lattice $\mathbb{L}^k$, and the choice is always well defined. If step [B2] is executed, then it is easily verified that every possible combination of a_1 and a_2 in $\mathbb{L}^2$ occurs in exactly one of the cases (i)–(iv) of [B2]. It is also clear that, in every case,

$$\text{if } \mathbf{a} \in \mathbb{L}, \text{ then } \mathbf{a}' \in \mathbb{L} \text{ and } \mathbf{a}' = \mathbf{a} \pm \mathbf{e}_t \text{ for some } t, \tag{4.2}$$

and the corresponding cells $\mathbb{C}(\mathbf{a})$ and $\mathbb{C}(\mathbf{a}')$ meet in a face (the face defined by $x_t = b_t + (\lambda/2m)(a_t + a_t' + 1)$: see (3.1)); that is, they are adjacent. Thus, any point of $\mathbb{C}(\mathbf{a})$ may be joined to any point of $\mathbb{C}(\mathbf{a}')$ by a chord of length less than $(\lambda/m)\sqrt{k+3}$ (since two adjacent cubes form a rectangular *brick* with $(k-1)$ sides of length λ/m and one of length $2\lambda/m$, whose *diameter* is $(\lambda/m)\,[(k-1)(1)^2 + 1(2)^2]^{1/2}$). We have

demonstrated that every cell has a well-defined successor cell to which it is adjacent. It remains to be shown that this relationship defines a single cyclic ordering of the lattice $\mathbb{L}^k$. We proceed inductively.

(iii) First, let $k=2$. Then (3.3) is impossible, and [B2] is always executed. The rules of succession embodied in cases (i)–(iv) of [B2] generate a tour, which can be described as follows: Begin at $(0,0)$; by case (iv), move in the $+\mathbf{e}_2$ direction until $(0, m-1)$ is reached; by case (ii), move in the $+\mathbf{e}_1$ direction until $(m-1, m-1)$ is reached; thereafter, if a_2 is even, we move in the direction of $-\mathbf{e}_1$ from $(m-1, a_2)$ to $(1, a_2)$ (or to $(0,0)$, when $a_2=0$) (this is case (i)), and if a_2 is odd, we move in the direction of $+\mathbf{e}_1$ from $(1, a_2)$ to $(m-1, a_2)$ (this is case (ii)); whenever the end of a segment parallel to the first axis is reached, the tour descends to the next one by moving in the $-\mathbf{e}_2$ direction from $(1, a_2)$ to $(1, a_2-1)$ or from $(m-1, a_2)$ to $(m-1, a_2-1)$. Because m is *even*, what we have described is indeed a tour of $\mathbb{L}^2$. (If m were to be *odd*, the point $(1, m-1)$ would be the successor of both $(0, m-1)$ and $(2, m-1)$, while $(m-1, m-1)$ would have no predecessor, and the algorithm would not yield a tour.) Figs. 1 and 2 illustrate these concepts for the cases of $m=8$ and 5, respectively.

Now, consider the application of Algorithm B to $\mathbb{L}^k$, and suppose that the algorithm has already been shown to generate a tour $\bar{\mathcal{T}}$ of $\mathbb{L}^{k-1}$. Denote the vector, whose first $(k-1)$ coordinates are the same as those of $\mathbf{a}$, by $\bar{\mathbf{a}} = (a_1, a_2, \cdots, a_{k-1})$. Then we see that, if $a_k + r_k \in \mathbb{L}$, by (3.3) the successor of $\mathbf{a}$ is $\mathbf{a} + r_k \mathbf{e}_k$; that is, the path generated by Algorithm B moves parallel to the kth axis, in the $r_k \mathbf{e}_k$ direction. Indeed, since r_k depends only on the coordinates of $\bar{\mathbf{a}}$ (which do not change when the path moves parallel to $\mathbf{e}_k$), we deduce that, when $r_k = +1$, the path crosses the cube $\mathbb{L}^k$ from $(\bar{\mathbf{a}}, 0)$ to $(\bar{\mathbf{a}}, m-1)$ and, when $r_k = -1$, the path crosses $\mathbb{L}^k$ from $(\bar{\mathbf{a}}, m-1)$ to $(\bar{\mathbf{a}}, 0)$. On reaching the end of such a segment parallel to the kth axis, we find that $a_k + r_k \notin \mathbb{L}$, so that (3.3) cannot hold for $t=k$. On perusal of [B1] for $t<k$ and of [B2], we see that the rules of succession in $\mathbb{L}^k$ are identical with those in the tour $\bar{\mathcal{T}}$ of $\mathbb{L}^{k-1}$. Observing further that, if $\mathbf{a}' - \mathbf{a}$ is perpendicular to $\mathbf{e}_k$, then r_k changes sign (since just one of $a_1, a_2, \cdots, a_{k-1}$ changes by ± 1), we can infer that the new $a_k + r_k \in \mathbb{L}$ and the path forthwith proceeds to cross $\mathbb{L}^k$ again in the reversed direction $r_k \mathbf{e}_k$.

Summing up, we see that, if a tour congruent to $\bar{\mathcal{T}}$ is drawn on each of the faces $a_k = 0$ and $a_k = m-1$ of $\mathbb{L}^k$ perpendicular to $\mathbf{e}_k$, then the path generated by Algorithm B in $\mathbb{L}^k$ zig-zags alternately between the two face tours, passing from a "zig" whose

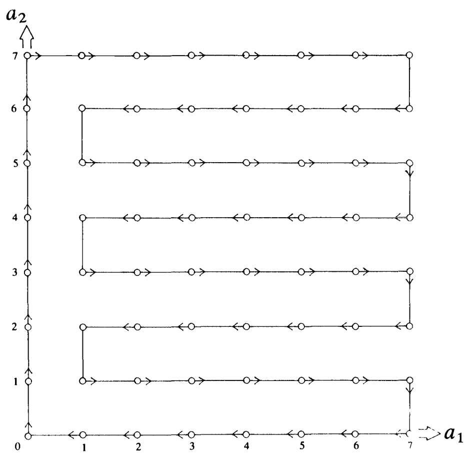


FIG. 1. *Tour of* $\mathbb{L}^2$ *by* [B2] *for the case when* $m=8$ (even).

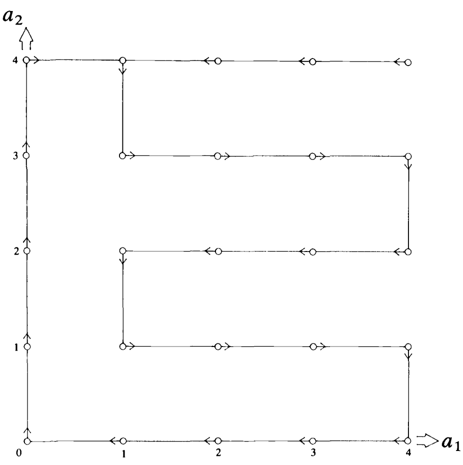


FIG. 2. *Path generated by* [B2] *for the (forbidden) case when* $m=5$ (odd).

first $(k-1)$ coordinates are given by $\bar{\mathbf{a}}$ to a "zag" whose first $(k-1)$ coordinates are given by the successor of $\bar{\mathbf{a}}$ in the tour $\bar{\mathcal{S}}$. Since $\bar{\mathcal{S}}$ passes through every point of $\mathbb{L}^{k-1}$, the path passes through every point of $\mathbb{L}^k$; and since the number of segments parallel to $\mathbf{e}_k$ equals the number of points in $\mathbb{L}^{k-1}$, namely m^{k-1}, which is even (because m is even), it follows that the number of "zigs" equals the number of "zags", and the path defined by Algorithm B in k dimensions is closed, and therefore is also a tour.

The form of the inductive step is illustrated in Fig. 3 for the case of $k=3$ and $m=6$. The two extreme tours in two dimensions, congruent to $\bar{\mathcal{S}}$, are seen as alternating

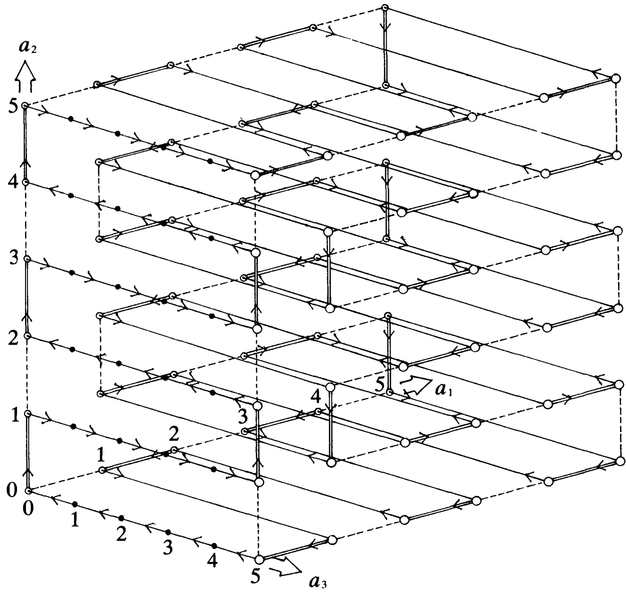


FIG. 3. *Tour of* $\mathbb{L}^3$ *generated by Algorithm* B. *Follow the arrows on single and double line-segments. This illustrates the inductive process described in part* (iii) *of the proof of Theorem* 1.

double and dotted line-segments. The "zigs" and "zags" parallel to the third axis are single lines (most of the interior points of $\mathbb{L}^3$ are omitted to make the path easier to see).

(iv) Having shown that Algorithm B does generate a tour of $\mathbb{L}^k$ (in (ii) and (iii) above), and that therefore Algorithm A does generate a tour of $\mathbb{A}$ (in (i)), we are left with the bound on the length $l_0(\mathbb{A})$ of this tour. The tour generated is described by the string (2.8). Each "piece" $\mathcal{S}_j$ of $\mathcal{S}$ is shorter by $d(A_{n_j}^{(j)}, A_1^{(j)})$ than $l(\mathbb{A}\mathbb{C}_j)$ because, by the definition of the tour-length and (2.5),

$$l(\mathbb{A}\mathbb{C}_j)=\sum_{i=2}^{n_j} d(A_{i-1}^{(j)}, A_i^{(j)})+d(A_{n_j}^{(j)}, A_1^{(j)}). \tag{4.3}$$

On the other hand (see (2.9)) the "pieces" of $\mathcal{S}$ are joined by segments $A_0^{(j)}A_1^{(j)}$, or more properly $A_{n_{j-1}}^{(j-1)}A_1^{(j)}$, joining a point of $\mathbb{C}_{j-1}$ to a point of $\mathbb{C}_j$ (for each of $j=1,2,\cdots,M$), and we have shown (in (ii) above) that any such segment cannot be longer than $(\lambda/m)\sqrt{k+3}$. Thus, since $M=m^k$, if $q=1-1/k$,

$$\begin{aligned} l_0(\mathbb{A}) &< \sum_{j=1}^{M} l(\mathbb{A}\mathbb{C}_j)+M\frac{\lambda}{m}\sqrt{k+3} \\ &= \sum_{j=1}^{M} l(\mathbb{A}\mathbb{C}_j)+\lambda M^q\sqrt{k+3}. \end{aligned} \tag{4.4}$$

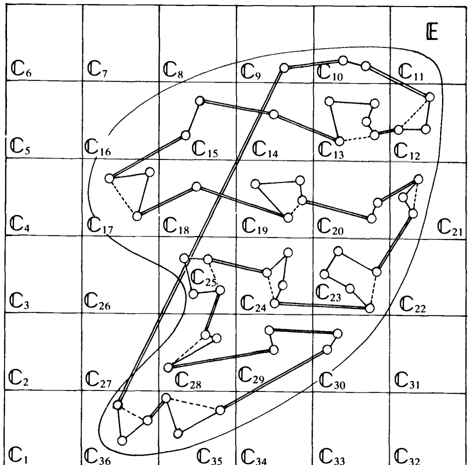


FIG. 4. *Example of a tour of* 53 *points in* $\mathbb{R}^2$ *generated by Algorithm* A *with* $\mathbb{E}$ *as shown and* $m=6$, $M=36$. *Follow single and double segments; omit dotted ones.*

Note that the inequality in (4.4) is *strict*, both because $A_1^{(j)} \neq A_{n_j}^{(j)}$ (and d is a metric) and because the cubes $\mathbb{C}_j$ are semi-open. Q.E.D.

5. The algorithms are fast.

THEOREM 2. *In probability, the time taken to execute Algorithms* A *and* B *will be asymptotic to* $An\sqrt{\sigma(n)}\log\sqrt{\sigma(n)}$ *as* $n \to \infty$; *that is, the time will be* $o[n\sigma(n)]$.

Proof. The execution time of our algorithm may be divided into several parts. T_1 is the time required to determine $\mathbb{E}$, $\mathbb{C}$, λ, m, and M; T_2 is the time required to determine which points of $\mathbb{A}$ are in each of the cells $\mathbb{C}_j$ $(j=1,2,\cdots,M)$; T_3 is the time required to determine the succession of cells (by Algorithm B); T_4 is the time required to obtain the cyclic order of the points in each individual cell (by Algorithm C); T_5 is the time required to compute the tour-length $l_0(\mathbb{A})$. We must prove that each of these five times is of the order of $n\sqrt{\sigma(n)}\log\sqrt{\sigma(n)}$ or less.

(i) We have already mentioned that $\mathbb{E}$ and $\mathbb{C}$ will either be known a priori, or will be determined in time of the order of n. Now, λ and $v(\mathbb{E})$ will be obtained in time independent of n, and generally we would say that $\sigma(n)$, and hence m and M, will also be computed (by (2.2) and (2.3)) in constant time. However, if n is *really large*, it will run to multiple precision, and $\sigma(n)$ may take a time $O(\log n)$ to compute. Nevertheless, we see that, at worst,

$$T_1 = O(n) = o[n\sqrt{\sigma(n)}\log\sqrt{\sigma(n)}]. \tag{5.1}$$

It is clear, also, that, given the tour $(\mathbb{A}, \omega)$ generated by our algorithm, its length $l_0(\mathbb{A})$ can be computed in time of the order of n (see (2.9), with (2.6)). Thus,

$$T_5 = O(n) = o[n\sqrt{\sigma(n)}\log\sqrt{\sigma(n)}]. \tag{5.2}$$

(ii) Let us suppose that the coordinates of the n points of $\mathbb{A}$ are each directly addressable in an array $\mathcal{X}$, occupying some kn locations. Define lists $\mathcal{L}_1, \mathcal{L}_2, \cdots, \mathcal{L}_M$, corresponding to the M cells: for instance, the M vectors $\mathbf{a} \in \mathbb{L}^k$ may be lexically ordered to identify the corresponding cells $\mathbb{C}(\mathbf{a})$ and lists $\mathcal{L}(\mathbf{a})$. In a time of the order of n, one may make a single pass through $\mathcal{X}$, determining for each point the cell $\mathbb{C}(\mathbf{a})$

in which it lies and entering its address in the corresponding list $\mathscr{L}(\mathbf{a})$. For each $\mathbf{a}$, the list $\mathscr{L}(\mathbf{a})$ of points in $\mathbb{A}\mathbb{C}(\mathbf{a})$ will have a length $2n(\mathbf{a})$ (where $n(\mathbf{a})$ is the number of points in $\mathbb{A}\mathbb{C}(\mathbf{a})$): each entry in the list will consist of an address in $\mathscr{H}$ and a pointer to the next entry in the list. By (2.6), this will add up to some $2n$ locations in all. Thus, with moderate storage capacity, we get

$$T_2 = O(n) = o[n\sqrt{\sigma(n)}\log\sqrt{\sigma(n)}]. \tag{5.3}$$

The procedure is thus to begin with one cell, say $\mathbb{C}(\mathbf{0})$, compute a minimal tour of the points of $\mathbb{A}\mathbb{C}(\mathbf{0})$ using the list $\mathscr{L}(\mathbf{0})$ and Algorithm C, and begin a new list $\mathscr{M}$, giving the ordering of the tour $(\mathbb{A}, \omega)$ as a string of addresses in $\mathscr{H}$, by entering the string $\mathscr{M}(\mathbf{0})$ of addresses generated by Algorithm C. We now use Algorithm B to determine the successor cell $\mathbb{C}(\mathbf{0}')$ to $\mathbb{C}(\mathbf{0})$, and use $\mathscr{L}(\mathbf{0}')$ and Algorithm C to generate the next piece $\mathscr{M}(\mathbf{0}')$ of $\mathscr{M}$. We repeat, from cell to cell, until all pieces $\mathscr{M}(\mathbf{0}^{(j)})$ have been constructed and entered in $\mathscr{M}$. The total time needed to compute the cell succession is then T_3, while the time needed to determine all the individual cell-tours is T_4.

It is clear that Algorithm B is independent of n (except through (2.2) and (2.3)), and that its execution for each cell does not depend on the number of cells. Thus,

$$T_3 = O(M) = O\left[\frac{n}{\log\sigma(n)}\right] = o[n\sqrt{\sigma(n)}\log\sqrt{\sigma(n)}]. \tag{5.4}$$

(iii) All that now remains to be estimated is the time T_4, and this will be shown to constitute the major part of the total time, in probability. We know that, if n_j points of $\mathbb{A}$ lie in $\mathbb{C}_j$, then, by (1.1), the time needed by Algorithm C to construct a minimal tour of $\mathbb{A}\mathbb{C}_j$ will be

$$t(\mathbb{A}\mathbb{C}_j) = t_{n_j} = \begin{cases} 2A(n_j-1)[2^{n_j-3}(n_j-2)+1] & \text{if } n_j > 0, \\ 0 & \text{if } n_j = 0, \end{cases} \tag{5.5}$$

and

$$T_4 = \sum_{j=1}^{M} t(\mathbb{A}\mathbb{C}_j). \tag{5.6}$$

At this stage, we introduce the *probabilistic structure* of our problem. Since the points of $\mathbb{A}$ are supposed to be independently uniformly distributed at random in the set $\mathbb{E}$, it follows that the probability that exactly s points of $\mathbb{A}$ fall into the cell $\mathbb{C}_j$ will be

$$\binom{n}{s}\alpha_j^s(1-\alpha_j)^{n-s}, \tag{5.7}$$

where

$$\alpha_j = \frac{v(\mathbb{E}\mathbb{C}_j)}{v(\mathbb{E})} \leqq \frac{v(\mathbb{C}_j)}{v(\mathbb{E})} = \alpha_0 = \frac{\lambda^k}{Mv(\mathbb{E})}, \tag{5.8}$$

with equality if and only if $v(\mathbb{E}^c\mathbb{C}_j) = 0$. Similarly, the probability that exactly r points of $\mathbb{A}$ will fall into $\mathbb{C}_i$ and exactly s points into $\mathbb{C}_j$, with $i \neq j$, will be

$$\binom{n}{r+s}\binom{r+s}{s}\alpha_i^r\alpha_j^s(1-\alpha_i-\alpha_j)^{n-r-s}. \tag{5.9}$$

Now partition the index set $\{1, 2, \cdots, M\}$ of the cells into

$$\begin{aligned} \mathbb{H}_0 &= \{j: \mathbb{C}_j \subseteq \mathbb{E}^c\}, \\ \mathbb{H}_1 &= \{j: \mathbb{C}_j \subseteq \mathbb{E}\}, \\ \mathbb{H}_2 &= \{j: \mathbb{C}_j\mathbb{E} \neq \varnothing \,\&\, \mathbb{C}_j\mathbb{E}^c \neq \varnothing\}. \end{aligned} \tag{5.10}$$

Denote the *cardinality* of any set $\mathbb{F}$ by $\mathrm{N}(\mathbb{F})$; and let

$$\mathrm{N}(\mathbb{H}_0) = N_0, \quad \mathrm{N}(\mathbb{H}_1) = N_1, \quad \mathrm{N}(\mathbb{H}_2) = N_2. \tag{5.11}$$

Then property (c) postulated for the set $\mathbb{E}$ tells us that $N_2 = O(M^q)$ as $n \to \infty$. Thus, if we write

$$\delta(n) = \log\sqrt{\sigma(n)} \quad \text{or} \quad \sigma(n) = e^{2\delta(n)},$$

whence, by (1.4),

$$\frac{e^{2\delta(n)}}{n} \to 0 \quad \text{and} \quad \frac{\delta(n)}{n} \to 0 \quad \text{as } n \to \infty, \tag{5.12}$$

then by (2.3), $M = O[n/\delta(n)]$ as $n \to \infty$, and so

$$N_2 = O\left\{\left[\frac{n}{\delta(n)}\right]^q\right\} \quad \text{and} \quad \frac{N_2}{M} \to 0 \quad \text{as } n \to \infty. \tag{5.13}$$

We also observe that, by (2.2) and (2.3),

$$\begin{aligned} m &= \lambda\left[\frac{n}{v(\mathbb{E})\delta(n)}\right]^p + O(1) = \lambda\left[\frac{n}{v(\mathbb{E})\delta(n)}\right]^p\left\{1 + O\left[\frac{\delta(n)}{n}\right]^p\right\}, \\ M &= m^k = \frac{\lambda^k}{v(\mathbb{E})}\left[\frac{n}{\delta(n)}\right]\left\{1 + O\left[\frac{\delta(n)}{n}\right]^p\right\} \quad \text{as } n \to \infty. \end{aligned} \tag{5.14}$$

Further, it is clear from (5.10) that

$$\bigcup_{j \in \mathbb{H}_1} \mathbb{C}_j \subseteq \mathbb{E} \subseteq \bigcup_{j \in \mathbb{H}_1 \cup \mathbb{H}_2} \mathbb{C}_j, \tag{5.15}$$

whence

$$\frac{N_1\lambda^k}{M} \leqq v(\mathbb{E}) \leqq \frac{(N_1 + N_2)\lambda^k}{M}. \tag{5.16}$$

It now follows from (5.13) that, since $q = 1 - p$,

$$N_1 = \frac{n}{\delta(n)}\left\{1 + O\left[\frac{\delta(n)}{n}\right]^p\right\} \quad \text{as } n \to \infty. \tag{5.17}$$

(iv) We now seek to obtain asymptotic forms for the *expected value* $\mathbf{E}[T_4]$ and *variance* var $[T_4]$ of the time T_4. By (5.6),

$$\mathbf{E}[T_4]=\sum_{j=1}^{M}\mathbf{E}[t(\mathbb{AC}_j)] \tag{5.18}$$

and

$$\begin{aligned}\operatorname{var}[T_4]&=\mathbf{E}\left[\left(\sum_{j=1}^{M}\{t(\mathbb{AC}_j)-\mathbf{E}[t(\mathbb{AC}_j)]\}\right)^2\right]\\&=\sum_{i=1}^{M}\sum_{j=1}^{M}\mathbf{E}[\{t(\mathbb{AC}_i)-\mathbf{E}[t(\mathbb{AC}_i)]\}\{t(\mathbb{AC}_j)-\mathbf{E}[t(\mathbb{AC}_j)]\}].\end{aligned} \tag{5.19}$$

Thus $\mathbf{E}[T_4]$ consists of terms $\mathbf{E}[t(\mathbb{AC}_j)]$, and var $[T_4]$ consists of products of such terms, together with $\mathbf{E}[\{t(\mathbb{AC}_j)\}^2]$ and $\mathbf{E}[t(\mathbb{AC}_i)t(\mathbb{AC}_j)]$ with $i\neq j$. If we adopt the usual notation, for integers n and positive integers ϕ, that

$$(n)_0=1,\qquad (n)_\phi=n(n-1)(n-2)\cdots(n-\phi+1)\quad (=0 \text{ for } \phi>n\geqq 0), \tag{5.20}$$

we readily verify that, by (1.1), for $s\geqq 1$,

$$\begin{aligned}t_s&=A[2^{s-2}(s)_2-2^{s-1}(s)_1+2(s)_1+\tfrac{1}{2}\times 2^s(s)_0-2(s)_0],\\ t_s^2&=A^2[16\times 4^{s-4}(s)_4+8\times 2^{s-3}(s)_3+2\times 4^{s-2}(s)_2-4\times 2^{s-2}(s)_2+4(s)_2\\&\qquad -4^{s-1}(s)_1+4\times 2^{s-1}(s)_1-4(s)_1+\tfrac{1}{4}\times 4^s(s)_0-2\times 2^s(s)_0+4(s)_0];\end{aligned} \tag{5.21}$$

so that we may write

$$t_s=A\sum_{\psi=0}^{2}\sum_{\theta=1}^{2}P_{\psi\theta}\theta^{s-\psi}(s)_\psi \quad\text{and}\quad t_s^2=A^2\sum_{\psi=0}^{4}\sum_{\theta=1}^{4}Q_{\psi\theta}\theta^{s-\psi}(s)_\psi, \tag{5.22}$$

where $P_{22}=1$, $P_{12}=-1$, $P_{11}=2$, $P_{02}=\frac{1}{2}$, $P_{01}=-2$, $Q_{44}=16$, $Q_{32}=8$, $Q_{24}=2$, $Q_{22}=-4$, $Q_{21}=4$, $Q_{14}=-1$, $Q_{12}=4$, $Q_{11}=-4$, $Q_{04}=\frac{1}{4}$, $Q_{02}=-2$, $Q_{01}=4$, and all other coefficients vanish. It follows from (5.7), (5.8), and (5.9) that

$$\begin{aligned}\mathbf{E}[t(\mathbb{AC}_j)]&=A\sum_{\psi=0}^{2}\sum_{\theta=1}^{2}P_{\psi\theta}J(n,\theta,\psi,\alpha_j),\\ \mathbf{E}[\{t(\mathbb{AC}_j)\}^2]&=A^2\sum_{\psi=0}^{4}\sum_{\theta=1}^{4}Q_{\psi\theta}J(n,\theta,\psi,\alpha_j),\\ \mathbf{E}[t(\mathbb{AC}_i)t(\mathbb{AC}_j)]&=A^2\sum_{\phi=0}^{2}\sum_{\psi=0}^{2}\sum_{\theta_1=1}^{2}\sum_{\theta_2=1}^{2}P_{\phi\theta_1}P_{\psi\theta_2}K(n;\theta_1,\theta_2;\phi,\psi;\alpha_i,\alpha_j),\end{aligned} \tag{5.23}$$

where we write

$$J(n,\theta,\psi,x)=\sum_{s=1}^{n}\binom{n}{s}x^s(1-x)^{n-s}\theta^{s-\psi}(s)_\psi, \tag{5.24}$$

$$\begin{aligned}&K(n,\theta_1,\theta_2;\phi,\psi;x,y)\\&\quad=\sum_{r=1}^{n}\sum_{s=1}^{n-r}\binom{n}{r+s}\binom{r+s}{s}x^ry^s(1-x-y)^{n-r-s}\theta_1^{r-\phi}(r)_\phi\theta_2^{s-\psi}(s)_\psi.\end{aligned} \tag{5.25}$$

(The sums are over indices from 1 to n because the time for index 0 is zero, not t_0: compare (5.5).) We may evaluate these sums as follows:

$$(5.26)\qquad \begin{aligned} J(n,\theta,\psi,x) &= (n)_\psi x^\psi \sum_{s=1}^{n} \binom{n-\psi}{s-\psi} (x\theta)^{s-\psi}(1-x)^{n-s} \\ &= (n)_\psi x^\psi \{1+x(\theta-1)\}^{n-\psi} - \delta_{\psi 0}(1-x)^n, \end{aligned}$$

where δ_{ij} is the *Kronecker delta* ($=1$ if $i=j$, $=0$ if $i \neq j$), and similarly,

$$(5.27)\qquad \begin{aligned} &K(n;\theta_1,\theta_2;\phi,\psi;x,y) \\ &= (n)_{\phi+\psi} x^\phi y^\psi \sum_{u=2}^{n} \binom{n-\phi-\psi}{u-\phi-\psi}(1-x-y)^{n-u} \sum_{s=1}^{u-1} \binom{u-\phi-\psi}{s-\psi}(x\theta_1)^{u-s-\phi}(y\theta_2)^{s-\psi} \\ &= (n)_{\phi+\psi} x^\phi y^\psi \sum_{u=2}^{n} \binom{n-\phi-\psi}{u-\phi-\psi}(1-x-y)^{n-u} \\ &\qquad\qquad \times \{(x\theta_1+y\theta_2)^{u-\phi-\psi} - \delta_{\psi 0}(x\theta_1)^{u-\phi} - \delta_{\phi 0}(y\theta_2)^{u-\psi}\} \\ &= (n)_{\phi+\psi} x^\phi y^\psi \{[1+x(\theta_1-1)+y(\theta_2-1)]^{n-\phi-\psi} - \delta_{\psi 0}[1+x(\theta_1-1)-y]^{n-\phi} \\ &\qquad\qquad - \delta_{\phi 0}[1-x+y(\theta_2-1)]^{n-\psi} - \delta_{\phi 0}\delta_{\psi 0}(1-x-y)^n\}. \end{aligned}$$

(We note that $\binom{a}{b} = 0$ whenever $b<0$ or $b>a$.)

Now, by a simple inductive argument on m, we observe that, for all nonnegative integers m, n, and ζ, with $n \geqq \zeta$,

$$(5.28)\qquad 0 \leqq n^m - (n-\zeta)_m \leqq mn^{m-1}[\zeta + \tfrac{1}{2}(m-1)].$$

Thus, since

$$(5.29)\qquad e^{nz} - (1+z)^{n-\zeta} = \sum_{m=1}^{\infty} \frac{1}{m!}[n^m - (n-\zeta)_m]z^m,$$

we have, for all $z \geqq 0$, that

$$0 \leqq e^{nz} - (1+z)^{n-\zeta} \leqq \sum_{m=1}^{\infty} \left\{ \frac{\zeta}{(m-1)!} + \frac{\frac{1}{2}}{(m-2)!} \right\} n^{m-1} z^m = (\zeta z + \tfrac{1}{2}nz^2)\, e^{nz},$$

whence

$$(5.30)\qquad e^{nz}(1-\zeta z - \tfrac{1}{2}nz^2) \leqq (1+z)^{n-\zeta} \leqq e^{nz}.$$

Thus, for all those J-terms and K-terms in the sums (5.23) for which $\phi \geqq 1$ and $\psi \geqq 1$,

$$(5.31)\qquad (n)_\psi \alpha_j^\psi e^{n\alpha_j(\theta-1)}\{1-\psi\alpha_j(\theta-1) - \tfrac{1}{2}n\alpha_j^2(\theta-1)^2\} \leqq J(n,\theta,\psi,\alpha_j) \leqq (n)_\psi \alpha_j^\psi e^{n\alpha_j(\theta-1)}$$

and

$$(5.32)\quad \begin{aligned}(n)_{\phi+\psi}\alpha_i^{\phi}\alpha_j^{\psi}\, e^{n\alpha_i(\theta_1-1)+n\alpha_j(\theta_2-1)}\{1-(\phi+\psi)[\alpha_i(\theta_1-1)+\alpha_j(\theta_2-1)]\\ -\tfrac{1}{2}n[\alpha_i(\theta_1-1)+\alpha_j(\theta_2-1)]^2\}\\ \leqq K(n;\theta_1,\theta_2;\phi,\psi;\alpha_i,\alpha_j)\leqq (n)_{\phi+\psi}\alpha_i^{\phi}\alpha_j^{\psi}\, e^{n\alpha_i(\theta_1-1)+n\alpha_j(\theta_2-1)}.\end{aligned}$$

By (5.8) and (5.14),

$$(5.33)\quad \begin{aligned}&\alpha_0=\frac{\delta(n)}{n}\left\{1+O\left[\frac{\delta(n)}{n}\right]^p\right\}\quad \text{as } n\to\infty \quad\text{and}\quad \alpha_j\leqq\alpha_0,\\ &(n)_\psi=n^\psi[1+O(1/n)],\quad \alpha_j=O[\delta(n)/n],\quad\text{and}\quad n\alpha_j^2=O\{[\delta(n)]^2/n\},\end{aligned}$$

whence, by (5.31),

$$(5.34)\quad J(n,\theta,\psi,\alpha_j)=(n\alpha_j)^\psi\, e^{n\alpha_j(\theta-1)}\left(1+O\left\{\frac{[\delta(n)]^2}{n}\right\}\right),$$

and similarly, by (5.32),

$$(5.35)\quad K(n;\theta_1,\theta_2;\phi,\psi;\alpha_i,\alpha_j)=(n\alpha_i)^\phi(n\alpha_j)^\psi\, e^{n\alpha_i(\theta_1-1)+n\alpha_j(\theta_2-1)}\left(1+O\left\{\frac{[\delta(n)]^2}{n}\right\}\right).$$

We note, further, that the correction terms for $\phi=0$ and $\psi=0$ in (5.26) and (5.27) are never of higher asymptotic order than the main terms, found in (5.34) and (5.35).

In calculating $\mathbf{E}[T_4]$, we may distribute the sum over the cells $\mathbb{C}_j$ among the several J-terms of the corresponding expression of (5.23). For $j\in\mathbb{H}_0$, there is no contribution; for $j\in\mathbb{H}_1$, each term equals $J(n,\theta,\psi,\alpha_0)$ and there are N_1 such terms; and for $j\in\mathbb{H}_2$, when n is sufficiently large, we see by (5.34) that the contributions are somewhat smaller, since the J-terms are monotonically increasing with α_j, and $\alpha_j\leqq\alpha_0$, by (5.33). Thus, by (5.13), (5.17), (5.33), and (5.34),

$$(5.36)\quad \begin{aligned}\sum_{j=1}^{M} J(n,\theta,\psi,\alpha_j)&=N_1J(n,\theta,\psi,\alpha_0)+\sum_{j\in\mathbb{H}_2}J(n,\theta,\psi,\alpha_j)\\ &=n[\delta(n)]^{\psi-1}\, e^{(\theta-1)\delta(n)}\left(1+O\left\{\frac{[\delta(n)]^{p+1}}{n^p}\right\}\right),\end{aligned}$$

since $e^{(\theta-1)n\alpha_0}=e^{(\theta-1)\delta(n)}\,e^{O\{[\delta(n)]^{p+1}/n^p\}}=e^{(\theta-1)\delta(n)}(1+O\{[\delta(n)]^{p+1}/n^p\}$, $[\delta(n)]^2/n=o\{[\delta(n)]^{p+1}/n^p\}$, $q=1-p$, and $[\delta(n)/n]^p=o\{[\delta(n)^{p+1}/n^p\}$. It follows at once from (5.18), (5.23), and (5.36) that

$$(5.37)\quad \begin{aligned}\mathbf{E}[T_4]&=AP_{22}\sum_{j=1}^{M}J(n,2,2,\alpha_j)\left\{1+O\left[\frac{1}{\delta(n)}\right]\right\}\\ &=An\,\delta(n)\,e^{\delta(n)}\left\{1+O\left[\frac{1}{\delta(n)}\right]\right\},\end{aligned}$$

since the terms in P_{22} dominate the result, and the terms of next highest order arise from P_{12} and are of the order of $ne^{\delta(n)}$ (a factor $1/\delta(n)$ lower), and since $[\delta(n)]^{p+1}/n^p=$

$o[1/\delta(n)]$ (because $[\delta(n)]^{p+2}/n^p = \{[\delta(n)]^{2k+1}/n\}^p = o\{[e^{2\delta(n)}/n]^p\} \to 0$ as $n \to \infty$, by (5.12)).

Similarly, by breaking up (5.19) into a single sum $\sum_j$ and a double sum $\sum_i \sum_{j>i}$, and calculating the asymptotic form of each term, we can obtain the corresponding form of var $[T_4]$. By (5.23), we get that

$$\begin{aligned}
\operatorname{var}[T_4] &= \sum_{j=1}^{M} \left(\mathbf{E}[\{t(\mathbb{AC}_j)\}^2] - \{\mathbf{E}[t(\mathbb{AC}_j)]\}^2 \right) \\
&\quad + 2 \sum_{i=1}^{M-1} \sum_{j=i+1}^{M} \left(\mathbf{E}[t(\mathbb{AC}_i) t(\mathbb{AC}_j)] - \mathbf{E}[t(\mathbb{AC}_i)] \mathbf{E}[t(\mathbb{AC}_j)] \right) \\
&= \sum_{j=1}^{M} A^2 \left(\sum_{\psi=0}^{4} \sum_{\theta=1}^{4} Q_{\psi\theta} J(n, \theta, \psi, a_j) - \left\{ \sum_{\psi=0}^{2} \sum_{\theta=1}^{2} P_{\psi\theta} J(n, \theta, \psi, \alpha_j) \right\}^2 \right) \\
&\quad + 2 \sum_{i=1}^{M-1} \sum_{j=i+1}^{M} A^2 \sum_{\phi=0}^{2} \sum_{\psi=0}^{2} \sum_{\theta_1=1}^{2} \sum_{\theta_2=1}^{2} P_{\phi\theta_1} P_{\psi\theta_2} \big(K(n; \theta_1, \theta_2; \phi, \psi; \alpha_i, \alpha_j) \\
&\qquad - J(n, \theta_1, \phi, \alpha_i) J(n, \theta_2, \psi, \alpha_j) \big).
\end{aligned} \tag{5.38}$$

By (5.36), we see that the contribution of the terms arising from the $\mathbf{E}[\{t(\mathbb{AC}_j)\}^2]$ is dominated by the terms with coefficient Q_{44}: these yield $16A^2 n[\delta(n)]^3 e^{3\delta(n)} (1 + O\{[\delta(n)]^{p+1}/n^p\})$, with the terms of next highest order coming from Q_{24} and being of the order of $n\,\delta(n)\, e^{3\delta(n)}$. The terms arising from $\{\mathbf{E}[t(\mathbb{AC}_j)]\}^2$ are dominated by those with coefficient P_{22}^2, and (by an argument analogous to that from (5.34) to (5.36)) these yield a contribution of the order of $n[\delta(n)]^3 e^{2\delta(n)}$, which is therefore asymptotically negligible. Thus, as in going from (5.36) to (5.37), the single sum in (5.38) is asymptotic to $16A^2 n[\delta(n)]^3 e^{3\delta(n)} (1 + O\{1/[\delta(n)]^2\})$.

Turning to the double sum, we observe by (5.26) and (5.27) that

$$\begin{aligned}
&K(n; \theta_1, \theta_2; \phi, \psi; \alpha_i, \alpha_j) - J(n, \theta_1, \phi, \alpha_i) J(n, \theta_2, \psi, \alpha_j) \\
&\quad = (n)_{\phi+\psi} \alpha_i^{\phi} \alpha_j^{\psi} \{ [1 + \alpha_i(\theta_1 - 1) + \alpha_j(\theta_2 - 1)]^{n-\phi-\psi} - \delta_{\psi 0} [1 + \alpha_i(\theta_1 - 1) - \alpha_j]^{n-\phi} \\
&\qquad\qquad - \delta_{\phi 0} [1 - \alpha_i + \alpha_j(\theta_2 - 1)]^{n-\psi} - \delta_{\phi 0}\, \delta_{\psi 0} [1 - \alpha_i - \alpha_j]^n \} \\
&\qquad - (n)_{\phi} (n)_{\psi} \alpha_i^{\phi} \alpha_j^{\psi} \{ [1 + \alpha_i(\theta_1 - 1)]^{n-\phi} - \delta_{\phi 0} [1 - \alpha_i]^n \} \\
&\qquad\qquad \times \{ [1 + \alpha_j(\theta_2 - 1)]^{n-\psi} - \delta_{\psi 0} [1 - \alpha_j]^n \}.
\end{aligned} \tag{5.39}$$

Note by (5.20), (5.30), and (5.33) that $(n)_\zeta = n^\zeta [1 + O(1/n)]$, $\{1 + O[\delta(n)/n]\}^\zeta = 1 + O[\delta(n)/n]$, $(1+z)^{n-\zeta} = e^{nz}(1 + O\{[\delta(n)]^2/n\})$ if $z = O[\delta(n)/n]$, for any ζ independent of n, and that the residue of greatest order is $O\{[\delta(n)]^2/n\}$, while, if $x = O[\delta(n)/n]$ and $y = O[\delta(n)/n]$, then

$$\left[\frac{1 + x + y}{(1+x)(1+y)} \right]^n = [1 - xy + xy(x+y) - \cdots]^n = 1 - nxy + O\left\{ \frac{[\delta(n)]^4}{n^2} \right\}. \tag{5.40}$$

Thus, so long as $\phi \geqq 1$ and $\psi \geqq 1$,

$$K(n;\theta_1,\theta_2;\phi,\psi;\alpha_i,\alpha_j)-J(n,\theta_1,\phi,\alpha_i)J(n,\theta_2,\psi,\alpha_j)$$

$$\begin{aligned}
&= -(n\alpha_i)^{\phi}(n\alpha_j)^{\psi}\left[1+O\left(\frac{1}{n}\right)\right] e^{n\alpha_i(\theta_1-1)+n\alpha_j(\theta_2-1)}\left(1+O\left\{\frac{[\delta(n)]^2}{n}\right\}\right)\\
&\quad\times\left[O\left(\frac{1}{n}\right)+\frac{(1+x+y)^{n-\phi-\psi}}{(1+x)^{n-\phi}(1+y)^{n-\psi}}-1\right] \quad (\text{where } x=n\alpha_i(\theta_1-1),\ y=n\alpha_j(\theta_2-1))\\
&= O\{[\delta(n)]^{\phi+\psi} e^{(\theta_1+\theta_2-2)\delta(n)}\}\left(1+O\left\{\frac{[\delta(n)]^{p+1}}{n^p}\right\}\right)O\left\{\frac{[\delta(n)]^2}{n}\right\}.
\end{aligned} \tag{5.41}$$

The dominant contribution to the double sum thus arises from terms with the coefficient P_{22}^2: these are less than M^2 in number, so that the contribution will be $O\{n[\delta(n)]^4 e^{2\delta(n)}\}$, which is again asymptotically negligible in comparison with the order of magnitude of the single sum in (5.38). Therefore,

$$\operatorname{var}[T_4]=16A^2n[\delta(n)]^3 e^{3\delta(n)}\left(1+O\left\{\frac{1}{[\delta(n)]^2}\right\}\right). \tag{5.42}$$

(v) We may now complete the proof of Theorem 2. First, we note that, for any $\varepsilon>0$ and all sufficiently large n (say, $n\geqq n_0(\varepsilon)$), by (5.37)

$$|\mathbf{E}[T_4]-An\delta(n)\, e^{\delta(n)}|\leqq\frac{\varepsilon}{2}An\delta(n)\, e^{\delta(n)}. \tag{5.43}$$

Next, we use Chebyshev's inequality with (5.12), (5.42), and (5.43) to obtain that

$$\begin{aligned}
&\operatorname{Prob}\left(\left|\frac{T_4}{An\delta(n)\, e^{\delta(n)}}-1\right|\leqq\varepsilon\right)\\
&\geqq \operatorname{Prob}\left(\left|\frac{T_4-\mathbf{E}[T_4]}{An\delta(n)\, e^{\delta(n)}}\right|\leqq\frac{\varepsilon}{2} \text{ and } \left|\frac{\mathbf{E}[T_4]}{An\delta(n)\, e^{\delta(n)}}-1\right|\leqq\frac{\varepsilon}{2}\right)\\
&= \operatorname{Prob}\left(\left|\frac{T_4-\mathbf{E}[T_4]}{An\delta(n)\, e^{\delta(n)}}\right|\leqq\frac{\varepsilon}{2}\right) \quad \text{for all } n\geqq n_0(\varepsilon)\\
&\geqq 1-\frac{\operatorname{var}[T_4]}{[(A\varepsilon/2)n\delta(n)\, e^{\delta(n)}]^2} \quad (\text{Chebyshev})\\
&\sim 1-\left(\frac{64}{\varepsilon^2}\right)\frac{\delta(n)\, e^{\delta(n)}}{n}\to 1 \quad \text{as } n\to\infty.
\end{aligned} \tag{5.44}$$

Thus, $T_4/An\delta(n)\, e^{\delta(n)}\to 1$ in probability as $n\to\infty$, or

$$T_4\sim An\delta(n)\, e^{\delta(n)} \quad \textit{in probability} \text{ as } n\to\infty. \tag{5.45}$$

Now, we have already shown that T_1, T_2, T_3, and T_5 are all $o[n\sqrt{\sigma(n)}\log\sqrt{\sigma(n)}]$ with *certainty* as $n\to\infty$ (see (5.1)–(5.4)). Therefore, since $\delta(n)=\log\sqrt{\sigma(n)}$ and $e^{\delta(n)}=\sqrt{\sigma(n)}$, by (5.12), it follows that the total time taken by the algorithms to

compute a tour of $\mathbb{A}$ will be

$$\sum_{r=1}^{5} T_r \sim An\sqrt{\sigma(n)} \log \sqrt{\sigma(n)}, \tag{5.46}$$

in probability, as $n \to \infty$. Q.E.D.

6. The algorithms are accurate.

THEOREM 3. *With probability one (that is, almost surely, a.s.), the length $l_0(\mathbb{A})$ of the tour of the n points of $\mathbb{A}$ generated by Algorithms* A *and* B *is asymptotic to the minimal tour length $l(\mathbb{A}) \sim \beta_k v(\mathbb{E})^p n^q$.*

Proof. (i) Since $l(\mathbb{A})$ is defined to be minimal, and since (by Theorem 1) the algorithms define a tour of the set A, we have that its length

$$l_0(\mathbb{A}) \geqq l(\mathbb{A}). \tag{6.1}$$

(ii) By (5.14),

$$\lambda M^q \sqrt{k+3} \sim \frac{\lambda^k}{v(\mathbb{E})^q} \left[\frac{n}{\delta(n)}\right]^q \sqrt{k+3} = o(n^q) \quad \text{as } n \to \infty, \tag{6.2}$$

so that, by (4.4),

$$l_0(\mathbb{A}) < \sum_{j=1}^{M} l(\mathbb{A}\mathbb{C}_j) + o(n^q) \quad \text{as } n \to \infty. \tag{6.3}$$

Now consider a minimal tour $(\mathbb{A}, \pi)$ of $\mathbb{A}$ and let $\mathbb{P}$ denote the polygonal path $A_1 A_2 \cdots A_n$ (that is, let $X \in \mathbb{P}$ iff $(\exists j \in \{1, 2, \cdots, n\})$ $(\exists x)$ $0 \leqq x \leqq 1$ and $X = xA_{j-1} + (1-x)A_j$), where the points of $\mathbb{A}$ are so numbered that $A_0 \pi A_1 \pi A_2 \pi \cdots \pi A_n = A_0$. Let $\mathbb{P}_j$ be the union of the closures of all connected pieces of $\mathbb{P}\mathbb{C}_j$ containing at least one point of $\mathbb{A}$. Then the number of such pieces will be

$$h_j \leqq n_j = \mathrm{N}(\mathbb{A}\mathbb{C}_j), \tag{6.4}$$

and the sum, l_j, of the lengths of these pieces will satisfy

$$\sum_{j=1}^{M} l_j \leqq l(\mathbb{A}). \tag{6.5}$$

The end-points of the pieces of $\mathbb{P}_j$ all lie in the boundary of $\mathbb{C}_j$, which consists of $2k$ *faces* $\mathbb{F}_{jf}$ $(f = 0, 1, \cdots, 2k-1$; with $\mathbb{F}_{j,2k} = \mathbb{F}_{j0})$; let $\mathbb{E}_{jf}$ be the set of end-points in $\mathbb{F}_{jf}$. We shall form a tour $(\mathbb{U}_j, \upsilon_j)$ of all the end-points, consisting of a tour $(\mathbb{V}_{jf}, \nu_{jf})$ of $\mathbb{E}_{jf}$, for each f, each connected to the next tour $\mathbb{V}_{j(f+1)}$ by a chord whose length cannot exceed the diameter of $\mathbb{C}_j$,

$$\Delta(\mathbb{C}_j) = \lambda M^{-p} \sqrt{k}. \tag{6.6}$$

It is proved in BHH, by a nontrivial combinatorial argument, that a tour $(\mathbb{T}_j, \tau_j)$ of $\mathbb{A}\mathbb{C}_j$ may be constructed by alternately traversing parts of $\mathbb{U}_j$ and pieces of $\mathbb{P}_j$ in such a way that $\mathbb{P}_j$ is traversed just once and $\mathbb{U}_j$ not more than twice. This means that

$$l(\mathbb{A}\mathbb{C}_j) \leqq l(\mathbb{T}_j, \tau_j) \leqq l_j + 2l(\mathbb{U}_j, \upsilon_j). \tag{6.7}$$

(The interested reader may find the above-mentioned proof under Lemma 2 in the appendix of BHH.) We note that

$$(6.8)\qquad l(\mathbb{U}_j, \upsilon_j) \leqq \sum_{f=1}^{2k} l(\mathbb{V}_{jf}, \nu_{jf}) + 2k\Delta(\mathbb{C}_j).$$

(iii) To construct $(\mathbb{V}_{jf}, \nu_{jf})$, we proceed as follows. First, we dissect the face $\mathbb{F}_{jf}$ of $\mathbb{C}_j$, which is a $(k-1)$-dimensional hypercube of side λM^{-p}, into L equal cells of side $\lambda M^{-p}L^{-p'}$, where $p' = 1/(k-1)$, just as in applying our algorithms, taking $L^{p'}$ to be an even integer; thus we may construct a tour of the cell-centers of length $L\lambda M^{-p}L^{-p'}$, using Algorithm B. Then for each cell we insert any point of $\mathbb{E}_{jf}$ therein into the tour, by connecting it to-and-fro to the nearest point of the tour, thereby increasing the length of the path by no more than $\lambda M^{-p}L^{-p'}\sqrt{k-1}$ for each point of $\mathbb{E}_{jf}$. Then

$$(6.9)\qquad l(\mathbb{V}_{jf}, \nu_{jf}) \leqq \lambda M^{-p}[L^{1-p'} + h_{jf}L^{-p'}\sqrt{k-1}],$$

where

$$(6.10)\qquad h_{jf} = \mathrm{N}(\mathbb{E}_{jf}), \qquad \sum_{f=1}^{2k} h_{jf} = 2h_j.$$

We now observe that

$$(6.11)\qquad \frac{\partial}{\partial L}[L^{1-p'} + h_{jf}L^{-p'}\sqrt{k-1}]\begin{cases} <0 & \text{if } L < L_0, \\ =0 & \text{if } L = L_0, \\ >0 & \text{if } L > L_0, \end{cases}$$

where

$$(6.12)\qquad L_0 = \left[\frac{\sqrt{k-1}}{k-2}\right] h_{jf}.$$

Choose L to be that integer multiple of 2^{k-1} satisfying

$$(6.13)\qquad L_0 \leqq L < L_0 + 2^{k-1};$$

hence, by (6.9),

$$(6.14)\qquad l(\mathbb{V}_{jf}, \nu_{jf}) \leqq \lambda M^{-p}\left(\frac{\sqrt{k-1}}{k-2}\right)^{-p'} h_{jf}^{-p'}\left[\frac{(k-1)^{3/2}}{k-2}h_{jf} + 2^{k-1}\right],$$

or, more simply,

$$(6.15)\qquad l(\mathbb{V}_{jf}, \nu_{jf}) \leqq \lambda M^{-p}(R_k h_{jf}^{q'} + S_k h_{jf}^{-p'}),$$

where $q' = 1 - p'$ and R_k and S_k are constants depending only on k.

(iv) We may now combine the foregoing results to yield the following (the subscripts attached to $\leqq$ and $<$ signs refer to the justifying assertion; e.g., the first

$\leqq_1$ refers to (6.1) and the first $<_3$ refers to (6.3)):

$$
\begin{aligned}
0 &\leqq_1 l_0(\mathbb{A}) - l(\mathbb{A}) <_3 \sum_{j=1}^{M} l(\mathbb{A}\mathbb{C}_j) + o(n^q) - l(\mathbb{A}) \\
&\leqq_7 \sum_{j=1}^{M} l_j + 2\sum_{j=1}^{M} l(\mathbb{U}_j, v_j) + o(n^q) - l(\mathbb{A}) \\
&\leqq_5 2\sum_{j=1}^{M} l(\mathbb{U}_j, v_j) + o(n^q) \\
&\leqq_8 2\sum_{j=1}^{M}\sum_{f=1}^{2k} l(\mathbb{V}_{jf}, \nu_{jf}) + 2k\sum_{j=1}^{M} \Delta(\mathbb{C}_j) + o(n^q) \\
&=_6 2\sum_{j=1}^{M}\sum_{f=1}^{2k} l(\mathbb{V}_{jf}, \nu_{jf}) + 2k^{3/2}\lambda M^q + o(n^q) \\
&\leqq_{15} 2\lambda M^{-p}\left(R_k \sum_{j=1}^{M}\sum_{f=1}^{2k} h_{jf}^{q'} + S_k \sum_{j=1}^{M}\sum_{f=1}^{2k} h_{jf}^{-p'}\right) + 2k^{3/2}\lambda M^q + o(n^q).
\end{aligned} \tag{6.16}
$$

We must remark that the bound (6.15) becomes infinite if $h_{jf} = 0$; but if there are no end-points in $\mathbb{F}_{jf}$, then there is no need to tour that face, and $l(\mathbb{V}_{jf}, \nu_{jf})$ becomes zero. Thus not only $h_{jf}^{q'}$ but also $h_{jf}^{-p'}$ should be interpreted as 0 in (6.15) and (6.16), when $h_{jf} = 0$. Therefore, since h_{jf} must be a nonnegative integer, we may replace $h_{jf}^{-p'}$ in (6.16) by 1 without decreasing the bound. Further, when $0 < q' < 1$, we may apply Hölder's inequality to the sum of $h_{jf}^{q'}$ to yield that, because $p' + q' = 1$,

$$
\begin{aligned}
\sum_{j=1}^{M}\sum_{f=1}^{2k} h_{jf}^{q'} &= \sum_{j=1}^{M}\sum_{f=1}^{2k} 1 \times h_{jf}^{q'} \leqq \left(\sum_{j=1}^{M}\sum_{f=1}^{2k} 1^{1/p'}\right)^{p'} \left(\sum_{j=1}^{M}\sum_{f=1}^{2k} (h_{jf}^{q'})^{1/q'}\right)^{q'} \\
&= (2kM)^{p'}\left(\sum_{j=1}^{M}\sum_{f=1}^{2k} h_{jf}\right)^{q'} \leqq_{10,4} (2kM)^{p'}(2n)^{q'}.
\end{aligned} \tag{6.17}
$$

(When $k = 2$, and so $p' = 1$ and $q' = 0$, the sum on the left of (6.17) becomes $2kM$, and the bound on the right becomes $2kM$ also; so that (6.17) still holds.) Applying these results to (6.16), we obtain that

$$
0 \leqq l_0(\mathbb{A}) - l(\mathbb{A}) < 2\lambda M^{-p}[2R_k(kM)^{p'} n^{q'} + S_k(2kM)] + 2k^{3/2}\lambda M^q + o(n^q), \tag{6.18}
$$

and since, by (5.14), $M = O[n/\delta(n)] = o(n)$, we have that $M^q = o(n^q)$ and $M^{p'-p} n^{q'} = o(n^{p'-p} n^{q'}) = o(n^q)$. Thus, finally,

$$
0 \leqq l_0(\mathbb{A}) - l(\mathbb{A}) < o(n^q) \quad \text{as } n \to \infty. \tag{6.19}
$$

(v) To complete the proof of our theorem, we observe that BHH have proved that (1.2) holds *with probability one*, when the set $\mathbb{A}$ is taken to be $\mathbf{P}^n$, the first n points of the infinite sequence $\mathbf{P}$, distributed independently and uniformly in the set $\mathbb{E}$. Since, under these circumstances,

$$
l(\mathbb{A}) = O(n^q) \quad \text{as } n \to \infty, \tag{6.20}
$$

we may conclude that

$$l_0(\mathbb{A}) \sim l(\mathbb{A}) \quad \text{as } n \to \infty. \tag{6.21}$$

Q.E.D.

Acknowledgment. We are grateful to referees for some helpful suggestions which have been incorporated in the present version of HT. Our main results are the same; but we have rearranged the material, made a few changes in the presentation, and, in reviewing the paper, have taken the opportunity to refine and simplify both the algorithm and the proofs of its speed and accuracy.

REFERENCES

J. BEARDWOOD, J. H. HALTON, AND J. M. HAMMERSLEY (1959), *The shortest path through many points*, Proc. Cambridge Philos. Soc., 55, pp. 299–327.

R. E. BELLMAN (1962), *Dynamic programming treatment of the traveling salesman problem*, J. Assoc. Comput. Mach., 9, pp. 61–63.

M. R. GAREY, R. L. GRAHAM, AND D. S. JOHNSON (1976), *Some* NP-*complete geometric problems*, Proc. 8th ACM Symposium on Theory of Computing, pp. 10–22.

M. R. GAREY AND D. S. JOHNSON (1979), *Computers and Intractability: A Guide to the Theory of* NP-*Completeness*, Freeman, San Francisco.

J. H. HALTON AND R. TERADA (1978), *An Almost Surely Optimal Algorithm for the Euclidean Traveling Salesman Problem*, Tech. Rep. no. 335, Computer Sciences Dept., University of Wisconsin.

M. HELD AND R. M. KARP (1962), *A dynamic programming approach to sequencing problems*, SIAM J. Appl. Math., 10, pp. 196–210.

R. M. KARP (1977), *Probabilistic analysis of partitioning algorithms for the traveling salesman problem in the plane*, Math. Oper. Res., 2, pp. 209–244.

P. D. KROLAK, W. FELTS, AND G. MARBLE (1970), *Efficient heuristics for solving large traveling salesman problems*, Proc. 7th Internat. Symposium on Mathematical Programming.

S. LIN AND B. W. KERNIGHAN (1973), *An effective heuristic algorithm for the traveling salesman problem*, J. Oper. Res., 21, pp. 498–516.

C. H. PAPADIMITRIOU (1977), *The Euclidean traveling salesman problem is* NP-*complete*, Theoret. Comput. Sci., 4, pp. 237–244.

SIAM J. COMPUT.
Vol. 11, No. 1, February 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1101-0004 $01.00/0

DYNAMIC PROGRAMMING IS OPTIMAL FOR NONSERIAL OPTIMIZATION PROBLEMS*

ARNON ROSENTHAL†

Abstract. We consider discrete optimization problems in which the only exploitable feature of the objective function is a limited form of decomposability. "Nonoverlapping comparison algorithms" are defined as a model of procedures which decompose the problem and apply Bellman's principle of optimality. Nonserial dynamic programming (DP), a simple elimination procedure, is shown to be optimal among all nonoverlapping comparison algorithms, including nondeterministic algorithms. These results can give an *exponential* lower bound on the shortest admissible proof that a solution is optimal. Furthermore, if part of the search space is ruled out, a subset of the comparisons made by DP optimally searches the remainder. We suggest that the running time of DP is a useful measure of the "interaction complexity" of a problem, and that because of its simplicity DP is of practical as well as theoretical interest.

Key words. nonserial dynamic programming, optimal algorithms, lower bound, exponential time, comparision algorithms, perfect elimination graph, decomposition, complexity, dynamic programming.

1. Introduction. An interesting, very general optimization problem is defined and extensively explored in [2]. An objective function is defined as a sum of *terms*, where each term is a (tabulated) function of only a few of the variables. In the restricted case where each term shares one variable with its predecessor and one with its successor, the problem has a serial structure and may be solved by ordinary serial dynamic programming. The unrestricted case is much harder (in fact, NP-hard), but problems with a favorable pattern of term interactions may be solved efficiently by the nonserial dynamic programming algorithm of [2]. Our major result is that nonserial dynamic programming is, within its class, an optimal algorithm for this problem. [2] does not define a formal model of computation, and obtains much weaker optimality results.

Let $x_1, x_2, \cdots, x_n$ be variables, each of which has a finite domain D_i. The optimization problem is to assign values to $x_1, x_2, \cdots, x_n$ so as to minimize $f(x_1, x_2, \cdots, x_n) = \sum f_i(X_i)$, where $X_i \subseteq \{x_1, \cdots, x_n\}$ and f_i is a function over the variables of X_i. An example function is $f(x_1, x_2, x_3, x_4, x_5) = f_1(x_1, x_2, x_4) + f_2(x_3, x_4) + f_3(x_3, x_5)$. The functions f_i are called *terms*. Our theorems assume that domain sizes may differ, but when giving explicit operation counts, we assume that all variable domains have the same size, denoted $|D|$.

2. Instances of the optimization problem. Many important problems are expressible as special cases of the optimization problem. Thus, a single optimization technique and program may be used for all such problems. The optimization problem is a particularly good model for problems with no apparent exploitable features except for decomposability. Some possible applications are listed below.

(1) Complex versions of some familiar problems. Set cover, satisfiability, vertex or edge cover and vertex coloring are classical problems for which sophisticated but exponential algorithms have been developed. In practice, it is frequently necessary to solve the problem with special local constraints and extra weights. In these cases the special purpose techniques may not apply. The optimization model which will be presented can handle the additional complications.

---

* Received by the editors April 19, 1979, and in final form March 3, 1981. This work was partially supported by a Rackham grant from the University of Michigan and by the National Science Foundation under grant MCS77-01753.

† Sperry Research Center, Sudbury, Massachusetts 01776.

(2) Routing traffic in communications networks [4].

(3) Many problems can be formulated as: Given a network, assign numbers (e.g., potentials) to the vertices so as to minimize the sum of the edges' costs. The *cost* of an edge is an arbitrary function of the potentials of incident vertices. A few examples are:

(a) The network represents the pattern of city streets, and the vertex numbers represent the (relative) timing of traffic lights. The cost of an edge is a complex (queuing-related) function of the difference in timing at the edge's incident vertices [1].

(b) The vertex labels represent voltages in a circuit, and the edge costs are the amount of power dissipated as a function of voltage drop.

(c) In a fluid-flow network vertex labels represent pressures, while flow rate and pumping cost depend on the pressure drops [12].

(d) For task scheduling, the vertices represent tasks to be assigned time slots, and the edges represent the cost of collisions between tasks, delay between related tasks, etc. The restriction to only two-way interactions is unnecessary—the optimization model can allow multiway interactions.

(4) Multi-dimensional smoothing for pattern recognition can be formulated as a discrete optimization problem, even if the loss function which measures the merit of an approximation is quite complicated [2].

(5) Some algorithms for analyzing probabilistic networks and combinational circuits are very similar to the nonserial dynamic programming algorithm considered here [9], [10].

The optimization problem is NP-hard, even if each term is required to be a 0–1 function of two 0–1 variables. (The reduction from max 2-satisfiability is immediate.)

3. Nonserial dynamic programming. In this section we present the nonserial dynamic programming algorithm (denoted DP) for the optimization problem. Let f be the objective function. Suppose $f(\cdot)=\sum f_i(\cdot)$; the dot represents the arguments which we prefer not to list explicitly. Renumber the terms so that the terms involving x_1 are numbered $f_p, f_{p+1}, \cdots, f_n$. Let $y_1, \cdots, y_q$ denote the variables appearing in terms $\{f_i | i \geqq p\}$, not including x_1. Now consider $\sum_{\{i \geqq p\}} f_i(\cdot)$ to be a single term $h(x_1, y_1, \cdots, y_q)=\sum_{\{i \geqq p\}} f_i$, so $f(\cdot)=\sum_{\{i<p\}} f_i(\cdot)+h(x_1, y_1, \cdots, y_q)$. In this expression, x_1 appears only in h. Given any assignment of values for $y_1, \cdots, y_q$, it is optimal to use the value of x_1 which will minimize h. (This is Bellman's principle of optimality.) To eliminate x_1 from the objective function, we replace h (i.e., all the terms involving x_1) by

$$h^*(y_1, \cdots, y_q) := \min_{\{\text{values of } x_1\}} h(x_1, y_1, \cdots, y_q).$$

DP now eliminates the remaining variables one at a time in an analogous manner. We shall always assume that the variables have been renumbered so that the order of elimination is $x_1, \cdots, x_n$. The elimination of x_j will henceforth be called "stage j", and the terms (tables) computed at that stage will be denoted $h_j(\cdot)$ and $h_j^*(\cdot)$. The computational cost is the time to compute h^*, namely $(|D|-1)$ comparisons for each of $|D|^{**}q$ entries in h^*. There may be some *fill* (i.e., variables together in h^* which were not previously together in any term).

The ordering of the variable elimination will affect the running time of the nonserial dynamic programming algorithm, because fill is order dependent. DP (Q) shall refer to the nonserial dynamic programming algorithm with the variables numbered according to Q.

The process may be summarized as:

ALGORITHM DP (Q)

1. Renumber the variables according to ordering Q.
2. For $i = 1$ until n eliminate x_i.

The appendix explores the properties of DP.

4. Graph-theoretic definitions. Considerable development is needed before we define our class of algorithms. We adopt the *interaction graph* of [2] as a natural way to exhibit the interaction pattern of variables in the objective function. Figure 1 will illustrate the definitions. The graph is denoted $G = (V, E)$. It has a vertex for each variable; vertices x and y are adjacent (denoted $x \sim y$) if and only if they appear together in a term of f (see Fig. 1). G, $D_1, \cdots, D_n$ and Q determine the number of comparisons performed by DP(Q). The actual values of f_i are irrelevant.

Henceforth, "sets" will refer to sets of variables, and the words "vertex" and "variable" will be used interchangeably. Let S_1, S_2 and S be sets of vertices, and let v be a vertex.

$S_1 \sim S_2$ (read "S_1 is *adjacent* to S_2") if there exists a vertex v_1 in S_1, and a vertex v_2 in S_2, such that $v_1 \sim v_2$. $v \sim S$ means $\{v\} \sim S$. By convention, $v \sim v$.

Nb (S) (read "the neighbors of S") $= \{v \notin S | v \sim S\}$. (In Fig. 1, Nb $(\{x_1, x_2\}) = \{x_4\}$ and Nb$(\{x_2, x_3\}) = \{x_1, x_4, x_5\}$.)

Int (S) (read "the interior of S") $= \{v \in S|$ all vertices adjacent to v are in $S\}$.

Bo (S) (read "the boundary of S") $= S - \text{Int}(S)$.

$S' = \{v | v \sim S\} = S \cup \text{Nb}(S)$. Sets created by the prime operator are called "primed" (e.g., S').

S_1 and S_2 are *nested* if and only if one is a subset of the other.

Consider the subgraph of the interaction graph induced by the vertices $\{x_1, x_2, \cdots, x_j\}$. The subgraph's connected component which contains x_j is denoted "C_j." In Fig. 1, $C_2 = \{x_1, x_2\}$ and $C_3 = \{x_3\}$. The variables in h_j^* are exactly the variables in Nb (C_j). (This result appears in [2, p. 33], in a different notation.)

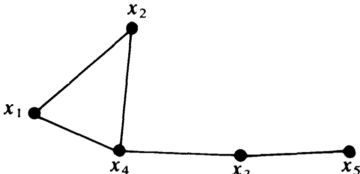


FIG. 1. *Interaction graph of* $f = f_1(x_1, x_2, x_4) + f_2(x_3, x_4) + f_3(x_3, x_5)$.
Int $(\{x_1, x_2, x_4\}) = \{x_1, x_2\}$;
Bo $(\{x_1, x_2, x_4\}) = \{x_4\}$;
Nb $(\{x_1, x_2, x_4\}) = \{x_3\}$;
$\{x_1, x_2, x_4\}' = \{x_1, x_2, x_3, x_4\}$;
$C_1 = \{x_1\}$, $C_2 = \{x_1, x_2\}$, $C_3 = \{x_3\}$, $C_4 = \{x_1, x_2, x_3, x_4\}$;
$C_5 = \{x_1, x_2, x_3, x_4, x_5\}$.

5. Comparison algorithms. An *assignment* to a set S assigns a value to each variable in S, and may be denoted $A(S)$. Two assignments $A_1(S)$ and $A_2(S)$ are *comparable* if identical values are assigned to all vertices in Bo (S), i.e., to all variables which appear in a term containing unassigned variables. $A_1(S_1)$ *extends* $A_2(S_2)$ if $S_2 \subseteq S_1$ and A_1 agrees with A_2 at each variable in S_2.

Given an assignment $A(S)$, define the *objective-value* of $A(S)$, [denoted $f(A(S))$] as $\sum (f_i(A(X_i)))$, where the sum is over terms satisfying:

(i) $X \subseteq S$ (i.e., every variable in X_i is assigned a value).

(ii) $X_i \not\subset \mathrm{Bo}\,(S)$. (Terms using only boundary variables are omitted because for comparable assignments, these variables will be assigned identical values, and hence will not affect the outcome of the comparison.)

Comparisons. Given a set of variables and an objective function f, a *comparison* compares the objective-values of two "comparable" assignments. The *loser* of a comparison is the assignment with the inferior (numerically higher) objective-value. Ties are broken lexicographically. For comparable assignments, Bellman's principle of optimality implies that any extension of the loser will be inferior to the corresponding extension of the winner. The optimum of each variable assignment cannot extend the loser of any comparison. A comparison of assignments to S will be denoted $A_1(S):A_2(S)$, or $A_{\mathrm{win}}(S):A_{\mathrm{lose}}(S)$. S is called the *carrier* of the comparison. $\mathrm{Int}\,(S)$, the set where comparable assignments may differ, will be called the *arena*.

The decision tree model will be used to decide "comparison algorithms." A comparison algorithm is designed for a fixed set of variables, with fixed domains, and a fixed interaction graph. The input consists of the term domains X_j (consistent with the interaction graph) and a term value for each assignment to the variables.

A comparison algorithm is a rule for choosing comparisons to perform. As a result of each comparison, all extensions of the loser are struck from a (conceptual) list of {all total assignments}. (Our model ignores the actual bookkeeping for strikeoffs.) The algorithm halts when only the optimal assignment survives all strikeoffs. For a fixed objective function, a collection of comparisons which strikes off every suboptimal total assignment is called a *verification*; it verifies the optimality of the survivor.

Henceforth, *collection* shall mean "collection of comparisions." The *cost* of any collection a is the number of comparisons. The *cost of an algorithm* on a function f is the cost of the verification produced for f. The appropriateness of this cost measure is addressed in later sections.

Formally, given an interaction graph G and the domain a *comparison algorithm* is a binary decision tree such that:

(i) The two edges descending from each internal node of the tree represent two assignments which are comparable for functions with graph G. The node itself represents a comparison whose result determines the choice of downward edge.

(ii) The comparisons along any root-leaf path supply a verification.

DP actually refers to a class of algorithms; to define a specific algorithm it is necessary to specify the variable ordering, interaction graph and domains of the variables. DP algorithms may be identified with comparison algorithms as follows: DP eliminates variables and compensates by altering terms of the objective function. [2, p. 33] shows that the value for h_j^* on an assignment to $\{x_j\} \cup \mathrm{Nb}\,(C_j)$ is equal to the original objective f on the best extension of A to C_j'. In this sense, the operation of DP on any problem is isomorphic to a comparison algorithm.

6. Main results.

Definition. The sets S_1 and S_2 are said to *overlap* if (i) S_1 and S_2 are not nested, and (ii) some term $f_i(X_i)$ is included in the calculations of objective-values for both S_1 and S_2.

Lemma 1.1. *T_1' and T_2' overlap if and only if they are not nested, and $T_1 \sim T_2$.*

Lemma 1.1 and many other simple results about graphs and overlap will be needed. Notation like G1 or O2(i), refers to a list of such results at the start of the appendix. Lemma 1.1 appears there as property O2(i).

We are principally concerned with nonoverlapping algorithms.

DEFINITION. Two comparisons *overlap* if their carriers overlap. A collection of comparisons, (also a comparison algorithm or a verification) is called *overlapping* if two of the comparisons overlap. Otherwise the collection, algorithm or verification is called *nonoverlapping.*

Consider a family of sets $\{T_1, T_2, \cdots\}$ such that any two sets are nested or disjoint (see Fig. 2). We define "innermost-out" orderings on the variables, as follows: Reordering if necessary, assume $T_i \subset T_j$ only if $i<j$. Now choose *any* ordering of $\{x_1, \cdots, x_n\}$ such that for every i, variables of $T_1, \cdots, T_{i-1}$ precede variables of T_i not in the previous sets. If the family is {the arenas of a nonoverlapping collection K}, the ordering will be denoted $Q\langle K\rangle$.

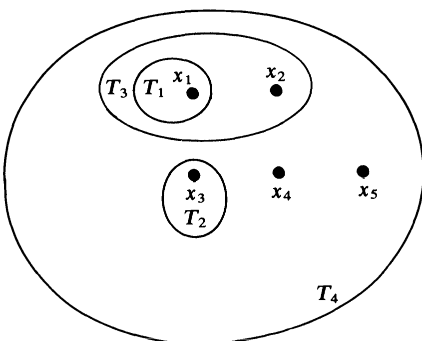


FIG. 2. *Suppose* $T_1=\{x_1\}$, $T_2=\{x_3\}$, $T_3=\{x_1, x_2\}$, $T_4=\{x_1, x_2, x_3, x_4, x_5\}$. *Two acceptable orderings are* $[x_1, x_2, x_3, x_4, x_5]$ *and* $[x_1, x_3, x_2, x_5, x_4]$.

LEMMA 1.2. *Suppose* $C_i \sim C_j$, *where* $i<j$. *Then* $C_i' \subseteq C_j'$.

Proof. Recall that C_i denotes the connected component of $\{x_1, \cdots, x_i\}$ which includes x_i. If $C_j \sim C_i$, then $C_j \cup C_i$ is a connected subset of $\{x_1, \cdots, x_j\}$. Since C_j is a connected component of $\{x_1, \cdots, x_j\}$, $C_i \subset C_j$. By G2 (in appendix), $C_i' \subseteq C_j'$. QED

DEFINITION. A carrier S is *reduced* if every vertex of S is in a term with an interior vertex, and Int (S) is connected. ([11] shows that there is no benefit to using nonreduced carriers.)

THEOREM 1 (Nonoverlapping sets theorem).

1(A). DP *is nonoverlapping, for every ordering* (*i.e., no verification produced by* DP *has overlapping carriers*).

1(B). *Let K denote a collection with reduced carriers. K is nonoverlapping if and only if each carrier of K is a carrier of* DP $(Q\langle K\rangle)$.

Proof. The carriers of DP are the sets C_i'. Lemma 1.2 and O2(i) imply that C_i' does not overlap C_j', so 1(A) is proved. If a collection has no pair of overlapping carriers, all subcollections are nonoverlapping. Thus 1(A) implies the $(\Leftarrow)$ direction of 1(B).

The other direction is proved in the appendix.

Consider any objective function f with graph G and variable domains $D_1, \cdots, D_n$. Let Q^* denote the DP elimination ordering which minimizes the number of comparisons performed. (Recall that the number of comparisons performed by DP for a function f depends only on $G, |D_1|, \cdots, |D_n|$ and Q^*.) This leads to the following result, proved in the appendix.

THEOREM 2 (Optimality theorem). *For every such function f,* DP (Q^*) *makes the fewest comparisons of any nonoverlapping comparison algorithm.* (Q^* denotes the optimal elimination ordering.)

DP seems to be the most useful nonoverlapping algorithm for the general optimization problem. The bookkeeping overhead for each comparison is reasonable. There is no known efficient algorithm to find the optimal DP ordering Q^*, but [2] gives heuristics which seem to find near optimal orderings. For the other algorithms presented in [2], the variable ordering problem seems even less tractable.

7. Other implications.

(1) *Adaptive (i.e., nonoblivious) and nondeterministic nonoverlapping algorithms have no advantage over* DP. A nondeterministic comparison algorithm nondeterministically chooses comparisons to perform until a verification is produced. Each node of a nondeterministic decision tree has several pairs of downward edges; one of these pairs is chosen. The *cost* on f is the length of the best verification obtainable (i.e., the tree's shortest root-leaf path which verifies f). Nonadaptive algorithms are "oblivious" to the exact values of the functions. (See R4 in § 8.)

Such algorithms must still produce a proof of optimality, i.e., a nonoverlapping verification (denoted VER) and VER must include as many comparisons as DP ($Q\langle\text{VER}\rangle$).

(2) *The efficiency of* DP *for a particular problem can measure the complexity of the problem's interaction graph.* If all variables have the same size domain, a good measure of structural complexity is $\log_{|D|}$ (running time), which is nearly independent of $|D|$, but depends on the interaction graph and the elimination ordering [2]. Loosely, if the system consists of local clusters of interacting variables, connected in a skinny, nearly treelike pattern, then DP will perform very well. A ladder network or star network is appropriately skinny; a square grid is not.

(3) *On some interaction graphs, an exponential lower bound can be obtained for* {*all nonoverlapping comparison algorithms*}. [5] shows that $|D|^n$ operations are required by DP on any function whose interaction graph is an $(n \times n)$ square grid. Such functions take $O(2n^2|D|^2)$ space to specify, assuming one location is used for each value of each term f_j. Hence any nonoverlapping comparison algorithm must take time exponential in problem size, for any problem on such a graph. A complete graph derived from 2-variable terms f_j also yields an exponential bound.

(4) *Familiar problems.* Many of the familiar NP-complete problems (e.g., satisfiability or set cover) can be formulated as instances of our optimization problem. Our theorem then provides a lower bound on nonoverlapping comparison algorithms for those problems.

8. Overlapping and nonoverlapping algorithms. This section provides a qualitative framework for comparing nonoverlapping algorithms, and compares out results with the results in [2]. One reason to restrict consideration to nonoverlapping algorithms is that they are easier to implement. We first consider two typical bookkeeping operations:

(a) Determine whether $A_1(T_1')$ and $A_2(T_2')$ agree on $T_1' \cap T_2'$.

Assuming the test at (a) was positive, let A_{union} denote the natural union of A_1 and A_2.

(b) Determine $f(A_{\text{union}}(T_1' \cup T_2'))$.

These operations are easy if T_1' and T_2' do not overlap. For operation (a), $A_1(\text{Bo}\,(T_1'))$ and $A_2(\text{Bo}\,(T_2'))$ are sufficient information if and only if T_1' and T_2' do not overlap (by O2(iv)). These boundary assignments will generally be available, as they are also needed to determine comparability. Similarly, to compute the objective

value of the union, one adds the two constituents' objective values plus perhaps some additional terms defined on Bo $(S_1) \cup$ Bo (S_2).

In contrast, if T_1' and T_2' overlap, (a) requires that variables not on the boundary be checked. The value assigned a boundary variable is fixed over all assignments in each equivalence class of the comparability relation; nonboundary variables must be explicitly checked. Operation (b) requires that terms using these additional variables be computed, and also that terms duplicated in the objective values on T_1' and T_2' be subtracted from the sum. Thus, considerable additional information must be retained from "solved" subproblems if later assignments may overlap.

DP verifications constitute a small subset of all nonoverlapping verifications. To illustrate this, we will consider restrictions R0–R4 obeyed by verifications produced by DP:

R0. The carriers are exactly the sets C_j' (where the variables have been numbered according to some ordering Q).

R1. The carriers are a subcollection of the sets C_j' for some ordering Q. (For reduced carriers, Theorem 1B implies that R1 is equivalent to "nonoverlapping".)

A key feature of DP is that it does all possible useful comparisons for Nb (C_j) before moving to the next stage assignments at stage j. It is difficult to express this restriction for arbitrary comparison algorithms, since stages (and hence the sets Nb (C_j)) are not defined. We define a comparison $A_1(Z):A_2(Z)$ to be *local to S* if $Z \subseteq S$. Only comparisons local to S can strike off assignments to S. Define Fixed $(S) = \{v \in S|$ for every comparison $A_1 : A_2$ local to S, $A_1(v) = A_2(v)\}$. (Undefined values are taken to be equal.) An assignment $A(S)$ is called a *beatable* extension of $A(Z)$ if some other extension to S has a better objective value.

R2. If VER includes $A_1(S):A_2(S)$, all beatable extensions of A (Fixed (S)) are struck off. That is, VER finds the optimal extension of A (Fixed (S)) to S.

R3. If S is a carrier in VER, then all beatable extensions of A (Fixed (S)) are struck off, and this is done for *every* assignment to Fixed (S).

R4. The algorithm is nonadaptive; i.e., for each fixed interaction graph, it is oblivious to the numerical values of the terms. Its comparison strategy can be represented by a circuit with comparator units and fixed interconnections. See [13] for other information on nonadaptive algorithms.

Verifications produced by DP can be shown to obey restrictions R0–R4. The optimality theorem implies that DP (Q^*) produces a verification which is optimal among the class of all verifications obeying R1.

[2] defines two generalizations of DP, "elimination in blocks", which eliminates several vertices at once, and a complex scheme called "regular multilevel elimination." Both generalizations obey R1–R4. (Nonregular elimination violates R1, but [2, Thm. 5.7.4] (or our Lemma A2) implies that nonregular schemes are no better than regular ones.) Theorems in [2] show that there is a DP ordering which is only slightly higher in (number of objective function evaluations) than elimination in blocks or multilevel elimination.

Our optimality results are cleaner and stronger. First, by using (number of comparisons) as our measure of work, we can show that DP is *absolutely* optimal among our class of algorithms. (The difference between the measures is small; it stems from the fact that finding the best of $|D|$ assignments requires $|D|$ evaluations and $|D|-1$ comparisons.) More important, our class of algorithms is far wider: we have shown DP to be optimal among comparison algorithms obeying just restriction R1 (i.e., nonoverlapping).

9. Overlapping algorithms, DP and perfect elimination graphs. Elimination of a vertex x_i causes *fill* in the interaction graph if two of the neighbors of x_i were nonadjacent before the elimination. (They will be adjacent after the elimination.) An ordering Q is called a *perfect elimination* ordering if DP (Q) causes no fill. (Perfect elimination orderings were originally defined for elimination in systems of linear equations.) We will now compare DP with verifications *which may include overlap.* Let G denote any interaction graph.

THEOREM 3 (Perfect elimination theorem).

3(A). *If Q is a perfect elimination ordering for G, then for every function with graph G,* DP (Q) *produces a shortest verification.*

3(B). *If ordering Q is not perfect elimination for G, then for some function f^0 with graph G, there exists an overlapping verification shorter than* DP (Q).

The theorem is proved in the appendix. The function f^0 shown there for 3(B) is degenerate in a way which can be exploited only if overlap is used. When the graph is a square grid with n vertices on a side, DP must make $O(|D|^n)$ comparisons [5], but there will be an overlapping verification whose length is polynomial, $O(n^2|D|^4)$. Hence overlap can make a dramatic difference.

10. Remarks, open questions and future work. We conjecture that for every interaction graph, there are some hard functions whose best overlapping verifications are as long as those produced by DP. If this conjecture is true, then on most graphs, none of our comparison algorithms can run in less than exponential worst-case time.

Many problems have additional structure (e.g., feasibility constraints) which rule out many possible assignments. A generalization of the optimality theorem provides some insight. Among nonoverlapping collections which strike off all remaining suboptimal assignments, some subcollection of the comparisons made by DP is smallest. The dominance theorem in the appendix contains more details.

The practical meaning of this optimality result is not clear. DP is fairly easy to implement, while it may be impractical to determine the appropriate subcollection, or to bookkeep the strikeoffs. [2] suggests one way of extending DP to handle feasibility constraints.

A decomposition procedure can be loosely defined as an algorithm which works by "solving" subsystems containing only a subset of the variables, and replacing these subsystems by an equivalent but simpler black box. To forbid later computations from looking inside this box, we require that solved subsystems be nested or effectively disjoint (here, nonoverlapping) and general properties applicable to other nonoverlapping "disjoint decomposition" algorithms (e.g., [3], [6]).

It would also be interesting to abstract properties of subsystems, boundaries and overlap. Perhaps our detailed arguments about overlap would be clearer in a setting more general than graphs (just as greedy arguments are simpler in matroids).

Some technical improvements might be made in the model [15] shows fully solving is always optimal if we measure (number of comparisons) + (number of concatenations of partial assignments), a more natural complexity measure than (number of comparisons).

We ruled out combining information from several comparisons to strike off assignments not struck off by any single comparison. Such combinations can give additional strikeoffs, but it is not known whether such operations can improve on DP. However, lower bounds for algorithm classes much larger than {comparison algorithms} will not be exponential. [7] and [8] show that for every objective function there exists "a linear proof" (i.e., a verification in a broader proof system) which uses only O

(number of term entries) comparisons. These short linear proofs cannot be found in polynomial time unless $P=NP$.

Appendix.

Notation.

$\subseteq$—"is a subset of"; $\subset$—"proper subset".

$S_1 \mathrm{I} S_2$—"S_1 and S_2 have nonempty intersection".

$S_1 \mathrm{O} S_2$—"S_1 overlaps S_2".

$\in$—"is a member of".

A slash through I or O will denote negation.

$\{x_1\}$ and $\{x_1\}'$ will often be written without set brackets (e.g., x_1').

A brief review of definitions follows.

A *loser* is a partial assignment which loses a comparison in the collection K being considered. A set is called a *carrier* or *arena* (in K) if it is the carrier (arena) of a comparison in K. Generally, T and T_i will be used to denote potential arenas; S, S_i, T' and T_i' will denote potential carriers. Z will denote any kind of set. C_j denotes the connected component of the induced subgraph on $\{x_1, \cdots, x_j\}$ which includes x_j. The connected components of $C_j-\{x_j\}$ will be denoted $\{C_{j1}, C_{j2}, \cdots\}$. Renumbering if necessary, we assume DP always eliminates variables in the order $x_1, \cdots, x_n$.

We hope to clarify our proofs by gathering all the graph-theoretic results here. G1, etc. refer to simple graph theoretic facts, and O1, etc. refer to facts concerning overlap. All these facts are proved in [11].

Graph theoretic facts.

G1. For any j, let $C_{j1}, C_{j2}, \cdots, C_{ji}$ denote the connected components of $C_j-\{x_j\}$. Then $(\bigcup_{\{i\}} \mathrm{Nb}\,(C_{ji})) \subseteq \mathrm{Nb}\,(C_j) \cup \{x_j\}$.

G2. $T_1 \subseteq T_2 \Rightarrow T_1' \subseteq T_2'$.

G3. Suppose $T_1 \mathrm{I} T_2$, $T_1 \not\subseteq T_2$, and $T1$ is connected. Then there is a vertex $x \in T_1 - T_2$ such that $x \in \mathrm{Nb}\,(T_2)$.

Overlap rules.

O1. The following are all equivalent: (i) $T_1 \sim T_2$; (ii) $T_1' \mathrm{I} T_2$; (iii) $T_1 \mathrm{I} T_2'$; (iv) $T_1' \cap T_2' \not\subseteq \mathrm{Bo}\,(T_1') \cap \mathrm{Bo}\,(T_2')$; (v) some term is duplicated in objective values computed on T_1' and T_2'.

DEFINITION. T_1' and T_2' are said to *collide* if the above conditions hold:

O2. T_1' overlaps T_2' if and only if $T_1' \not\subseteq T_2'$ and $T_2' \not\subseteq T_1'$ and T_1' collides with T_2'. ("O2(iii)" shall mean O2 with case (iii) from O1.)

O3. Let $\{T_1', T_2', \cdots\}$ be a family of (distinct) nonoverlapping sets, such that every vertex is in some T_i, and $T_i = \mathrm{Int}\,(T_i')$. Then, for each v, there is a unique smallest T_i which contains v.

O4. If $S_1 \subset S_2$ and $S_1 \mathrm{O} Z$ and $S_2 \not{\mathrm{O}} Z$ then $Z \subset S_2$.

THEOREM 1 (Nonoverlapping sets theorem).

1(A). DP *is nonoverlapping, for every ordering.*

1(B). *Let K denote a collection with reduced carriers. K is nonoverlapping if and only if each carrier of K is a carrier of* DP $(Q\langle K\rangle)$.

Proof. 1(A) and the ($\Leftarrow$) direction of 1(B) were proved in the text. We prove ($\Rightarrow$) for 1(B).

Let T denote an arena in K and T' a carrier. x_H will denote the highest numbered vertex in T. T is a connected subset of $\{x_1, \cdots, x_H\}$, and C_H is defined as the connected component of $\{x_1, \cdots, x_H\}$ containing x_H, so $T \subseteq C_H$. To prove the theorem we need only show $C_H \subseteq T$.

If $C_H \not\subseteq T$, then let x_p be the vertex (existent by G3) which is simultaneously in $\mathrm{Nb}\,(T)$ and in C_H. Let T_1 denote the unique (O3) smallest arena of K containing x_p.

T' and T_1' are both carriers and thus do not overlap, but are adjacent (at x_p); since $x_p \notin T$, by O2(i), we get $T \subset T_1$. Hence T_1 must have followed T in the ordering of sets which produced the variable ordering. No smaller arena contained x_p, so all vertices of T must have preceded x_p. Hence $H < p$, which contradicts the definition of H. QED

We now consider the properties of the comparisons associated with DP for some fixed function f and the elimination ordering $1, 2, \cdots, n$. We say an assignment $A(C_j')$ is *beatable at stage j* if $A(C_j')$ is a beatable extension of $A(\text{Nb}(C_j))$. This definition is consistent with the use of "beatable" in R2 of § 8.

We now assume DP uses ordering $x_1, \cdots x_n$, and let j denote an arbitrary stage, A an arbitrary assignment.

THEOREM A1 (Properties of DP).

DP1 (Strikes off all losers). *Stages* 1 *through j of* DP *strike off all assignments beatable at stage j.*

DP2 (Acts at first opportunity). *If $A(C_j')$ is compared at stage j, then it was not beatable at any earlier stage. Equivalently, if $A(C_j')$ is beatable at stage j, then no extension of $A(C_j')$ is compared at any later stage.*

Let A^*_{lose} denote the best total assignment extending $A_{\text{lose}}(C_j')$.

DP3 (Nonredundant). *Suppose some comparison $A_{\text{lose}}(C_j'):A_{\text{win}}(C_j')$ is omitted from* DP. *Then A^*_{lose} is not struck off by any remaining comparisons.*

Proof of DP1 *and* DP2. Inductive hypothesis: Every assignment A beatable at stages before j has been struck off at some stage $1, 2, \cdots$, or $j-1$.

Now at stage j, for each $A(\text{Nb}(C_j))$, DP compares the assignments $A_v(C_j')$ obtained as follows: For each $v \in \text{Domain}(x_j)$, extend $A(\text{Nb}(C_j))$ to $\{\text{Nb}(C_j) \cup \{x_j\}\}$ by assigning value v to x_j. Let $C_{j1}, C_{j2}, \cdots, C_{jt}$ denote the connected components of $C_j - \{x_j\}$. By G1, all vertices of $(\bigcup_{\{t\}} \text{Nb}(C_{jt}))$ have been assigned values. By the inductive hypothesis, all beatable assignments to each C_{jt} have been struck off, and only the single optimal extension to C_{jt}' survives. Using the surviving assignment for each C_{jt}, we can assign values to all the vertices of $C_j - \{x_j\}$. Denote this assignment $A_v(C_j')$. Clearly, no assignments so formed could be struck off at an earlier stage, so DP2 holds.

DP compares the assignments A_v ($v \in \text{domain}(x_j)$). To show DP1, consider any assignment $A(C_j')$ which is beatable at stage j, but is not struck off at stages $1, \cdots, j-1$. Let z denote $A(x_j)$. A must be the assignment A_z, since by the inductive hypothesis and the assumption that A was unbeaten at previous stages, A agrees on each C_{jt}' with the optimal extension of $A(\text{Nb}(C_{jt}))$ to C_{jt}'. But then $A = A_z$ was compared with the best extension of $A(\text{Nb}(C_j))$ (which must have been A_v for some other v). Hence $A(C_j')$ was struck off at stage j. DP1 is proved.

LEMMA A1. *Given a set T_1 and assignment $A(T_1')$, let A^* denote the best total assignment which extends $A(T_1')$. For a comparison with arena T_2 to strike off A^*, it is necessary that T_1' and T_2' collide.*

Proof. Take the assignments $A_{\text{win}}(T_2')$ which beat A^*, and form a total assignment A^0 from $A_{\text{win}}(T_2')$ and $A^*(\{v | v \notin T_2'\})$. If T_1' and T_2' do not collide, then $T_1' \cap T_2'$ B Bo (T_2') (by O2 (iv)). But for comparability, A_{win} and A^* must assign the same values to Bo (T_2') so $A^0(T_1') = A(T_1')$. Thus A^0 is an extension of $A(T_1')$ which is superior to A^*, a contradiction.

Proof of DP3. Suppose A^*_{lose} were struck off by a remaining comparison on a set denoted T'. It is clear from the definition of DP that $T' \neq C_j'$. By Lemma A1, T' collides with C_j'. Now if C_j' and T' were nested and unequal, by DP2, the comparison on the larger set would not take place. Hence, T' and C_j' must overlap. This contradicts the fact that DP is nonoverlapping. QED

DEFINITION. Given 2 collections K_{new} and K, K_{new} *dominates* K if $|K_{new}| \leqq |K|$, and K_{new} strikes off every assignment which K strikes off. ($|K|$ is the number of comparisons in K.)

DEFINITION. A comparison (or subcollection) k_1 is *replaceable by* k_2 in a collection K if $K \cup \{k_2\} - \{k_1\}$ dominates K. k_1 is *uniformly replaceable* by k_2 if it is replaceable by k_2 in *every* collection K.

Dominance, "replaceable in K" and "uniformly replaceable" are transitive.

LEMMA A2. *Suppose $A(S)$ extends $A(Z)$, and that k_S and k_Z denote comparisons lost by $A(S)$ and $A(Z)$ respectively. Then k_S is uniformly replaceable by k_Z.*

Proof. Every assignment struck off as an extension of the loser $A(S)$ is also an extension of the new loser $A(T)$.

An assignment $A(S)$ is 1-*optimal* if either x_1 is unassigned, or $A(x'_1)$ is an unbeatable extension of $A(\text{Nb}(x_1))$. DP1 implies that the first stage of DP strikes off exactly those assignments to x'_1 which are not 1-optimal. A collection is 1-optimal if every assignment evaluated is 1-optimal.

LEMMA A3. *Any collection such that no comparison overlaps x'_1 is dominated by a collection $K_{new} = K^1 \cup K^2$ such that K^1 includes only comparisons from stage* 1 *of* DP, and K^2 *is* 1-*optimal.*

Proof. K_{new} is obtained by the following algorithm:

For each comparison $A_{lose} : A_{win}$

If A_{win} is not 1-optimal then reassign x_1 as the best value to extend $A(\text{Nb}(x_1))$.

If A_{lose} is not 1-optimal then replace this comparison by the DP comparison which strikes off A_{lose}.

The replacement of A_{win} makes all winners 1-optimal, but does not change the identity of losers. The next replacement removes comparisons whose losers are not 1-optimal, and in place leaves comparisons from stage 1 of DP. By Lemma A2, the new collection dominates the old. QED

1-optimal assignments correspond to assignments for the objective function (denoted f^2) obtained by eliminating x_1. The elimination replaces h_1 by h_1^*; h_1^* simply assumes that the 1-optimal value of x_1 is used. In this way, K^2 is isomorphic to a collection (denoted $K^2(f^2)$) of comparisons for f^2. Similarly, stages $2, 3, \cdots n$ of DP may be identified with the DP algorithm of f^2 (denoted $\text{DP}^2(f^2)$). The following lemma now follows from the definitions.

LEMMA A4. (i) *If K^2 strikes off all* 1-*optimal assignments for f then $K^2(f^2)$ is a verification for f^2.*

(ii) ($x_1, \cdots, x_n$ *is an admissible ordering for the original collection*) $\Rightarrow$ ($x_2, \cdots, x_n$ *is an admissible ordering for* K^2).

THEOREM A2 (Optimality theorem). *For every function f,* DP (Q^*) *makes the fewest comparisons of any nonoverlapping comparison algorithm.* (Q^* denotes the optimal elimination ordering.)

Proof. We use induction on the number n of variables. The result clearly holds for $n = 1$. For larger n, consider any nonoverlapping verification VER. Assume without loss of generality that generality that ($Q\langle\text{VER}\rangle = x_1, \cdots, x_n$ Perform the construction of Lemma A3. K^1 will include all stage 1 comparisons. By the inductive hypothesis, $K^2(f^2)$ is dominated by $\text{DP}^2(f^2)$, which corresponds $1-1$ with the comparisons of stages $2, \cdots, n$. Thus $|\text{DP}(x_1, \cdots, x_n)| \leqq |K^1 \cup K^2|$, which was constructed to dominate VER. Hence DP can provide a shortest nonoverlapping verification for functions of n variables, and the induction is completed. QED.

The dominance theorem below is a stronger form of the optimality theorem. We have not yet found a simple proof; a long proof is given in [11].

THEOREM A2′ (Dominance theorem). *For every nonoverlapping collection K, there is a subcollection of* DP $(Q\langle K\rangle)$ *which dominates K.*

THEOREM A3. *The verification produced by* DP (Q^*) *is a shortest verification (overlap permitted) if and only if Q^* is a perfect elimination ordering.*

Proof of Theorem 3. (Θ) Consider the following function and the following (overlapping) sequence of comparisons. We shall show that the comparisons form a verification which is shorter than DP.

The function: Suppose each variable's domain consists of $\{1, 2, \cdots, |D|\}$, where $|D| \geqq 2$. Suppose the interaction graph was obtained from functions denoted $f_j(X_j)$, $j = 1, 2, \cdots$. For each j, define a term of f^0 to be $7_{xj\in X_j} x_j$. f^0 is degenerate in that for each j, the optimal value of x_j is 1, regardless of the assignment to any other variables.

The verification. As usual, assume the ordering is $(x_1, x_2, \cdots)$. Define $\mathrm{Nb}^+(j)$ ($\mathrm{Nb}^-(j)$) to be the neighbors of x_j which follow (precede) x_j in the elimination ordering. The comparisons at stage j are formed as follows: For each assignment to $\mathrm{Nb}^+(j)$, keeping $\mathrm{Nb}^-(j)$ assigned 1, compare all values for x_j. $x_j = 1$ will always win. Given an assignment A, let p be the lowest index such that x_p is assigned a value other than 1. Then A will be struck off at stage p, when $A(C'_p)$ loses to the assignment obtained from A by assigning 1 to x_p. Hence the collection is a verification.

Efficiency. For all i, $\mathrm{Nb}^+(i) = \mathrm{Nb}(\{x_i\}) - \mathrm{Nb}^-(i) \subseteq \mathrm{Nb}(C_i)$. Now suppose Q is not perfect elimination, so some fill edge (x_j, x_w) is added during the elimination. Assume $j < w$. Then by [2, p. 33], $x_w \in \mathrm{Nb}(C_j)$ and $x_w \notin \mathrm{Nb}^+(j)$, so $|\mathrm{Nb}^+(j)| < |\mathrm{Nb}(C_j)|$. The work at any stage i of the overlapping scheme is $(|D|-1)^*|D|^{**}|\mathrm{Nb}^+(i)|$ comparisons, while DP performs $(|D|-1)^*(|D|^{**}|\mathrm{Nb}(C_i)|)$. Thus at no stage does the overlapping verification do more comparisons than DP, and at stage j it does less.

Proof of 3(B). First we need a lemma.

LEMMA A5. *Suppose x_1 can be eliminated with no fill. Then no set S' can overlap x'_1.*

Proof. If x'_1 collides with S', then by O1 (ii), x_1 is adjacent to some vertex $v \notin S$. Now since x_1 may be eliminated with no fill, all vertices of x'_1 must be mutually adjacent, and hence $x'_1 \subseteq v'$. But $v' \subseteq S'$, so $x'_1 \subseteq S'$, and there is no overlap. The lemma is proved.

Now apply the construction and inductive proof from the optimality theorem. That is, produce K^1 and K^2 and invoke the inductive hypothesis to replace K^2 by stages $2, \cdots, n$ of DP. We omit the details. QED

Conjecture. For *every* function, the best overlapping verification requires at least as many comparisons as the scheme described in 3(A) (which is a verification only because $x_i = 1$ is always the winner for the degenerate function f^0). That is, overlap will do no better than remove the cost due to fill.

REFERENCES

[1] R. E. ALLSOP, *Selection of offsets to minimize delay to traffic in a network controlled by fixed-time signals*, Transportation Sci., 2 (1968), pp. 1–13.

[2] U. BERTELE AND F. BRIOSCHI, *Nonserial Dynamic Programming*, Academic Press, New York, 1972.

[3] R. M. KARP, *Functional decomposition and switching circuit design*, J. Soc. Ind. Appl. Math, 11 (1963), pp. 693–718.

[4] A. KORSAK, *A proposed algorithm for globally optimal nonlinear-cost multidimensional flows in networks and some special applications*, presented at Fifth International Symposium on Traffic and Transportation, Berkeley, CA, June, 1971.

[5] A. MARTELLI AND U. MONTANARI, *Dynamic programming schemata* in Automata, Language and Programming, 2nd Colloquium, University of Saarbruecken, Lecture Notes on Computer Science, 14, Springer-Verlag, New York, 1974.

[6] H. MINE, K. OHNO AND M. FUKISHIMA, *Multilevel decomposition of nonlinear programming by dynamic programming*, J. Math. Anal. Appl., 53 (1976), pp. 7–32.

[7] J. MORAVEK, *A note upon minimal path problem*, J. Math. Anal. Appl., 30 (1970), pp. 702–717.

[8] M. RABIN, *Proving simultaneous positivity of linear forms*, JCCS, 6 (1972), pp. 639–650.

[9] A. ROSENTHAL, *Computing the reliability of complex networks*, SIAM J. Appl. Math., 32 (1977) pp. 384–393.

[10] A. ROSENTHAL, *Decomposition algorithms for probabilistic circuits and fault trees*, submitted for publication.

[11] A. ROSENTHAL AND V. JECZEN, *Additional proofs for dynamic programming is optimal for nonserial optimization problems*, working paper, available from the authors.

[12] B. ROTHFARB, H. FRANK, D. ROSENBAUM, K. STEIGLITZ AND D. KLEITMAN, *Optimal design of offshore natural-gas pipeline systems*, Oper. Res. 6 (1970), pp. 992–1020.

[13] A. BORODIN, M. J. FISCHER, D. G. KIRKPATRICK, N. A. LYNCH AND M. TOMPA, *A time-space tradeoff for sorting and related non-oblivious computations*, Proc. 20th IEEE Symposium on Foundations of Computer Science, 1979, pp. 312–318.

[14] A. ROSENTHAL AND P. HELMAN, *A general theory of discrete dynamic programming*, in preparation.

[15] P. HELMAN, *A new theory of dynamic programming*, Ph.D. Thesis, University of Michigan, Ann. Arbor, 1981.

SIAM J. COMPUT.
Vol. 11, No. 1, February 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1101-0005 $01.00/0

ON THE EXPECTED PERFORMANCE OF SCANNING DISKS*

E. G. COFFMAN, JR.† AND MICHA HOFRI‡

Abstract. This paper describes and analyzes the SCAN policy, used to schedule read/write requests at a moving-arm disk device, when fast response over the entire disk area is at a premium. An analysis is presented which handles precisely the dependence structure between queues accumulated at different cylinders. The arrival process of requests to each cylinder is assumed Poisson and homogeneous in time. A relatively efficient algorithm for evaluating numerically the mean waiting time at each cylinder is presented and its complexity analyzed. We discuss further extensions intended to capture additional details of realistic situations. These include distributed record lengths, skipping unreferenced cylinders and letting successive arrivals' target cylinders be dependent variables.

Key words. disk system performance evaluation, disk SCAN policy analysis, movable-arm disk system analysis

1. Introduction. The quality of service provided by a computing system depends largely, if not critically, on the techniques it uses to handle its secondary memory requirements. Thus, it is important to obtain a precise account of the performance of these memory devices, and its dependence on the physical characteristics and methods of operation. This paper presents such an account for a specific case—a disk-like device used under the so-called SCAN policy.

The term "disk-like device" is intended to aggregate semirandom access devices, on which the recording/reading mechanism can assume a limited number of positions. From each position, only a section of the device can be serviced continuously without further mechanical motion, excluding the rotation of surfaces. The following analysis is expressed in disk terminology, which we assume is familiar. If further specification of these devices is desired, references [7]–[8] may be consulted.

A disk is a nonrandom access device in the sense that it must perform a relatively long operation, called a seek, between accesses to distinct cylinders. Thus the time to process a batch of requests may depend on the order of service. The purpose of a scheduling policy is to devise a processing order which optimizes a suitable measure of performance. It is natural that the waiting time of a request is the variable most often chosen as the basis of such measures, and thus its calculation is the main objective of the analyses we present.

The simplest scheduling policy to implement, in terms of the required data structures and hardware, is a linear first-come-first-served (FCFS) queue. It is easily shown and intuitively clear that this results in many more, and relatively longer seeks than alternatives require. Any improvement over FCFS requires keeping track of the requests according to their target cylinders. Due to the relatively long time required to perform a seek, most of the improved policies schedule all the requests for the cylinder currently under the read/write heads before all other pending requests. Under these conditions the scheduling of requests reduces to selection of cylinders. Scheduling to reduce rotational delays, in the manner that is done for drums, is outside the scope of this paper.

---

* Received by the editors September 29, 1978, and in final revised form February 18, 1981. This paper is a revision of an earlier paper presented at the International Symposium on Performance Evaluation, Stresa, Italy, 1977.

† Bell Laboratories, Murray Hill, New Jersey 07974.

‡ The Technion, Haifa, Israel.

There are two basic policies that have been adopted for the solution of this decision problem. The first is SSTF, an acronym for shortest-seek-time-first. Under this policy, after all the requests to the current cylinder are served, the arm performs a seek to the nearest cylinder that has a waiting request. Under certain conditions this policy is credited with achieving the shortest overall mean waiting time, although no successful analysis has yet appeared. At the same time, however, the variance of this time may be undesirably large when the input of requests is inhomogeneous in time. These issues are further discussed in [4].

The second is SCAN. Under this policy the motion of the arm is organized so as to reduce the variance of the waiting time when compared with SSTF. The sacrifice made for this improvement is an increase in the mean waiting time. More specifically, at any given time instant the arm is in one of two modes, "in", or "out". When a seek is required in the first mode, the arm moves toward the disk spindle until a cylinder with pending requests is encountered. When no such cylinder exists, the mode of the arm is changed and its direction of motion reversed. Arm movement in the out mode is similar; thus the arm performs a shuttle service across the disk.

The SCAN policy and certain of its variants have been studied extensively in the literature. Analyses based on approximations have been presented in [2], [7], [8] and exact analyses of idealized models have been attempted in [1], [5]. However, in both of [1], [5] errors have been found which also render these results approximate at best.

In the sequel we shall provide an exact analysis of the basic SCAN policy which is based on results of Eisenberg [3]. This paper improves on the analysis in [3] in two important respects: The derivation of all the major equations is via probabilistic reasoning, and we propose a computational procedure to evaluate the quantities of interest which is a major improvement over the one outlined in [3]. The next section describes the mathematical model along with the necessary background results, and in § 3 expressions for mean waiting times are derived. In § 4 a procedure for computing numerical results is presented and its complexity analyzed. In the final section conclusions are drawn and a number of open problems outlined. In the Appendix we collect the notation used in the paper.

2. The mathematical model and preliminary results. The scanning-disk model developed below is the appropriate specialization of a model [3] in which multiple queues receive periodic service in fixed but arbitrary cycles. In particular, the disk is viewed as comprising M service points (cylinders) arranged along a line. At any point in time the server (arm) is either located at one of these points (possibly performing an I/O operation) or it is in motion between them (seeking). In this model, seeks are done to adjacent cylinders only and require a time units. Each I/O operation requires a constant, fixed amount of time, T, to complete. Note that we make no distinction here between reads and writes. Request arrivals for the mth cylinder constitute a homogeneous Poisson process with rate λ_m. The arrival processes to distinct cylinders are assumed independent of each other and the state of the system. The order of service at each cylinder is FCFS. Transition times between seek termination and beginning of service, as well as between services are assumed to be zero.

Elements of the model. We describe arm motion in terms of *stages*. The arm moves in cycles of stages numbered 1 to $2M-2$; the correspondence between the ith stage and cylinder m_i is given by

$$m_i = \begin{cases} i, & 1 \leqq i < M, \\ 2M-i, & M \leqq i \leqq 2M-2. \end{cases} \tag{1}$$

Note that only one stage corresponds to each of cylinders 1 and M. Otherwise, stages i and $2M-i$ correspond to the servicing of requests at cylinder m_i, and a cycle through the $2M-2$ stages corresponds to a complete scan of the arm across the disk and back.

The main process we investigate is $\vec{N}(t)$, the occupancies of the cylinder queues. The state of the disk and the collection of queues is observed at instants when a specification of the value of $\vec{N}(t)$ and the position of the arm gives a complete (i.e., Markovian) description of the disk facility. A value of $\vec{N}(\cdot)$ is denoted by an M-vector $\vec{n}=(n_1, \cdots, n_M)$. The specification of the position of the arm is explicit: We observe the state of the system at stage terminations and define $\beta^i_{\vec{n}}$ as the probability that immediately after a stage-i termination the queues are in state $\vec{n}$. These states describe an irreducible aperiodic Markov chain.

We state without proof the intuitive claim that the chain is recurrent when $\lambda=\sum_{m=1}^M \lambda_m < 1/T$, which is the same as requiring $\sum \rho_m < 1$, where $\rho_m = \lambda_m T$ is the traffic intensity of queue m. Consequently, the states of the system at stage terminations may be assumed to have the stationary probabilities $\beta^i_{\vec{n}}$. Note that the recurrence condition depends only on λ and T, and not on a. Indeed, the fraction of time the arm spends seeking vanishes in the limit $\sum \rho_m \to 1$. At the end of this section we will be able to comment further on this point, following (16).

In order to calculate the probabilities we find it expedient to consider an additional set of regeneration points, viz. stage beginnings, and define $\alpha^i_{\vec{n}}$ as the probability that immediately before a stage-i beginning the system is in state $\vec{n}$. The computational tools are probability generating functions (pgf's), e.g.,

$$\beta^i(\vec{z}) = \sum_{n_1=0}^{\infty} \sum_{n_2=0}^{\infty} \cdots \sum_{n_M=0}^{\infty} \beta^i_{\vec{n}} z_1^{n_1} z_2^{n_2} \cdots z_M^{n_M}, \tag{2}$$

with $\alpha^i(\vec{z})$ similarly defined. Note that in (2) the sum over n_{m_i} contributes only when $n_{m_i}=0$ by virtue of its being a stage-i termination state, and thus could be suppressed.

The probability distribution function (pdf) of the numbers of arrivals $(n_1, n_2, \cdots, n_M)$ to the M cylinders during a period with a pdf $F(\cdot)$ is given by

$$p(F; n_1, \cdots, n_M) = \int_{t=0}^{\infty} \frac{(\lambda_1 t)^{n_1}}{n_1!} \cdots \frac{(\lambda_M t)^{n_M}}{n_M!} e^{-\lambda t}\, dF(t). \tag{3}$$

If $L(\cdot)$ is the Laplace–Stieltjes transform (LST) corresponding to $F(\cdot)$ then the pgf of $p(\cdot\,;\cdot)$ is given by

$$\begin{aligned} p(F; \vec{z}) &= L(\lambda_1 - \lambda_1 z_1 \cdots + \lambda_m - \lambda_M z_M) \\ &\equiv \tilde{L}(\vec{z}). \end{aligned} \tag{4}$$

Note that $\tilde{L}(\cdot)$ has a vector argument.

Let $A(s)$ and $C(s)$ denote the LSTs of the seek and service periods, respectively, with $\tilde{A}(\vec{z})$ and $\tilde{C}(\vec{z})$ defined in analogy with $L(\vec{z})$. Since these periods are in fact constants we have

$$\tilde{A}(\vec{z}) = \exp\left(-a \sum_m \lambda_m (1-z_m)\right), \qquad \tilde{C}(\vec{z}) = \exp\left(-T \sum_m \lambda_m (1-z_m)\right). \tag{5}$$

Calculation of $\beta^i(\vec{z})$. Since the queues at stage beginnings comprise those requests that were there at the last stage termination plus those that arrived during the seek,

and since these two components are independent, we have

(6) $$\alpha^i(\vec{z}) = \begin{cases} \beta^{i-1}(\vec{z})\tilde{A}(\vec{z}), & i > 1, \\ \beta^{2M-2}(\vec{z})\tilde{A}(\vec{z}), & i = 1, \end{cases}$$

relating states "across" a seek.

To obtain an equation satisfied by the $\beta(\cdot)$ alone we need to relate the states of the system "across" a complete servicing of a stage. Let stage i of queue m_i start with k_{m_i} requests in the queue. If we restrict our view to this queue until it empties, the analysis of a standard M/G/1 system applies. Denote by $G_{m_i}(\cdot)$ the pdf of a busy period in such a queue and by B_{m_i} its LST. Then $B_{m_i}(\cdot)$ satisfies the equation

(7) $$B_{m_i}(s) = C(s + \lambda_{m_i} - \lambda_{m_i} B_{m_i}(s)).$$

We use this relation later in § 4.

A "compound" busy period, which begins with k_{m_i} customers present, is the sum of k_{m_i} i.i.d. such "simple" busy periods. Thus it has the pdf $G_{m_i}^{*k_{m_i}}(\cdot)$. Observing queue occupancies when this stage terminates we find for the number, h_i, of requests in queue m at stage-i termination

(8) $$h_i = \begin{cases} 0, & m = m_i, \\ k_m + c\,(\lambda_{m_i}; G_{m_i}^{*k_{m_i}}), & m \neq m_i. \end{cases}$$

On the right-hand side, for $m \neq m_i$, k_m is the number of customers at queue m when stage-i started, and is distributed according to $\alpha^i_{\vec{n}}$; $c(\lambda; F)$ is a variable distributed as the number of events counted by a homogeneous Poisson process with rate λ during a period with the pdf $F(\cdot)$. Adapting (3) we may write $p_{m_i}^{(i)}(G_{m_i}^{*k_{m_i}}; \cdot)$ for the joint pgf of this last variable over $M-1$ queues, excluding m_i; the superscript signifies the suppression. Thus

(9) $$\beta^i_{\vec{n}} = \sum_{\vec{k}} \alpha^i_{\vec{k}} p_{m_i}^{(i)}(G_{m_i}^{*k_{m_i}}; \vec{n} - \vec{k}),$$

where the sum is over all k such that $\vec{n} \geqq \vec{k}$ componentwise (excepting the m_ith component), and $\vec{n} = \vec{k}$ when $k_{m_i} = 0$. Substituting for $p^{(i)}(\cdot\,;\cdot)$:

$$\beta^i_{\vec{n}} = \sum_{\vec{n} \geqq \vec{k}} \alpha^i_{\vec{k}} \int_{t=0}^{\infty} e^{-(\lambda - \lambda_{m_i})t} \prod_{m \neq m_i} \frac{(\lambda_m t)^{n_m - k_m}}{(n_m - k_m)!} \, dG_{m_i}^{*k_{m_i}}(t)$$

where $\vec{k} \leqq \vec{n}$ should be construed as above. Multiplying by $\prod_{r \neq m_i} z_r^{n_r}$ on both sides we obtain, summing on all $\vec{n}$,

$$\begin{aligned} \beta^i(\vec{z}) &= \sum_{\vec{n}} \prod_{r \neq m_i} z_r^{n_r} \beta^i_{\vec{n}} \\ &= \sum_{\vec{k}} \alpha^i_{\vec{k}} \prod_{r \neq m_i} z_r^{k_r} \int_{t=0}^{\infty} e^{-(\lambda - \lambda_{m_i})t} \sum_{\vec{n} \geqq \vec{k}} \prod_{m \neq m_i} \frac{(\lambda_m z_m t)^{n_m - k_m}}{(n_m - k_m)!} \, dG_{m_i}^{*k_{m_i}}(t) \\ &= \sum_{\vec{k}} \alpha^i_{\vec{k}} \prod_{r \neq m_i} z_r^{k_r} \int_{t=0}^{\infty} \exp\Big(-(\lambda - \lambda_{m_i})t + \sum_{m \neq m_i} \lambda_m z_m t\Big) \, dG_{m_i}^{*k_{m_i}}(t). \end{aligned}$$

Performing the integration and noting that $\lambda - \lambda_{m_i} = \sum_{m \neq m_i} \lambda_m$, one gets

$$\beta^i(\vec{z}) = \sum_{\vec{k}} \alpha^i_{\vec{k}} \prod_{r \neq m_i} z_r^{k_r} B_{m_i}^{k_{m_i}}\Big(\sum_{m \neq m_i} \lambda_m (1 - z_m)\Big),$$

and summing over $\vec{k}$ we obtain

$$\beta^i(\vec{z}) = \alpha^i(\vec{z}^{(i)}), \tag{10}$$

where the superscript i over $\vec{z}$ means that the m_ith component is replaced by $B_{m_i}(\sum_{k\neq m_i}\lambda_k - \lambda_k z_k)$. Combining (6) with (10) we finally obtain

$$\beta^i(\vec{z}) = \beta^{i-1}(\vec{z}^{(i)})\tilde{A}(\vec{z}^{(i)}), \tag{11}$$

which is the basis for the numerical calculations we shall develop. This equation could also be used, as in [3], for the basis of a formal solution for the β functions, but as this solution is of limited utility for us, it will not be presented here.

Refining the chains. The chain embedded at stage terminations gives a description of the evolution of $\vec{N}(t)$ that is too "coarse" to define the waiting times of individual requests. To evaluate these, we embed in $\vec{N}(t)$ a finer chain defined at service beginning and completion epochs. These epochs are regeneration points for $\vec{N}(t)$, and thus its values there also constitute an ergodic Markov chain. Let $\pi_{i,\vec{n}}$ be the probability that a service termination occurs in stage-i, and the value of $\vec{N}$ just after the termination is $\vec{n}$.[1] Following the relation between $\beta^i_{\vec{n}}$ and $\alpha^i_{\vec{n}}$, we may relate $\pi_{i,\vec{n}}$ to the joint probability $\omega_{i,\vec{n}}$, of observing the system just before a service initiation in stage-i and at state $\vec{n}$. A relation similar to (6) between the pgf's of $\pi_{i,\vec{n}}$ and $\omega_{i,\vec{n}}$ is simple, since they are related "across" a single service duration. Hence

$$\pi_i(\vec{z}) = \frac{\omega_i(\vec{z})\tilde{C}(\vec{z})}{z_{m_i}}. \tag{12}$$

The remainder of this section is devoted to finding a relation between the $\pi_{i,\vec{n}}$ and $\beta^i_{\vec{n}}$. The analytical development in [3] will be replaced here by one that exploits somewhat more intuitive arguments.

We focus on the ith stage and consider observations made just after completions of stage-i services and seeks from stage $i-1$ to stage i, i.e., the union of those epochs that define transitions of the chains described by $\pi_{i,\vec{n}}$ and $\alpha^i_{\vec{n}}$. Next, suppose there have been K requests served by the system. For large K, the number of epochs at which state $\vec{n}$ occurred is approximately $\alpha^i(K)\alpha^i_{\vec{n}} + K\pi_{i,n}$, where $\alpha^i(K)$ was the number of stage-i beginnings.

Now consider the epochs just after the *beginnings* of stage-i services and seeks from stage i to stage $i+1$. These have a one-to-one correspondence with the set of epochs defined above, and correspond also to the union of the set of epochs of transition of the chains described by $\omega_{i,\vec{n}}$ and $\beta^i_{\vec{n}}$. But from the point of view of these latter epochs we have $\beta^i(K)\beta^i_{\vec{n}} + K\omega_{i,\vec{n}}$ as the expected number of epochs at which state $\vec{n}$ occurred, where $\beta^i(K)$ was the number of stage-i completions. Thus, dividing by K and taking the limit $K\to\infty$, we have by the law of large numbers

$$\gamma\beta^i_{\vec{n}} + \omega_{i,\vec{n}} = \gamma\alpha^i_{\vec{n}} + \pi_{i,\vec{n}}, \tag{13}$$

where $\gamma = \lim_{K\to\infty}\beta^i(K)/K = \lim_{K\to\infty}\alpha^i(K)/K$ is the limiting ratio of the number of stage-i visits to the number of requests served, and must be the same for all i, since each stage occurs once per cycle. In terms of generating functions

$$\beta^i(\vec{z}) + \omega_i(\vec{z}) = \gamma^i(\vec{z}) + \pi_i(\vec{z}). \tag{14}$$

Substituting for $\omega_i(\vec{z})$ from (12) we obtain

[1] The difference in notation between $\beta^i_{\vec{n}}$ and $\pi_{i,\vec{n}}$ reflects a difference in definition; in β^i_i, i specifies a conditioning event, but in $\pi_{i,\vec{n}}$, i is a part of the state descriptor.

$$\pi_i(\vec{z}) = \gamma C(\vec{z}) \frac{\alpha^i(\vec{z}) - \beta^i(\vec{z})}{z_{m_i} - C(\vec{z})}, \tag{15}$$

which, in conjunction with (6), gives us the desired relation between the $\pi_i(\vec{z})$ and $\beta^i(\vec{z})$. To evaluate γ we may proceed by the following brief expected-value argument.

In equilibrium, the average number served per cycle is given by λD where D is the average cycle length. Since for each i there is exactly one stage-i visit per cycle, we have $\gamma = 1/\lambda D$. Since the total seek time per cycle is $2(M-1)a$, we have for the average cycle time $D = \lambda DT + 2(M-1)a$. Hence,

$$D = \frac{2(M-1)a}{1-\lambda T} \quad \text{and} \quad \gamma = \frac{1-\lambda T}{2(M-1)a\lambda}. \tag{16}$$

The linear dependence of the average cycle duration D on the seek time a may seem strange when the limit $a \to 0$ is considered, as it would predict the vanishing of D regardless of ρ. This however merely represents the ability of the arm to make, in the limit $a \to 0$, an unbounded number of cycles during each "idle period" (idle in the sense that no requests are available); since the time between idle periods has a finite mean, D would be "biased away". The unboundedness of γ is then self-evident.

This effect also makes the term "utilization" a rather inappropriate term for $\rho = \lambda T$. In queueing models, one associates high or low values of with sluggish or prompt response. Here, λT merely denotes the fraction of the cycle time used for actual transmission. The responsiveness of the device depends upon the cycle *duration* and hence on a. For low values of λ the mean response time is essentially determined by a.

3. Calculation of waiting times. Let W_m and W^i denote respectively the waiting times of a request at queue m and one which is served during stage i. Consider for the moment one queue in isolation. We observe that since the service order is FCFS, the requests queued at service termination must have arrived during the waiting or service time of the request just completed. Now let $\pi_{i,j} = \sum_{\vec{k} \varepsilon \{\vec{n} | \vec{n}_{m_i} = j\}} \pi_{i,\vec{k}}$ be the marginal queue-length distribution of queue m_i at stage-i service completions. Define $\pi_i = \sum_{\vec{n}} \pi_{i,\vec{n}} = \sum_j \pi_{i,j} = \pi_i(\vec{1})$. Then from the above observation we have for the probability that queue m_i holds $n_{m_i} = j$ customers at service completion epochs

$$\frac{\pi_{i,j}}{\pi_i} = \int_0^\infty \frac{(\lambda_{m_i} t)^j}{j!} \varepsilon^{-\lambda_{m_i} t} \, d\{F_{W^i} * F_T(t)\}, \tag{17}$$

where $F_{W^i} * F_T(t)$ corresponds to the convolution of the distributions for stage-i waiting times and service times. $F_T(\cdot)$ is the distribution of a constant T; therefore, calculating the generating functions of both sides of (17) we obtain for the LST of F_{W^i}

$$L_{W^i}(s) = \frac{\pi_i(1, \cdots, 1 - s/\lambda_{m_i}, \cdots, 1)}{\pi_i \exp(-Ts)}. \tag{18}$$

Denoting the pgf of $\pi_{i,j}$ by $\pi_i(z)$ gives the numerator of (18) as $\pi_i(1 - s/\lambda_{m_i})$. For the distribution $F_{W_m}(\cdot)$ of waiting time at cylinder m, we average over the stages where $m_i = m$ and get

$$\begin{aligned} F_{W_1}(t) &= F_{W^1}(t), \\ F_{W_m}(t) &= \frac{\pi_m F_{W^m}(t) + \pi_{2M-m} F_{W^{2M-m}}(t)}{\pi_m + \pi_{2M-m}}, \qquad 1 < m \leqq M. \end{aligned} \tag{19}$$

Next, from (6) and (15) we can express $\pi_i = \pi_i(\vec{1})$ in terms of first derivatives of $\beta^{i-1}(\vec{z})$ at $\vec{z} = \vec{1}$,

$$\pi_i = \frac{\gamma}{1-\rho_{m_i}}(\beta_{m_i}^{i-1} + a\lambda_{m_i}), \tag{20}$$

where $\beta_m^i \equiv \partial\beta^i(\vec{z})/\partial z_m|_{\vec{z}=\vec{1}}$ and $\rho_{m_i} = T\lambda_{m_i}$. Finally, we differentiate (18) at $s = 0$ using (15) and then again (6) to eliminate $\alpha^i(\cdot)$. In the resulting expression we substitute for γ and π^i from (16) and (20) and obtain

$$E(W^i) = \frac{\beta_{m_im_i}^{i-1} + 2a\lambda_{m_i}\beta_{m_i}^{i-1} + a^2\lambda_{m_i}^2}{2\lambda_{m_i}(a\lambda_{m_i} + \beta_{m_i}^{i-1})} + \frac{T\rho_{m_i}}{2(1-\rho_{m_i})}, \tag{21}$$

where $\beta_{m,n}^i \equiv \partial^2\beta^i(\vec{z})/\partial z_m\partial z_n|_{\vec{z}=\vec{1}}$. From (19) the desired mean waiting time is

$$\begin{aligned} E(W_1) &= E(W^1), \\ E(W_m) &= \frac{\pi_m E(W^m) + \pi_{2M-m}E(W^{2M-m})}{\pi_m + \pi_{2M-m}}, \qquad 1 < m \leqq M. \end{aligned} \tag{22}$$

The lengths, N_m, of queues accumulated at the individual cylinders are of practical interest as well. From Little's theorem, the mean value, $E(N_m)$, at request completion epochs is given by $\lambda_m E(W_m)$. To obtain higher moments one needs only to differentiate (15) further, and accumulate contributions as in (19). However, the complexity of these calculations can be expected to increase significantly.

4. Numerical calculations. The expressions we derived for $E(W^i)$ and higher moments include the partial derivatives of the functions $\beta^i(\vec{z})$ at $\vec{z} = \vec{1}$. We show here how these can be calculated based on (11). We shall also use the following values and notation:

$$-B_m'(0) = \frac{T}{1-\rho_m} \equiv T\gamma_m, \tag{23}$$

$$B_m''(0) = T^2\gamma_m^3, \tag{24}$$

$$\frac{\partial z_{m_i}^{(i)}}{\partial z_m} \equiv \zeta_m^{m_i}(\vec{z}) = -(1-\delta_{m,m_i})\lambda_m B_{m_i}'\left[\sum_{j\neq m_i}\lambda_j(1-z_j)\right], \tag{25}$$

$$\zeta_m^{m_i}(\vec{1}) \equiv \zeta_m^{m_i} = (1-\delta_{m,m_i})\rho_m\gamma_{m_i}. \tag{26}$$

We get by straightforward calculation

$$\frac{\partial A(\vec{z}^{(i)})}{\partial z_m} = a(1-\delta_{m,m_i})\tilde{A}(\vec{z}^{(i)})[\lambda_m + \lambda_{m_i}\zeta_m^{m_i}(\vec{z})].$$

At $\vec{z} = \vec{1}$ the right-hand side becomes $(1-\delta_{m,m_i})a\lambda_m\lambda_{m_i}$.

Proceeding from (11) we get

$$\begin{aligned} \beta_m^i(\vec{z}) &\equiv \frac{\partial\beta^i(\vec{z})}{\partial z_m} \\ &= (1-\delta_{m,m_i})\{\beta_m^{i-1}(\vec{z}^{(i)})\tilde{A}(\vec{z}^{(i)}) + \beta_{m_i}^{i-1}(\vec{z}^{(i)})\tilde{A}(\vec{z}^{(i)})\zeta_m^{m_i}(\vec{z}) + a\beta^i(\vec{z})[\lambda_m + \lambda_{m_i}\zeta_m^{m_i}(\vec{z})]\}. \end{aligned} \tag{27}$$

At $\vec{z} = \vec{1}$ these derivatives yield

$$\beta_m^i = (1-\delta_{m,m_i})\{\beta_m^{i-1} + \beta_{m_i}^{i-1}\rho_m\gamma_{m_i} + a\lambda_m\gamma_{m_i}\}, \qquad 1 \leqq i \leqq 2M-2, \quad 1 \leqq m \leqq M. \tag{28}$$

Differentiating $\beta^i_m(\vec{z})$ with respect to z_n, and evaluating at $\vec{z}=\vec{1}$, we find

$$\begin{aligned}\beta^i_{m,n} = (1-\delta_{m,m_i})(1-\delta_{n,m_i})\{&\beta^{i-1}_{m,n}+\beta^{i-1}_{m,m_i}\zeta^{m_i}_n+\beta^{i-1}_{n,m_i}\zeta^{m_i}_m \\ &+\beta^{i-1}_{m_i,m_i}\zeta^{m_i}_m\zeta^{m_i}_n+\beta^{i-1}_m a\lambda_n\gamma_{m_i}+\beta^{i-1}_{m_i}\zeta^{m_i}_m a\lambda_n\gamma_{m_i} \\ &+\beta^{i-1}_{m_i}\zeta^{m_i}_m\zeta^{m_i}_n\gamma_{m_i}+\beta^i_n a\lambda_m\gamma_{m_i}+a\lambda_{m_i}\zeta^{m_i}_m\zeta^{m_i}_n\gamma_{m_i}\}.\end{aligned} \tag{29}$$

Higher derivatives are readily formed in this manner.

Equation (28) can be used to calculate all β^i_m. These equations, while quite cumbersome for symbolic manipulation, are very well suited to numerical solution by computer. Equation (28) is applied $2M-2$ times (each time for all values of m) in order to collect coefficients for a set of M equations, linear in (say) β^1_m. These are straightforward to solve; re-applying (28) yields us the first derivatives that we need.

A very similar procedure for (29) is used to determine values for the second order derivatives. We note in passing that although the calculation of the mean waiting time required the values of only a small fraction of these derivatives, the form of the only expression we found that was relatively convenient to solve required the evaluation of many more in the process.

The regularity of the system of equations (29) suggests that the corresponding matrices might be explicitly inverted. This does not seem to be the case; they can be inverted in terms of suitably defined $M\times M$ blocks, but inverting these blocks results in an overall effort of the same complexity.

Since $\beta^{i-1}_{m_i}$, as required in (21), has the value of the expected queue size at queue m_i when stage $i-1$ is terminated, the $\beta^i_{m_i}$ may be determined in the following more direct way.

Define

g_i = expected time between end of stage i and the last time at which the head departed from queue m_i.

d_i = expected time between end of stage $i-1$ and end of stage i.

We state without proof the following mean-value relationships. They should be obvious on inspection.

$$\begin{aligned}&\beta^{i+1}_{m_i}=\lambda_{m_i}d_{i+1}, \\ &d_i=\lambda_{m_i}g_iT+a=\rho_{m_i}g_i+a, \\ &g_1=g_M=D=\sum_{i=1}^{2M-2}d_i, \\ &g_2=d_1+d_2=(d_1+a)/(1-\rho_{m_2}), \\ &g_i=g_{i-1}+d_i+d_{2M-i+1}, \qquad 3\leqq i\leqq M-1, \\ &g_i=D-g_{2M-i}, \qquad 2\leqq i\leqq M-1, \\ &g_i=\frac{1}{1-\rho_{m_i}}\left(\frac{\beta^{i-1}_{m_i}}{\lambda_{m_i}}+a\right), \qquad 1\leqq i\leqq 2M-2.\end{aligned}$$

The last equation results first from breaking g_i down into the time interval beginning with the departure of the arm from queue m_i and ending with the termination of stage $i-1$, with mean length $\beta^{i-1}_{m_i}/\lambda_{m_i}$, plus the time interval during which queue m_i served. Next, we observe that if the load at queue m_i is A when its processing begins, the arm's expected stay there is $A/(1-\rho_{m_i})$. This is also obtainable by differentiating (9)

in [3] at $\vec{z}=\vec{1}$. Finally, then

$$\beta^1_{m_2}=\lambda_{m_2}d_1, \qquad \beta^{i-1}_{m_i}=\lambda_{m_i}(d_{2M-i+1}+g_{i-1}), \qquad 3\leqq i\leqq M-1.$$

Thus, the $\beta^{i-1}_{m_i}$ may be obtained recursively, beginning with (16). Unfortunately, no such procedure was discovered for the second derivatives.

It is remarkable that a very similar, though slightly simpler, queueing model treated by Konheim and Swartz [5] requires substantially fewer calculations to find $E(W_i)$, precisely because a "direct" approach to the evaluation of the second order derivatives can be implemented there. Higher moments seem to be equally hard to calculate in both cases.

The calculations we have outlined were performed for a number of input rates and distributions (across the disk). The following characterizes the results:

- Total input rate (λ) and cylinder number (m) are the main determinants of $E(W_m)$. It is quite robust under changes in the distribution of λ_m.
- This mean was also only very slightly influenced by relative changes in λ and T, so long as ρ was not affected.
- "Popular" cylinders require shorter waiting times than those with low λ_m.
- When all λ_m are equal, the graph of $E(W_m)$ versus m is parabolic. This is not true for any non-uniform distribution.

Complexity of the calculations. Each application of (28) uses $O(M^2)$ operations (the number of substitutions per line of the equations increasing from 3 to M), for a total of $O(M^3)$. This is also the time complexity of the solution of the resultant set of $M-1$ equations and the back substitution through (28); the total is then still $O(M^3)$.

Similarly, each application of (29) requires $O(M^3)$ operations; here the solution of the final set of equations dominates, with $O(M^6)$ operations required. Thus, the numerical evaluation of these equations for realistic devices, where M assumes values in the low hundreds, is too expensive for most purposes. One should use it, however, to investigate qualitative features, such as:

- Dependence of performance measures on hardware parameters, and on details of the algorithms used for scheduling, as well as for comparison with other scheduling methods on the same hardware.
- Evaluation of the quality of various approximations to the above analysis before their application to a full-fledged system.

It is of interest to note that the above described scheme of calculations is very much cheaper than the one suggested in [3], since the repeated (chain) differentiations used there are not required in our method, and the subsequent calculations are fewer in number (e.g., $O(M^4)$ are required in [3] to calculate all the first order derivatives).

5. Elaborations of the model. The model, as presented in § 2, captures a number of the features and properties of a disk system which are critical to its performance. For the sake of simplicity, however, it does contain a good many assumptions that would seem at once to be both arbitrary and at variance with the real state of affairs in such systems. We discuss here a number of these assumptions and what their removal or modification entails, mainly in terms of model tractability.

Request service duration. This quantity was assumed constant and equal for all cylinders simply to keep down the level of detail of the model (and use somewhat less storage during numerical calculations). Since these properties are not used explicitly in the procedures we developed, there is absolutely no effect on the analysis when we assume that the time to service a request has a general distribution function, which may indeed be cylinder-specific. Let S_m be the variable representing the service

duration at cylinder m, with distribution function and LST, $F_{S_m}(\cdot)$ and $\phi_m(\cdot)$, respectively; the LST of the busy period distribution is then the solution of the equation $B_m(s)=\phi_m(s+\lambda_m-\lambda_m B_m(s))$. $\phi_m(\cdot)$ itself should also be substituted for $\exp(-Ts)$ in the various equations in § 2 and § 3. No essential change need be made, and the calculations proceed in precisely the same manner.

Meaningful seeks. By this rather fanciful name we refer to the incorporation into the model of the following modification: At stage completion a seek is initiated to the next nonempty cylinder queue in the cycle, as given in (1). We recognize here the fact that, due to acceleration effects, the time to traverse k cylinders in a single seek is appreciably less than k times the duration of a single-cylinder seek. This certainly represents better the way a disk facility is managed, but it also introduces a number of complicating factors:

- Instead of a single seek time, we have now a multitude; this may be as high as $M(M-1)$, but it can be reduced to $M-1$ by assumptions that are quite borne out in practice.
- The movements of the arm are not prescribed as before, but rather depend on the state of the system, a feature that has been the undoing of many queueing-theoretical models.
- As a consequence, the key relations of § 3 do not hold, and a new approach must be found.

Nevertheless, the authors believe that this modification still yields a tractable situation, and are preparing an analysis of its main features.

Dependent arrivals. The model as described in § 2 can be rephrased to postulate a single Poisson stream of arrivals, at rate λ, with each request destined to cylinder m with fixed probability $p_m=\lambda_m/\lambda$, independently of the state of the system and, in particular, of the history of the arrival process itself. We propose in this paragraph a modification which deviates from this last assumption. In particular, we wish to consider the situation where the destinations of successive arrivals form a first order Markov chain.

Thus, to the description of the state of the system a further index has to be added—the identity of the last cylinder addressed by the arrival process. This is a major departure from the model analyzed above, and the methods used there would be ineffective in handling it. Still, its analysis would be desirable, inasmuch as it represents many common situations better than the simpler version, in particular when one considers systems that employ a large number of disk drives (i.e., there are fewer tasks active at each drive at any time).

Appendix. Notation. We collect below the notation used in the paper.

a—seek time between adjacent cylinders.
$A(s)$—LST of the (point) distribution of the seek time, $A(s)=\exp(-as)$.
$A(\vec{z})$, $C(\vec{z})$—cf. (5).
$B_m(s)$—LST of the distribution G_m.
$C(s)$—LST of the (point) distribution of the service time $=\exp(-Ts)$.
D—duration of a cycle (time to complete $2M-2$ stages).
F—generic probability distribution function (pdf).
G_m—the pdf of a busy period in cylinder m. Eq. (7).
i—generic stage index.
L—generic LST.
$L(\vec{z})$—cf. (4).
m—generic queue index.

M—number of cylinders (queues).
m_i—the cylinder served at stage i (cf. (1)).
$\vec{n}$—state of the queues; n_m requests for cylinder m.
$\vec{N}(t)$—the process of cylinder-queue occupancies.
$p(F; \vec{k})$—the probability of $\vec{k}$ arrivals during a period with pdf F (cf. (3)).
$c(\lambda; F)$—a variable, equal to the number of counts of a Poisson process of rate λ during a period with the pdf $F(\cdot)$.
T—duration of I/O request execution.
W^i—the waiting time of a request processed at stage i.
W_m—the waiting time of a request addressed to cylinder m.
$\vec{z}^{(i)}$—cf. (10).
α^i_n—the probability that at the beginning of stage i the queues are at state $\vec{n}$.
$\alpha^i(\vec{z})$—pgf for $\alpha^i_{\vec{n}}$.
$\beta^i_{\vec{n}}$—the probability that just after stage-i termination the queues are at state $\vec{n}$.
$\beta^i(\vec{z})$—pgf for $\beta^i_{\vec{n}}$.
$\beta^i_m(\vec{z})$—$\partial\beta^i(\vec{n})/\partial z_m$.
β^i_m—$\beta^i_m(\vec{1})$.
$\beta^i_{m,n}$—$\partial^2\beta^i(\vec{z})/\partial z_m \partial z_n|_{\vec{z}=\vec{1}}$.
γ—cf. (13).
γ_m—$1/(1-\rho_m)$, cf. (23).
$\delta_{m,n}$—Kronecker's delta.
$\zeta^{m_i}_m(\vec{z})$—cf. (25).
$\zeta^{m_i}_m$—$\zeta^{m_i}_m(\vec{1})$.
λ_m—input rate to cylinder m.
λ—$\sum_{m=1}^M \lambda_m$.
π_i—the probability that a service completion occurs at stage i.
$\pi_{i,\vec{n}}$—the probability that just after a service termination the system is at stage i and the queues are at state $\vec{n}$.
$\pi_i(\vec{n})$—pgf for $\pi_{i,\vec{n}}$.
ρ_m—$\lambda_m T$.
$\omega_{i,\vec{n}}$—the probability that a service beginning is at stage i and the state of the queues is $\vec{n}$.
$\omega_i(\vec{z})$—pgf for $\omega_{i,\vec{n}}$.

Acknowledgments. Comments of the referees helped in organizing the paper. Dr. Kimming So (IBM Research, Yorktown Heights, NY) performed the numerical calculations and suggested the direct evaluation of the first order derivatives.

REFERENCES

[1] E. G. COFFMAN, JR., L. A. KLIMKO AND B. RYAN, *Analysis of scanning policies for reducing disk seek times*, this Journal, 1 (1972), pp. 269–279.
[2] P. J. DENNING, *Effects of scheduling on file memory operations*, Proc. AFIPS SJCC, 31 (1967), pp. 9–21.
[3] M. EISENBERG, *Queues with periodic service and changeover time*, Oper. Res., 20 (1972), pp. 440–451.
[4] M. HOFRI, *Disk scheduling*: FCFS *vs.* SSTF *revisited*, Comm. ACM, 23 (1980), p. 11.
[5] C. W. ONEY, *Queueing analysis of the scan policy for moving-head disks*, J. Assoc. Comput. Mach., 22 (1975), pp. 397–412.
[6] A. KONHEIM AND G. B. SWARTZ, *Polling in a LOOP system*, J. Assoc. Comput. Mach., 27 (1980), pp. 42–59.
[7] T. J. TEOREY, *Properties of disk scheduling policies in multiprogrammed computer systems*, Proc. AFIPS FJCC 41 (1972), pp. 1–14.
[8] T. J. TEOREY AND B. T. PINKERTON, *A comparative analysis of disk scheduling policies*, Comm. ACM, 15 (1972), pp. 177–184.

SIAM J. COMPUT.
Vol. 11, No. 1, February 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1101-0006 $01.00/0

A HIDDEN-LINE ALGORITHM FOR HYPERSPACE*

ROBERT P. BURTON† AND DAVID R. SMITH‡

Abstract. An object-space hidden-line algorithm for higher-dimensional scenes has been designed and implemented. Scenes consist of convex hulls of any dimension, each of which is compared against the edges of all convex hulls not eliminated by a hyperdimensional clipper, a depth test after sorting and a minimax text.

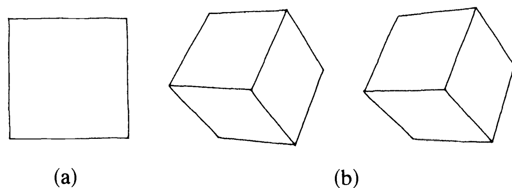
Hidden and visible elements are determined in accordance with the dimensionality of the selected viewing hyperspace. When shape alone is the attribute of interest, hidden-line elimination need be performed only in that hyperspace.

The algorithm is of value in the production of shadows of hyperdimensional models, including but not limited to four-dimensional space-time models, the hyperdimensional elementary catastrophe models and multivariate statistical models.

Key words. hyperspace, hidden-line elimination

Introduction. This paper describes an algorithm for solving the hidden-line problem in hyperspace. The lines are edges of convex hulls approximating the surfaces of hyperdimensional objects. The algorithm removes edges which would be invisible in a hyperdimensional scene. The scene may then be projected to lower dimensions. The development of a hidden-line eliminator for hyperspace is part of an ongoing effort to display and gain insight into the structures of higher-dimensional space. Of particular interest are four-dimensional space-time models, the seven elementary catastrophe models, of which five are hyperdimensional [1] and multivariate statistical models. While these models are numerically useful to some extent, they are of limited general utility in the absence of adequate hyperdimensional presentation techniques. Without an ability to present visible lines only, the four- (and higher-) dimensional analogues of front, rear and depth become hopelessly garbled in the generalized view.

Previous efforts to display hyperobjects [2], [3] have employed techniques which either discard one or more variables or hold them constant so as to restrict structures to three dimensions. Such techniques impose unacceptable constraints. To illustrate by analogy, consider a cube aligned with x-, y- and z-axes. The cube can be restricted to two dimensions by either eliminating or holding constant one of the coordinates. The cube then appears to be nothing more than a square (see Fig. 1a). A generalized technique, which permits the cube to be viewed from any position, with any orientation and in stereo, provides substantially more information, especially when hidden elements of the cube are removed (see Fig. 1b). Similarly, views of hyperobjects are significantly enriched when a generalized viewing capability is combined with a hyperdimensional hidden-line eliminator such as the one described in this paper.



(a) (b)

FIG. 1. (a) *A restricted view of a cube*; (b) *A generalized stereo-pair view of the cube with hidden lines removed.*

* Received by the editors November 6, 1979, and in final form March, 1981.

† Computer Science Department, Brigham Young University, Provo, Utah 84602.

‡ Department of Computer Science, Carnegie–Mellon University, Pittsburgh, Pennsylvania 15213.

The meanings of *visible* and *hidden*. The development of an algorithm for removing hidden lines is necessarily preceded by a determination of the meanings of *visible* and *hidden*. The definitions offered here accommodate the geometry of space and are hypothesized to accommodate the geometry of hyperspace.

An object J is defined to be *visible* in a viewing space of dimension m to the extent that the points P constituting J or any section(s) of J intersected with the viewing space collectively extend at least into the $m-1$ dimensions of the viewing space orthogonal to the ray of vision and are not hidden. A point P on an object J is defined to be *hidden* from view at V by an object H if and only if a neighborhood of at least dimension $m-1$ exists about an intersection of the line segment VP and H, completely contained in H and extending into each of the $m-1$ dimensions orthogonal to VP (see Fig. 2). Opacity of objects is assumed.

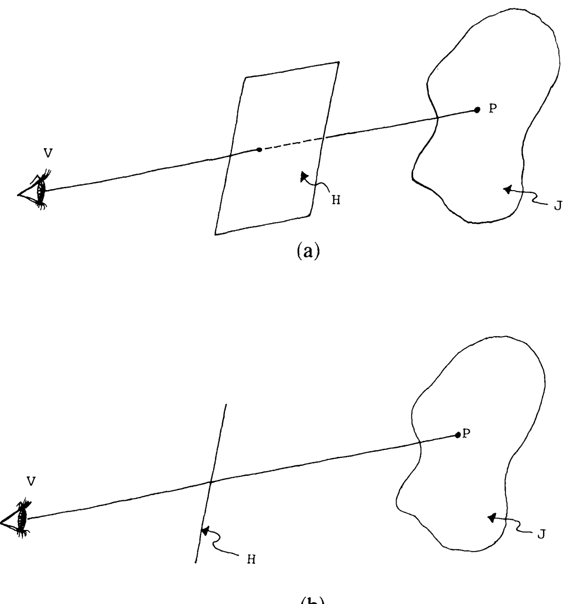


FIG. 2. *Hidden and visible.* (a) *An obscuring intervening object*; (b) *A nonobscuring intervening object.*

Each snapshot of the viewing space involves the viewer at V, the object(s) J which may be partially or completely hidden and the potential hider(s) H. Vision is limited to, but always includes, the $m-1$ dimensions which can be perceived plus depth which can be inferred, which together span the viewing space. Objects possessing additional dimensions become part of the scene to the extent that they intersect it or are projected into it.

Surface approximation. Being piecewise smooth, the surfaces of n-dimensional forms intended for graphic presentation are topologically equivalent to segments of $(n-1)$-space. Thus, the surface of a form in n-space can be approximated by polyhedra of dimension $n-1$. This is an extension of the practice of approximating surfaces of three-dimensional forms with polygonal patches.

Simplification. Both the initial absence of an intuitive feeling and a lack of experience in four and higher dimensions encouraged conceptual simplicity in the design of algorithms for producing, transforming and presenting hyperdimensional scenes. With this in mind, the initial hidden-element algorithms required all surfaces to

be approximated by simplices.[1] The final algorithm was extended to accommodate general convex hulls. The representation of objects as convex hulls provides both a disadvantage and an advantage. The advantage is robustness. Consider a patch in three-dimensional space, supposed to be a quadrilateral, but which actually consists of four nonplanar points. Its manifests itself as a tetrahedron, but nevertheless hides points in a predictable manner. The disadvantage is that concave objects must be built from multiple convex objects. Restriction of the effort to the development of a hidden-*line* algorithm was another obvious step. Attributes other than shape, such as color, for example, were ignored. These restrictions were easily accepted in part because the graphics equipment on which the algorithm was to be implemented was monochromatic and vector oriented.

Simplification was also achieved by collapsing successively from n dimensions to $n-1$ dimensions, and so on, resulting ultimately, for our inspection, in a stereo pair or a single two-dimensional image. The several levels of computation implied by these stages of collapse suggested an object-space algorithm rather than a screen-space algorithm [4]. Furthermore, the algorithm needed to provide output of the same form as its input. After some experience with the algorithm, however, we observed and were able to establish that hidden lines need be eliminated at only one dimensional level prior to projection into R^3 or R^2, making successive application of the algorithm unnecessary. Repeated viewing transformations are still required, nevertheless. Finally, the hidden-line algorithm was simplified by preprocessing the scene with a viewing transformation which can include perspective. The effect of this transformation is to place the viewer at infinity looking in the negative direction along the mth axis with the rays of vision parallel to the mth axis.

The viewing transformation. A generalized viewing transformation in $R^m (m \geqq 2)$ includes:

(1) translation of the scene so that the point to be *looked from* is at the origin 0_0 of object coordinates; and

(2) rotation through the appropriate angles in the planes determined by the axis pairs $(1, m), (2, m), \cdots, (m-1, m)$. Rotation in the (i, m)-plane directs the gaze so as to ultimately place the point to be *looked at* on the viewing axis. Rotation in planes where the axis pair does not include m is implicitly restricted. Simply specifying *look at* and *look from* positions without an order for rotations leaves the scene free to tumble with $\binom{m-1}{2}$ unrestricted degrees of freedom. While this would not affect spatial relationships or the visibleness of elements of the scene, it would significantly affect the orientation of the scene relative to the viewer as well as subsequent projections to lower dimensions.

The rotation scheme based on lexicographic ordering of axis pairs accommodates the human habit of keeping the eyes parallel to the horizon in three-dimensional space; if this third degree of rotational freedom were unrestricted the gaze would remain fixed, but the scene would be free to rotate about the viewing axis. Lexicographic ordering also facilitates easy calculation of the Euler angles. Corresponding advantages are experienced in higher dimensions.

Scenes are projected orthogonally to lower dimensions except possibly during the final projection.

Hidden-line elimination. Hidden-line elimination is carried out by comparing each convex hull H against each edge E to determine which portion of the edge, if any,

[1] An *n-simplex* is a convex hull of $n+1$ affinely independent points. A set of points $\{x_i\}$ is *affinely independent* if and only if for some fixed j the set of points $\{(x_i - x_j) | i \neq j\}$ is linearly independent.

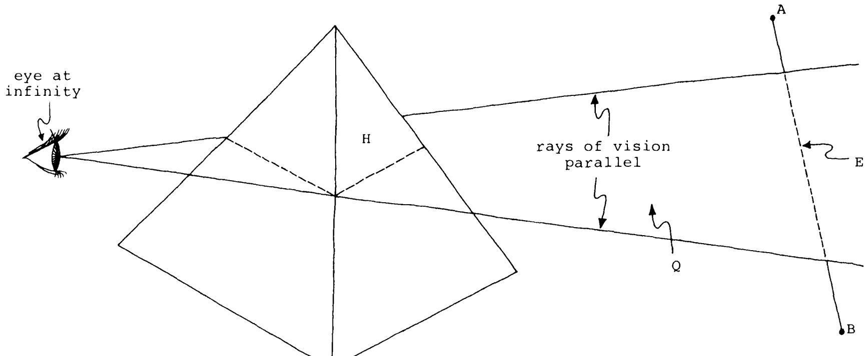


FIG. 3. *A partially obscured edge.*

is hidden by the convex hull. Requiring the hull H to be convex assures that all points of E which H hides are contained in a connected interval.

An edge E with endpoints $A=(a_1, a_2, \cdots, a_m)$ and $B=(b_1, b_2, \cdots, b_m)$ may be hidden at least partially by a convex hull H if H intersects the partial plane of view Q (see Fig. 3). From the criterion of Carathéodory,

$$X \in \textit{convex hull} \text{ of a finite set } A=\{X_i \mid i=1,2,\cdots,m\}$$

if and only if

$$X=\sum_{i=1}^{m} \alpha_i x_i, \quad \text{where } \alpha_i \geqq 0, \quad i=1,\cdots,m, \quad \sum_{i=1}^{m} \alpha_i=1.$$

The definition of a convex hull is less restrictive than the definition of an n-simplex. The edge E can be expressed parametrically as

$$E=\{X \in R^m \mid X=A+(B-A)t, 0 \leqq t \leqq 1\}. \tag{1}$$

The partial plane of view Q can be expressed as the directed sum

$$\begin{aligned} Q &= E+e_m s, \qquad 0 \leqq s \\ &= \{X \in R^m \mid X=A+(B-A)t+e_m s, 0 \leqq t \leqq 1, 0 \leqq s\}, \end{aligned} \tag{2}$$

where e_m is the mth vector in the natural basis of R^m. Letting $X_i=(x_{1i}, x_{2i}, \cdots, x_{mi})^T$ be the ith vertex of the convex hull, we have

$$H=\left\{X \in R^m \middle| X=\sum_{i=1}^{m} \begin{bmatrix} x_{1i} \\ \vdots \\ x_{mi} \end{bmatrix} \alpha_i, 0 \leqq \alpha_i, \sum_{i=1}^{n} \alpha_i=1\right\}. \tag{3}$$

Intersecting Q and H yields

$$\begin{bmatrix} x_{11} \\ \vdots \\ x_{m1} \end{bmatrix} \alpha_1+\cdots+\begin{bmatrix} x_{1n} \\ \vdots \\ x_{mn} \end{bmatrix} \alpha_n=A+(B-A)t+e_m s=\begin{bmatrix} a_1 \\ \vdots \\ a_m \end{bmatrix}+\begin{bmatrix} b_1-a_1 \\ \vdots \\ b_m-a_m \end{bmatrix} t+\begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} s. \tag{4a}$$

Rearranging, we get

$$\text{(4b)} \qquad \begin{bmatrix} a_1 - b_1 \\ \vdots \\ a_m - b_m \end{bmatrix} t + \begin{bmatrix} x_{11} \\ \vdots \\ x_{m1} \end{bmatrix} \alpha_1 + \cdots + \begin{bmatrix} x_{1n} \\ \vdots \\ x_{mm} \end{bmatrix} \alpha_n + \begin{bmatrix} 0 \\ \vdots \\ 0 \\ -1 \end{bmatrix} s = \begin{bmatrix} a_1 \\ \vdots \\ a_m \end{bmatrix}.$$

Incorporating the restriction $\sum_{i=1}^{n} \alpha_i = 1$ yields

$$\text{(4c)} \qquad \begin{bmatrix} a_1 - b_1 \\ \vdots \\ a_m - b_m \\ 0 \end{bmatrix} t + \begin{bmatrix} x_{11} \\ \vdots \\ x_{m1} \\ 1 \end{bmatrix} \alpha_1 + \cdots + \begin{bmatrix} x_{1n} \\ \vdots \\ x_{mn} \\ 1 \end{bmatrix} \alpha_n + \begin{bmatrix} 0 \\ \vdots \\ 0 \\ -1 \\ 0 \end{bmatrix} s = \begin{bmatrix} a_1 \\ \vdots \\ a_m \\ 1 \end{bmatrix},$$

which corresponds to

$$\text{(4d)} \qquad \begin{bmatrix} a_1 - b_1 & x_{11} & \cdots & x_{1n} & 0 \\ \vdots & \vdots & & \vdots & \vdots \\ & & & & 0 \\ a_m - b_m & x_{m1} & \cdots & & -1 \\ 0 & 1 & \cdots & & 0 \end{bmatrix} \begin{bmatrix} t \\ \alpha_1 \\ \vdots \\ \alpha_n \\ s \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_m \\ 1 \end{bmatrix}$$

or, as the adjoint matrix

$$\text{(4e)} \qquad \begin{bmatrix} a_1 - b_1 & x_{11} & \cdots & x_{1n} & 0 & a_1 \\ \vdots & \vdots & & \vdots & \vdots & \vdots \\ & & & & 0. & \\ a_m - b_m & x_{m1} & \cdots & x_{mn} & -1 & a_m \\ 0 & 1 & \cdots & 1 & 0 & 1 \end{bmatrix}$$

still restricted by $0 \leqq t \leqq 1$, $0 \leqq s$ and $0 \leqq \alpha_i$.

The portion of E hidden by H can be determined by finding the values of t for which the adjoint matrix has a solution. Since H is convex the solution will be a connected interval.

A problem arises in the elimination process when all convex hulls are compared against all edges. If s is allowed to be zero, a convex hull will eliminate its own edges. One solution might be to avoid testing a convex hull against the edges which bound it. However, convex hulls may hide some of their own edges. Furthermore, an edge may be shared by two or more convex hulls; the task of remembering all the convex hulls which a given edge bounds is cumbersome. A simple solution to the problem requires only slight modification of the adjoint matrix. By adding a small number $\varepsilon > 0$ to the mth coordinates of the endpoints A and B, the edge is moved a distance ε closer to the viewer. The visible or hidden status of the edge is easily determined now because it no longer lies on the surface of the convex hull from which it arose. The modification

appears in the $(m, n+3)$ element of the adjoint matrix:

$$\begin{bmatrix} a_1-b_1 & x_{11} & \cdots & x_{1n} & 0 & a_1 \\ \vdots & \vdots & & \vdots & \vdots & \vdots \\ a_m-b_m & x_{m1} & \cdots & x_{mn} & 0 & a_m+\varepsilon \\ 0 & 1 & \cdots & 1 & -1 & 1 \\ & & & 0 & & \end{bmatrix}. \tag{4f}$$

The value of ε is suggested by the depth sort information. Experience has indicated that a value of 10^{-5} times the maximum depth works well.

The adjoint matrix is transformed to permit solution for $t_{\min}$ and $t_{\max}$ by the simplex method of linear programming [5]. The adjoint matrix is modified to yield t as a function of s and α_i. One step of Gauss elimination suffices, using an element of column 1 as the pivot for partial pivoting. Row independence is assured by completing the forward Gauss elimination process on the adjoint matrix and ignoring the zero rows remaining at the bottom. If after elimination the last nonzero row has a nonzero element in the adjoint $(n+3)$ column only, the system is inconsistent and there is no solution for t, in which case H does not hide any portion of E.

The adjoint matrix is now in tableau form for the simplex method, except that column 1 is superfluous and can be ignored. The value of t is the element $(1, n+3)$ of the adjoint matrix. The simplex method transforms the matrix to yield the values of $t_{\min}$ and $t_{\max}$. Although details of the procedure are not presented here, it should be noted that the initial basic feasible solution is found using artificial variables. A special loop in effect accomplishes Gauss elimination by pivoting on the artificial variables without expanding the matrix to contain them explicitly. A value for *Big M* [6, p. 63], the arbitrary large number needed for the initial basic feasible solution, can be determined during the depth sort which is discussed below. Experience indicates that good results can be obtained with *Big M* equal to 10^2 times the maximum depth. Although the simplex method enforces the restrictions $0 \leqq s$ and $0 \leqq \alpha_i$, it does not restrict the range of t. Therefore, the interval $[t_{\min}, t_{\max}]$ must be intersected with the interval $[0, 1]$. If t is unbounded, then E is parallel to e_m; the projected image of E is a single point and can be ignored.

Depth and minimax tests. By performing a depth test and a minimax test, the situation can often be resolved without employing the adjoint matrix described above. Edges determined from coordinates of vertices and lists of vertices comprising convex hulls are entered into a hash table which is heap sorted into a linear table implicitly eliminating shared or otherwise redundant edges. Edges are sorted according to depth so that the edge with the greatest depth appears first in the list. The depth of each convex hull is compared against the depth of each edge. If both endpoints of the edge are closer than the convex hull to the viewer, the convex hull cannot possibly obscure any part of the edge. Furthermore, it cannot obscure any of the subsequent edges since edges are listed in order of decreasing depth. When the depth test fails, a minimax test is applied to the other $m-1$ dimensions to identify cases where the convex hull cannot hide the edge, i.e., cases where the ith coordinate values of the edge entirely exceed or entirely fall short of the ith coordinate values of the convex hull.

Partially obscured edges. If a convex hull hides a middle portion of an edge, the edge is divided into two segments. To avoid entering the resultant new endpoints into the vertex array and to avoid additional edge definitions, the hidden intervals of an edge

are placed in a singly-linked list. Each node in this list gives the minimum and maximum values of t ($0 \leqq t \leqq 1$) in (1) for which part of the edge is obscured. The use of linked lists saves computer time since all separate, visible segments of an edge can be checked simultaneously against possible obscuring convex hulls. This would be impossible if new endpoints were calculated and additional edges generated. Within the list, obscured portions of an edge which are found to overlap or nearly abut are combined to minimize the number of linked nodes. If the combined interval is [0, 1], the edge is completely hidden; the associated node is subsequently returned to the pool of available storage.

Clipping. Clipping in hyperspace is conceptually simple, being an extension of three-dimensional clipping [7]. Similar to its analogue in three dimensions, the hyper-pyramid of vision in R^4, for example, imposes the following constraints:

$$-w \leqq c_x x \leqq w,$$

$$-w \leqq c_y y \leqq w,$$

$$-w \leqq c_z z \leqq w,$$

where each of the c_x, c_y and c_z represents the cotangent of half the angle of vision in the given direction.

Even though the introduction of new points into the point array should be avoided, a new endpoint must be generated for an edge which lies partly within the field of vision, but crosses the zero-depth plane. In other cases it is sufficient to treat a clipped edge as a partially obscured edge, using the linked list. While this approach saves storage, it necessitates modification by the perspective routine of the t-parameters in the hidden-segment linked list [7].

The convex hulls should be processed by an object clipper. However, a convex hull totally within or outside the field of vision poses no problems. Even if a convex hull lies partly within the field of vision it need not be clipped in most cases; only the edges, not the convex hull itself, are ever visible. However, when a partially visible convex hull crosses the zero-depth hyperplane, division by depth in the perspective transformation becomes troublesome [7]. In such cases the convex hull must be clipped to a hyperplane slightly in front of the viewer. The convex hull is clipped by regenerating the edges and clipping them against this limiting hyperplane. Intersection points are entered into the hash table. The description of the convex hull now includes the hyperplane intersection points, but excludes vertices on the viewer's side of the clipping hyperplane. The hash table is placed in vacated storage locations in the vertex array, permitting all points to be treated uniformly.

The key to the hash table can be the sum of the vertex's first $m-1$ (for R^m) coordinates, multiplied by a constant [8]. Even though most transformations will cause a vertex to generate a key which differs from the one by which it was entered into the table, the objective of avoiding redundant points is realized at each dimensional level of clipping.

Classification and performance. The hidden-line algorithm presented in this paper is an object-space algorithm. The algorithm is closely related to Roberts' algorithm [9], particularly because it sweeps the area from an edge out to infinity. Basic relationships are formulated as matrix systems which are solved by standard methods.

As is true for Roberts' algorithm, computation required is roughly proportional to the square of the complexity of the scene. Minimization of computation is heavily dependent upon the ability of the preliminary tests to resolve the situation, thereby avoiding the need for the matrix solution. Experience with three-dimensional scenes

having from 400 to 1400 triangles with 700 to 2150 edges yields execution times (transformation, clipping, sorting, hidden-line elimination and plot file generation all divided by the product of the number of edges and triangles) of 83 μsec to 130 μsec to test one triangle against one edge, running on a DECsystem-1070. Failures of the depth test and minimax test result in higher execution times.

Discussion. The algorithm presented in this paper performs hidden-line elimination in R^n ($n \geqq 3$) and differs from conventional hidden-line elimination algorithms in its hyperdimensional capabilities. When an image is projected into R^3 or R^2 after hyperdimensional hidden-line elimination, information is preserved which would be lost if hidden-line elimination were performed in R^3 using conventional algorithms.

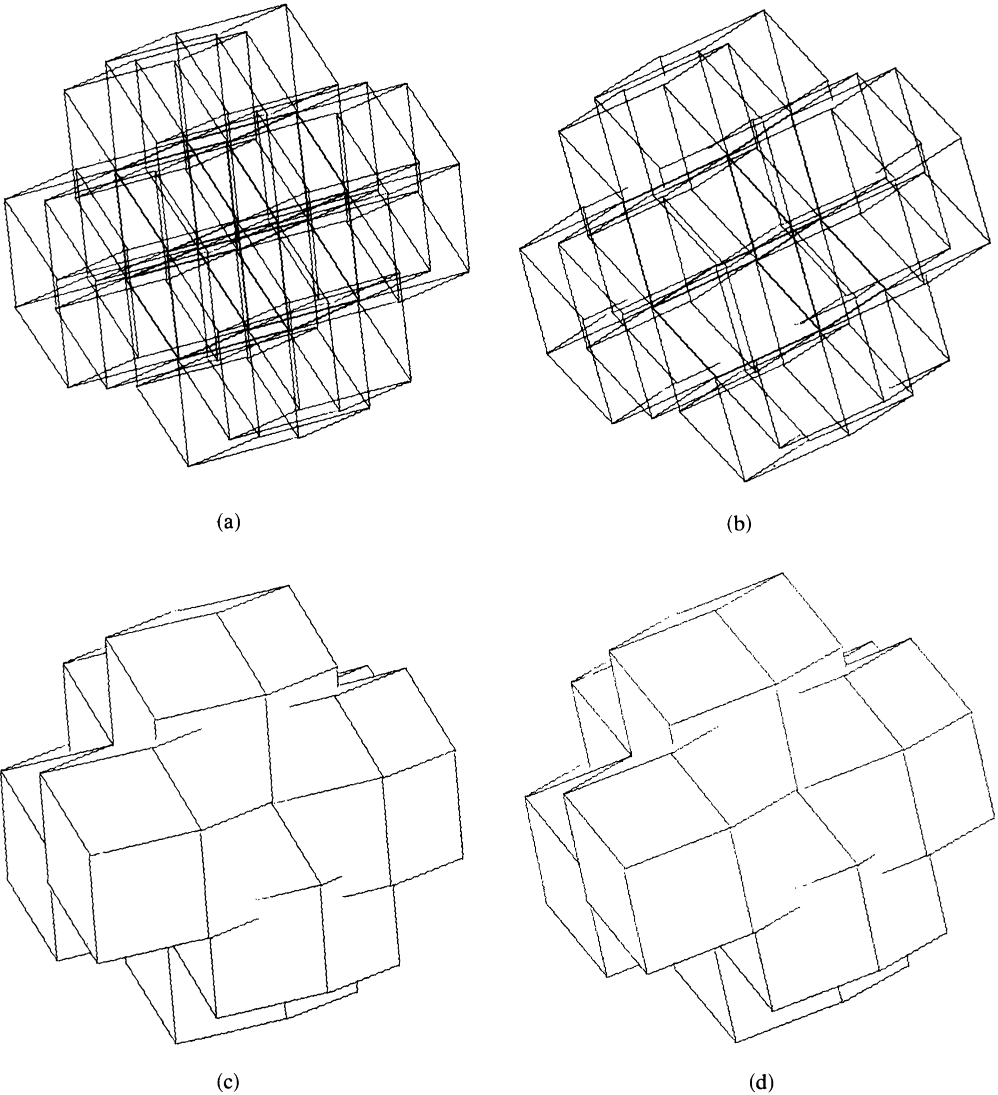


FIG. 4. *A generalized view of tesseracts (hypercubes) on each of the eight hyperfaces of a central tesseract.* (a) *No hidden-line elimination.* (b) *Four-dimensional hidden-line elimination.* (c) *Four-dimensional hidden-line elimination followed by three-dimensional hidden-line elimination.* (d) *Three-dimensional hidden-line elimination only.*

The information which is preserved is that which would be visible from hyperspace in a three- or two-dimensional projection of hyperspace.[2] Figure 4a shows a generalized view of a four-dimensional object with no hidden lines removed, projected into R^2; Fig. 4b shows the same object after four-dimensional hidden-line elimination and subsequent projection into R^2. Information is preserved in Fig. 4b which would have been lost had the object first been projected into R^3, followed by hidden-line elimination, followed by projection into R^2, as in Fig. 4d.

Hidden-line elimination need be performed only once when shape alone is the attribute of interest. Any subsequent hidden-line elimination would obliterate the results of all previous hidden-line elimination and alone would yield a shape equivalent to the aggregate shape that would result from successive applications of the algorithm from higher dimensions. By way of illustration, consider Fig. 4c, which shows the results of four-dimensional hidden-line elimination followed by three-dimensional hidden-line elimination. Figure 4d shows the results of three-dimensional hidden-line elimination alone.[3] The apparent equivalence of Figs. 4c and 4d can be explained as follows.

Assume that a point P is hidden from a viewer at V by a hypervolume H in four-dimensional space (see Fig. 5). The line segment VP intersects H in one point R provided only that VP and H are not contained in the same three-dimensional

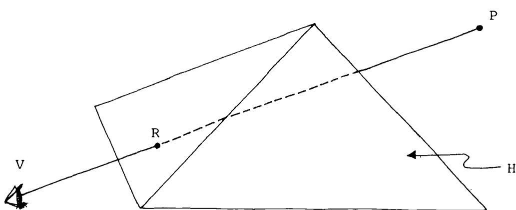


FIG. 5. *A point hidden in R^4 is hidden in R^3.*

hyperplane. Since the collapse from four to three dimensions takes place along the axis which contains V, R and P, the points R and P coincide after projection into R^3. P is simply absorbed into H. The shape of H is unchanged. In the case under consideration here, three-dimensional hidden-element elimination leaves no clue that four-dimensional hidden-element elimination took place previously. Attributes such as color would persist, however, unless hidden-element elimination took place in each successive dimension. When shape is the only attribute of interest, a significant savings is realized because the costly process of removing hidden lines need be performed only once. Computation is also minimized because the space in which hidden-line elimination is performed is dimensionally the least of all the spaces in which hidden-line elimination would be performed if successive application of the algorithm were necessary.

Acknowledgments. The authors gratefully acknowledge the consultation and contributions of other members of the Hyperspace Research Group. Appreciation for their support is extended to Brigham Young University and Eyring Research Institute.

[2] In the same sense, a photograph represents hidden-element elimination performed in R^3 and subsequent projection to R^2 to be viewed from R^3.

[3] The algorithm would never be used in R^3; a variety of superior algorithms could be summoned for that purpose. An illustration involving R^3 is chosen to simplify the presentation of the concept.

REFERENCES

[1] E. ZEEMAN, *Catastrophe theory*, Scientific American, 234 (April, 1976), pp. 65–83.
[2] C. PANATI, *Catastrophe theory*, Newsweek, 83 (January 19, 1976), p. 55.
[3] J. GORMAN, *The shape of change*, The Sciences, (September/October 1976), p. 21.
[4] I. SUTHERLAND, R. SPROULL AND R. SCHUMACKER, *A characterization of ten hidden-surface algorithms*, ACM Computing Surveys, 6 (1974), pp. 1–55.
[5] D. STEINBERG, *Computational Matrix Algebra*, McGraw-Hill, New York, 1974.
[6] F. HILLIER AND G. LIEBERMAN, *Operations Research*, Holden-Day, San Francisco, 1974.
[7] W. NEWMAN AND R. SPROULL, *Principles of Interactive Computer Graphics*, 2nd ed., McGraw-Hill, New York, 1979.
[8] M. STEPHENSON AND H. CHRISTIANSEN, *A polyhedron clipping and capping algorithm and a display system for three-dimensional finite element models*, Computer Graphics, 9 (Fall 1975), pp. 1–16.
[9] L. ROBERTS, *Machine perception of three dimensional solids*, MIT Lincoln Laboratory TR 315, 1963.

SIAM J. COMPUT.
Vol. 11, No. 1, February 1981

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1101-0007 $01.00/0

SYMBOLIC PROGRAM ANALYSIS IN ALMOST-LINEAR TIME*

JOHN H. REIF† AND ROBERT E. TARJAN‡

Abstract. This paper describes an algorithm to construct, for each expression in a given program text, a symbolic expression whose value is equal to the value of the text expression for all executions of the program. We call such a mapping from text expressions to symbolic expressions a *cover*. Covers are useful in such program optimization techniques as constant propagation and code motion. The particular cover constructed by our methods is in general weaker than the covers obtainable by the methods of [Ki], [FKU], [RL], [R2] but our method has the advantage of being very efficient. It requires $O(m\alpha(m, n)+l)$ operations if extended bit vector operations have unit cost, where n is the number of vertices in the control flow graph of the program, m is the number of edges, l is the length of the program text, and α is related to a functional inverse of Ackermann's function [T2]. Our method does not require that the program be well-structured nor that the flow graph be reducible.

Key words. code movement, code optimization, constant propagation, data flow analysis, symbolic evaluation.

1. Introduction.

Let $\mathscr{E}$ be an expression which appears somewhere in a computer program. If $\mathscr{E}$ evaluates to a constant independent of the particular execution of the program, then the program can be improved by substituting the appropriate constant for $\mathscr{E}$ in the program text. The systematic application of this technique is called constant propagation. Another kind of improvement is possible if $\mathscr{E}$ occurs within a loop but has the same value for every execution of the loop; in this case the program may be improved by moving the computation of $\mathscr{E}$ outside the loop. (Note that this is not an improvement if the loop is executed less than twice.) Constant propagation and code motion require for their application a mapping from text expressions to symbolic expressions such that in any program execution every symbolic expression has the same value as its corresponding text expression. We call such a mapping a *cover*. We desire a cover which is as simple as possible in some appropriately defined sense, but even determining whether a given text expression always evaluates to a constant is an undecidable problem. In this paper we describe an algorithm for efficiently computing a reasonably good cover.

In order to address this problem, we need some definitions. We represent the flow of control through a program π by a flow graph[1] $G=(V, E, r)$ where each vertex v represents a consecutive block of assignment statements and each edge $(u, v)\in E$ specifies a possible flow of control caused by a branch from a test statement. An execution of π induces a path in G beginning at the *start vertex* r. We shall denote the number of vertices in G by n and the number of edges in G by m.

Let $\Sigma=\{X, Y, Z, \cdots\}$ be the set of program variables occurring within π. A program variable $X\in\Sigma$ is *defined* at $v\in V$ if X occurs on the left-hand side of an assignment statment of v. For each program variable $X\in\Sigma$ and vertex $v\in V$, we let the *entry variable* X^v denote the value of X on entry to v.

* Received by the editors June 5, 1979, and in final form March 2, 1981.

† Aiken Computer Laboratory, Harvard University, Cambridge, Massachusetts 02138. The work of this author was supported by Naval Electronics System Command contract N00039-76-C-0168 and Rome Air Development Center contract F30602-76-C-0032.

‡ Department of Computer Science, Stanford University, Stanford, California 94305. Present address: Bell Laboratories, 600 Mountain Ave., Murray Hill, New Jersey 07974. The work of this author was supported by National Science Foundation grant MCS-75-22870 A02, by Office of Naval Research contract N00014-76-C-0330, by a Guggenheim fellowship, and by Bell Laboratories.

[1] The appendix contains the graph-theoretic terminology we employ.

Let θ be the set of function signs occurring in the program. For simplicity, we assume a domain D such that every k-ary function represented by a sign in θ has the same domain D^k. Let C be a set of constant signs containing a unique sign for every element in D. Let EXP be the set of expressions built from entry variables, constant signs in C, and function signs in θ. To each expression $\mathscr{E} \in \text{EXP}$ corresponds a unique *reduced expression* $\mathscr{E}_R$ formed by repeatedly substituting the appropriate constant sign for each subexpression of $\mathscr{E}$ consisting of a function sign applied to constant signs.

For any expression $\mathscr{E} \in \text{EXP}$ and any execution of the program π, the *value* of $\mathscr{E}$ on exit from a vertex v is defined as follows: If $\mathscr{E}$ contains an entry variable X^u such that control has never entered u, then the value of $\mathscr{E}$ is undefined. Otherwise the value of $\mathscr{E}$ is computed by substituting for each entry variable X^u the value of X when control last entered u, and evaluating the resulting expression.

For each vertex $v \in V$ and program variable $X \in \Sigma$ defined at v, the *exit expression* $\mathscr{E}(X, v) \in \text{EXP}$ is formed as follows. Begin by letting the expression $\mathscr{E}$ be X. Process each assignment statement of v, starting from the last assignment defining X and working backwards to the first assignment in v. To process an assignment $Y := \mathscr{E}'$, replace each occurrence of Y in $\mathscr{E}$ by $\mathscr{E}'$. After all assignments are processed, reduce $\mathscr{E}$ and replace each occurrence of a variable Y by the corresponding entry variable Y^v. The resulting exit expression $\mathscr{E}(X, v)$ represents the value of X on exit from v in terms of constants and values of variables on entry to v. For example, $\mathscr{E}(Z, v_2) = Z^{v_2} + (X^{v_2} * Y^{v_2})$ represents the value of Z on exit from vertex v_2 in the flow graph of Fig. 1.

A *text expression* is any subexpression of an exit expression $\mathscr{E}(X, v)$ (including the expression itself); we say the text expression *occurs at* v. An expression $\mathscr{E} \in \text{EXP}$ *covers* a text expression t occurring at v if for any execution of program π, $\mathscr{E}$ and t have the

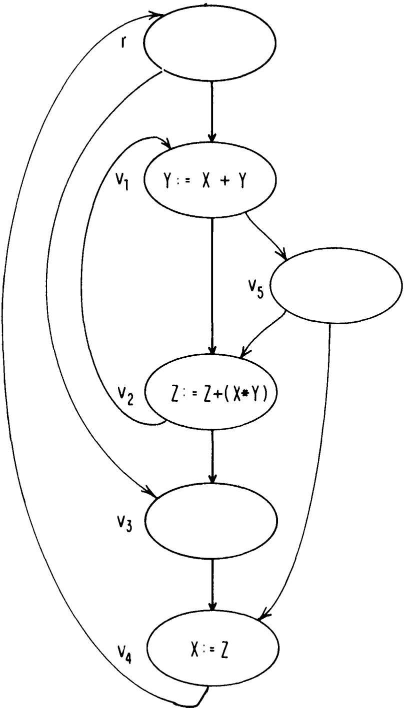


FIG. 1. *A program flow graph.*

same value on any exit from v. See Fig. 1. This definition implies that if X^u appears in $\mathscr{E}$ then u dominates v. Thus there is a unique vertex v which is minimal (i.e., closest to the start vertex) with respect to the dominator relation and such that for all entry variables X^u in $\mathscr{E}$, u dominates v. We call such a vertex the *origin* of $\mathscr{E}$; it is the earliest point in the program at which $\mathscr{E}$ can be computed.

A cover of π is a mapping Ψ from all text expressions to reduced expressions in EXP, such that, for each text expression t, $\Psi(t)$ covers t. We would like to construct a cover whose origins are minimal with respect to the dominator relation. We can use such a cover for constant propagation: if a constant sign c covers a text expression t, we may substitute c in line in the program text for the computation associated with c.

We can also use a cover in code motion. If we define the *birthpoint* of a text expression t to be the minimal vertex to which the computation of t may be moved, then the birthpoint of t is precisely the origin of a minimal cover of t. For example, in Fig. 1 the birthpoint of text expression $t = X^{v_2} * Y^{v_2}$ is v_1; $X^r * (X^r + Y^{v_1})$ covers t. Code

Text expression	Covering expressions
X^{v_2}	X^r
Y^{v_2}	$X^r + Y^{v_1}$
Z^{v_2}	Z^{v_1}
$\mathscr{E}(Z, v_2) = Z^{v_2} + (X^{v_2} * Y^{v_2})$	$Z^{v_1} + (X^r * (X^r + Y^{v_1}))$

FIG. 2. *Symbolic analysis of the program in Fig.* 1.

motion requires approximations to birthpoints (i.e., vertices which are dominated by the true birthpoints) and other knowledge including knowledge of the cycle structure of the flow graph of π. (We may not wish to move code as far as the birthpoint since the birthpoint may be contained in control cycles avoiding the original location of the code.) [R1] presents efficient algorithms which utilize approximate birthpoints for code motion optimization. See [AU], [CA], [E], [G] for further discussion of code motion optimizations. Other practical uses of covers have been made by [FK] in their optimizing Pascal compiler.

Unfortunately, for programs which manipulate the natural numbers using ordinary arithmetic the problem of computing a minimal cover is recursively unsolvable [R2]. The usual approach in program optimization is to trade accuracy for speed; [FKU], [Ki], [RL], [R2] present fast algorithms which compute reasonably good covers whose origins yield approximate birthpoints. The fastest of these [RL], [R2] has a time bound almost linear in $m \cdot |\Sigma| + l$, where l is the length of the program text.

In this paper we describe a very fast algorithm for computing a rather weak cover. This *simple cover* can be used directly for code optimization, or it can serve as input to a more powerful method for symbolic evaluation presented in [RL], [R2]. From a data structure called a *global value graph* (which is related to the use-definition chains of [AU], [Sc] used to represent the flow of values through a program), the algorithm of [RL], [R2] constructs a cover which yields better approximate birthpoints than does the simple cover. This algorithm runs in time almost linear in the size of the input global value graph, which is very compact when constructed from the simple cover [RL], [R2].

In order to define the simple cover we need one more concept. A variable X is *definition-free* between distinct vertices u and v if no u-avoiding path from a successor of u to a predecessor of v contains a definition of X. By convention any program variable X is definition-free between v and v for any vertex v. For any entry variable X^v which is a text expression, the *simple origin* of X^v is the minimal vertex u (with respect

to the dominator relation) such that X is definition-free between u and v. In the example of Fig. 1, X^{v_2} has a simple origin r, and Y^{v_2} and Z^{v_2} have simple origin v_1. If X^v has simple origin $u \neq v$, then on any execution of π the program variable X has the same value on entry to v as it did after the most recent execution of u; we take the simple origin as an approximation to the birthpoint of X^v.

We recursively define the simple cover Ψ using simple origins. If t contains no entry variables then $\Psi(t)=t$. Otherwise we form $\Psi(t)$ from t by applying the following transformation.

(i) Repeat the following step for all entry variables X^v occurring in t: Let u be the simple origin of X^v. If $u=v$ do nothing. Otherwise replace X^v in t by $\Psi(\mathscr{E}(X, u))$ if X is defined at u or by X^u if X is not defined at u.

(ii) Reduce the resulting expression.

Our algorithm for computing the simple cover consists of three parts, described in §§ 2–4 of this paper. First, we determine for each vertex v the set of program variables defined between the immediate dominator of v and v itself. We call this set of variables idef (v). The idef computation can be regarded as a path problem of the kind studied in [GW], [T3], but another approach is more fruitful: a straightforward modification of the dominator-finding algorithm of [LT] computes idef in $O(m\alpha(m, n)+l)$ time, assuming that logical bit vector operations on vectors of length $|\Sigma|$ have unit cost, where l is the length of the program text and α is related to an inverse of Ackermann's function [T2].

Second, we use idef to compute the simple origins of all entry variables appearing as text expressions. This computation requires a variable-length shift operation on bit vectors (shift left to the first nonzero bit) and requires $O(n+l)$ time. Third, we construct a directed acyclic graph representing the simple cover (we save space by combining common subexpressions). This algorithm also requires $O(n+l)$ time but uses no bit vector operations. The total running time of our algorithm is thus $O(m\alpha(m, n)+l)$ if extended bit vector operations require constant time.

2. An algorithm for computing idef based on finding dominators. In this section we shall describe an algorithm for computing idef (v) for all vertices $v \in V$ in the flow graph $G=(V, E, r)$ of a computer program. We obtain the algorithm by adding appropriate extra steps to the dominators algorithm of [LT], and we shall assume that the reader is familiar with [LT]. Our algorithm requires def $(w)=\{X | X$ is defined at $w\}$ for each vertex $w \in V$ as input and uses set union as a basic operation. If each subset of Σ is represented as a bit vector of length $|\Sigma|$, then a set union is equivalent to an "or" operation on bit vectors; we shall assume each set union requires constant time. Construction of def (w) for all vertices w is easy and requires time proportional to the length of the program text.

Properties of idef. For any vertex $w \neq r$, let idom (w) be the immediate dominator of w in G. For $w \neq r$, we define idef$(w)= \bigcup$ {def $(v)|$ there is a nonempty path from v to w which avoids idom $(w)\}$. Note that def (w) is a term in the union defining idef (w) if and only if there is a cycle containing w but avoiding idom(w). To compute idom and idef, we first perform a depth-first search on G, starting from vertex r and numbering the vertices from 1 to n as they are reached during the search. The search generates a spanning tree T rooted at r, with vertices numbered in preorder [T1]. For convenience in stating our results, we shall assume in this subsection that all vertices are identified by number, and we shall use $\rightarrow$, $\overset{*}{\rightarrow}$, $\overset{+}{\rightarrow}$ to denote ancestor-descendant relationships in T (see the appendix).

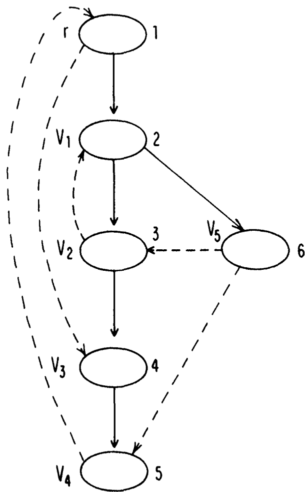


FIG. 3. *Depth-first search of the flow graph given in Fig. 1. Solid edges denote tree edges and dotted edges denote nontree edges. The depth-first search number is given to the right of each vertex.*

vertex	number	idom	sdom	def	idef	sdef
r	1	—	—	$\varnothing$	—	—
v_1	2	r	r	$\{Y\}$	$\{Y, Z\}$	$\{Y, Z\}$
v_2	3	v_1	v_1	$\{Z\}$	$\varnothing$	$\varnothing$
v_3	4	r	r	$\varnothing$	$\{Y, Z\}$	$\varnothing$
v_4	5	r	v_1	$\{X\}$	$\{Y, Z\}$	$\varnothing$
v_5	6	v_1	v_1	$\varnothing$	$\varnothing$	$\varnothing$

FIG. 4. *Tabulation of information calculated for the program flow graph given in Fig. 1.*

The following *paths lemma* is an important property of depth-first search and is crucial to the correctness of our algorithm.

LEMMA 2.1 [T1]. *If v and w are vertices of G such that $v \leqq w$, then any path from v to w must contain a common ancestor of v and w in T.*

As an intermediate step, the dominators algorithm computes a value for each vertex $w \neq r$ called its *semi-dominator*, denoted sdom (w) and defined by

$$\text{sdom}(w) = \min \{v \mid \text{there is a path } v = v_0, v_1, \cdots, v_k = w \text{ such that } v_i > w \text{ for } 1 \leqq i < k\}. \tag{1}$$

We shall in addition compute a value sdef (w) for each vertex $w \neq r$ defined by

$$\text{sdef}(w) = \bigcup \{\text{def}(v) \mid \text{there is a nonempty path } v = v_0, v_1, \cdots, v_k = w \text{ such that } v_i \geqq w \text{ for } 0 \leqq i \leqq k\}. \tag{2}$$

The following properties of semi-dominators and dominators justify the dominators algorithm.

LEMMA 2.2 [LT]. *Let $w \neq r$. Then* idom $(w) \overset{*}{\rightarrow}$ sdom $(w) \overset{+}{\rightarrow} w$.

THEOREM 2.1 [LT]. *For any vertex $w \neq r$,*

$$\text{sdom}(w) = \min (\{v \mid (v, w) \in E \text{ and } v < w\} \cup \{\text{sdom}(u) \mid u > w \text{ and there is an edge } (v, w) \text{ such that } u \overset{*}{\rightarrow} v\}). \tag{3}$$

THEOREM 2.2 [LT]. *Let $w \neq r$ and let u be a vertex for which* sdom (u) *is minimum among vertices u satisfying* sdom $(w) \overset{+}{\rightarrow} u \overset{*}{\rightarrow} w$. *Then*

$$\text{(4)} \qquad \text{idom}(w) = \begin{cases} \text{sdom}(w) & \textit{if } \text{sdom}(w) = \text{sdom}(u), \\ \text{idom}(u) & \textit{otherwise.} \end{cases}$$

The dominators algorithm uses Theorem 2.1 to efficiently compute semi-dominators and Theorem 2.2 to efficiently compute immediate dominators. We shall use two analogous results to efficiently compute sdef and idef.

THEOREM 2.3. *Let $w \neq r$ and let*

$$\text{adef}(w) = \{\text{def}(u) \cup \text{sdef}(u) \mid u > w \text{ and there is an edge } (v, w) \text{ such that } \overset{*}{\rightarrow} v\}.$$

Then

$$\text{(5)} \qquad \text{sdef}(w) = \begin{cases} \text{def}(w) \cup \text{adef}(w) & \textit{if there is an edge } (v, w) \textit{ such that } w \overset{*}{\rightarrow} v, \\ \text{adef}(w) & \textit{otherwise.} \end{cases}$$

Proof. First we show that the right side of (5) contains sdef (w). Let $v = v_0, v_1, \cdots, v_k = w$ be a nonempty path such that $v_i \geqq w$ for $0 \leqq i \leqq k$. We can assume without loss of generality that the path $v_0, v_1, \cdots, v_{k-1}$ is simple and $v_i \neq w$ for $1 \leqq i \leqq k-1$. Let j be minimum such that $v_j \overset{*}{\rightarrow} v_{k-1}$. By Lemma 2.1, $v_i > v_j$ for $0 \leqq i \leqq j-1$. We consider three cases. If $j \neq 0$, then $v_j \neq w$, and def $(v) \subseteq$ sdef $(v_j) \subseteq$ adef (w). If $j = 0$ and $v \neq w$, then def $(v) \subseteq$ adef (w). If $j = 0$ and $v = w$, then the edge (v_{k-1}, w) must satisfy $w \overset{*}{\rightarrow} v_{k-1}$, and the right side of (5) explicitly contains def (v). Thus in any case the right side of (5) contains def (v). Since this is true for any appropriate v, the right side of (5) contains sdef (w).

Now we show that sdef (w) contains the right side of (5). Suppose there is an edge (v, w) such that $w \overset{*}{\rightarrow} v$. Then the path consisting of the tree path from w to v followed by the edge (v, w) contains no vertices smaller than w, and def $(w) \subseteq$ sdef (w). Let u be a vertex such that $u > w$ and there is an edge (v, w) such that $u \overset{*}{\rightarrow} v$. Let x be any vertex for which there is a nonempty path $x = v_0, v_1, \cdots, v_k = u$ such that $v_i \geqq u$ for $0 \leqq i \leqq k$. Then this path, followed by the tree path from u to v, followed by the edge (v, w), contains no vertices smaller than w. Thus def $(x) \subseteq$ sdef (w). Since this is true for any such x, sdef $(u) \subseteq$ sdef (w). Furthermore def $(u) \subseteq$ sdef (w). It follows that adef $(w) \subseteq$ sdef (w), and the theorem is true. $\square$

THEOREM 2.4. *Let $w \neq r$. Let u be a vertex for which* sdom (u) *is minimum among vertices u satisfying* sdom $(w) \overset{+}{\rightarrow} u \overset{*}{\rightarrow} w$. *Let* tdef $(w) = \bigcup\{\text{def}(x) \cup \text{sdef}(x) \mid \text{sdom}(w) \overset{+}{\rightarrow} x \overset{+}{\rightarrow} w\}$. *Then*

$$\text{(6)} \qquad \text{idef}(w) = \begin{cases} \text{tdef}(w) \cup \text{sdef}(w) & \textit{if } \text{sdom}(w) = \text{sdom}(u), \\ \text{idef}(u) \cup \text{tdef}(w) \cup \text{sdef}(w) & \textit{otherwise.} \end{cases}$$

Proof. First we show that the right side of (6) contains idef (w). Let $v = v_0, v_1, \cdots, v_k = w$ be a nonempty path which avoids idom (w). Let v_i be the minimum vertex on this path such that $i \leqq k-1$. If $v_i \geqq w$, then def $(v) \subseteq$ sdef (w) by the definition of sdef.

Suppose on the other hand that $v_i < w$. By Lemma 2.1, there is some j in the range $i \leqq j \leqq k$ such that v_j is an ancestor of both v_i and w. This means $v_j \leqq v_i$. But by the definition of v_i, $v_i \leqq v_j$. Thus, $v_i = v_j$ and $v_i \overset{+}{\rightarrow} w$. We must consider three cases.

(i) Suppose sdom $(w) \overset{+}{\rightarrow} v_i$ and $i = 0$. Then def $(v) =$ def $(v_i) \subseteq$ tdef.

(ii) Suppose sdom $(w) \overset{+}{\rightarrow} v_i$ and $i \neq 0$. Then def $(v) \subseteq$ sdef $(v_i) \subseteq$ tdef.

(iii) Suppose $v_i \overset{*}{\to}$ sdom (w). The path from r to w consisting of the tree path from r to v_i followed by the path $v_i, v_{i+1}, \cdots, w$ must contain idom (w); thus idom $(w) \overset{+}{\to} v_i$. By Theorem 2.2, sdom $(w) \neq$ sdom (u) (which means the second half of (6) applies) and idom $(w) =$ idom (u). The path from v to u consisting of $v = v_0, v_1, \cdots, v_i$ followed by the tree path from v_i to u avoids idom (u), which means def $(v) \subseteq$ idef (u).

In all cases def (v) is contained in the right side of (6); since this is true for any appropriate v, idef (w) is contained in the right side of (6) by the definition of idef.

It remains to show that idef (w) contains the right side of (6). Let x be any vertex such that sdom $(w) \overset{+}{\to} x \overset{+}{\to} w$, and let $v = v_0, v_1, \cdots, v_k = x$ be any path such that $v_i \geqq x$ for $0 \leqq i \leqq k$. Since idom $(w) \overset{*}{\to}$ sdom (w), the path from v to w consisting of the path $v = v_0, v_1, \cdots, v_k = x$ followed by the tree path from x to w avoids idom (w). It follows that tdef $\subseteq$ idef (w). Since idom $(w) < w$, it is immediate that sdef $(w) \subseteq$ idef (w). If sdom $(w) \neq$ sdom (u), then idom $(w) =$ idom $(u) \overset{+}{\to} u$, and any idom (u)-avoiding path to u can be extended to an idom (w)-avoiding path to w by adding the tree path from u to w. Thus in this case idef $(u) \subseteq$ idef (w) □

Details of the algorithm. The algorithm for computing immediate dominators and idef consists of the following four steps.

Step 1. Carry out a depth-first search of the problem graph. Number the vertices from 1 to n as they are reached during the search. Initialize the variables used in succeeding steps.

Step 2. Compute the semi-dominators of all vertices by applying Theorem 2.1 and the sdef values by applying Theorem 2.3. Carry out the computation vertex-by-vertex in decreasing order by number.

Step 3. Implicitly define the immediate dominator of each vertex by applying Theorem 2.2 and partially compute idef values by applying Theorem 2.4.

Step 4. Explicitly define the immediate dominator of each vertex and finish computing idef. Carry out the computation vertex-by-vertex in increasing order by number.

The dominators algorithm used the following arrays.

Input

succ (v): The set of vertices w such that (v, w) is an edge of the graph.

Computed

parent (w): The vertex which is the parent of vertex w in the spanning tree generated by the search.

pred (w): The set of vertices v such that (v, w) is an edge of the graph.

semi (w): A number defined as follows:
- (i) Before vertex w is numbered, semi $(v) = 0$.
- (ii) After w is numbered but before its semi-dominator is computed, semi (w) is the number of w.
- (iii) After the semi-dominator of w is computed, semi (w) is the number of the semi-dominator of w.

vertex (i): The vertex whose number is i.

bucket (w): A set of vertices whose semi-dominator is w.

dom (w): A vertex defined as follows:
- (i) After Step 3, if the semi-dominator of w is its immediate dominator, then dom (w) is the immediate dominator of w. Otherwise dom (w) is a vertex v whose number is smaller than that of w and whose immediate dominator is also the immediate dominator of w.
- (ii) After Step 4, dom (w) is the immediate dominator of w.

In addition, our algorithm uses def (w) as input and computes sdef (w) and idef (w).

Rather than converting vertex names to numbers during Step 1 and converting numbers back to names at the end of the computation, the dominators algorithm refers to vertices as much as possible by name. Arrays semi and vertex include all necessary information about vertex numbers. Array semi serves a dual purpose, representing (though not simultaneously) both the number of a vertex and its semi-dominator.

During Step 1, our algorithm initializes parent, pred, semi, vertex, and sdef. When a vertex w receives a number i, the algorithm assigns semi $(w)=i$ and vertex $(i)=w$. Step 1 also initializes sdef $(w)=\emptyset$ and updates sdef $(w)=$ def (w) if it finds an edge (v, w) such that $w \stackrel{*}{\rightarrow} v$. Implementation of Step 1 is straightforward, and we omit the details.

The algorithm carries out Steps 2 and 3 simultaneously, processing the vertices $w \neq r$ in decreasing order by number. During this computation the algorithm maintains an auxiliary data structure that represents a forest contained in the depth-first spanning tree. More precisely, the forest consists of vertex set V and edge set $\{(\text{parent}(w), w) | \text{vertex } w \text{ has been processed}\}$. The algorithm uses one procedure to construct the forest and two procedures to extract information from it.

LINK (v, w):	Add edge (v, w) to the forest.
EVAL (v):	If v is the root of a tree in the forest, return v. Otherwise, let r be the root of the tree in the forest which contains v. Return any vertex $u \neq r$ of minimum semi (u) on the path $r \stackrel{*}{\rightarrow} v$.
EVALDEF (v):	If v is a tree root, return $\emptyset$. Otherwise, let $r = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k = v$ be the tree path from the root of the tree containing v to v. Return $\bigcup\{\text{def}(v_i) \cup \text{sdef}(v_i) | 1 \leqq i \leqq k\}$.

Reference [LT] explains how to use EVAL to compute semi-dominators and dominators; we shall describe how to use EVALDEF analogously to compute sdef and idef. When a vertex w is processed, the algorithm examines each edge $(v, w) \in E$ and updates sdef by assigning sdef $(w) :=$ sdef $(w) \cup$ EVALDEF (v). After w is processed, sdef (w) has the proper value by Theorem 2.3. To verify this claim, consider any edge $(v, w) \in E$. If v is numbered no greater than w, then v is unprocessed when (v, w) is examined, which means v is the root of a tree in the forest and EVALDEF (v) returns $\emptyset$. If v is numbered greater than w, then EVALDEF returns $\bigcup\{\text{def}(u) \cup \text{sdef}(u) | u > w \text{ and } u \rightarrow^* v\}$. Thus the algorithm computes sdef exactly as specified in Theorem 2.3.

After processing w to compute semi (w) and sdef (w), the algorithm adds w to bucket (vertex (semi (w))) and adds a new edge to the forest using LINK (parent (w), w). This completes Step 2 for w. The algorithm then empties bucket (parent (w)), carrying out Step 3 for each vertex v in the bucket. By applying EVAL (v), the algorithm obtains a vertex u satisfying the condition in Theorem 2.2 and 2.4. Using this u, the algorithm implicitly computes the immediate dominator of v. The algorithm also partially computes idef (v) by assigning idef (v):=sdef $(v) \cup$ EVALDEF (parent (v)). (EVALDEF (parent (v)) = tdef (v) as defined in Theorem 2.4.) In Step 4, the algorithm examines vertices in increasing order by number, filling in the immediate dominators not explicitly computed by Step 3 and completing the computation of idef. Here is an Algol-like version of Steps 2–4. The bracketed statements are those added to the original dominators algorithm to compute sdef and idef.

```
            comment initialize variables;
            for i := n by −1 until 2 do
                w := vertex (i);
Step 2:         for each v ∈ pred (w) do
                    u := EVAL (v);
                    if semi (u) < semi (w) then semi (w) := semi (u) fi;
                    [sdef (w) := sdef (w) ∪ EVALDEF (v)] od;
                add w to bucket (vertex (semi (w)));
                LINK (parent (w), w);
Step 3:         for each v ∈ bucket (parent (w)) do
                    delete v from bucket (parent (w));
                    u := EVAL (v);
                    dom (v) := if semi (u) < semi (v) then u
                               else parent (w) fi;
                    [idef (v) := sdef (v) ∪ EVALDEF (parent (v))] od od;
Step 4:     for i := 2 until n do
                w := vertex (i);
                if dom (w) ≠ vertex (semi (w)) then
                    [idef (w) := idef (dom) (w)) ∪ idef (w);]
                    dom (w) := dom (dom (w)) fi od;
```

Reference [T2] offers two ways to implement LINK, EVAL, and EVALDEF. The simpler method has an $O(m \log n)$ time bound and the more complicated one has an $O(m\alpha(m, n))$ time bound. Farrow [F] provides another $O(m\alpha(m, n))$ method. If we include the $O(l)$ time required to construct def from the program text, then the entire algorithm for computing idef requires $O(m\alpha(m, n)+l)$ time, assuming that each set union requires constant time.

3. Computing simple origins. Once we know def and idef, we can employ the following theorem to compute simple origins. It is convenient for us to assume that idef $(r)=\Sigma$.

THEOREM 3.1. *Let X^v be an entry variable which is a text expression. Then*

$$\text{simple origin } (X^v) = \begin{cases} v & \text{if } X \in \text{idef}(v), \\ u & \text{if } X \notin \text{idef}(v) \text{ and } u \text{ is the maximal proper dominator of} \\ & \quad v \text{ such that } X \in \text{def}(u) \cup \text{idef}(u). \end{cases} \tag{7}$$

Proof. Recall that X^v occurs at v. The theorem is immediate from the definitions of simple origin, def, and idef, using the fact that idef $(r)=\Sigma$. □

In order to use Theorem 3.1 efficiently, we need to compute two additional subsets of variables for each vertex. For any vertex $v \in V$, text (v) is the set of variables X such that X^v is a text expression. We can compute text in $O(l)$ time by scanning the program text. For any vertex $v \in V$, relevant (v) is the set of variables X such that, for some vertex w properly dominated by v, X^w is a text expression and X is definition-free between v and w.

THEOREM 3.2. *For any vertex v,*

$$\text{relevant }(v) = \bigcup \{(\text{text }(w) \cup \text{relevant }(w)) - \text{idef }(w) \mid w \in V \text{ and idom }(w) = v\}.$$

Proof. Immediate. □

We can compute relevant in $O(n)$ time by carrying out a depth-first traversal of the dominator tree and processing the vertices in postorder. Note that, for any vertex v, the

set relevant $(v) \cap (\text{def}(v) \cup \text{idef}(v))$ contains exactly the variables X such that, for some vertex w, v is the simple origin of the text expression X^w.

Given text and relevant, we compute simple origins in another depth-first traversal of the dominator tree. During the traversal, we maintain a stack for each variable X. When the traversal reaches a vertex $v \neq r$, stack (X) contains (in dominator order) all proper dominators u of v such that $X \in \text{relevant}(u) \cap (\text{def}(u) \cup \text{idef}(u))$. These vertices are all the candidates (other than v) for the simple origin of X^v. If $X \in \text{idef}(v)$, then the simple origin of X^v is v; otherwise the simple origin of X^v is the top vertex on stack (X) when v is reached during the traversal. The following algorithm computes simple origins using this method.

```
procedure TRAVERSE (v);
    begin
        for each X ∈ test(v) do
            simple origin (X^v) := if X ∈ idef (v) then v
                                   else top of stack (X) fi od;
        for each X ∈ relevant (v) ∩ (def (v) ∪ (idef (v)) do
            push v on stack (X) od;
        for each w in {w|idom (w) = v} do TRAVERSE (w) od;
        for each X ∈ relevant (v) ∩ (def (v) ∪ idef (v)) do
            pop v from stack (X) od
    end TRAVERSE;
for each X ∈ Σ do stack (X) = ∅ od;
TRAVERSE (r);
```

The correctness of the algorithm is immediate. To get the algorithm to run fast, we need a method to convert a bit vector representing a set into a list of elements of the set. We can do this in time proportional to the size of the set if we have a variable-length shift operation which shifts a bit vector left to the first nonzero bit and returns the length of the shift. Since such an operation is required to normalize floating-point numbers, it is a machine-language instruction on many computers. Assuming that a variable-length shift requires constant time, the time required to compute simple origins is

$$O\left(n + \sum_{v \in V} (|\text{text}(v)| + |\text{relevant}(v) \cap (\text{def}(v) \cup \text{idef}(v))|)\right) = O(n + l)$$

since each variable $X \in \text{text}(v)$ corresponds to an appearance of X in the program text at vertex v, and each variable $X \in \text{relevant}(v) \cap (\text{def}(v) \cup \text{idef}(v))$ corresponds to a text expression X^w for which v is the simple origin.

4. Computing the simple cover and approximate birthpoints. From the simple origins, it is easy to construct the simple cover Ψ and an approximate birthpoint for each text expression. We begin by constructing a directed acyclic graph (*dag*) to represent all text expressions in the program. We shall call the vertices in this dag *nodes* to distinguish them from the vertices of the control flow graph. The dag has one node representing each text expression. An expression which is a constant sign or an entry variable X^v is represented by a sink labeled by the appropriate constant sign or entry variable; an expression of the form $\theta(E_1, E_2, \cdots, E_k)$ is represented by a node labeled with θ having k (ordered) successors representing the expressions $E_1, E_2, \cdots, E_k$. An example appears in Fig. 5. See [AU], [FKU] for further discussion of this representation. It is easy to construct a dag representing the text expressions in $O(l)$ time.

We convert the dag representing the text expressions into a dag representing the simple cover as follows. We process the sinks of the dag labeled by entry variables X^v in

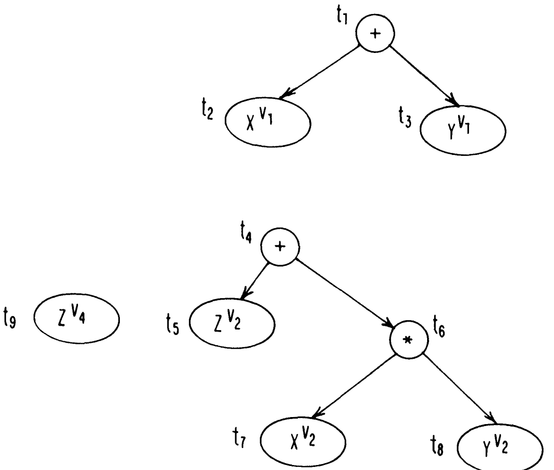


FIG. 5. *Dags representing the text expression of the program in Fig.* 1.

an order consistent with the dominator order; i.e., if v dominates w, we process sinks labeled X^v before sinks labeled X^w. We process sinks labeled X^v as follows. Let u be the simple origin of X^v. If $u = v$ we do nothing. *If* $u \neq v$ and X is defined at u, we replace all edges leading to sinks labeled X^v by edges leading to the node corresponding to exit expression $\mathscr{E}(X, u)$. (This node now represents $\Psi(\mathscr{E}(S, u))$.) If $u \neq v$ and X is not defined at u, we replace the labels X^v by labels X^u. This method requires $O(l)$ time.

We apply two more steps to simplify the resulting dag. First we replace each node all of whose successors represent constants by a sink representing an appropriate constant. We repeat this transformation until it is no longer applicable. This requires $O(l)$ time and produces a dag representing a set of reduced expressions. Next, we merge

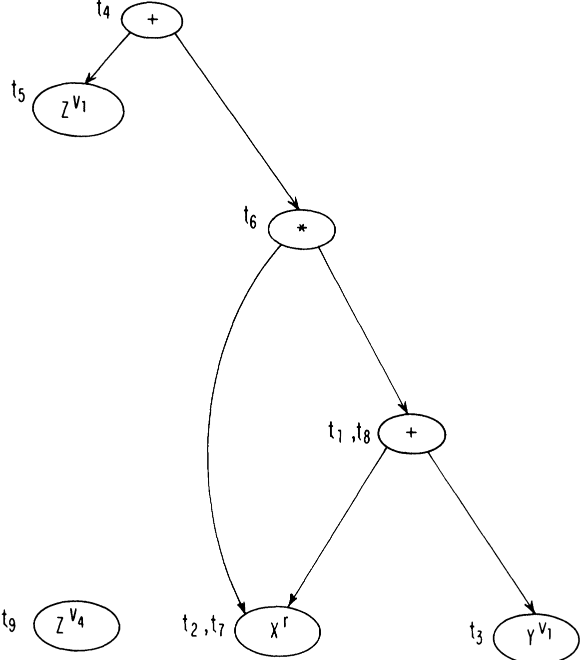


FIG. 6. *Dag representing the simple cover of the program in Fig.* 1.

all nodes representing common subexpressions. This can be done in $O(l)$ time using the acyclic congruence closure algorithm described in [DST]. The result is a dag representing the simple cover. See Fig. 6.

We can compute an approximate birthpoint for each text expression by processing the nodes of the dag representing the simple cover in reverse topological order. Each sink labeled by a constant has approximate birthpoint r. Each sink labeled X^v has approximate birthpoint v. Each node with successors has an approximate birthpoint which is the maximal vertex (with respect to the dominator relation) of the approximate birthpoints of its successors. The approximate birthpoint of a text expression is the approximate birthpoint of the corresponding node in the simple cover dag. (Thus our birthpoints are approximated in part by the simple origins which we computed in § 3.) This computation also requires $O(l)$ time, giving a total of $O(l)$ time to compute both a simple cover and approximate birthpoints.

By combining the algorithms of §§ 2, 3, and 4, we obtain a symbolic evaluation method which requires $O(m\alpha(m, n)+l)$ time if extended bit vector operations require constant time.

Appendix. Graph-theoretic terminology. A *directed graph* $G=(V, E)$ consists of a finite set V of *vertices* and a set E of ordered pairs (v, w) of vertices, called *edges*. If (v, w) is an edge, w is a *successor* of v and v is a *predecessor* of w. A *sink* is a vertex with no successors. A graph $G_1=(V_1, E_1)$ is a *subgraph* of G if $V_1 \subseteq V$ and $E_1 \subseteq E$. A *path* p of *length* k from v to w in G is a sequence of vertices $p=(v=v_0, v_1, \cdots, v_k=w)$ such that $(v_i, v_{i+1}) \in E$ for $0 \leqq i < k$. The path is *simple* if $v_0, \cdots, v_k$ are distinct (except possibly $v_0=v_k$) and the path is a *cycle* if $v_0=v_k$. By convention there is a path of no edges from every vertex to itself but a cycle must contain at least one edge. If $p_1=(u=u_0, u_1, \cdots, u_k=v)$ is a path from u to v and $p_2=(v=v_0, v_1, \cdots, v_l=w)$ is a path from v to w, the path p_1 *followed by* p_2 is $p=(u=u_0, u_1, \cdots, u_k=v=v_0, v_1, \cdots, v_l=w)$. A directed graph is *acyclic* if it contains no cycles. A *topological order* on an acyclic graph is a total ordering of the vertices such that, for each edge (v, w), v is ordered before w.

A *flow graph* $G=(V, E, r)$ is a directed graph (V, E) with a distinguished *start vertex* r such that for any vertex $v \in V$ there is a path from r to v. A (*directed, rooted*) *tree* $T=(V, E, r)$ is a flow graph such that $|E|=|V|-1$. The start vertex r is the *root* of the tree. Any tree is acyclic, and if v is any vertex in a tree T, there is a unique path from r to v. If v and w are vertices in a tree T and there is a tree path from v to w, then v is an *ancestor* of w and w is a *descendant* of v (*denoted by* $v \overset{*}{\rightarrow} w$). If in addition $v \neq w$, then v is a *proper ancestor* of w and w is a *proper descendant* of v (denoted by $v \overset{+}{\rightarrow} w$). If $v \overset{*}{\rightarrow} w$ and (v, w) is an edge of T (denoted by $v \rightarrow w$), then v is the *parent* of w and w is a *child* of v. In a tree each vertex has a unique parent (except the root, which has no parent). If $G=(V, E)$ is a graph and $T=(V', E', r)$ is a tree such that (V', E') is a subgraph of G and $V'=V$, then T is a *spanning tree* of G.

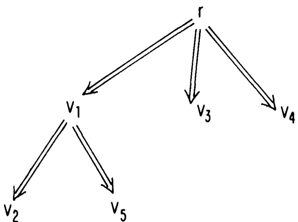


FIG. 7. *Dominator tree of the flow graph given in Fig.* 1. *The symbol* $\Rightarrow$ *leads from* idom (v) *to vertex* v.

If $G = (V, E, r)$ is a flow graph and $u, v \in V$, then *u dominates v* if all paths from r to v contain u. The dominator relation is a partial ordering with minimal element r. If u dominates v and $u \neq v$, then *u properly dominates v*. It can be shown that, for each vertex $v \neq r$, there is a unique vertex u called the *immediate dominator* of v which properly dominates v and is dominated by all other dominators of v. We denote the immediate dominator of v by idom (v). The tree $T = (V, E', r)$ with $E' = \{(\text{idom}(v), v) | v \neq r\}$ is the *dominator tree* of G.

REFERENCES

[AU] A. V. AHO AND J. D. ULLMAN, *Introduction to Compiler Design*, Addison-Wesley, Reading, MA, 1977, pp. 441–477.

[CA] J. COCKE AND F. E. ALLEN, *A catalogue of optimization transformations*, Design and Optimization of Computers, R. Rustin, ed., Prentice-Hall, Englewood Cliffs, NJ, 1971, pp. 1–31.

[DST] P. J. DOWNEY, R. SETHI AND R. E. TARJAN, *Variations on the common subexpression problem*, J. Assoc. Comput. Mach., 27 (1980), pp. 758–771.

[E] C. EARNEST, *Some topics in code optimization*, J. Assoc. Comput. Mach., 21 (1974), pp. 76–102.

[F] R. FARROW, *Efficient variants of path compression in unbalanced trees*, unpublished manuscript (1978).

[FK] R. N. FAIMAN AND A. A. KORTESOJA, *An optimizing Pascal compiler*, IEEE Trans. Software Engineering, SE-6 (1980), pp. 512–519.

[FKU] E. A. FONG, J. B. KAM AND J. D. ULLMAN, *Application of lattice algebra to loop optimization*, Conf. Record Second ACM Symposium on Principles of Programming Languages, January, 1975, pp. 1–9.

[G] C. M. GESCHKE, *Global program optimizations*, Ph.D. thesis, Computer Science Dept., Carnegie-Mellon University, Pittsburgh, PA, 1972.

[GW] S. GRAHAM AND M. WEGMAN, *A fast and usually linear algorithm for global flow analysis*, J. Assoc. Comput. Mach., 23 (1976), pp. 172–202.

[HU] M. S. HECHT AND J. D. ULLMAN, *Flow graph reducibility*, this Journal, 2 (1972), pp. 188–202.

[Ki] G. A. KILDALL, *A unified approach to global program optimization*, Proc. ACM Symposium on Principles of Programming Languages, Boston, 1973, pp. 194–206.

[LT] R. LENGAUER AND R. E. TARJAN, *A fast algorithm for finding dominators in a flow graph*, ACM Trans. Programming Languages and Systems, 1 (1979), pp. 121–141.

[R1] J. H. REIF, *Code motion*, this Journal, 9 (1980), pp. 375–395.

[R2] ———, *Combinatorial aspects of symbolic program analysis*, Ph.D. thesis, Division of Engineering and Applied Physics, Harvard University, Cambridge, MA, 1977.

[RL] J. H. REIF AND H. R. LEWIS, *Symbolic evaluation and the global value graph*, Proc. 4th ACM Symposium on Principles of Programming Languages, 1977.

[Sc] J. T. SCHWARTZ, *Optimization of very high level languages—value transmission and its corollaries*, Computer Languages, 1 (1975), pp. 161–194.

[T1] R. E. TARJAN, *Depth-first search and linear graph algorithms*, this Journal, 1 (1972), pp. 146–160.

[T2] ———, *Applications of path compression on balanced trees*, J. Assoc. Comput. Mach., 26 (1979), pp. 690–715.

[T3] ———, *A unified approach to path problems*, J. Assoc. Comput. Mach., 22 (1981), to appear.

SIAM J. COMPUT.
Vol. 11, No. 1, February 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1101-0008 $01.00/0

ANALYSIS OF A GENERAL MASS STORAGE SYSTEM*

D. COPPERSMITH,† D. S. PARKER,‡ AND C. K. WONG†

Abstract. A model of a general mass storage system is presented and its performance analyzed. The system is composed of a square two-dimensional grid of storage cells over which a single read/write head moves freely. The head can contain at most some fixed number b of cell contents. Algorithms for realizing an arbitrary permutation of the memory contents are presented for all ranges of b, particularly the important case $b = 1$; in each case the algorithms' performances are explicitly characterized. Open problems, especially regarding the development of good heuristics, are then discussed.

Key words. memory systems, mass storage systems, permutations, L_p-metrics

1. Introduction. With the explosive development of new technologies in the past few years, the design and analysis of memory systems has become more and more complicated. As the shapes of cost-benefit curves have changed and more alternatives have become available in all ranges of memory performance, the task of producing a design for a mass storage system has expanded to require many complex decisions on the nature of the system (whether a homogeneous or hierarchical structure is to be used, which technologies provide the cheapest solution within a given performance range of each part of the system, etc.). Since there are so many possible memory structures, little has been written about the analysis of memory systems in general; this situation has no doubt been exacerbated by the rapidity with which technological advances are being made and the state of flux of the spectrum of design tradeoffs, which can only have intimidated researchers from making general analyses.

This paper is concerned with the analysis of the general mass storage system shown in Fig. 1. The system is composed of a square $n \times n$ grid of memory "cells", on which a single read/write head is permitted to move to and fro. Each cell contains, uniformly, some memory subsystem with a given capacity; and it is assumed that the read/write head, or its controller, has a fixed number b of "registers" which are each large enough to contain a cell's contents. (Hence we are concerned with the range of values $1 \leqq b \leqq n^2$, and the limits $b = 1$ and $b = n^2$ are of particular interest.) In addition to this, the movement of the read/write head is restricted in ways so that the distance between points on the grid (i.e., the amount of time required by the head to move from one point to another) is reflected by the L_1, L_2 or L_∞-metric on the grid. That is, the head can either move:

(a) horizontally or vertically, but not both simultaneously, at uniform speed (in which case distance between two points is given by the L_1-metric);
(b) horizontally or vertically or both at uniform speed (L_∞-metric);
(c) in any direction at uniform speed (L_2-metric).

Note that with $b = 1$ and the L_1-metric the memory system can be made to model an elaborate bubble memory of the type discussed in [1], and with $b = 1$ and the L_∞-metric a tape mass storage system like the IBM 3850 can be modelled. (See, for example, [2], [11], [15].) For most applications a small value of b, like one, seems reasonable.

In addition to the grid and read/write head, we assume the existence of a control unit, and some control memory, connected to the head and to the channel making

* Received by the editors November 6, 1979, and in revised form February 9, 1981.
† IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.
‡ Department of Computer Science, University of California, Los Angeles, California 90024.

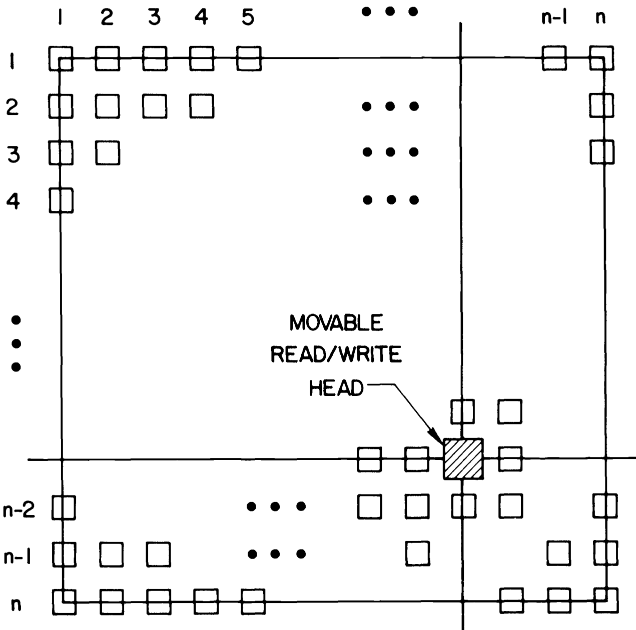


FIG. 1. *General mass storage system on $n \times n$ grid.*

requests on the memory. In the "*online*" mode, requests on the memory are accepted by the control unit and serviced by scheduling a tour for the head (which might be dynamically modified as new requests come in). One problem that might be addressed here is therefore the development of good online scheduling algorithms; if requests are scattered randomly about the grid then possible solutions might resemble the algorithms already developed for disk-like units (see, e.g., [3], [13]). In fact the memory system here can be regarded as a "four-dimensional drum"—a drum with two seek dimensions (each cell containing a track of information).

We will be concerned here with what we call the "*offline*", or stand-alone use of this memory, however. Note that, if requests are *not* randomly distributed on the grid but instead favor given cells over others with a definite probability distribution, then memory performance will be enhanced when all the "popular" cells are located close to one another. With a given access probability distribution, in fact, the best arrangement of the cells for the minimization of average access time is a sort of spiral, with the most popular cells in the center and least popular on the fringes. (This has been discussed in [4] and [2], [5]; the exact nature of the spiral depends on the metric being used, i.e., the restrictions on head movement.) The idea here is that statistics on access frequency might be collected for all the cells while the system is run in online mode; subsequently the memory system could then be switched offline and the cell contents permuted to realize the spiral organization. In this way average access time in online operation can be reduced, even without a sophisticated scheduling algorithm.

The problem we are addressing is therefore: *What is a good way to realize a permutation of the cell contents in the offline mode*? A solution will permit us to operate the storage system efficiently in the online/offline manner just described, and has independent interest as well (it is the two-dimensional generalization of the elevator

scheduling problem solved by Karp [6, pp. 358–361]). Due to some symmetry considerations the solution of the problem is somewhat more difficult than might be expected. Below, after suitable definitions and machinery are set up, the average and worst case costs (i.e., time required to realize a permutation offline, average implying that all permutations are assumed equally likely) are derived for $b=1$, then $b=n^2$ and finally for intermediate values of b. The "cycle algorithm" analyzed for the $b=1$ case is asymptotically optimal, so this case may be viewed as resolved (asymptotically at least). For larger b, unfortunately, currently only good algorithms are provided, but these algorithms are shown to give performance within a small constant factor of optimal.

2. Definitions and general considerations. As just indicated, we are given a square *grid* G, of size $n \times n$, and are concerned with realizing a *permutation* μ selected from P, the set of all permutations of grid points. Thus P is the symmetric group on ^{2} objects. The permutation μ indicates how the memory's contents are to be moved: if $\mu(i)=j$, then the contents of cell i are to be moved to cell j. (Cells in the grid may be indexed in any convenient way.) Thus our problem is to produce an optimal, or near-optimal, schedule of head movements and reads or writes (or exchanges) which realize a given permutation μ. The number of head movements is assumed to be the dominating cost factor, and we will concentrate all of our efforts below on analyzing the movements required by different schedules. Since each head movement takes, as assumed above, a fixed amount of *time* determined by an L_p metric, we will therefore also be studying time requirements of schedules. Below we will use the terms "head movements" and "time" interchangeably.

Because of the symmetries of the square grid, certain permutations may be effectively realized using fewer head movements than might initially seem necessary. Consider the realization of the 180° rotation permutation pictured in Fig. 2. When $b=1$, if we naively go ahead and move the grid contents around as indicated then it turns out that we require time of at least $cn^3+O(n^2)$, where c is the metric-determined constant

$$c=\begin{cases}1 & \text{in } L_1,\\ .76519572\cdots = 1/3\,[\ln(1+\sqrt{2})+\sqrt{2}] & \text{in } L_2,\\ 2/3 & \text{in } L_\infty.\end{cases}$$

(The time required is reflected directly by distance under a metric. Although the L_2 read/write head may seem more powerful than the L_∞ head since it can move in any direction, it really is not, since the L_∞ head moves simultaneously at uniform speed horizontally *and* vertically. Thus to move from (0, 0) to (1, 1) the L_∞ head takes time 1, while the L_2 head takes time $\sqrt{2}$.) In all three cases this is a great deal of time when one considers that one can get away with *zero* time: if the controller simply remembers that the memory is in the $-180°$ rotated "state", it can translate all future requests on the memory with negligible overhead—and the offline rearrangement never need be made.

We generalize the above idea as follows. Suppose that

$$\text{Cost}(\pi)$$

denotes the least possible cost in time required to (naively) realize the permutation $\pi \in P$. Suppose further that a user requests the memory be permuted according to $\mu \in P$. Instead of just taking Cost (μ) time, we employ the following more clever approach.

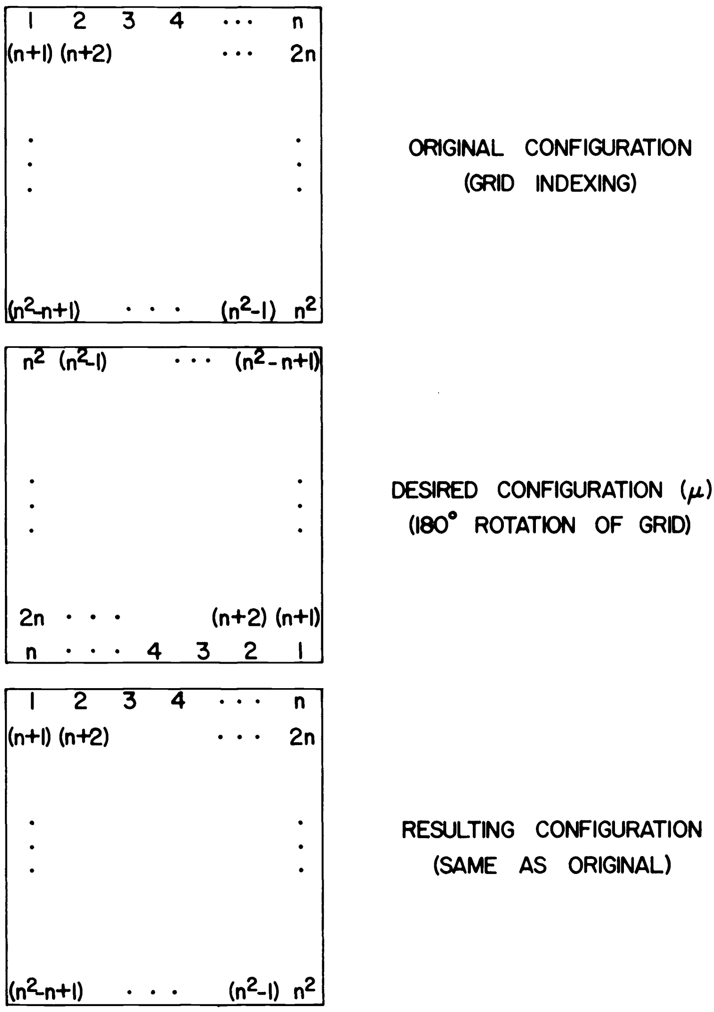


FIG. 2. *Taking advantage of grid symmetry.*

The group S of symmetry operations on the square consists of 8 elements

$$S=\{1, \rho, \rho^2, \rho^3, \tau, \rho\tau, \rho^2\tau, \rho^3\tau\}\subset P,$$

where ρ represents a 90° clockwise rotation and τ a flip about the square's horizontal axis of symmetry, so $\rho^4=1$, $\tau^2=1$, $\rho\tau=\tau\rho^3$, etc. When we say that the grid is in *state* σ we mean the user must apply σ to his conception of the grid's contents to get the actual grid's contents. The grid/state pair (G, σ) is equivalent to the pair $(\sigma'(G), \sigma'\circ\sigma)$ for all $\sigma'\in S$, $\circ$ denoting composition of permutations; the user always thinks of $(\sigma^{-1}(G), 1)$. Note that before Fig. 2 the memory's state σ is 1, since the user's conception of the memory is correct, whereas after Fig. 2 the state becomes $\sigma=\mu^{-1}=-180°$ rotation $=\rho^{-2}$.

We may solve the problem posed in § 1 in three steps. Assume that the grid is in state σ and that the user requests a permutation μ be realized (relative to his perception of the grid's contents), then

(1) Determine which state $\sigma'\in S$ minimizes Cost $(\sigma'\circ\mu\circ\sigma^{-1})$.

(2) Realize $\pi = \sigma' \circ \mu \circ \sigma^{-1}$ directly on the grid.

(3) Mark the grid's state to be σ'.

Note that when the user is kind enough to choose μ as an element of S, this process is trivial, since then one can always find $\sigma' \in S$ such that $\sigma' \circ \mu \circ \sigma^{-1} = 1$, and with any reasonable permuting algorithm, we have

$$\text{Cost}\,(1) = 0.$$

In other words when $\mu \in S$ there is no work to be done except change the grid's state, as was shown in Fig. 2. The point is, however, that even when μ is not in S significant savings in time can result by using this approach of choosing the cheapest grid state. We will quantify this statement in the following section.

Now, the only remaining difficulty is to exhibit an optimal algorithm which produces head movement schedules for realizing a given permutation π (for example, $\pi = \sigma' \circ \mu \circ \sigma^{-1}$) directly/naively on the grid. Unfortunately this is not so simple, if one wants the algorithm to terminate quickly. Here we want the time to *produce* a schedule for realizing π to be significantly less than the time actually needed to *execute* the schedule by the read/write head. We are assuming that algorithms of moderate time complexity (say, between $O(n^2)$ and $O(n^4)$ steps, where again there are n^2 grid elements) will have this property. Much of the rest of this paper concerns itself with finding algorithms of moderate complexity for various values of b which produce near-optimal schedules.

It can be shown that the problem of producing an optimal schedule of head operations is NP-hard, no matter how large b is (i.e., no matter how many registers the head has), by reducing the following path travelling salesman problem (PTSP) [7], [8] to it: given a set of m city coordinates $\{c_i = (x_i, y_i) | i = 1, \cdots, m\}$, where x_i, y_i are integers between 1 and m, find an optimal path through all the cities, i.e., the travelling salesman is not constrained to return to his starting city, he merely is required to visit each city once. Note that, except for the L_2-metric, we can easily show that the problem is in NP [8]. The reduction is as follows: we construct a set of $2m$ points $\{p_i, q_i | i = 1, \cdots, m\}$ on a suitably large grid, namely,

$$p_i = (4x_i, 4y_i), \qquad q_i = (4x_i + 1, 4y_i)$$

so n, the grid size, is actually $4m$.

Note that if we can produce an optimal read/write head tour for the permutation

$$\mu = (p_1 q_1)(p_2 q_2) \cdots (p_m q_m)$$

expressed in cycle notation—so $\mu(p_i) = q_i$, $\mu(q_i) = p_i$ for all i and $\mu(x) = x$ otherwise—then an optimal tour for the original PTSP can be easily extracted. This reduction works no matter how large b is (since the optimal head tour will always just exchange p_i and q_i before moving on to another pair, so $b = 1$ will always suffice) and no matter which L_p metric is used (the reduction of the L_2 PTSP to the planar Hamiltonian path problem given in [7] generalizes for L_1 and L_∞ as well; see also [8].)

It is therefore clear that the best we can probably do here is produce a heuristic polynomial algorithm that finds *near*-optimal tours. Fortunately this is not hard for most permutations, as we shall see. The permutation μ produced in the above reduction is very *sparse* as a permutation, since it leaves most of the grid undisturbed. We shall see that "good" tours for nonsparse permutations can always be found quickly. Interestingly, it is a very rare occurrence for a randomly selected permutation to be

sparse; for example, [14] shows

$$\lim_{n\to\infty} \Pr\,[\text{random } \mu \in P \text{ does not satisfy } \mu(x)=x \text{ for } \textit{any } x \in G] = \frac{1}{e}.$$

3. The case $b=1$. The case $b=1$, that is, the case that the read/write head contains only one register, is probably the most important for practical applications and will correspondingly be given most of the attention of this paper. In this case the contention for the use of the head is extreme, in fact, so extreme that as we will see the cost of realizing a permutation is determined almost wholly by the contention and not very much by the precise form of the scheduling algorithm. This simplification permits a thorough analysis of this case, a task which becomes more difficult as b grows large.

We present first a simple but effective algorithm for generating a read/write head schedule in realizing a permutation $\pi \in P$. (Here π is viewed as an "absolute" permutation—symmetry operations on the grid are not taken into consideration.)

Cycle algorithm. Given $b=1$, permutation π to be realized, head initially in any location x on grid.

Step 1. Determine cycles (orbits) of permutation π.

Step 2. Repeatedly:

(a) schedule the head to permute all the elements in the cycle of its current grid location, in the obvious way (i.e., move x to $\pi(x)$, $\pi(x)$ to $\pi(\pi(x))$, etc., until the head returns to point x).

(b) schedule the head to go to the nearest location whose contents have yet to be moved.

Although this algorithm is extremely simple-minded, it is clear that the only possible waste in time it might make would come from Step 2(b), since all the moves made in Step 2(a) are necessary when $b=1$. Let CACost (π) denote the cost of realizing π with the cycle algorithm;

$$\text{CACost}(\pi) = \sum_{i\in G} d(i, \pi(i)) + (\text{Cost of Step 2(b) for } \pi),$$

where d is the L_p-metric under consideration. The contribution to the total cost from Step 2(a) is directly related to the intrinsic difficulty of the permutation, while that from Step 2(b) is directly dependent on the algorithm. Fortunately, as will be derived below, for most permutations π[1]

$$\sum_{i\in G} d(i, \pi(i)) = \theta(n^3),$$

whereas we can show

PROPOSITION 1. (Cost of Step (2b) for π) $= O(n^2)$.

Proof. This is easy to show; in fact, the coefficient of n^2 will be less than one. The only observation that need be made is that, in moving from cycle to nearest cycle, the head will traverse the entire grid less than once. And traversing the entire $n \times n$ grid takes time $n^2 + O(n)$. □

Thus the cycle algorithm is asymptotically optimal for most permutations, although it could conceivably perform badly for "sparse" permutations. As an example of how

[1] $\theta(n^3)$ means exactly order n^3.

bad it can get, consider the permutation π of 4 cycles illustrated in Fig. 3, given that the head starts at point a and the L_1-metric is being used. The cycle algorithm processes π as (abc); (de); (fg); (hi) requiring $8n+O(1)$ head motions. (From a to b to c and back to a, we need $4n$; from a to d, we need n; from d to f, another n; and finally from f to h, we need $2n$.) However the optimal method is to process the small cycles while working on the large one, i.e.,

$$(a\,b\,(hi)\,(de)\,(fg)\,c),$$

which takes $4n+O(1)$ head motions. It therefore may be worthwhile to consider refinements of the cycle algorithm, particularly if head movements are much slower than the computation speed of the control unit (which is devising the head's schedule), as we assume here. One good alternative is the following:

Minimal spanning tree/Euler circuit/cycle algorithm.

Step 1. Determine cycles of permutation π.

Step 2. Derive distances between cycles of π (i.e., for each pair of cycles C_1, C_2 in π derive $\min_{x\in C_1 y\in C_2} d(x, y)$ and record this in a matrix as the distance between C_1 and C_2).

Step 3. Form a minimal spanning tree for the cycles. This tree corresponds very closely to a "Euler circuit" for π.

Step 4. Traverse the minimal spanning tree (Euler circuit) in the obvious way. Effectively this changes π to look like one enormous cycle, but a cycle which touches itself.

Note that this algorithm is fairly effective in reducing obvious waste: for the permutation in Fig. 3 it produces the schedule

$$(a\,b\,(hi)\,c\,(fg))\,(de)$$

with a cost of $5n+O(1)$ steps if cycles are joined in one way, and

$$(a\,(hi)\,b\,(fg)\,c\,(de))$$

with the same cost if they are joined in another (note that this latter schedule is better under the L_2 and L_∞ metrics than the former). However, Step 2 can be extremely expensive, requiring as much as $O(n^4)$ time or $O(n^2)$ words of memory depending on π's cycle structure. If the cost of Step 2 is not felt to be exorbitant, however, the user may consider enlarging it to capitalize on the fact that the distance between two cycles is often less than the minimum distance between their elements; this was shown in Fig. 3 with the cycles (abc) and (de).

For the rest of this section we will assume that the read/write head control unit uses something like the cycle algorithm so that the dominant term in the cost of realizing a permutation depends solely on the permutation. In fact we *define*

$$\text{Cost}\,(\pi)=\sum_{i\in G} d(i, \pi(i)),$$

d being again the metric under consideration, since then CACost asymptotically approaches Cost as n grows large, and since this simplification permits us to ignore algorithm structure in the following analyses of costs.

Ignoring for the moment the symmetry operations S mentioned in § 2, we ask the *average and worst-case values of* Cost (π), where average means that all permutations π are considered equally likely. Essentially then we are asking how much time we would require to realize permutations offline if we did not worry about "grid

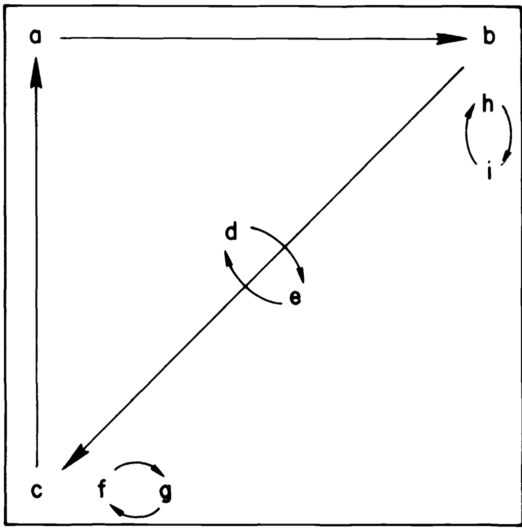


FIG. 3. *Permutation for which cycle algorithm is poor.*

states"; this will serve as a basis for comparison when the analysis *with* grid states is made below.

THEOREM 1.

$$\operatorname*{Average}_{\pi\in P}[\text{Cost}(\pi)] = \begin{cases} 2/3n^3 + O(n), & L_1 \textit{ metric}, \\ (.5214)n^3 + O(n^2), & L_2 \textit{ metric}, \\ 7/15n^3 + O(n'), & L_\infty \textit{ metric}. \end{cases}$$

Proof. For all three metrics we have

$$\begin{aligned} \operatorname*{Average}_{\pi\in P}[\text{Cost}(\pi)] &= \frac{1}{(n^2)!} \sum_{\pi\in P} \sum_{i\in G} d(i, \pi(i)) \\ &= \frac{1}{(n^2)!} \sum_{i\in G} \sum_{\pi\in P} d(i, \pi(i)) \\ &= \frac{1}{(n^2)!} \sum_{i\in G} \sum_{j\in G} N(i, j)\, d(i, j), \end{aligned}$$

where

$$\begin{aligned} N(i, j) &= [\text{number of permutations } \pi \in P \text{ such that } \pi(i) = j] \\ &= (n^2 - 1)!, \end{aligned}$$

so

$$\operatorname*{Average}_{\pi}[\text{Cost}(\pi)] = \frac{1}{n^2} \sum_{i\in G} \sum_{j\in G} d(i, j).$$

This sum must now be analyzed independently for each of the three metrics. In all three cases we represent grid points with the matrix-like indexing $i \Leftrightarrow (i_1, i_2)$, where i_1 and i_2 have values between 1 and n.

In the L_1 case we have

$$\begin{aligned}\frac{1}{n^2}\sum_{i\in G}\sum_{j\in G} d(i,j) &= \frac{1}{n^2}\sum_{i_1=1}^{n}\sum_{i_2=1}^{n}\sum_{j_1=1}^{n}\sum_{j_2=1}^{n}(|i_1-j_1|+|i_2-j_2|)\\ &= \frac{1}{n^2}\left(\left(\sum_{i_1=1}^{n}\sum_{j_1=1}^{n} n^2|i_1-j_1|\right)+\left(\sum_{i_2=1}^{n}\sum_{j_2=1}^{n} n^2|i_2-j_2|\right)\right)\\ &= 2\sum_{k,l=1}^{n}|k-l|.\end{aligned}$$

Let Δ denote the *multiset* $\{|k-l|, k=1,\cdots,n, l=1,\cdots,n\}$. Then it is easy to verify that Δ contains (n) zeros, $2(n-1)$ ones, $2(n-2)$ twos, $\cdots$, and $2(1)$ $n-1$'s. Concisely, if δ is a positive integer less than n, there are $2(n-\delta)$ copies of δ in Δ. Thus, continuing,

$$\begin{aligned}&= 2\sum_{\delta\in\Delta}\delta\\ &= 2\sum_{\delta=1}^{n-1}2(n-\delta)\delta\\ &= 2/3(n^3-n) = 2/3n^3+O(n) \quad \text{as stated.}\end{aligned}$$

In the L_∞ case we find

$$\begin{aligned}\frac{1}{n^2}\sum_{i\in G}\sum_{j\in G} d(i,j) &= \frac{1}{n^2}\sum_{i_1,i_2=1}^{n}\sum_{j_1,j_2=1}\max(|i_1-j_1|,|i_2-j_2|)\\ &= \frac{1}{n^2}\sum_{\delta_1\in\Delta}\sum_{\delta_2\in\Delta}\max(\delta_1,\delta_2).\end{aligned}$$

By carefully manipulating this sum we can show this is

$$\begin{aligned}&= \frac{1}{15n^2}(7n^5-5n^3-2n)\\ &= \frac{7}{15}n^3+O(n) \quad \text{as stated.}\end{aligned}$$

The constant $\frac{7}{15}$ has been independently verified in [10], where it was shown the average L_∞ distance in a square with edge 2 is $\frac{14}{15}$ (implying the average distance in a square of edge 1 is $\frac{7}{15}$).

The L_2 derivation is, not surprisingly, more complicated. By appealing to the Euler–Maclaurin summation theorem we have

$$\begin{aligned}\frac{1}{n^2}\sum_{i\in G}\sum_{j\in G} d(i,j) &= \frac{1}{n^2}\sum_{i_1,i_2=1}^{n}\sum_{j_1,j_2=1}^{n}\sqrt{|i_1-j_1|^2+|i_2-j_2|^2}\\ &= \frac{1}{n^2}\sum_{\delta_1\in\Delta}\sum_{\delta_2\in\Delta}\sqrt{\delta_1^2+\delta_2^2}\\ &= \frac{1}{n^2}\sum_{\delta_1=1}^{n}\sum_{\delta_2=1}^{n}2(n-\delta_1)2(n-\delta_2)\sqrt{\delta_1^2+\delta_2^2}+O(n)\\ &= \frac{4}{n^2}\int_0^n\int_0^n(n-x)(n-y)\sqrt{x^2+y^2}\,dx\,dy+O(n^2).\end{aligned}$$

The double integral can be evaluated as $(4/n^2)\int_0^n (n-x)F(x,n)\,dx$, where

$$F(x,n)=\int_0^n (n-y)\sqrt{x^2+y^2}\,dy$$

$$=\frac{n}{2}x^2 \operatorname{arcsinh}\left(\frac{n}{x}\right)+\frac{n^2}{2}\sqrt{n^2+x^2}-\frac{1}{3}(n^2+x^2)^{3/2}+\frac{x^3}{3}$$

(fudging at the boundary $x=0$ is harmless), and by taking the asymptotic behavior of this integral we find that, since $\operatorname{arcsinh}(z)=\ln(z+\sqrt{z^2+1})$, we have in closed form

$$\frac{1}{n^2}\sum_{i,j\in G} d(i,j)=\left[\frac{1}{3}\ln(1+\sqrt{2})+\frac{2+\sqrt{2}}{15}\right]n^3+O(n^2)$$

$$=[.52140\,54331\,64720\,67833\cdots]n^3+O(n^2)$$

as claimed. This asymptotic form of the average cost agrees well with the exact values or moderate n. For several values the n^3 coefficient

$$\frac{1}{n^3}\left[\frac{4}{n^2}\sum_{\delta_1,\delta_2=1}^{n}(n-\delta_1)(n-\delta_2)\sqrt{\delta_1^2+\delta_2^2}\right]$$

is tabulated in Table 1, and a polynomial regression on the table shows

$$\text{Average } L_2 \text{ cost}=(.52140)n^3-(.66319)n^2-(2.98662)n+(476.07),$$

with small residuals and enormous F-statistics. □

TABLE 1
Asymptotic behavior of average L_2 cost.

n	*Average L_2 cost/n^3*
100	0.51471 44257
200	0.51806 59602
300	0.51918 04742
400	0.52007 11123
500	0.52073 85192
1000	0.52096 08787
1500	0.52107 20379

Frequently the evaluation of average complexity is of limited use, since the *standard deviation* can be large, suggesting that behavior much less and much greater than the average will occur reasonably often. It is interesting to note that this is not the case here.

THEOREM 2. Standard Deviation [Cost (π)] $= O(n^2)$ *in all* 3 *metrics.*

Proof. Recall that standard deviation $=\sqrt{\text{Variance}}$, and if we let $A=$ Average_π [Cost (π)] then

$$\operatorname*{Variance}_{\pi}[\text{Cost}(\pi)]=\frac{1}{(n^2)!}\sum_{\pi}\left(\sum_{i\in G} d(i,\pi(i))\right)^2-A^2$$

$$=\left(\frac{1}{(n^2)!}\sum_{\pi}\sum_{i\in G} d(i,\pi(i))^2+\frac{1}{(n^2)!}\sum_{\pi}\sum_{i\neq j} d(i,\pi(i))\,d(j,\pi(j))\right)-A^2$$

$$=\left(\frac{1}{n^2}\sum_{i,j} d(i,j)^2\right)+\left(\frac{1}{n^2(n^2-1)}\sum_{\substack{i\neq j\\ k\neq l}} d(i,k)\,d(j,l)\right)-A^2.$$

It can be shown that the second term in this expression is

$$\frac{1}{n^2(n^2-1)}\left[n^4A^2-2\sum_k\Big(\sum_i d(i,k)\Big)^2+\sum_{i,k} d(i,k)^2\right],$$

and since we can put

$$A^2=c_1n^6+o(n^6),$$

$$\sum_{i,j} d(i,j)^2=c_2n^6+o(n^6),$$

$$\sum_k\Big(\sum_i d(i,k)\Big)^2=c_3n^8+o(n^8)$$

in all 3 metrics, we find

$$\begin{aligned}\underset{\pi}{\text{Variance}}\,[\text{Cost}\,(\pi)]&=\frac{1}{n^2}(c_2n^6+o(n^6))\\&\quad+\frac{1}{n^2(n^2-1)}(n^4A^2-2(c_3n^8+o(n^8))+(c_2n^6+o(n^6)))-A^2\\&=n^4(c_2-2c_3)+A^2\Big(\frac{1}{1-1/n^2}-1\Big)+o(n^4)\\&=n^4(c_2-2c_3)+(c_1n^6+o(n^6))(1/n^2+1/n^4+\cdots)+o(n^4)\\&=n^4(c_1+c_2-2c_3)+o(n^4).\end{aligned}$$

Thus, by taking the square root, we find that the standard deviation of the average cost is $O(n^2)$ in all three metrics. Actually evaluating the leading coefficient $\sqrt{c_1+c_2-2c_3}$ is tedious, but, for example, in the L_1 case (in which case the coefficient is larger than in L_2 or L_∞ since L_1 costs vary more than do the others) we can determine that

$$\begin{aligned}\underset{\pi\in P}{\text{Standard Deviation}}\,[\text{Cost}\,(\pi)]&=\sqrt{\tfrac{4}{45}n^4+O(n^2)}\\&=\Big(\frac{2\sqrt{5}}{15}\Big)n^2+O(1)\\&=(.29814)n^2+O(1).\end{aligned}$$

□

It is thus apparent that the average cost figures given by Theorem 1 are very good predictors of the running time of a read/write head schedule for a random permutation, especially as n gets large. We can also get precise bounds on the worst-case running time for any permutation.

THEOREM 3.

$$\underset{\pi\in P}{\text{Worst Case}}\,[\text{Cost}\,(\pi)]=\begin{cases}n^3-n, & n \text{ odd}, \quad L_1 \text{ metric},\\ n^3, & n \text{ even}, \quad L_1 \text{ metric},\\ \tfrac{1}{3}[\ln(1+\sqrt{2})+\sqrt{2}]n^3+O(n^2), & L_2 \text{ metric},\\ 2/3(n^3-n), & L_\infty \text{ metric}.\end{cases}$$

Proof. Letting π be any permutation and p be any point on the grid G, we apply the triangle inequality for the metric d to get the upper bound

$$\text{Cost}(\pi) = \sum_{i \in G} d(i, \pi(i))$$

$$\leqq \sum_{i \in G} (d(i,p) + d(p, \pi(i))) = 2 \sum_{i \in G} d(i,p).$$

This right-hand expression is maximized when p is the center of the grid (when n is even p is not actually a cell location). This gives us the upper bounds stated in the theorem, because we can actually find a permutation which attains this upper bound: Note that the 180°-*rotation permutation* π shown in Fig. 2 is a worst-case permutation since it satisfies

$$d(i, \pi(i)) = d(i,p) + d(p, \pi(i))$$

for all $i \in G$. Any permutation satisfying this equality for all i must necessarily be a worst-case permutation; also a simple symmetry argument shows that the only way a permutation π can satisfy this equality is for p to be the grid center. The values stated in the theorem reflect the cost of this permutation in each of the metrics. □

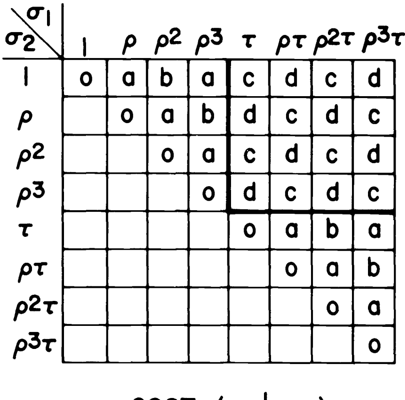
Up to this point we have ignored the possible savings that are made by taking advantage of the symmetry of the grid, as discussed in § 2. For the rest of this section we examine how the symmetry operations affect the average and worst-case cost statistics derived above.

We begin first by studying the relative costs of the symmetry permutations themselves. All the necessary information is listed in Fig. 4 and Table 2, derived in the same manner as in Theorems 1 and 2, using the matrix notation

$$\rho((i_1, i_2)) = (i_2, n+1-i_1),$$

$$\tau((i_1, i_2)) = (n+1-i_1, i_2),$$

and so forth for the other members of the group S. The only expressions that present



$\sigma_2 \backslash \sigma_1$	1	ρ	ρ^2	ρ^3	τ	$\rho\tau$	$\rho^2\tau$	$\rho^3\tau$
1	o	a	b	a	c	d	c	d
ρ		o	a	b	d	c	d	c
ρ^2			o	a	c	d	c	d
ρ^3				o	d	c	d	c
τ					o	a	b	a
$\rho\tau$						o	a	b
$\rho^2\tau$							o	a
$\rho^3\tau$								o

COST $(\sigma_2^{-1} \circ \sigma_1)$

MATRIX IS SYMMETRIC

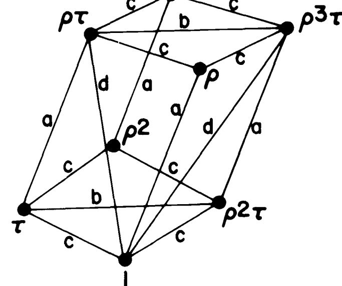


FIG. 4. *Distance matrix for grid symmetry operations.*

TABLE 2
Values of distances in Fig. 4

	a	b	c	d
$d=L_1$ distance	$\frac{2}{3}(n^3-n)$	(n^3-n), n odd (n^3), n even	$\frac{1}{2}(n^3-n)$, n odd $\frac{1}{2}(n^3)$, n even	$\frac{2}{3}(n^3-n)$
$d=L_2$ distance	$(.54107)n^3$ $+O(n^2)$	$\frac{1}{3}[\ln(1+\sqrt{2})+\sqrt{2}]n^3$ $+O(n^2)$	$\frac{1}{2}(n^3-n)$, n odd $\frac{1}{2}(n^3)$, n even	$(\sqrt{2})/3(n^3-n)$
$d=L_\infty$ distance	$\frac{1}{2}(n^3-n)$, n odd $\frac{1}{2}(n^3)$, n even	$\frac{2}{3}(n^3-n)$	$\frac{1}{2}(n^3-n)$, n odd $\frac{1}{2}(n^3)$, n even	$\frac{1}{3}(n^3-n)$

any difficulty are the values of a and b under the L_2-metric, for which we have

$$a=\sum_{i\in G} d(i,\rho(i))=\sum_{k,l=1}^{n}\sqrt{|k-l|^2+|n+1-(k+l)|^2},$$

$$b=\sum_{i\in G} d(i,\rho^2(i))=\sum_{k,l=1}^{n}\sqrt{|n+1-2k|^2+|n+1-2l|^2};$$

in the latter case, the techniques of Theorem 1 can be applied directly, but a closed form has not yet been derived for a.

We now examine the effect on cost of the symmetry group S. It turns out that symmetry operations do not significantly reduce the *average* running time, but they reduce the *worst-case* running time significantly. We formalize this as follows: given a permutation $\pi\in P$, define the *symmetrized cost* SCost by

$$\text{SCost}(\pi)\overset{\text{def}}{=}\min_{\sigma\in S}\text{Cost}(\sigma\circ\pi).$$

It follows obviously that $\text{SCost}(\pi)\leqq\text{Cost}(\pi)$ for all permutations π. Nevertheless we have the surprising result of Theorem 4.

THEOREM 4.

$$\underset{\pi}{\text{Average}}[\text{SCost}(\pi)]=\underset{\pi}{\text{Average}}[\text{Cost}(\pi)]-O(n^2).$$

Proof. Let $f(x)=[\text{Number of permutations } \pi \text{ such that } \text{Cost}(\pi)=x]/(n^2)!$ for any integer x be the "probability density" for Cost (π), with corresponding distribution $F(y)=\sum_{x\leqq y}f(x)$. The point is that f looks very much like a "spike". If A denotes the average cost as in Theorem 2, then

$$A=\sum_{x=0}^{\infty}xf(x),$$

and the variance B^2 is given by

$$B^2=\sum_{x=0}^{\infty}(x-A)^2f(x).$$

Using a Chebyshev inequality process we have for each positive integer $C<A$

$$B^2 \geqq \sum_{x=0}^{C} (x-A)^2 f(x)$$
$$\geqq (A-C)^2 \sum_{x=0}^{C} f(x) = (A-C)^2 F(C).$$

Thus $(A-C) \leqq B/\sqrt{F(C)}$ for $0<C<A$. Now note that, for every permutation π, SCost (π) is the minimum Cost of the eight translates $\sigma \circ \pi$, with $\sigma \in S$, and of course all of these translates are again permutations. Thus the distribution function F_S, giving this distribution of costs with S, is at best the left-hand $\frac{1}{8}$ of F (renormalized by a factor of 8). That is, if $\lambda = \max\{x \mid F(x) \leqq \frac{1}{8}\}$ then $F_S(x) \leqq G(x)$, where

$$G(x) = \begin{cases} 8F(x), & x<\lambda, \\ 1, & x \geqq \lambda. \end{cases}$$

Correspondingly, if

$$g(x) = \begin{cases} 8f(x), & x<\lambda, \\ 1-8F(x), & x=\lambda, \\ 0, & x>\lambda, \end{cases}$$

then we have the bound

$$A' = \sum_{x>0} xg(x) \leqq \underset{\pi}{\text{Average}}\,[\text{SCost}\,(\pi)] < \sum_{x>0} xf(x) = A.$$

Now it is clear that $A' = \sum_{x>0} xg(x) = F^{-1}(\frac{1}{16})$, since we are finding the average (or midpoint) of the left-hand $\frac{1}{8}$ of F. Then applying the Chebyshev bound derived above for $C=A'$ we find

$$A - A' \leqq \frac{B}{\sqrt{(1/16)}} = 4B,$$

so A' is within 4 standard deviations of A; however from Theorem 2 we know that B is only of order $O(n^2)$. Thus

$$A - O(n^2) \leqq \text{Average}\,[\text{SCost}\,(\pi)] < A,$$

which is what was to be proved. □

Theorem 4 suggests that using the symmetry operations S will not significantly reduce execution time, on the average. However, we now show that using S does reduce considerably the worst-case time. To do this, we establish first the following lemma.

LEMMA. *Given $\pi \in P$, we can construct $\pi' \in P$ such that*

$$\text{SCost}\,(\pi') = \underset{\sigma \in S}{\text{Average}}\,[\text{Cost}\,(\sigma \circ \pi)] + O(n^2).$$

Proof. This statement can be proved, but instead we show the simpler result that we can construct a permutation π' of the $4n \times 4n$ grid having the corresponding property

$$\frac{1}{(4)^3}\,\text{SCost}\,(\pi') = \underset{\sigma \in S}{\text{Average}}\,[\text{Cost}\,(\sigma \circ \pi)] + O(n^2).$$

Since we are not really concerned about the size of n, only the form of the worst case permutation, this shift of grid size is not important.

We construct π' from π by "interleaving" 2 copies of each of the 8 translates $\sigma\circ\pi$ of π. We do this by dividing the $4n\times 4n$ grid G' into 16 $n\times n$ subgrids G_{pq}, with p and q being integers between 1 and 4, defined by

$$G_{pq}=\{(4k_1+p, 4k_2+q)|0\leqq k_1, k_2\leqq n-1\}.$$

π' is then constructed as being a permutation mapping each G_{pq} into G_{pq} for all p, q. For each point $(i_1, i_2)=(4k_1+p,\ 4k_2+q)$ in G', if $\sigma_{pq}(1\leqq p, q\leqq 4)$ constitutes an enumeration of S such that each element of S appears twice, and if $\sigma_{pq}\circ\pi(k_1, k_2)=(l_1, l_2)$, then we define

$$\pi'((i_1, i_2))=(4l_1+p, 4l_2+q).$$

In other words, π' is $\sigma_{pq}\circ\pi$ when restricted to G_{pq}. From this it follows immediately that

$$\begin{aligned}\text{Cost}(\pi')&=\sum_{p,q} 4\ \text{Cost}(\sigma_{pq}\circ\pi)\\ &=2\sum_{\sigma\in S} 4\ \text{Cost}(\sigma\circ\pi)\\ &=64 \operatorname*{Average}_{\sigma\in S}[\text{Cost}(\sigma\circ\pi)],\end{aligned}$$

where the factor 4 comes from the fact that the grids G_{pq} have a distance of 4 between cells. Moreover, since $\sigma\circ S=S$ for all $\sigma\in S$, we know

$$\text{Cost}(\sigma\circ\pi')=\text{Cost}(\pi')+O(n^2)\quad \text{for all } \sigma\in S$$

(where it is understood now that by σ we mean the symmetry operations on the $4n\times 4n$ grid G'), the $O(n^2)$ term resulting from the minor rotations of each of the 4×4 chunks $\{(4k_1+p, 4k_2+q)|1\leqq p, q\leqq 4\}$ of G' under σ. From this we get

$$\text{SCost}(\pi')=\text{Cost}(\pi')-O(n^2)$$

and the lemma follows. □

COROLLARY.

$$\operatorname*{Worst\ Case}_{\pi\in P}[\text{SCost}(\pi)]=\operatorname*{Worst\ Case}_{\pi\in P}[\operatorname*{Average}_{\sigma\in S}[\text{Cost}(\sigma\circ\pi)]]+O(n^2).$$

Proof. Suppose π is the worst case, i.e., π maximizes SCost (π). Then by the above lemma there exists π' such that

$$\begin{aligned}\text{SCost}(\pi')&=\operatorname*{Average}_{\sigma\in S}[\text{Cost}(\sigma\circ\pi)]+O(n^2)\\ &\geqq \text{SCost}(\pi)+O(n^2).\end{aligned}$$

Thus π' is essentially a worst-case too. However,

$$\text{SCost}(\pi')=\operatorname*{Average}_{\sigma\in S}[\text{Cost}(\sigma\circ\pi')]+O(n^2);$$

so the corollary is proved. □

THEOREM 5.

$$\operatorname*{Worst\ Case}_{\pi\in P}[\text{SCost}(\pi)]=\alpha n^3+o(n^3),$$

where $\alpha = 0.72096$ *for the* L_1*-metric*, $\alpha = .4387826$ *for the* L_∞*-metric and* $0.53956 \leqq \alpha \leqq 0.6202$ *for* L_2*-metric.*

Proof. By the above corollary it suffices to consider

$$A = \underset{\pi \in P}{\text{Worst Case}} \left[\underset{\sigma \in S}{\text{Average}} \left[\text{Cost} \left(\sigma \circ \pi \right) \right] \right].$$

We study the L_1-metric first.

For convenience of later discussion, we identify the $n \times n$ grid G with the square with vertices (1, 1), (1, −1), (−1, −1), (−1, 1). Since

$$\underset{\sigma \in S}{\text{Average}} \left[\text{Cost} \left(\sigma \circ \pi \right) \right] = \frac{1}{8} \sum_{x \in G} \sum_{\sigma \in S} d(x, \sigma \circ \pi(x)),$$

clearly, we can assume without loss of generality that x and $\pi(x)$ are in the same quadrant, i.e., π maps a quadrant into itself. Let $x = (a, b)$ be a point in the first quadrant G_1 and let $\pi(x) = (u, v)$ be its image. Then, $a, b, u, v \geqq 0$;

$$\begin{aligned} \sum_{\sigma \in S} d(x, \sigma \circ \pi(x)) &= 2(|u-a| + |\nu - b| + |u - b| \\ &\qquad + |\nu - a| + (u+a) + (\nu + b) + (u+b) + (\nu + a)) \\ &= 2(d(x, \pi(x)) + d(x, \eta \circ \pi(x))) + 4(u + \nu + a + b), \end{aligned}$$

where $\eta(u, \nu) = (\nu, u)$. Thus

$$\sum_{x \in G_1} \sum_{\sigma \in S} d(x, \sigma \circ \pi(x)) = 4 \sum_{x \in G_1} (u + \nu + a + b) + 2 \sum_{x \in G_1} d(x, \pi(x)) + d(x, \eta\pi(x)).$$

Note that $\sum_{x \in G_1} (u + \nu + a + b)$ is the same for all π. Hence if we can construct a permutation π from G_1 onto G_1 such that $\sum_{x \in G_1} d(x, \pi(x)) + d(x, \eta \circ \pi(x))$ is maximized, then we can extend it to G by reflection to obtain the worst-case permutation.

Since we are interested in the coefficient of the n^3 term in A only, we need only consider continuous transformations from G_1 onto G_1 with Jacobians equal to ± 1. (An area-preserving continuous map is roughly the limit of one-one permutations.)

Divide G_1 into 10 regions as in Fig. 5, where $e = (\frac{1}{2}, \frac{1}{2})$, $t = ((3 - \sqrt{6})/2, 1 - (3 - \sqrt{6})/2)$ such that area B_1 = area B_3, area B_2 = area B_4, area A = area C, area D_1 = area D_3, area D_2 = area D_4.

Next we define a real-valued function f on G_1 as follows, where d is again the L_1 metric:

$$f(x) = \begin{cases} 2\, d(x, e) & \text{for } x \in A, C, \\ d(x, e) + d(x, t) & \text{for } x \in B_2, B_3, B_4, \\ d(x, e) + d(x, t') & \text{for } x \in D_2, D_3, D_4, \\ d(e, t) & \text{for } x \in B_1, D_1. \end{cases}$$

By direct verification, we have for the L_1-metric

(i) $d(x, y) + d(x, \eta(y)) = f(x) + f(y)$

for $x \in A$, $y \in C$; $x \in B_1$, $y \in B_3$; $x \in B_2$, $y \in B_4$; $x \in D_1$, $y \in D_3$; $x \in D_2$, $y \in D_4$.

(ii) $d(x, y) + d(x, \eta(y)) \leqq f(x) + f(y)$ for $x, y \in G_1$.

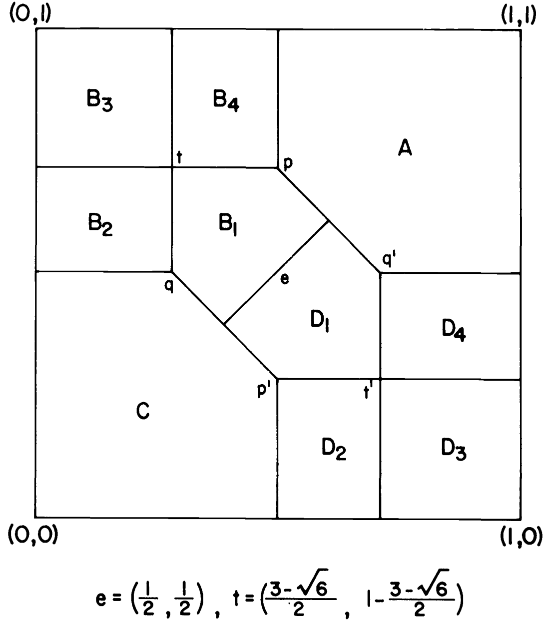


FIG. 5. *Worst-case permutation for the L_1-metric.*

Inequality (ii) implies that, for any transformation π,

$$d(x, \pi(x))+d(x, \eta\pi(x))\leqq f(x)+f(\pi(x)).$$

On the other hand, for any transformation π_0 such that it maps A onto C, B_1 onto B_3, B_2 onto B_4, D_1 onto D_3 and D_2 onto D_4, by (i)

$$d(x, \pi_0(x))+d(x, \eta\pi_0(x))=f(x)+f(\pi_0(x)).$$

It follows that π_0 maximizes $\sum_{x\in G_1} d(x, \pi(x))+d(x, \eta\pi(x))$.

Direct calculation of $\text{Average}_{\sigma\in S}[\text{Cost}(\sigma\circ\pi_0)]$ yields the result stated in the theorem.

To achieve the worst case for the L_∞-metric, we note that the mapping $g(x, y)=((y+x)/2, (y-x)/2)$ is an isometry between the plane with L_∞-metric and that with the L_1 metric. Thus, instead of working with G_1 (with the L_∞-metric), it suffices to consider the triangle T_1 with vertices $(0,0)$, $(\sqrt{2}, 0)$, $(0, \sqrt{2})$ (with the L_1-metric). In other words, we have to construct a transformation form T_1 onto T_1 such that $\sum_{x\in T_1} d(x, \pi(x))+d(x, \eta\pi(x))$ is maximized, where d corresponds to the L_1-metric.

For convenience, we normalize T_1 to a triangle with vertices $(0,0)$, $(1,0)$, $(0,1)$. As before, we divide T_1 into 10 regions, such that area $B_1=$ area B_3, area $B_2=$ area B_4, area $A=$ area C, area $D_1=$ area D_3, area $D_2=$ area D_4. (See Fig. 6.) To do this, X, Y, Z must satisfy the equations:

$$2X^2-2XY-2XZ-Y^2-3Z^2+4Z=0,$$
$$X^2-2X+2XY-2XZ-Z^2+2Z=0,$$
$$2X^2-4X+2XY-2XZ-Y^2+Z^2+1=0.$$

which means $X=0.27677$, $Y=0.531439$ and $Z=0.139882$. Again, any transformation mapping A onto C, B_1 onto B_3, B_2 onto B_4, D_1 onto D_3 and D_2 onto D_4 will

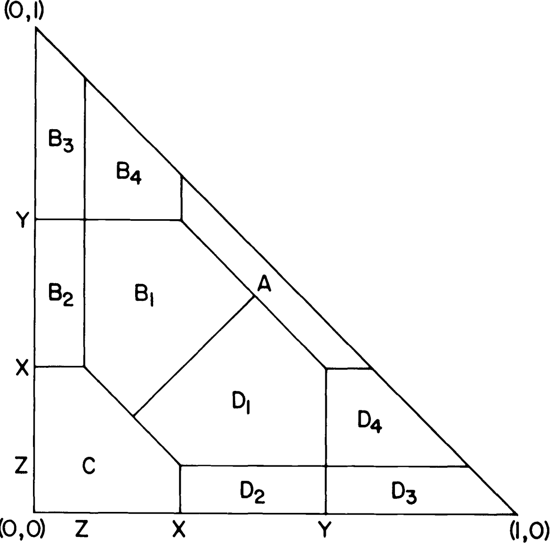


FIG. 6. *Worst-case permutation for the L_∞-metric.*

maximize the desired sum and will be our solution. We compute, as before, that the mean distance is

$$\tfrac{2}{3}-X+3X^2-\tfrac{4}{3}X^3+\tfrac{1}{6}Y^3+Z^2-\tfrac{1}{2}Z^3-X^2Y+3XY^2$$
$$-2XY+2X^2Z-5XZ^2-\tfrac{9}{2}Y^2Z+\tfrac{9}{2}YZ^2+2XYZ=0.43878265158.$$

The case of the L_2-metric seems to be much more difficult, and we are unable to obtain the coefficient of n^3 exactly. Only some simple bounds are presented here.

Consider the triangle T with vertices (0, 0), (1, 0), (1, 1). Let w be an arbitrary but fixed point in T. Then, assuming that π maps all triangles in the grid onto themselves,

$$\begin{aligned}\frac{1}{8}\sum_{\sigma\in S}\sum_{x\in T} d(x,\sigma\circ\pi(x)) &\leqq \frac{1}{8}\sum_{\sigma}\sum_{x}[d(x,\sigma(w))+d(\sigma(w),\sigma\circ\pi(x))]\\ &=\frac{1}{8}\sum_{\sigma}\sum_{x} d(x,\sigma(w))+\frac{1}{8}\sum_{\sigma}\sum_{x} d(w,\pi(x))\\ &=\frac{1}{8}\sum_{\sigma}\sum_{x} d(x,\sigma(w))+\sum_{x} d(w,x).\end{aligned}$$

Let $T'=\sigma'(T)$ denote the translate of T by $\sigma'\in S$ and $w'=\sigma'(w)$. Then we have exactly the same inequality:

$$\begin{aligned}\frac{1}{8}\sum_{\sigma\in S}\sum_{X\in T} d(x,\sigma\circ\pi(x)) &\leqq \frac{1}{8}\sum_{\sigma}\sum_{x\in T'}[d(x,\sigma(w'))+d(\sigma(w'),\sigma\circ\pi(x))]\\ &=\frac{1}{8}\sum_{\sigma}\sum_{x\in T} d(x,\sigma(w))+\sum_{x\in T} d(w,x).\end{aligned}$$

Thus,

$$\frac{1}{8}\sum_{\sigma\in S}\sum_{x\in G} d(x,\sigma\circ\pi(x))\leqq \min_{w\in T}\left\{\sum_{\sigma\in S}\sum_{x\in T} d(x,\sigma(w))+8\sum_{x\in T} d(w,x)\right\}.$$

For the L_1-metric, the minimization point turns out to be $w_0 = (\frac{1}{2}, (5-\sqrt{17})/4)$. While we are unable to determine the minimization point for either the L_∞ or L_2-metric, we can use w_0 to obtain an upper bound:

$$\frac{1}{8}\sum_{\sigma\in S}\sum_{x\in G} d(x, \sigma\circ\pi(x)) \leqq 0.6202n^3.$$

To obtain a lower bound, let G_1, G_2, G_3, G_4 be the four quadrants of G and define a permutation π_0 mapping G_i onto itself, for $i = 1, 2, 3, 4$:

$$\pi_0 = \begin{cases} \rho^3\tau & \text{in } G_1, \\ \rho\tau & \text{in } G_2, \\ \rho^3\tau & \text{in } G_3, \\ \rho\tau & \text{in } G_4, \end{cases}$$

where by ρ, τ we mean the symmetry operations on the appropriate quadrants. Direct computation shows that

$$\text{SCost}(\pi_0) = 0.53956n^3.$$

Therefore α is between 0.53956 and 0.6202. □

To summarize, when a user gives us a permutation μ while the memory is in state σ, we form $\mu\circ\sigma^{-1}$, determine SCost $(\mu\circ\sigma^{-1}) = \text{Cost}(\sigma'\circ\mu\circ\sigma^{-1})$ and realize the permutation $\sigma'\circ\mu\circ\sigma^{-1}$ using "cycle algorithm" or some other similar algorithm. Theorems 4 and 1 give the average cost, and Theorem 5 gives the worst-case cost.

The case $b = n^2$. In the case $b = n^2$, the read/write head contains enough memory to save the entire contents of the mass storage device. The possibility could arise, for example, if some large random-access devicė were available during the offline permuting period. The value $b = n^2$ is perhaps exorbitant, but it serves a useful limiting value with which performance for smaller values of b may be compared.

Below we produce a straightforward algorithm for generating any permutation $\pi \in P$, always taking time $2n^2$. Although this algorithm is very suboptimal for some permutations, it is not bad in the general case. As before, π is viewed here as an "absolute" permutation, and symmetry operations of the grid are ignored. In fact, no benefit whatsoever is gained by considering symmetry translates of a permutation if the final permutation is to be realized with the following algorithm.

Two-pass algorithm.

Given $b = n^2$, permutation π to be realized, head initially in any location x on the grid.

Step 1. Read in entire contents of grid in a single pass across all n^2 cells.

Step 2. Write out the cell contents in their target locations in a second pass across the grid.

It is obvious that this algorithm always takes $2n^2$ steps, which is suboptimal for most permutations. However we claim the algorithm is within a factor of optimal for almost all permutations.

Note first of all that *any* algorithm for realizing permutations with $b = n^2$ will usually take at least n^2 steps. To see this, note that

$$\begin{aligned}\Pr[\text{random } \pi \text{ has at least } k \text{ unit cycles}] &\leqq \binom{n^2}{k}(n^2-k)!/(n^2)! \\ &= 1/k!.\end{aligned}$$

(This probability may be evaluated precisely using the principle of inclusion and exclusion; see Liu [9].) Now the algorithm must visit every point x on the grid for which $\pi(x) \neq x$, i.e., for which π is not a unit cycle. However, as the above inequality shows, the number of permutations having many unit cycles is a very small percentage of the total set (asymptotically negligible). So n^2 steps are necessary almost all the time, and the two-pass algorithm is at worst a factor of two away from optimal.

The two-pass algorithm *can* be improved upon somewhat. Observe that if we can devise a schedule for the read/write head which reads in many cells' contents *before* the head moves to the permutation targets for these contents during the initial read-in pass, then these contents can be dropped off when the target contents are read in. If enough contents can be dropped off in this manner, then the second write-out pass will only require the read/write head to visit some fraction $p (0 < p < 1)$ of the grid locations. The whole process might only take time $(1+p)n^2$.

In fact we can guarantee $p \leqq \frac{1}{2}$. Consider *any* pass over the grid. Then either (1) at least half the points x in the pass are visited before $\pi(x)$ is visited in the pass, or else (2) this statement is true if the pass is reversed (done backwards). This observation leads to the following algorithm:

More intelligent two-pass algorithm.

Step 1. Make a read-in pass over the grid, which has the property that at least half of the points x on the grid are passed over before $\pi(x)$ is passed over. For each such point x, drop off its contents when $\pi(x)$ is passed over and the $\pi(x)$ contents are read in.

Step 2. Make a write-out pass over the grid which visits those points where contents must still be dropped off, and as few other points as possible.

This algorithm requires at most as much time as its predecessor, and has the additional benefit that it only uses $n^2/2$ registers at any given time. Hence: *$b = n^2/2$ is the most registers we could ever need*, and $b = n^2$ is wasteful.

We leave the precise analysis of this latter algorithm as an interesting open question. Is it possible to always choose a read-in pass schedule that guarantees p *smaller* than $\frac{1}{2}$, thereby improving Step 1? What algorithms may be used to generate efficient schedules for Step 2? etc.

5. The case $1 < b < n^2$. We have now established that, when $b = 1$, $O(n^3)$ time is necessary to realize the average permutation, while when $b = n^2$ only $O(n^2)$ time is necessary. It is interesting to ask what sort of behavior we get if we choose some intermediate value of b. It is obvious that

$$(\text{Time required } (b = 1)) \leqq k \text{ (Time required } (b = k))$$

for any k between 1 and n^2, but it is not obvious whether the inequality can be replaced by equality. We show that, modulo constant factors, it can. That is, for $1 < b \leqq n$ only $O(n^3/b)$ time is necessary. This suggests that having a large number of registers may not be cost effective.

We give an algorithm for $b = n$ which uses time $6n^2$ (in any metric) to realize any permutation. The algorithm is based on the operation of the three-stage rearrangeable switching network studied by Beneš [12] and others. It comprises three passes, each taking time $2n^2$ and modelling one stage of the three-stage network.

Permutation Algorithm for $b = n$.

Step 1. For each of the n rows of the grid;

(a) read the row in,

(b) write the row back out such that, at the end of Step 1, each column contains n items whose destinations are all in different rows.

Step 2. For each of the n columns of the grid;
(a) read the column in,
(b) write the column back out so that, at the end of Step 2, every item in the grid is in the same row as its destination.

Step 3. For each of the n rows of the grid;
(a) read the row in,
(b) write the row back out in permuted order.

It is obvious that the algorithm works if Step 1 can be made to do what it says it does. That it *can* is a consequence of the Slepian-Duguid theorem ([12], p. 86), the details of which are omitted here. An example is given in Fig. 7. The algorithm makes

$$(3 \text{ steps}) \times (n \text{ rows/cols per step}) \times (2n \text{ head movements per row/col})$$
$$= 6n^2 \text{ head movements}$$

as claimed.

11	7	19	23	24
6	1	10	12	25
16	4	18	15	17
22	2	5	9	21
14	13	20	3	8

(a) INITIAL CONFIGURATION (NUMBERS INDICATE ROW-MAJOR ORDERED DESTINATIONS)

11	7	19	23	24
6	1	25	12	10
18	15	4	17	16
22	21	9	2	5
3	20	14	8	13

(b) AFTER FIRST STEP: EACH COLUMN NOW CONTAINS 5 ITEMS WHOSE DESTINATIONS ARE IN DIFFERENT ROWS.

3	1	4	2	5
6	7	9	8	10
11	15	14	12	13
18	20	19	17	16
22	21	25	23	20

(c) AFTER SECOND STEP: COLUMNS HAVE BEEN PERMUTED SO EACH ITEM IS IN CORRECT ROW.

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

(d) AFTER THIRD STEP: GRID IS IN SORTED ORDER.

FIG. 7. *Examples of $b = n$ permutation algorithm execution (for case $n = 5$).*

This algorithm may be used recursively to derive interesting algorithms for $b < n$. Note that to permute a *row* (or column) of n cells using $b = \sqrt{n}$ registers using the same basic algorithm can be done by breaking the row (column) into $\sqrt{n}$ pieces of length $\sqrt{n}$. These pieces are then thought of as forming a $\sqrt{n} \times \sqrt{n}$ grid, except here moving from one piece to the next takes time $\sqrt{n}$ instead of 1. Permuting one row of the grid using the algorithm then requires essentially.

$$\underbrace{(2\sqrt{n}+\sqrt{n})\times\sqrt{n}}_{\text{Step 1 (subrows)}} + \underbrace{2(n-\sqrt{n})\times n}_{\text{Step 2 ("cols")}} + \underbrace{(2\sqrt{n}+\sqrt{n})\times\sqrt{n}}_{\text{Step 3 (subrows)}}$$
$$= 2n^{3/2} + 4n \text{ head movements.}$$

Therefore to permute the entire grid with $b=\sqrt{n}$ we make

$$(3 \text{ steps}) \times (n \text{ rows/cols per step}) \times (2n^{3/2}+4n \text{ head movements/row or col})$$

$$=6n^{5/2}+O(n^2) \text{ head movements.}$$

In general, recursive application of this algorithm with $b=n^{1/2^j}$ for integral j produces an algorithm requiring on the order of

$$6n^3/b \text{ head movements,}$$

so, for at least the values $k=n^{1/2^j}$, the inequality at the beginning of this section can be replaced by equality (within a constant factor near 6).

For really small values of b this approach will be inefficient. It would seem better in this situation to develop heuristics extending the basic cycle algorithm of § 3. One possible extension is a "greedy" heuristic which reads in the contents of b cells and then proceeds to drop off the item whose destination is closest. A new item is read in when the old item is dropped off, again the head moves to drop off the item whose destination is closest, and so forth. However we do not elaborate any further on this subject, leaving the development of algorithms for very small b as an interesting open problem.

One final comment should be made on the $b=n$ algorithm. Namely, it may be generalized immediately to an algorithm for a system with *n read/write heads, which only takes $O(n)$ time.* Assuming each head has $b=n$ registers, each pass over a row or column in each of the three steps may be handled by a single head. Obviously the heads can be coordinated so that they do not conflict with one another's movement. This approach may be used when the grid may be read both horizontally and vertically (a mild generalization of the scheme in [1]): in effect the grid becomes a "torus" with a set of $2n$ read/write heads permanently fixed on the torus axes.

6. Conclusions. A model of a general mass storage system was assumed, in which a single read/write head moves freely across a two-dimensional $n \times n$ grid of storage cells. Head motions were assumed to take time proportional to one of the L_1, L_2 or L_∞-metrics, and the head was capable of holding some fixed number b of cell values. The problem of finding efficient algorithms for rearranging the grid's contents according to some permutation μ was addressed.

For the important case $b=1$, a near-optimal algorithm (the "cycle algorithm") was presented and analyzed at length. Average and worst-case performance were determined for all three metrics, even under the complicating assumption that symmetry operations of the grid be used to reduce permuting cost. The performance figures given are excellent estimates of behavior since the corresponding standard deviations are asymptotically negligible.

For the cases $b=n^2$ and $1<b<n^2$ good permuting algorithms were presented and shown to be within a constant factor of optimal, but not analyzed in detail. The general behavior of algorithms for b in these ranges was determined, but it remains open to develop the optimal such algorithms, or good heuristics, especially for the case where b is very small but greater than one.

Acknowledgment. The authors are grateful to the referee, whose comments have made the presentation of this paper much clearer.

REFERENCES

[1] A. K. CHANDRA, HSU CHANG AND C. K. WONG, *Two-dimensional bubble domain memory*, U.S. Patent No. 4,174,538, Nov. 13, 1979.

[2] P. C. YUE AND C. K. WONG, *Near-optimal heuristics for an assignment problem in mass storage*, Internat. J. Comp. Inform. Sci., 4 (1975), pp. 281–294.

[3] S. H. FULLER, *Analysis of drum and disk storage units*, Lecture Notes in Computer Science, 31, Springer-Verlag, N.Y. 1975.

[4] P. P. BERGMANS, *Minimizing expected travel time on geometrical patterns by optimal probability rearrangements*, Inf. Cont., 20 (1972), pp. 331–350.

[5] R. M. KARP, A. C. MCKELLAR AND C. K. WONG, *Near-optimal solutions to a 2-dimensional placement problem*, this Journal, 4 (1975), pp. 271–286.

[6] D. E. KNUTH, *The Art of Computer Programming*, Vol. 3, Addison-Wesley Publishing Co., Reading, MA, 1973.

[7] C. H. PAPADIMITRIOU AND K. STEIGLITZ, *Some complexity results for the traveling salesman problem*, Proc. 8th ACM Symposium on Theory of Computing, Hershey, PA, May 3–5, 1976, pp. 1–9.

[8] M. R. GAREY, R. L. GRAHAM AND D. S. JOHNSON, *Some NP-complete geometric problems*, Proc. 8th ACM Symposium on Theory of Computing, Hershey, PA, May 3–5, 1976, pp. 10–22.

[9] C. L. LIU, *Introduction to Combinatorial Mathematics*, McGraw-Hill, New York, 1968, pp. 106–7.

[10] C. K. WONG AND K. C. CHU, *Average distances in L_p disks*, SIAM Rev., 19 (1977), pp. 320–324.

[11] D. T. LEE AND C. K. WONG, *Voronoi diagrams in $L_1(L_\infty)$ metrics with 2-dimensional storage applications*, this Journal, 9 (1980), pp. 200–211.

[12] V. BENEŠ, *Mathematical Theory of Connecting Networks and Telephone Traffic*, Academic Press, New York, 1965.

[13] C. K. WONG, C. L. LIU AND J. APTER, *A drum scheduling algorithm*, Lecture Notes in Computer Science, 2, Springer-Verlag, New York, 1973, pp. 267–275.

[14] J. RIORDAN, *An Introduction to Combinatorial Analysis*, John Wiley, New York, 1958.

[15] C. K. WONG, *Minimizing expected head movement in one-dimensional and two-dimensional mass storage systems*, Comp. Surv., 12 (1980), pp. 167–178.

SIAM J. COMPUT.
Vol. 11, No. 1, February 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1101-0009 $01.00/0

ALGORITHMS FOR EDGE COLORING BIPARTITE GRAPHS AND MULTIGRAPHS*

HAROLD N. GABOW† AND ODED KARIV‡

Abstract. A *minimum edge coloring* of a bipartite graph is a partition of the edges into Δ matchings, where Δ is the maximum degree in the graph. Coloring algorithms that run in time $O(\min(m(\log n)^2, n^2 \log n))$ are presented. The algorithms rely on an efficient procedure for the special case of Δ an exact power of two. The coloring algorithms can be used to find maximum cardinality matchings on regular bipartite graphs in the above time bound. An algorithm for coloring multigraphs with large multiplicities is also presented.

Key words. edge coloring, matching, bipartite graph, multigraph, open shop scheduling, euler partition

1. Introduction. Given a bipartite graph or multigraph, we seek a minimum edge coloring. An (*edge*) *coloring* is an assignment of a *color* to each edge of the graph so the edges incident to any vertex have distinct colors; equivalently, each color forms a matching. A *minimum coloring* uses as few colors as possible.

This problem arises in a number of settings. Examples are routing in a permutation network [LPV], preemptive scheduling of an open shop [GS], preemptive scheduling of unrelated parallel processors [LL], and the class-teacher timetable problem [Go†]. For instance, in the last of these we are given a collection of classes and teachers. Certain meetings must take place between classes and teachers; each meeting is specified by the class, the teacher and the number of periods the meeting lasts. A teacher can meet only one class at a time; a class can be taught by only one teacher at a time. A meeting that lasts more than one period can be scheduled as a number of one-period meetings, not necessarily consecutive in time. The problem is to schedule all meetings in as few periods as possible.

In the corresponding coloring problem, the bipartite multigraph has a vertex for each class and for each teacher. There is an edge joining a class and a teacher if they meet; the edge's multiplicity is the length of the meeting. A matching corresponds to meetings that can be scheduled in the same period. A coloring is a schedule (the colors, i.e., periods, can be ordered arbitrarily); a minimum coloring is a schedule using the fewest periods.

In practice, the timetable problem has additional constraints that make it difficult. For instance, if some teachers are available only during a restricted set of periods, it is NP-complete [E].

In contrast, the basic problem of finding a minimum edge coloring can be solved efficiently. Section 3 presents two algorithms that color graphs. The time bounds are $O(m(\log n)^2)$ and $O(n^2 \log n)$, where n and m are the number of vertices and edges, respectively. The first algorithm is superior to the second for nondense graphs ($m = O(n^2/\log n)$); it is also superior to previous coloring algorithms [Ga] [GK], the best of which is $O(m\sqrt{n} \log n)$. A preliminary version of the second algorithm appears in [GK]. Both algorithms are based on the fact that graphs with maximum degree a power of two can be colored rapidly.

* Received by the editors August 16, 1979, and in final form January 13, 1981.

† Computer Science Department, University of Colorado, Boulder, Colorado 80309. The work of this author was supported in part by the National Science Foundation under grant NSF78-18909.

‡ Computer Science Department, Technion, Haifa, Israel. This work was done while O. Kariv was with the Department of Computer Science of State University of New York, Albany, and his work was supported in part by that department.

Section 3 includes an application to matching. "High-low" bipartite graphs are defined; this class includes the regular graphs. A maximum cardinality matching on a high-low graph can be found in time $O(\min(m(\log n)^2, n^2 \log n))$. This improves on the bound of $O(m\sqrt{n})$ for general bipartite graphs [HK].

The algorithms of § 3 also work on multigraphs. However they become less efficient as edge multiplicities increase. Section 4 discusses coloring multigraphs with large multiplicities. An algorithm using time $O(nm_g \log K)$ is presented. Here m_g is the number of edges (not counting multiplicities) and K is the maximum edge multiplicity. The algorithm uses augmenting paths for colorings. A previous algorithm, based on matching theory, uses time $O(m_g^2)$ [GS]. For a large class of graphs (i.e., $n \log K = o(m_g)$), our algorithm is faster.

2. Preliminaries. This section introduces terminology and reviews previous coloring algorithms as a basis for the algorithms of §§ 3 and 4.

Throughout the paper, G denotes a given bipartite graph. n is the number of vertices, m the number of edges, and Δ the maximum degree. In § 4, G is a bipartite multigraph containing duplicate edges but no self-loops. We make a convention to distinguish between a multigraph and its underlying graph, in which edge multiplicities are ignored: The prefix "multi" indicates a reference to the multigraph; its absence indicates a reference to the underlying graph. For example, m_m denotes the number of multiedges (i.e., edges are counted according to their multiplicity), and m_g denotes the number of edges (in the underlying graph). So, $m_m \geqq m_g$.

A coloring that uses exactly k-colors is a *k-coloring.* It is well known that a minimum coloring (in a bipartite graph) is a Δ-coloring. (This follows from the fact that, by the König–Hall theorem, a graph has a matching that covers all vertices of degree Δ [B].) So, our problem is to find a Δ-coloring. We denote the colors as $1, \cdots, \Delta$. A *partial coloring* is a coloring of a subset of the edges. Figures 1 and 2 illustrate these terms (edge labels specify colors).

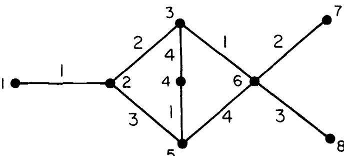


FIG. 1 *A 4-colored graph.*

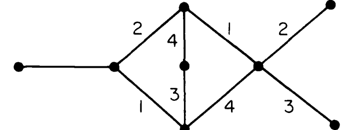


FIG. 2. *A partial coloring.*

A well-known approach to coloring [O] uses the method of augmenting paths. As such it resembles algorithms for network flows and matchings [L]. We begin with some definitions. Consider a partial Δ-coloring. *Vertex v misses color α* if no edge colored α is incident to v. Uncolored *edge vw misses color pair $\alpha\beta$* if v misses α and w misses β. If vw misses $\alpha\beta$, an *$\alpha\beta$ path (from vw)* is a path P that starts at v or w, has edges that are colored alternately α and β, and has maximal length. P is also called an *alternating path.* Note an alternating path can have no edges, if $\alpha = \beta$, or more generally, if v (or w) misses both α and β.

An alternating path can be used to color an uncolored edge. The following algorithm does this in linear space; the more direct approach of [GK] does not.

procedure *augment* (vw) **comment** vw is an uncolored edge in a partial Δ-coloring. vw gets colored;

begin

1. let vw miss color pair $\alpha\beta$;

2. let S be the subgraph of edges colored α or β **comment** the connected components of S are paths and cycles;
3. let P be a connected component of S incident to v or w (P can be $\emptyset$ if no edge of S meets v, or w) **comment** P is an $\alpha\beta$ path from vw;
4. interchange colors α and β on the edges of P **comment** now a color γ, $\gamma = \alpha$ or β, is missing at both v and w;
5. color edge vw **comment** use γ;

end *augment*;

As an example, consider the uncolored edge 12 in Fig. 2. For $\alpha\beta = 13$, subgraph S contains two paths; augmenting on 254 gives Fig. 1.

LEMMA 1. *"Augment" colors an uncolored edge in time $O(n)$. A graph can thus be colored in time $O(nm)$; the space is $O(n+m)$.*

Proof. First note *augment* works correctly. In line 1, colors α and β exist (there are Δ colors, and since vw is uncolored, both v and w miss a color). In line 2, the comment is true since α and β are matchings. Thus in line 3, P is in fact an $\alpha\beta$ path from vw. The comment in line 4 relies on the fact that P, if nonnull, does not end at v or w. (If it did, there would be an odd cycle in the graph). So in line 5, vw is colored correctly.

An appropriate data structure allows the time and space bounds to be achieved. One possibility is to represent the graph by adjacency lists; each edge in an adjacency list indicates its color; also, each color has a list of all edges with that color. Note line 1 is $O(n)$, if we bucket sort the colors occurring at v and at w. Further details are left to the reader. □

Another approach to edge coloring [Ga] uses divide-and-conquer. An *euler partition* is a partition of the edges of G into open and closed paths, so that each vertex of odd (even) degree is the end of exactly one (zero) open path. Any graph has an euler partition, which can be found in time $O(n+m)$ [B], [Ga]. The partition can be used to divide G into two edge-disjoint subgraphs G_1 and G_2. Traverse each path of the partition, placing edges alternately in G_1 and G_2. Then G_1 and G_2 both have maximum degree $\lfloor \Delta/2 \rfloor$ or $\lceil \Delta/2 \rceil$. (Since if uv and vw are consecutive edges in the partition, one is placed in G_1 and the other in G_2; thus v's degree gets halved.) This suggests the following recursive algorithm:

procedure *euler-color* (G) **comment** G is a bipartite graph with all edges uncolored. A $2^{\lceil \lg \Delta \rceil}$-coloring of G is found;

begin

1. let Δ be the maximum degree in G;
2. **if** $\Delta = 1$ **then** color all edges in G, using a new color **else begin**
3. divide G into edge-disjoint subgraphs G_1, G_2, each with maximum degree at most $\lceil \Delta/2 \rceil$ (use an euler partition);
4. *euler-color* (G_1);
5. *euler-color* (G_2);

end end *euler-color*;

For example, consider Fig. 1. Using three open paths, 12, 3254365 and 768, we get G_1 and G_2 of Fig. 3. The final coloring obtained is Fig. 1.[1]

Euler-color does not necessarily find a Δ-coloring. Suppose Δ is odd and both G_1 and G_2 have maximum degree $\lceil \Delta/2 \rceil$. Even if the recursive calls of lines 4–5 find

[1] In all examples of this paper the algorithms make choices. For convenience we do not point out that different results are possible if different choices are made.

FIG. 3. *Dividing the graph by an euler partition.*

minimum colorings, these colorings combine to give a $(\Delta+1)$-coloring of G, i.e., an extra color is used. However, the coloring uses at most twice the minimum number of colors.

LEMMA 2. *"Euler-color" finds a* $2^{\lceil \lg \Delta \rceil}$*-coloring.*[2] *The time is* $O(m \log n)$ *and the space is* $O(n+m)$.

Proof. Correctness follows from a simple induction: G_1 and G_2 have maximum degree $\lceil \Delta/2 \rceil$. So *euler-color* finds $(2^{\lceil \lg \Delta \rceil - 1})$-colorings of these graphs, which combine to give the desired coloring. A similar induction shows the time is $O(m \log \Delta)$. The space bound follows from careful programming of the recursion. Further details are in [Ga]. □

Note if Δ is an exact power of two, *euler-color* finds a minimum coloring. This is the basis of our algorithms.

3. Algorithms for graphs. This section presents two edge coloring algorithms, with time bounds $O(m(\log n)^2)$ and $O(n^2 \log n)$. On nondense graphs ($m = O(n^2/\log n)$) the first algorithm is faster. Both algorithms are based on the fact that graphs with Δ an exact power of two can be colored fast.

The first algorithm works by repeatedly enlarging a partial Δ-coloring. Each iteration constructs a subgraph S of maximum degree $2^{\lfloor \lg \Delta \rfloor}$, containing a large number of uncolored edges (and possibly some colored edges). S is colored, by erasing all its colors and using *euler-color*. This gives G more colored edges. The process is repeated until all edges are colored.

S is constructed by assigning color pairs to the uncolored edges. If edge vw is assigned the pair $\alpha\beta$, we say $\alpha(\beta)$ occurs at $v(w)$. Color pairs are assigned in such a way that a color α occurs at most once at a vertex v—in an edge colored α or in an edge assigned a pair with α at v. The exact role of the pairs in forming S will become clearer as the discussion proceeds.

procedure *color-by-pairs*; **comment** given is a bipartite graph G with maximum degree Δ, and all edges uncolored. A minimum coloring (using colors $1, \cdots, \Delta$) is found;

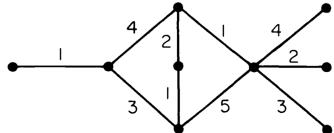
begin

1. **while** G has uncolored edges **do**
   begin
2. assign a color pair $\alpha\beta$ to each uncolored edge, so any color occurs at most once at any vertex;
3. find a set of $2^{\lfloor \lg \Delta \rfloor}$ colors C, such that $>\frac{1}{4}$ the uncolored edges are assigned a color pair $\alpha\beta$ with both $\alpha, \beta \in C$;
4. let S contain all edges whose colors are in C, i.e., edges colored α with $\alpha \in C$ and edges assigned $\alpha\beta$ with both $\alpha, \beta \in C$;
5. color the edges of S, using the colors C **comment** erase all colors in S and use *euler-color*;

end end *color-by-pairs*;

[2] Throughout this paper "lg" denotes logarithm base 2.

Consider the graph of Fig. 4. Figure 5 gives an assignment of color pairs. Choosing $C=\{1,2,3,4\}$, S is the graph of Fig. 1, and it gets the coloring shown (see § 2). The 5-coloring of Fig. 4 results if the next iteration chooses $C=\{2,3,4,5\}$.



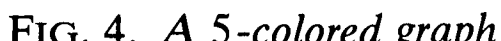


FIG. 4. *A 5-colored graph.*

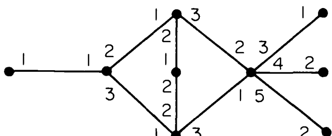


FIG. 5. *A color pair assignment.*

Now we analyze the algorithm. It is not obvious that line 3 can be done. For clarity of presentation we assume this; i.e., Lemmas 3 and 4 explicitly assume the following:

LEMMA 6. *Line 3 of "color-by-pairs" can be done, using time $O(m \log n)$ and space $O(m+n)$.*

Of course, the proof of Lemma 6 (see below) is independent of Lemmas 3 and 4.

LEMMA 3. *"Color-by-pairs" finds a minimum coloring (if we assume Lemma 6).*

Proof. It is easy to see that lines 1–4 can be done, assuming Lemma 6. Note S has maximum degree $\leqq |C| = 2^{\lfloor \lg \Delta \rfloor}$. So Lemma 2 shows *euler-color* can be used to color S, as claimed in line 5.

After line 5, G still has a valid partial coloring. This follows because there are no edges colored $\alpha \in C$ that are not in S.

Finally note that the **while** loop of line 1 eventually halts. In fact, the algorithm loops $O(\log m)$ times. For by line 3, after i iterations of the loop there are $<(\frac{3}{4})^i m$ uncolored edges. So there are $\leqq \lceil \log_{4/3} m \rceil$ iterations.

Now it is clear that *color-by-pairs* halts with a valid Δ-coloring. (This is true even if G is a multigraph). □

To analyze the timing, we start with a detailed implementation of line 2.

comment this code implements line 2 of *color-by-pairs*;

2.1 sort the adjacency lists of G in order of increasing color, using the order "uncolored" $<1<2\cdots<\Delta$;

2.2 **for** each uncolored edge vw **do**

begin let $\alpha(\beta)$ be the next largest color missing at $v(w)$ **comment** α is missing at v if no edge incident to α is colored α or assigned a pair with α at v;

assign $\alpha\beta$ to vw;

end;

Lines 2.1–2 use time $O(m+n)$. Line 2.1 is a bucket sort: The edges of G are placed into buckets, one for each color; then the edges of each bucket are placed on the appropriate adjacency list. Line 2.2 uses pointers that scan down every adjacency list. To find the next color missing at v, advance v's pointer until it reaches the next gap in the color sequence.

LEMMA 4. *"Color-by-pairs" uses time $O(m(\log n)^2)$ and space $O(m+n)$ (if we assume Lemma 6).*

Proof. The proof of Lemma 3 shows the **while** loop does $O(\log n)$ iterations. So it suffices to show each of lines 2–5 is $O(m \log n)$. Line 2 is $O(m+n)$, as indicated above. Line 4 is clearly linear. Lines 3 and 5 are $O(m \log n)$ by Lemmas 6 and 2, respectively. □

Now we discuss line 3. It is convenient to restate line 3 in terms of a multigraph M derived from the color pair assignment. M has a vertex for each color α, $1 \leqq \alpha \leqq \Delta$. It has an edge $\alpha\beta$ of multiplicity k if G has $k > 0$ uncolored edges assigned the pair $\alpha\beta$. Fig. 6 shows M for the assignment of Fig. 5 (edge labels give multiplicities). The task of line 3 is this: Given a multigraph M, with n vertices, m_g edges, and m_m multiedges (recall m_m counts each edge according to its multiplicity), find p vertices whose induced subgraph P has a large number of multiedges. (In line 3, $p = 2^{\lfloor \lg \Delta \rfloor}$, and $> m_m/4$ multiedges are required. Also, n (of $M) = \Delta$.)

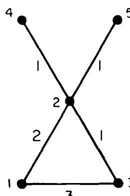


FIG. 6. *The multigraph derived from the assignment.*

A natural approach is to build P up in a greedy way: repeatedly add the vertex that in some sense has the highest degree. Variations of this method work (see below). However, better results come by reducing M down to P in a greedy way.

```
    procedure dense-graph; comment given is M, a multigraph, and p, an integer ≦n.
              The algorithm finds P, a subgraph induced on p vertices, containing a
              large number of multiedges;
      begin
1.    P := M;
2.    for i := n step −1 to p+1 do
        begin
3.      let v be the vertex of least multidegree in P;
4.      P := P − v;
      end end dense-graph;
```

In Fig. 6, for $p = 4$ the vertices of P are 1, 2, 3, 4 (as desired in Fig. 5). For $p = 2$, P contains 1 and 3.

LEMMA 5. *"Dense-graph" halts with P a subgraph of M having p vertices and* $\geqq m_m \cdot (p(p-1)/n(n-1))$ *multiedges. The time is* $O((m_g + n) \log n)$ *and the space is* $O(m_g + n)$.

Proof. First we discuss correctness. Let v be the vertex with least multidegree in M. Since the sum of all multidegrees in M is $2m_m$, v has multidegree $\leqq 2m_m/n$. Thus $M - v$ has $\geqq m_m(1-(2/n))$ multiedges. This reasoning shows that when *dense-graph* halts, the number of multiedges in P is at least

$$m_m \cdot \prod_{i=p+1}^{n} \left(1 - \frac{2}{i}\right) = m_m \cdot \prod_{i=p+1}^{n} \left(\frac{i-2}{i}\right) = m_m \cdot \frac{p(p-1)}{n(n-1)},$$

as desired.

To achieve the time bound, use a priority queue containing the multidegree of each vertex of P. Line 3 finds the vertex v of least multidegree and removes it from the queue. Line 4 removes the vertices adjacent to v from the queue, and reinserts them with multidegree appropriately reduced. Charging these queue operations to the corresponding vertex or edge shows the total time is $O((m_g + n) \log n)$. □

Lemma 6 follows easily:

LEMMA 6. *Line 3 of "color-by-pairs" can be done, using time $O(m \log n)$ and space $O(m+n)$.*

Proof. To do line 3 execute *dense-graph*, with M constructed as described above, and $p=2^{\lfloor \lg \Delta \rfloor}$. The desired colors C correspond to the vertices of P.

Note $p=2^{\lfloor \lg \Delta \rfloor} \geqq (\Delta+1)/2$, $n=\Delta$, and m_m is the number of uncolored edges. Lemma 5 implies P contains $>m_m/4$ multiedges. So C is as desired in line 3.

As for time and space, M can be constructed in linear time, using a bucket sort. Lemma 5 shows *dense-graph* runs in the desired time. □

Several remarks about *dense-graph* are in order. First, note that we may view *dense-graph* as an approximation algorithm for the clique problem. This problem is, given a graph G and an integer p, find a complete subgraph on p vertices, i.e., a subgraph with p vertices and $p(p-1)/2$ edges. The clique problem is NP-complete and so, probably intractable [K]. *Dense-graph* does not solve this problem but a related one: Find a subgraph with a guaranteed number of edges. For this related problem, *dense-graph's* bound of $m_m \cdot (p(p-1)/n(n-1))$ multiedges is the best possible. This can been seen by taking M as the complete graph on n vertices.

As expected, decreasing the bound on the number of multiedges allows faster algorithms. For example, consider this alternative algorithm: First, from M, delete the $(n-p)/2$ vertices with smallest multidegree; then, from the resultant graph, delete the $(n-p)/2$ vertices with smallest multidegree. This algorithm is faster than *dense-graph*—the time is $O(m_g+n)$, using linear median finding. It is straightforward to calculate the bound on the number of edges. For instance, when $p \geqq (n+1)/2$ (the region of interest for *color-by-pairs*), the bound is $\geqq m_m/6$ multiedges.

Other variations and generalizations of *dense-graph* are possible. For example, the above linear algorithm works if we build P up rather than reduce M. A similar but simpler linear algorithm is given in [LPV]. Any method that, when $p \geqq (n+1)/2$, achieves a bound of cm_m multiedges (for some constant $c>0$) in time $O((m_g+n)\log n)$ suffices for *color-by-pairs*.

Lemmas 3, 4, and 6 complete the analysis.

THEOREM 1. *"Color-by-pairs" finds a minimum coloring, in time $O(m(\log n)^2)$ and space $O(m+n)$.*

An interesting variant of *color-by-pairs* that runs efficiently on a parallel machine (the Parallel Random Access Computer) is given in [LPV]. The time is $O((\log n)^3)$.

Now we present the second coloring algorithm. It also repeatedly colors subgraphs with maximum degree a power of two. However, it proceeds recursively. The graph is divided in two, using an euler partition. The first subgraph is colored recursively. This coloring is used to enlarge the second subgraph, so its maximum degree is a power of two; it is colored by *euler-color*.

procedure *color-by-partition* (G); **comment** G is a bipartite graph, all of whose edges are uncolored. A minimum coloring of G is found;

begin

1. let Δ be the maximum degree in G;
2. **if** $\Delta=1$ **then** color all edges in G, using a new color
   else begin
3. divide G into edge-disjoint subgraphs G_1 and G_2 having maximum degree Δ_1 and Δ_2, where $\Delta_1, \Delta_2 \leqq \lceil \Delta/2 \rceil$ and G_1 has no more edges than G_2 (use an euler-partition);
4. *color-by-partition* (G_1);

5. remove the edges of r colors from G_1 and add them to G_2, where $r = 2^{\lceil lg \Delta/2 \rceil} - \Delta_2$;
6. *euler-color* (G_2); **comment** now the colorings of G_1 and G_2 (as modified in line 5) give a Δ- or $(\Delta+1)$-coloring of G;
7. **if** G is not Δ-colored **then**
   begin
8. make all edges of some color α uncolored;
9. **for** all uncolored edges e **do** *augment* (e);
   end end end *color-by-partition*;

For Fig. 4, an euler partition gives the subgraphs of Fig. 7; the recursive call of line 4 colors the left subgraph G_1 as shown. Line 5 enlarges G_2 to Fig. 1, which gets colored as shown. Adding the edge of color 5 from G_1 gives the desired 5-coloring.

Figure 8 gives the call tree for this example. The root represents the original call to *color-by-partition*, on a graph of maximum degree 5 (Fig. 4). The left son of the

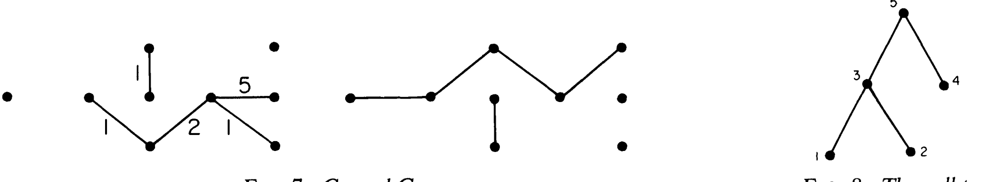


FIG. 7. G_1 *and* G_2.

FIG. 8. *The call tree.*

root represents the recursive call, on a graph of maximum degree 3 (Fig. 7); the right son represents the call to *euler-color*, on a graph of maximum degree 4 (Fig. 1). Other nodes are interpreted similarly.

LEMMA 7. *"Color-by-partition" finds a minimum coloring.*

Proof. The argument is by induction on Δ. If $\Delta = 1$, line 2 colors the graph correctly. Now assume $\Delta > 1$. By induction, the recursive call of line 4 correctly colors G_1.

In line 5, note the transfer of edges can be done, i.e., $\Delta_1 \geqq r \geqq 0$. For $\Delta_1 \geqq r$, note $\Delta_1 \geqq \Delta - \Delta_2$ and $\Delta > 2^{\lceil \lg \Delta/2 \rceil}$; for $r \geqq 0$, note $2^{\lceil \lg \Delta/2 \rceil} \geqq \lceil \Delta/2 \rceil \geqq \Delta_2$.

After the transfer of line 5, G_2 has maximum degree $\leqq \Delta_2 + r = 2^{\lceil \lg \Delta/2 \rceil}$. So, in line 6, *euler-color* finds a $(\Delta_2 + r)$-coloring of G_2. G_1 is colored with $\Delta_1 - r$ additional colors. Thus, after line 6, G is $(\Delta_1 + \Delta_2)$-colored. $\Delta_1 + \Delta_2$ is Δ or $\Delta + 1$, by the euler partition of line 3. So there is at most one extra color. It is eliminated by the augments of lines 7–9.

Now by induction, *color-by-partition* finds a minimum coloring (this is true even if G is a multigraph). $\square$

LEMMA 8. *"Color-by-partition" uses time* $O(n^2 \log n)$ *and space* $O(m+n)$.

Proof. For the time bound, consider the call tree illustrated in Fig. 8. In general this tree has nodes representing calls to *color-by-partition* and *euler-color*. The root represents the original call, *color-by-partition* (G). A *color-by-partition* node can have two sons; the left son represents the call *color-by-partition* (G_1) (line 4), and the right son represents *euler-color* (G_2) (line 6). *Euler-color* nodes are leaves. Suppose the levels of the tree are numbered, with 0 at the root. So each level except 0 has two nodes.

We note two facts about this tree. First, it has at most $\lceil \lg \Delta \rceil + 1$ levels. For the *color-by-partition* son of a *color-by-partition* node satisfies $\Delta_1 \leqq \lceil \Delta/2 \rceil$ (see line 3). Hence a *color-by-partition* node on level i has degree at most $\lceil \Delta/2^i \rceil$, and the conclusion follows.

Second, let m_i be the number of edges in the graph of a node on level i. For a *color-by-partition* node, $m_i \leqq m/2^i$, by the euler partition of line 3. So for an *euler-color* node, $m_i \leqq m/2^{i-1}$ (recall that line 5 enlarges G_2).

Now we estimate the time. We first consider the total time spent in *color-by-partition* nodes, and then the time in *euler-color* nodes.

For *color-by-partition* nodes, we estimate the time in a typical node, excluding the calls in lines 4 and 6. Lines 7–9 do $\leqq n$ augments. Each of these is $O(n)$, giving $O(n^2)$ time. The other lines are $O(m+n)$. So the time in a typical node is $O(n^2)$, and the total time for *color-by-partition* nodes is $O(n^2 \log n)$.

Next consider the *euler-color* nodes. For a node on level i, the time is $O(m_i \log n)$ (by Lemma 2). Summing over all levels and noting $m_i \leqq m/2^{i-1}$ shows the total time is $O(m \log n)$.

Thus the total time for *color-by-partition* is $O(n^2 \log n)$, as desired.

The linear space bound follows from careful programming, making sure that an edge is represented only once in all levels of the recursion. See also Lemmas 1–2. □

Lemmas 7 and 8 give the desired result:

THEOREM 2. *"Color-by-partition" finds a minimum coloring, in time $O(n^2 \log n)$ and space $O(m+n)$.*

We close this section by noting how coloring algorithms can be used to find matchings for a large class of graphs.

DEFINITION. A bipartite graph is *high-low* if for some integer k, one vertex set contains only vertices of degree $\geqq k$, and the other contains only vertices of degree $\leqq k$.

High-low graphs include both regular graphs and semiregular graphs. (A bipartite graph is *semiregular* if one vertex set (or both) contains only vertices of degree Δ.)

A coloring algorithm can be used to find a maximum cardinality matching on a high-low graph H. First prune H to a semiregular graph S (with $\Delta = k$). Any color of a minimum coloring covers all degree Δ vertices. So a coloring of S gives a maximum matching of H.

THEOREM 3. *A maximum cardinality matching can be found in a high-low graph in time $O(\min(m(\log n)^2, n^2 \log n))$ and space $O(m+n)$.*

The general bipartite matching algorithm of [HK] uses time $O(m\sqrt{n})$. Theorem 3 improves this for high-low graphs. Other improvements, based on *euler-color*, are given in [Ga], [GK]. For example, in a semiregular graph with Δ a power of two, a maximum matching can be found in time $O(m+n)$. (Use *euler-color*, only recurring on G_1).

4. Large multiplicities. This section discusses coloring multigraphs with large multiplicities. An algorithm using $O(nm_g \log K)$ time is presented. This is faster than previous algorithms on a large class of graphs (more precisely, if $n \log K = o(m_g)$).

The difference between coloring graphs and multigraphs is that in a multigraph a matching can be used for more than one color, if each of its edges has multiplicity greater than one. Hence the term *multicolor* denotes a set of colors, each of which uses the same matching; the *multiplicity* (of a multicolor) is the number of colors in the set. (When the meaning is clear from context, we use "color" instead of "multicolor," and speak of the "multiplicity of a color," etc.) Thus a minimum edge coloring can be viewed as a collection of δ multicolors of multiplicity k_i, $i = 1, \cdots, \delta$, where $\Delta = \sum_{i=1}^{\delta} k_i$.

Figure 9 shows a multigraph (edge labels give multiplicites). Figure 10 shows a coloring that uses multicolors $1, \cdots, 5$ (edge labels give multicolors). Note that multicolors 1 and 5 can be combined to give a coloring with four multicolors.

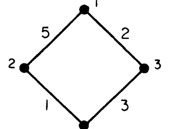
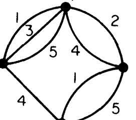


FIG. 9. *A multigraph.*



multicolor	1	2	3	4	5
multiplicity	2	1	2	1	1

FIG. 10. *A 7-coloring with 5 multicolors.*

Our problem is to find a minimum coloring of a bipartite multigraph. It is also desirable to economize on the number of multicolors. For instance, in scheduling applications, the colors of a multicolor can be scheduled in consecutive time periods. In the class-teacher timetable problem, this reduces the number of interrupted meetings; in open shop scheduling it reduces the number of preemptions [GS], [LL]. (The number of preemptions is less than n times the number of multicolors.)

The algorithms of § 3 color multigraphs. However, they do not take advantage of large multiplicities of multiedges and multicolors. Here we give an algorithm that does. If K is the largest edge multiplicity, the time and space are both $O(nm_g \log K)$; the number of multicolors is $O(m_g \log K)$. (The algorithms of § 3 use time $O(m_m(\log m_m)^2)$ and $O((n^2+m_m)\log m_m)$, and $O(m_m)$ multicolors.)

Gonzalez and Sahni [GS] give an algorithm for this problem based on augmenting paths for matchings. Their algorithm uses time $O(m_g^2)$.[3] The space is $O(nm_g)$ (or if colors can be output as they are found, $O(m_g+n)$). The number multicolors is $O(m_g)$.

Our algorithm runs faster than [GS] if

$$n \log K = o(m_g). \tag{1}$$

For example, suppose $K = O(n^a)$ for some constant $a > 0$. (This is not unreasonable, since otherwise the time for doing arithmetic on multiplicities cannot be ignored, as it is in the time bounds.) Then (1) becomes $n \log n = o(m_g)$. So, for instance, if $m_g = \Omega(n^{1+\varepsilon})$ for some $\varepsilon > 0$, (1) holds and our algorithm is faster. Thus, our algorithm is faster on a large collection of graphs. However, it is clear that our algorithm uses a factor $O(\log K)$ more space and more multicolors.

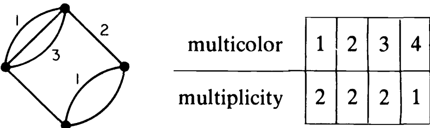
Our algorithm is based on augmenting paths for colorings. In a multigraph an augment works as follows. Let e be an uncolored edge of multiplicity k_e. Suppose e misses color pair $\alpha\beta$, where colors α, β have multiplicity k_α, k_β, respectively. One $\alpha\beta$ path P can be used to augment k copies of e, where $k = \min(k_\alpha, k_\beta, k_e)$. (Imagine doing a series of standard augments along P, each time using copies of α, β, and e.) The result is a new coloring where α and β have multiplicities $k_\alpha - k$ and $k_\beta - k$, respectively; new colors α' and β' have multiplicities k; and an uncolored copy of e has multiplicity $k_e - k$. (Note a color or edge of multiplicity 0 can be discarded, so either α, β, or the uncolored edge e disappears).

For example, consider edge 24 in Fig. 11. By choosing $\alpha\beta = 22$, $P = \emptyset$, and augmenting, multicolors 2 and 4 of Fig. 10 are formed.

Ideally we would like to augment just once for each multiedge. This will be the case if

$$k_e \leqq \min(k_\alpha, k_\beta) \tag{2}$$

[3] This time bound is for the simpler of two algorithms in [GS]. The other algorithm has a lower bound when $n_1^2 < m_g$ (here n_1 is the number of vertices in the smaller of the two vertex sets of the bipartite graph). In our algorithm's time bound, n can be replaced by n_1, allowing a comparison with the second algorithm. Since the result is similar but more involved, we omit it. Also, see [Gon].



multicolor	1	2	3	4
multiplicity	2	2	2	1

FIG. 11. *A partial coloring.*

(for if (2) holds, $k = k_e$, whence all copies of e get colored). However, (2) is difficult to achieve, since an augment introduces color multiplicities $k_\alpha - k$ and $k_\beta - k$, and these quantities can be small. This may cause (2) to fail in a later augment.

A special case where (2) always holds is when the edge multiplicities, say $k_1, \cdots, k_m$, satisfy $k_i \geqq 2k_{i+1}$. In this case, we can augment edges in order of decreasing multiplicity. Then before any augment $k_\alpha, k_\beta \geqq 2k_e$ (since if this holds before an augment, the new color multiplicities $(k_\alpha - k_e, k_\beta - k_e, k_e)$ are $\geqq k_e$, whence it holds for the next augment.) Hence we do only one augment per multiedge.

The general case can be transformed to the special one, at a slight price: An edge of multiplicity $k > 1$ is split into $\leqq \lceil \lg k \rceil$ copies, each with multiplicity a power of two, using the binary expansion of k. Then all multiplicities are a power of two, and essentially the above strategy works.

The algorithm below uses this approach. We assume the multigraph is given as a collection of adjacency lists, where each edge specifies its (integer) multiplicity. Similarly, the output coloring is specified as a list of colors (i.e., matchings) each with its multiplicity.

The algorithm starts with only one multicolor, Λ, corresponding to the null matching, having multiplicity Δ. New multicolors are formed by splitting off from old ones in augments. However, the number of colors is always Δ.

procedure *color-by-multiplicity* **comment** given is M, a multigraph with large multiplicities. A coloring with large multiplicities is found;
begin
1. let Λ be (the only) multicolor, with a matching that is empty, and multiplicity Δ (where Δ is the largest multidegree of a vertex);
2. let K be the largest multiplicity of an edge;
3. **for** $p := 2^{\lfloor \lg K \rfloor}, 2^{\lfloor \lg K \rfloor - 1}, \cdots, 2^1, 2^0$ **do**
   begin
4. **for** each uncolored edge e of multiplicity $k \geqq p$ **do**
   begin
5. let e miss $\alpha\beta$, where α and β are (existing) multicolors, of multiplicities k_α and k_β, and $\alpha, \beta \neq \Lambda$ if possible **comment** $k_\alpha, k_\beta \geqq p$;
6. split e into uncolored multiedges f, h of multiplicites $p, k - p$, respectively; **comment** discard edges of multiplicity 0;
7. color multiedge f by augmenting along an $\alpha\beta$ path, forming multicolors α' and β' (multiplicity p) and original multicolors α, β (multiplicities $k_\alpha - p, k_\beta - p$) **comment** discard colors of multiplicity 0;

end end end *color-by-multiplicity*;

Figure 11 shows the coloring for Fig. 9 after the iteration for $p = 2$. The coloring of Fig. 10 can be obtained, if the algorithm chooses pairs $\alpha\beta$ so the augmenting path is always empty.

LEMMA 9. *"Color-by-multiplicity" finds a minimum coloring of a multigraph, using $O(m_g \log K)$ multicolors.*

Proof. We first prove that *color-by-multiplicity* finds a minimum coloring. To start we show that at all times the partial coloring uses Δ colors; further, for any multicolor $\alpha \neq \Lambda, p|k_\alpha$. The argument is by induction on the number of augments. Clearly the inductive hypothesis holds before any augment has been done.

Now consider an augment in lines 5–7. Note in line 5, k_α, $k_\beta \geqq p$. This is true by induction if $\alpha(\beta) \neq \Lambda$. Otherwise if $\alpha(\beta) = \Lambda$, then by the choice rule in line 5, one end of e misses only one multicolor, Λ. Thus $\Delta \geqq k + \sum_{\gamma \neq \Lambda} k_\gamma$. Also, by induction, $\Delta = k_\Lambda + \sum_{\gamma \neq \Lambda} k_\gamma$. So, $k_\Lambda \geqq k \geqq p$, as desired.

Thus, the augment can always be done as specified in lines 6–7. It is easy to see the augment conserves the number of colors, and further, $p|k_\gamma$ for $\gamma \neq \Lambda$.

The halving of p in line 3 also preserves the condition $p|k_\gamma$. This completes the induction.

In the last iteration of line 3, $p = 1$. So the loop of lines 4–7 colors all remaining uncolored edges. Hence the algorithm halts with a Δ-coloring, as desired.

Now consider the number of multicolors. It is easy to see that at most $\lfloor \lg K \rfloor + 1$ augments are done for each multiedge e. (More precisely, an augment is done for each one in the binary expansion of e's multiplicity). Each augment creates at most two new multicolors α', β'. So there are $\leqq 2m_g(\lfloor \lg K \rfloor + 1)$ multicolors. □

We sketch some implementation details. Line 5 requires finding a multicolor missing at a given vertex. To do this, each vertex has a list of the multicolors it misses. Line 7 requires finding an (augmenting) $\alpha\beta$ path from f. To do this, there is a table $T(v, \alpha)$ that specifies, for each vertex v and each multicolor α, the edge (if any) of color α incident to v. (Note this approach to finding the path uses more space than the approach in § 2.) Finally, each multicolor α specifies its multiplicity. (Some auxiliary pointers are left to the reader.)

LEMMA 10. *"Color-by-multiplicity" uses time and space* $O(nm_g \log K)$.

Proof. For the time bound, there are $O(m_g \log K)$ augments (as noted in the proof of Lemma 9). An augment takes time $O(n)$, if the above data structure is used. For the space bound, the lists of missing multicolors and the table T use space $O(nm_g \log K)$. Further details are left to the reader. □

Lemmas 9–10 are summarized as follows.

THEOREM 4. *"Color-by-multiplicity" finds a minimum coloring of a bipartite multigraph. It uses* $O(nm_g \log K)$ *time and space, and* $O(m_g \log K)$ *multicolors.*

Note that in the special case $k_i \geqq 2k_{i+1}$ mentioned above, the bounds on time, space and multicolors all decrease by a factor $O(\log K)$.

Acknowledgment. We thank Adi Shamir for his stimulating conversations and for defining "high-low" graphs.

REFERENCES

[B] C. BERGE, *Graphs and Hypergraphs*, North-Holland, Amsterdam, 1973.

[E] S. EVEN, A. ITAI AND A. SHAMIR, *On the complexity of time-table and multicommodity flow problems*, this Journal, 5 (1976), pp. 691–703.

[Ga] H. GABOW, *Using euler partitions to edge color bipartite multigraphs*, Inter. J. Comp. and Inf. Sci., 5 (1976), pp. 345–355.

[GK] H. GABOW AND O. KARIV, *Algorithms for edge coloring bipartite graphs*, Proc. Tenth Ann. ACM Symposium on Theory of Comp., San Diego, CA., 1978, pp. 184–192.

[Gon] T. GONZALEZ, *A note on open shop preemptive schedules*, IEEE Trans. Comput., C-28 (1979), pp. 782–786.

[GS] T. GONZALEZ AND S. SAHNI, *Open shop scheduling to minimize finish time*, J. Assoc. Comput. Mach., 23 (1976), pp. 665–679.

[Got] C. C. GOTLIEB, *The construction of class-teacher time-tables*, Proc. IFIP Congress 62, Munich, North-Holland, Amsterdam, 1963, pp. 73–77.

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

[K] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum, New York, 1972, pp. 85–103.

[L] E. L. LAWLER, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.

[LL] E. L. LAWLER AND J. LABETOULLE, *On preemptive scheduling of unrelated parallel processors by linear programming*, J. Assoc. Comput. Mach., 25 (1978), pp. 612–619.

[LVP] G. LEV, N. PIPPENGER AND L. G. VALIANT, *A fast parallel algorithm for routing in permutation networks*, IEEE Trans. Comput., C-30 (1981), pp. 93–110.

[O] O. ORE, *Theory of Graphs*, AMS Colloquium Publications 38, American Mathematical Society, Providence, RI, 1962.

SIAM J. COMPUT.
Vol. 11, No. 1, February 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1101-0010 $01.00/0

TWO FAMILIAR TRANSITIVE CLOSURE ALGORITHMS WHICH ADMIT NO POLYNOMIAL TIME, SUBLINEAR SPACE IMPLEMENTATIONS*

MARTIN TOMPA†

Abstract. Any Boolean straight-line program which computes the transitive closure of an $n \times n$ Boolean matrix by successive squaring requires time exceeding any polynomial in n if the space used is $o(n)$. This is the first demonstration of a "natural" algorithm which (1) has a polynomial time implementation and (2) has a small (e.g., $O(\log^2 n)$) space implementation, but (3) has no implementation running in polynomial time and small space simultaneously. It is also shown that any implementation of Warshall's transitive closure algorithm requires $\Omega(n)$ space, and that many familiar sorting algorithms exhibit similar behavior.

Key words. transitive closure, sorting, time-space tradeoff, pebbling, straight-line program

1. Introduction. The transitive closure of an $n \times n$ Boolean matrix can be computed by a Boolean circuit of polynomial size and $O(\log^2 n)$ depth, by $O(\log n)$ matrix squaring operations. There are two obvious implementations of this algorithm on a sequential machine, one running in polynomial time (corresponding to a breadth-first evaluation of the circuit), the other in $O(\log^2 n)$ space (corresponding to depth-first evaluation). However, a recent conjecture maintains that there is no algorithm which computes transitive closure and runs in polynomial time and $(\log n)^{O(1)}$ space simultaneously (see, for example, Cook [1979]). This paper provides two pieces of evidence relevant to the conjecture:

(1) Any implementation of the aforementioned successive squaring algorithm requires time exceeding any polynomial in n if the space used is $o(n)$ (independent of the subprocedure chosen for matrix squaring).

(2) Any implementation of Warshall's transitive closure algorithm (Warshall [1962]) requires space $\Omega(n)$.

Most of the recent results concerning time-space tradeoffs fall into two categories, those which demonstrate modest tradeoffs for algorithms which solve "natural" problems (Abelson [1978], Borodin and Cook [1980], Borodin et al. [1979], Grigoryev [1976], Ja'Ja' [1980], Munro and Paterson [1978], Savage and Swamy [1978], [1979], Tompa [1980], and Yao [1979]) and those which introduce algorithms designed specifically to demonstrate that a decrease in space in their implementation causes the required time to increase from polynomial to superpolynomial (Carlson and Savage [1980], Lengauer and Tarjan [1979], Lingas [1978], Paul and Tarjan [1978], and van Emde Boas and van Leeuwen [1978]). The time-space tradeoff presented in this paper for transitive closure by successive squaring is the first demonstration of a "natural" algorithm which exhibits the time and space behavior of the latter category.

A method for determining the time and space requirements of straight-line implementations of circuits comes from a well-known "pebbling game" played on the circuit (for a survey, see Pippenger [1980]). The programmer is given a supply of pebbles which may be placed on the vertices of the circuit in a sequence of moves. Each move consists of picking up 0 or more pebbles, and putting down exactly 1. There is no restriction on which pebbles may be removed, but a pebble may only be

* Received by the editors September 24, 1980, and in revised form February 17, 1981. This material is based upon work supported by the National Science Foundation under grant MCS77-02474.

† Department of Computer Science, FR-35, University of Washington, Seattle, Washington 98195.

placed on a vertex v (called *pebbling* v) if all vertices with edges directed into v were pebbled at the beginning of the move. The goal of the game is to *pebble the circuit*, which means that each output must have been pebbled at some point. Intuitively, each pebble corresponds to a register, and pebbling a vertex v corresponds to storing in that register the subexpression computed at v. Each pebbling of a circuit then corresponds to a single straight-line implementation, using time equal to the number of moves in the pebbling and space equal to the maximum number of pebbles simultaneously on the circuit.

2. A time-space tradeoff for successive squaring. The transitive closure of an $n \times n$ Boolean matrix A is simply $(I \vee A)^{n-1}$, and hence can be computed by $\lceil \log_2 (n-1) \rceil$ matrix squaring operations. This section applies a method of Grigoryev [1976] to demonstrate a modest time-space tradeoff for any Boolean straight-line program which squares matrices. Using a technique of Paul and Tarjan [1978], this modest tradeoff for a single squaring operation is compounded into a dramatic tradeoff for $\lceil \log_2 (n-1) \rceil$ successive squaring operations. Notice that the result is independent of the particular subprocedure chosen for matrix squaring.

The following independence notion is central in Grigoryev's work: A function $f:\{0,1\}^p \to \{0,1\}^q$ is *m-independent* if and only if:

(a) for all $k \leqq m$,

(b) for all sets of k inputs $x_{i_1}, x_{i_2}, \cdots, x_{i_k}$, and

(c) for all sets of $m-k$ outputs $y_{j_1}, y_{j_2}, \cdots, y_{j_{m-k}}$,

there is an assignment of values to $x_{i_1}, x_{i_2}, \cdots, x_{i_k}$ such that $y_{j_1}, y_{j_2}, \cdots, y_{j_{m-k}}$ assume at least $2^{m-k-1}+1$ of their possible 2^{m-k} values when only the other $p-k$ inputs are allowed to vary.

The following lemma establishes the relationship between independence of a function and the time and space requirements to compute it:

LEMMA 1 (Grigoryev [1976]). *Let G be any Boolean circuit which computes an m-independent function. Then in the course of pebbling any $S+1$ outputs, starting from any configuration of S pebbles on G, at least $m-S$ inputs must be pebbled.*

Proof. Assume only $k \leqq m-S-1$ inputs $x_{i_1}, x_{i_2}, \cdots, x_{i_k}$ are pebbled in the course of pebbling outputs $y_{j_1}, y_{j_2}, \cdots, y_{j_{S+1}}$. Consider the Boolean straight-line program which corresponds to this pebbling. The values of its $S+1$ outputs are determined solely by the values of the k inputs read and the initial values of the S Boolean registers corresponding to pebbles. By the definition of m-independence, there is some fixed assignment of values to the inputs $x_{i_1}, x_{i_2}, \cdots, x_{i_k}$ such that $y_{j_1}, y_{j_2}, \cdots, y_{j_{S+1}}$ assume at least 2^S+1 combinations of values. But these $S+1$ outputs are determined completely by the initial contents of the S Boolean registers once the input values read are fixed, and these registers can assume only 2^S distinct combinations of values. □

COROLLARY (Grigoryev [1976]). *The time T and space S required to compute an m-independent function $f:\{0,1\}^p \to \{0,1\}^q$ satisfies $(S+1)T \geqq (q-S)(m-S)$.*

Proof. For every set of $S+1$ outputs pebbled, $m-S$ inputs must be pebbled. Hence,

$$T \geqq \lfloor q/(S+1) \rfloor (m-S) \geqq (q-S)(m-S)/(S+1). \quad \square$$

Grigoryev applied this result to prove a time-space tradeoff for matrix multiplication. An examination of the technique used to prove n-independence for matrix multiplication will shed some light on how a similar independence result may be proved for matrix squaring:

LEMMA 2 (Grigoryev [1976]). *Let* mult: $\{0, 1\}^{2n^2} \to \{0, 1\}^{n^2}$ *be the function which multiplies two* $n \times n$ *Boolean matrices* A *and* B. *Then* mult *is* n*-independent.*

Proof. Let $k \leqq n$, and suppose k inputs and $n-k$ outputs are specified. By fixing B to be a permutation matrix, the columns of AB can be made an arbitrary permutation of the columns of A. Since the specified inputs occupy at most k columns of A and the specified outputs at most $n-k$ columns of AB, B can be fixed in such a way that the $n-k$ specified outputs are identically $n-k$ unspecified inputs. Thus, these outputs assume all 2^{n-k} possible combinations of values when only unspecified inputs are allowed to vary. □

Let sq: $\{0, 1\}^{n^2-n} \to \{0, 1\}^{n^2-n}$ be the function which maps the off-diagonal entries of an $n \times n$ Boolean matrix A into the off-diagonal entries of $(I \vee A)^2$. There are two minor obstacles to proving a result about sq similar to Lemma 2: there is no "extra" matrix B to manipulate, and some care must be taken in choosing the column permutation in order to avoid the 1s on the main diagonal of $I \vee A$. To overcome the first obstacle, the roles of A and B in the proof of lemma 2 will be combined into one matrix, and we will be satisfied with (approximately) $n/2$-independence. The following lemma will be useful in overcoming the second obstacle:

LEMMA 3. *Let* $K, C_1, C_2, \cdots, C_r$ *be nonempty sets. If* $|K| > \sum_{i=1}^{r} |C_i|$, *then there are distinct elements* $k_1, k_2, \cdots, k_r$ *in* K *such that* $k_i \notin C_i$.

Proof. For each C_i there must be at least r elements in $K - C_i$, since the other C_j are nonempty. The lemma then follows by an easy application of P. Hall's theorem on systems of distinct representatives. (Consult any of several combinatorics texts, e.g., Liu [1968, Thm. 11-1].) □

LEMMA 4. sq *is* $\lfloor (n-1)/2 \rfloor$*-independent.*

Proof. Let $m = \lfloor (n-1)/2 \rfloor$. Let $0 \leqq k \leqq m$, and suppose k inputs and $m-k$ outputs are specified. Let K be the set of column numbers which contain neither specified inputs nor specified outputs, so $|K| \geqq n - m > m$. Let $C_j = \{i | \text{output position } (i, j) \text{ is specified}\}$. By Lemma 3, there is an injection $f: \{j | C_j \neq \emptyset\} \to K$ such that $f(j) \notin C_j$. For each j such that $C_j \neq \emptyset$, fix

$$A_{i,j} = \begin{cases} 1 & \text{if } i = j \text{ or } i = f(j), \\ 0 & \text{otherwise.} \end{cases}$$

and fix all others of the k specified inputs arbitrarily. Then if (i, j) is a specified output position, $(A \vee I)^2_{i,j} = A_{i,j} \vee A_{i,f(j)}$. But $A_{i,j}$ is already fixed at 0 ($i \neq j$ since (i, j) is specified, and $i \neq f(j)$ since $i \in C_j$ and $f(j) \notin C_j$) so $(A \vee I)^2_{i,j}$ is identically $A_{i,f(j)}$. Note that by the choice of K, column $f(j)$ of A has no entries fixed other than the diagonal entry, and $A_{i,f(j)}$ is not the diagonal entry since, as already observed, $i \neq f(j)$. Finally, the mapping from specified outputs (i, j) to $(i, f(j))$ is injective, so the $m-k$ specified outputs assume all 2^{m-k} possible combinations of values when only unspecified inputs are allowed to vary. □

COROLLARY. *Any Boolean straight-line program which computes* sq *using* S *Boolean registers and* T *steps requires* $(S+1)T \geqq n^3/4 - O(n^2)$.

Proof.

Case 1 ($S \geqq n/4$). Since $T \geqq n^2 - n$, $(S+1)T \geqq n^3/4 + 3n^2/4 - n$.

Case 2 ($S \leqq n/4$). By Lemma 4 and the corollary to Lemma 1,

$$(S+1)T \geqq (n^2 - n - S)(m - S) \geqq n^3/4 - 21n^2/16 + 5n/4. \qquad \square$$

The main result of this section follows easily from Lemma 4:

THEOREM 1. *Let* G *be any Boolean circuit which computes the transitive closure of an* $n \times n$ *matrix* A *by computing* $(A \vee I)$, $(A \vee I)^2$, $(A \vee I)^4, \cdots, (A \vee I)^{2^{\lceil \log_2 (n-1) \rceil}}$,

iteratively using any subcircuit which computes the function sq. Then pebbling G with S pebbles requires time

$$T \geqq [(n-4S-3)/(2S+2)]^{\log_2 n}.$$

Proof. The argument is similar to one used by Paul and Tarjan [1978, Lemma 4]. Let $m = \lfloor (n-1)/2 \rfloor$ and $k = \lceil \log_2 (n-1) \rceil$. G is composed of subcircuits $C_1, C_2, \cdots, C_k$, each of which computes sq, and the inputs of C_{i+1} are the outputs of C_i. Using induction on i, it is straightforward to show that pebbling any $S+1$ outputs of C_{i+1}, beginning with any configuration of the S pebbles on G, requires $\lfloor (m-S)/(S+1) \rfloor^i (m-S)$ moves in which pebbles are placed on inputs of G:

Basis ($i=0$): follows directly from Lemmas 1 and 4.

Induction: In the course of pebbling any $S+1$ outputs of C_{i+1}, Lemmas 1 and 4 show that $m-S$ outputs of C_i must be pebbled. Applying the induction hypothesis to sets of $S+1$ of these as they are pebbled yields the claimed result.

Thus, the total number of times inputs of G are pebbled is

$$\begin{aligned} T &\geqq \lfloor (m-S)/(S+1) \rfloor^{k-1} (m-S) \lfloor (n^2-n)/(S+1) \rfloor \\ &\geqq \lfloor (\lfloor (n-1)/2 \rfloor - S)/(S+1) \rfloor^{\lceil \log_2 (n-1) \rceil + 1} \\ &\geqq \lfloor (n-2-2S)/(2S+2) \rfloor^{\log_2 n} \\ &\geqq [(n-4S-3)/(2S+2)]^{\log_2 n}. \end{aligned}$$ □

COROLLARY. *For any circuit satisfying the statement of Theorem 1, if $S = o(n)$ then T exceeds any polynomial in n. In fact, if $S = O(n^{1-\varepsilon})$ for any fixed $\varepsilon > 0$, then $T = 2^{\Omega(\log^2 n)}$.*

3. Warshall's algorithm requires linear space. The result of § 2 prompts the investigation of other transitive closure algorithms. Warshall's algorithm (Warshall [1962]) suggests itself because of its familiarity and its contrast to the successive squaring algorithm. Warshall's algorithm computes the transitive closure A^* of an $n \times n$ Boolean matrix A as follows:

$$C^0_{ij} \leftarrow (I \vee A)_{ij} \textbf{ for all } 1 \leqq i, j \leqq n;$$

$$\textbf{for } k \textbf{ from } 1 \textbf{ to } n \textbf{ do}$$

$$C^k_{ij} \leftarrow C^{k-1}_{ij} \vee (C^{k-1}_{ik} \,\&\, C^{k-1}_{kj}) \textbf{ for all } 1 \leqq i, j \leqq n;$$

A^*_{ij} is then given by C^n_{ij}, for all $1 \leqq i, j \leqq n$. The circuit corresponding to this algorithm has size $O(n^3)$ and depth n, and each internal vertex has indegree 3.

The main result of this section is to show that $n-1$ pebbles are necessary to pebble this circuit. (Notice that, unlike the discussion in § 2, this section deals with pebblings of a single circuit, rather than any of a family of circuits.) The method is due to Cook [1974], who showed a lower bound on the number of pebbles required to pebble a certain "pyramid" graph. The circuit corresponding to Warshall's algorithm contains a similar subcircuit, which will be called a *Warshall pyramid* and is defined recursively:

(a) Warshall pyramid P^0_{ij} consists of a single vertex labelled C^0_{ij}.

(b) Warshall pyramid P^k_{ij} consists of pyramids P^{k-1}_{ik} and P^{k-1}_{kj}, whose identically labelled vertices have been identified, together with new edges from C^{k-1}_{ik} and C^{k-1}_{kj} to a new vertex labelled C^k_{ij}.

Warshall pyramid P^4_{57} is shown in Fig. 1.

LEMMA 5. *If $k < i$ and $k < j$, then $k+1$ pebbles are required to pebble P^k_{ij}.*

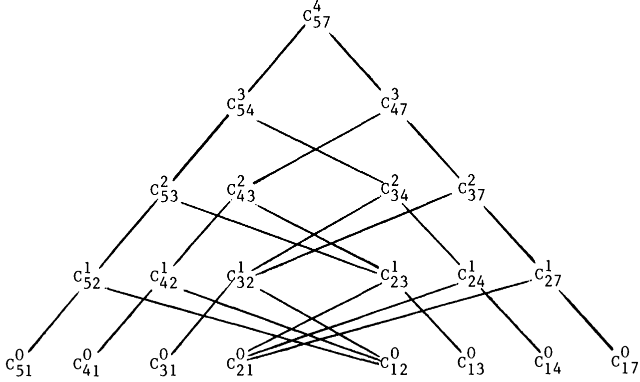


FIG. 1. *Warshall pyramid* P^4_{57}.

Proof. The argument is similar to one given by Cook [1974, Thm. 5]. The pebbling begins with all paths from inputs to C^k_{ij} pebble-free, and ends with no paths from inputs to C^k_{ij} pebble-free. Consider the first step after which this latter condition is attained; some input v must have been pebbled in this step, closing an otherwise pebble-free path p from v to C^k_{ij}, and every other path from any input to C^k_{ij} contains a pebble. The path p contains one of C^{k-1}_{ik} or C^{k-1}_{kj}, say C^{k-1}_{ik}. Notice that the (unique) path from C^{k-1}_{kj} to C^0_{1j} contains at least one pebble, and is disjoint from P^{k-1}_{ik} since $j > k$. Continuing inductively, P^{k-1}_{ik} contains k additional pebbles. □

THEOREM 2. *Any straight-line program which implements Warshall's algorithm for $n \times n$ transitive closure requires space $n-1$.*

Proof. The circuit corresponding to Warshall's algorithm contains $P^{n-2}_{n,n-1}$ whose pebbling, by Lemma 5, requires $n-1$ pebbles. □

Notice that Theorem 2 is optimal to within a constant factor, since Warshall's circuit can be pebbled using $2n+1$ pebbles.

It should be remarked that Theorem 2 applies as well to Floyd's shortest-paths algorithm and Kleene's algorithm for converting finite automata to regular expressions. (See Aho, Hopcroft, and Ullman [1974, § 5.6] for the appropriate generalization of Warshall's algorithm.)

4. Ramifications of the generalization of these results. Sections 2 and 3 analyzed the time and space requirements of two common transitive closure algorithms, and found that neither admits a polynomial time, sublinear space implementation. The obvious direction for further research is to attempt to generalize these results to broader classes of algorithms. However, demonstrating that no Boolean straight-line program computes transitive closure in polynomial time and small space simultaneously will prove to be as difficult as some of the more "classical" open problems of complexity. For instance, if no circuit which computes a function f can be pebbled with $O(\log n)$ pebbles in polynomial time, then no circuit which computes f has $O(\log n)$ depth. In the case of transitive closure, f would be a function computable in polynomial time but provably not in logarithmic Boolean depth. Such a result would be a breakthrough in complexity theory.

Cook [personal communication] and Pippenger [1979] made a more important observation about the ramifications of demonstrating that no Boolean straight-line program computes transitive closure in polynomial time and small space simultaneously: namely, if no Boolean straight-line program computes a function f in polynomial time and $(\log n)^{O(1)}$ space simultaneously, then neither does any Turing

machine. In particular, if f can be computed in polynomial time, as in the case of transitive closure, then polynomial time would have been proved more powerful than logarithmic space. Their result is obtained by simulating Turing machines by straight-line programs as follows: Let a *semioblivious* Turing machine be one with two tapes, one of which is a read-only, oblivious input tape and the other a read-write non-oblivious work tape. For simplicity, assume in what follows that S is a function of n which is $\Omega(\log n)$ and $O(n)$. Then a multitape Turing machine using time T and space S can be simulated by a semioblivious Turing machine using time $O(nT)$ and space $O(S)$, by recording the original input head position on a work tape track. This machine can be simulated in turn by a Boolean circuit of depth $O(nT)$ and "width" $O(S)$, using a well-known construction (see, for example, Ladner [1975]). Such a circuit can be pebbled in $O(nST)$ steps using $O(S)$ pebbles.

Lipton [personal communication] has observed that this simulation can be carried out even if the Turing machine is probabilistic, by generalizing a result of Adleman [1978]. (Consult that reference for a discussion of probabilistic Turing machines and their simulation by circuits.) The simulating Boolean circuits are deterministic but nonuniform (i.e., there is no efficient algorithm which, given n, constructs the nth circuit), have width $O(S)$, and depth $O(n^2T)$. As a relevant application, Lipton pointed out that *symmetric* transitive closure can be computed by (nonuniform) Boolean straight-line programs which use only $O(\log n)$ space and polynomial time, by the main result of Aleliunas et al. [1979].

5. Sorting algorithms which exhibit similar behavior. This section shows that the behaviors of transitive closure algorithms described in §§ 2 and 3 are also exhibited by certain familiar sorting algorithms. The main result is that any straight-line program which executes a recursive merge-sort using only the binary operators max and min requires time $2^{\Omega(\log^2 n)}$ if the space is restricted to $n^{1-\varepsilon}$, independent of the merging subprocedure chosen. Examples of such sorting algorithms include Batcher's odd-even merge-sort, Batcher's bitonic merge-sort, and Stone's perfect-shuffle sort (see Knuth [1973] for descriptions). The proof of this result is similar to that of Theorem 1, but Lemmas 1 and 4 are replaced by

LEMMA 6. *Let G be any max-min circuit which merges two sorted lists of length m. Then in the course of pebbling any $2S$ outputs in the middle third of the outputs of G, starting from any configuration of S pebbles on G, at least $(m-6S-1)/6$ inputs in the middle thirds of each of the input lists must be pebbled.*

Proof. The argument is similar to one given in Tompa [1980, Thm. 2]. Let Y be any set of $2S$ outputs in the middle third of the outputs of G, and consider a partition of the middle third of either input list into blocks X_i each consisting of $2S$ *consecutive* inputs. Since there is an assignment of distinct values to the $2m$ inputs which causes the $2S$ inputs in X_i to end at the output positions in Y, there must be $2S$ vertex-disjoint paths from each X_i to Y, of which at least S must be pebble-free initially. Hence to pebble the $2S$ outputs in Y, at least

$$\lfloor \lfloor m/3 \rfloor /2S \rfloor S \geqq \lfloor (m-2)/6S \rfloor S \geqq (m-6S-1)/6$$

of the middle third inputs must be pebbled. □

THEOREM 3. *Let G be any max-min circuit which sorts n inputs by recursively sorting the first and second halves, and merging the resulting sorted lists. Then pebbling G with S pebbles requires time*

$$T \geqq 2^{(\log_2 n - \log_2 S - 6)^2/2}.$$

Proof. Let $k=\lfloor \log_2 n-\log_2 S-6\rfloor$. As in Theorem 1, repeated application of Lemma 6 reveals that the number of pebble placements at level k of G is at least

$$
\begin{aligned}
&\left\lfloor \frac{\lfloor n/3\rfloor}{2S}\right\rfloor \cdot \left\lfloor \frac{(n/2-6S-1)/6}{2S}\right\rfloor \cdot \left\lfloor \frac{(n/4-6S-1)/6}{2S}\right\rfloor \cdots (n/2^k-6S-1)/6\\
&\qquad \geqq \frac{n-18S}{12S}\cdot \frac{n/2-18S}{12S}\cdot \frac{n/4-18S}{12S}\cdots \frac{n/2^k-18S}{12S}\\
&\qquad \geqq (n/24S)^{k+1}/2^{k(k+1)/2}\\
&\qquad \geqq 2^{(k+1)^2/2}. \qquad \square
\end{aligned}
$$

COROLLARY. *For the circuit of Theorem* 3, *if* $S=O(n^{1-\varepsilon})$ *for any fixed* $\varepsilon>0$, *then* $T=2^{\Omega(\log^2 n)}$.

It should be noted that Theorem 3 applies to a generalization of max-min circuits called "ordering networks" by Pippenger and Valiant [1976].

Other max-min sorting algorithms mentioned in Knuth [1973], namely straight insertion, bubble sort, and the odd-even transposition sort, are readily seen to require $\Omega(n)$ space, using the same technique as in § 3.

Acknowledgments. I am grateful to Allan Borodin, Mike Fischer, Richard Ladner, and Larry Ruzzo for enjoyable and fruitful discussions concerning this material.

REFERENCES

H. ABELSON [1979], *A note on time-space tradeoffs for computing continuous functions*, Inform. Process. Lett., 8, pp. 215–217.

L. ADLEMAN [1978], *Two theorems on random polynomial time*, Proc. 19th IEEE Symposium on Foundations of Computer Science, October 1978, pp. 75–83.

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

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

A. BORODIN AND S. A. COOK [1980], *A time-space tradeoff for sorting on a general sequential model of computation*, Proc. 12th ACM Symposium on Theory of Computing, April 1980, pp. 294–301.

A. BORODIN, M. J. FISCHER, D. G. KIRKPATRICK, N. A. LYNCH AND M. TOMPA [1979], *A time-space tradeoff for sorting on non-oblivious machines*, Proc. 20th IEEE Symposium on Foundations of Computer Science, October 1979, pp. 319–327.

D. A. CARLSON AND J. E. SAVAGE [1980], *Graph pebbling with many free pebbles can be difficult*, Proc. 12th ACM Symposium on Theory of Computing, April 1980, pp. 326–332.

S. A. COOK [1974], *An observation on time-storage trade off*, J. Comput. System Sci., 9, pp. 308–316.

——— [1979], *Deterministic CFL's are accepted simultaneously in polynomial time and log squared space*, Proc. 11th ACM Symposium on Theory of Computing, April–May 1979, pp. 338–345.

D. YU. GRIGORYEV [1976], *An application of separability and independence notions for proving lower bounds of circuit complexity*, Notes of Scientific Seminars, 60, Steklov Mathematical Institute, Leningrad Department, pp. 38–48 (in Russian).

J. JA' JA' [1980], *Time-space tradeoffs for some algebraic problems*, Proc. 12th Annual ACM Symposium on Theory of Computing, April 1980, pp. 339–350.

D. E. KNUTH [1973], *The Art of Computer Programming: Sorting and Searching*, vol. 3, Addison-Wesley, Reading, MA.

R. E. LADNER [1975], *The circuit value problem is log space complete for P*, SIGACT News, 7, pp. 18–20.

T. LENGAUER AND R. E. TARJAN [1979], *Upper and lower bounds on time-space tradeoffs*, Proc. 11th ACM Symposium on Theory of Computing, April–May 1979, pp. 262–277.

A. LINGAS [1978], *A PSPACE-complete problem related to a pebble game*, in Automata, Languages and Programming, Lecture Notes in Computer Science 62, Springer-Verlag, Berlin, pp. 300–321.

C. L. LIU [1968], *Introduction to Combinatorial Mathematics*, McGraw-Hill, New York.

J. I. MUNRO AND M. S. PATERSON [1978], *Selection and sorting with limited storage*, Proc. 19th IEEE Symposium on Foundations of Computer Science, October 1978, pp. 253–258.

W. J. PAUL AND R. E. TARJAN [1978], *Time-space trade-offs in a pebble game*, Acta Informat., 10, pp. 111–115.

N. PIPPENGER [1979], *On simultaneous resource bounds*, Proc. 20th IEEE Symposium on Foundations of Computer Science, October 1979, pp. 307–311.

——— [1980], *Pebbling*, preprint, IBM Thomas J. Watson Research Center, Yorktown Heights, NY.

N. PIPPENGER AND L. G. VALIANT [1976], *Shifting graphs and their applications*, J. Assoc. Comput. Mach., 23, pp. 423–432.

J. E. SAVAGE AND S. SWAMY [1978], *Space-time tradeoffs on the FFT algorithm*, IEEE Trans. Inform. Theory, IT-24, pp. 563–568.

——— [1979], *Space-time tradeoffs for oblivious integer multiplication*, in Automata, Languages and Programming, Lecture Notes in Computer Science 71, Springer-Verlag, Berlin, pp. 498–504.

M. TOMPA [1980], *Time-space tradeoffs for computing functions, using connectivity properties of their circuits*, J. Comput. System Sci., 20, pp. 118–132.

P. VAN EMDE BOAS AND J. VAN LEEUWEN [1978], *Move rules and trade-offs in the pebble game*, University of Utrecht Technical Report RUU-CS-78-4.

S. WARSHALL [1962], *A theorem on Boolean matrices*, J. Assoc. Comput. Mach., 9, pp. 11–12.

A. C.-C. YAO [1979], *On the time-space tradeoff for sorting with linear queries*, preprint, Stanford University.

SIAM J. COMPUT.
Vol. 11, No. 1, February 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1101-0011 $01.00/0

MINIMUM VARIANCE HUFFMAN CODES*

LAWRENCE T. KOU†

Abstract. Huffman's well-known coding method constructs a minimum redundancy code which minimizes the expected value of the word length. In this paper, we characterize the minimum redundancy code with the minimum variance of the word length. An algorithm is given to construct such a code. It is shown that the code is in a certain sense unique. Furthermore, the code is also shown to have a strong property in that it minimizes a general class of functions of the minimum redundancy codes as long as the functions are nondecreasing with respect to the path lengths from the root to the internal nodes of the corresponding decoding trees.

Key words. Huffman code, redundancy, mean, variance, decoding tree, weights at leaves, weights at internal nodes

1. Introduction. Let $\mathcal{S}=\{s_1, s_2, \cdots, s_q\}$ be the set of $q \geqq 1$ source symbols composed in the messages to be sent, $\mathcal{C}=\{c_1, c_2, \cdots, c_r\}$ be the set of $r \geqq 2$ code letters used in sending the enclosed messages and $p: \mathcal{S} \to \mathcal{R}$ be the discrete probability density function on $\mathcal{S}$, where $\mathcal{R}$ is the set of real numbers and for each $s_i \in \mathcal{S}$, $p(s_i)$ gives the probability that the source symbol s_i appears in the messages. For every coding scheme, we have the set $\mathcal{W}=\{w_i | w_i$ is the string of letters in $\mathcal{C}$ encoded for the source symbol $s_i\}$. We shall call w_i a word and $\mathcal{W}$ the code for $\mathcal{S}$ in terms of $\mathcal{C}$. $\mathcal{W}$ is an instantaneous code if and only if for every pair of distinct words in $\mathcal{W}$, neither is a prefix of the other. We shall be interested only in the instantaneous codes in this paper. Given $\mathcal{S}$, $\mathcal{C}$ and p, each encoding scheme defines the word length, or more precisely, the random point $l: \mathcal{S} \to \mathcal{I}$, where s is the set of nonnegative integers and, for every $s_i \in \mathcal{S}$, $l(s_i)$ is the length of w_i, i.e., the number of letters in w_i. We shall adopt the notion of the null word, Λ, as the word of no letter and define the length of Λ to be zero. The induced discrete probability density function for l, $p_l: \mathcal{I} \to \mathcal{R}$, is defined such that, for every $k \in \mathcal{I}$, $p_l(k)=\sum_{s_i \in \mathcal{S}_k} p(s_i)$, where $\mathcal{S}_k=\{s_i | s_i \in \mathcal{S}$ and $l(s_i)=k\}$. With respect to the given $\mathcal{S}$, $\mathcal{C}$ and p, the instantaneous code with the minimum expected value of l is not necessarily unique as can be illustrated by the following example.

Example 1. Let $\mathcal{S}=\{s_1, s_2, \cdots, s_9\}$, $\mathcal{C}=\{c_1, c_2, c_3\}$, $p(s_1)=p(s_2)=p(s_3)=\frac{1}{30}$, $p(s_4)=p(s_5)=p(s_6)=p(s_7)=\frac{2}{30}$, $p(s_8)=\frac{6}{30}$ and $p(s_9)=\frac{13}{30}$. Consider the following two instantaneous codes.

$$\mathcal{W}=\{w_1=c_1c_1c_1,\ w_2=c_1c_1c_2,\ w_3=c_1c_1c_3,\ w_4=c_1c_2c_1,\ w_5=c_1c_2c_2,$$
$$w_6=c_1c_2c_3,\ w_7=c_1c_3,\ w_8=c_2,\ w_9=c_3\},$$
$$\mathcal{W}'=\{w'_1=c_1c_1c_1,\ w'_2=c_1c_1c_2,\ w'_3=c_1c_1c_3,\ w'_4=c_1c_2,\ w'_5=c_1c_3,$$
$$w'_6=c_2c_1,\ w'_7=c_2c_2,\ w'_8=c_2c_3,\ w'_9=c_3\}.$$

Both codes have expected word length equal to $\frac{5}{3}$, which, in fact, is the minimum value for the given inputs.

It is convenient to introduce the construction of the decoding tree for a coding scheme. The following example is used to demonstrate several terminologies in connection with the decoding tree for an instantaneous code.

Example 2. Let $\mathcal{S}$, $\mathcal{C}$, p, $\mathcal{W}$ and $\mathcal{W}'$ be the same as in Example 1. The corresponding decoding trees are shown in Fig. 1. Every decoding tree consists of two types of

* Received by the editors June 30, 1980, and in revised form April 15, 1981.

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

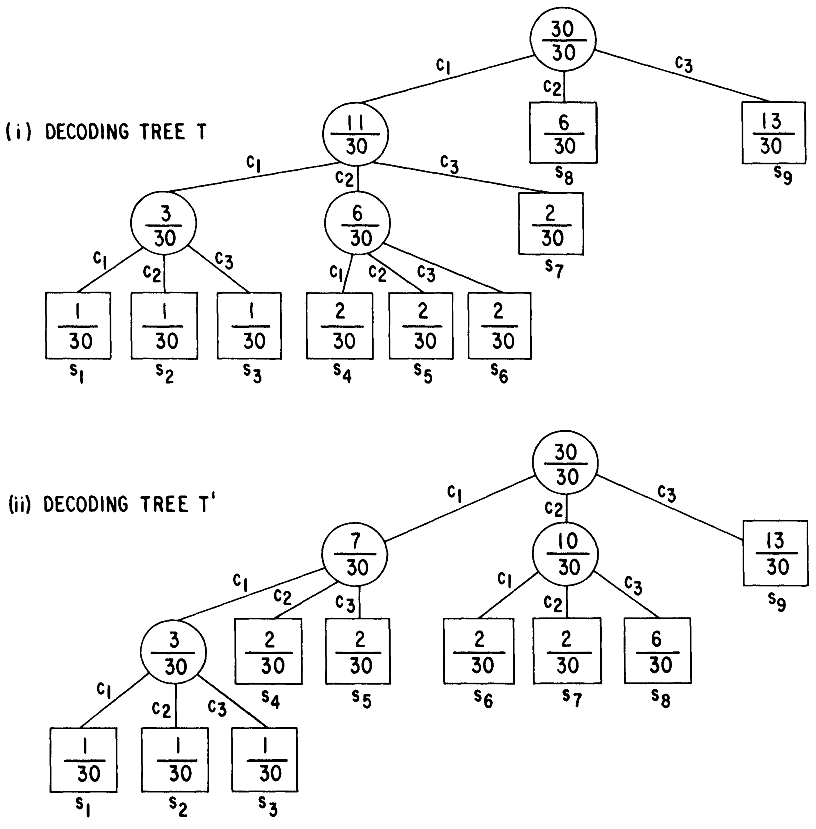


FIG. 1

nodes, the leaves (the square-shaped nodes) and the internal nodes (circular nodes). Written next to each leaf is the corresponding source symbol. Each node, leaf or internal node, is labeled with a weight. The weight at the leaf associated with source symbol s_i is the probability $p(s_i)$. The weight at an internal node is the sum of the weights of its sons, or equivalently, the sum of the weights labeled at the leaves of the subtree rooted at that particular internal node. Edges of the tree are systematically labelled with the letters in $\mathscr{C}$. The path from the root to the leaf associated with the source symbol s_i specifies the encoded word w_i and the number of edges on the path equals $l(s_i)$. The weighted path length [2] of a decoding tree, $\sum_{s_i \in \mathscr{S}} p(s_i) l(s_i)$, equals to $\sum_{k \in \mathscr{I}} p_l(k)k$ which, by definition, is the mean of the corresponding random point l. For both decoding trees in this example, each of the weighted path lengths is equal to $\frac{5}{3}$. However, the corresponding variances differ. For the decoding tree T, the corresponding variance is $\frac{111}{135}$ while for the decoding tree T', the corresponding variance is $\frac{57}{135}$.

Let T be a decoding tree for the code $\mathscr{W}$ with respect to $\mathscr{S}$, $\mathscr{C}$ and p. The following terminologies associated with T are used in this paper.

DEFINITION 1.

$\text{MEAN}(T) \triangleq$ mean of the random point l associated with T;

$\text{VAR}(T) \triangleq$ variance of the random point l associated with T;

$WL(T) \triangleq ((p(s_{i_1}), l(s_{i_1})), (p(s_{i_2}), l(s_{i_2})), \cdots, (p(s_{i_q}), l(s_{i_q})))$, where $\bigcup_{k=1}^{q} \{s_{i_k}\} = \mathcal{S}$ and for all $k = 1, 2, \cdots, (q-1)$, $(p(s_{i_k}), l(s_{i_k}))$ is lexicographically equal to or smaller than $(p(s_{i_{k+1}}), l(s_{i_{k+1}}))$.

$WL(T)^+ \triangleq ((p(s_{j_1}), l(s_{j_1})), (p(s_{j_2}), l(s_{j_2})), \cdots, (p(s_{j_g}), l(s_{j_g})))$, where $g \leqq q = |\mathcal{S}|$, $\bigcup_{k=1}^{g} \{s_{j_k}\} = \{s_i | s_i \in \mathcal{S}$ and $p(s_i) \neq 0\}$ and for all $k = 1, 2, \cdots, (g-1)$, $(p(s_{j_k}), l(s_{j_k}))$ is lexicographically equal to or smaller than $(p(s_{j_{k+1}}), l(s_{j_{k+1}}))$.

For any tree T with weighted nodes, we define the following.

DEFINITION 2.

$L(T) \triangleq$ the set of the leaves of T;

$I(T) \triangleq$ the set of the internal nodes of T;

$W_L(\mathrm{T}) \triangleq$ sum of weights labeled at the leaves of T;

$W_I(T) \triangleq$ sum of weights labeled at the internal nodes of T;

$WPL(T) \triangleq$ weighted path length of T.

We shall often use the following convention in this paper for naming the nodes and subtrees of a decoding tree T. Node i in T is the ith node counting, level by level, from the top level to the bottom level and, for each level, from the leftmost node to the rightmost node. The notation $T(i)$ is used for the subtree of T with node i as its root. Thus, for the decoding tree T' in Fig. 1, node 5 has weight $\frac{3}{30}$ and $T'(5)$ is the subtree of T' that has node 5 as its root which has node 11, node 12, and node 13 as the three sons labeled with source symbols s_1, s_2 and s_3 respectively.

Since there is a one-to-one correspondence between the set of instantaneous codes and the set of their decoding trees, the outputs of the coding algorithms discussed in this paper are chosen, for convenience, to be the decoding trees instead of the actual codes they decode. With respect to $\mathcal{S}$, $\mathcal{C}$ and p, Huffman's well-known algorithm [1] constructs a decoding tree for a minimum redundancy code and in terms of the word length, l, Huffman code minimizes the mean of l. To describe Huffman's algorithm, we start with the family $\mathcal{F}$ of $q = |\mathcal{S}|$ weighted trees, $\bigcup_{i=1}^{q} \{T_i\}$, where T_i contains just a single node labeled with the source symbol s_i and the weight $p(s_i)$. For any weighted tree X, let us temporarily consider $W_L(X)$ as the weight of X. Huffman's algorithm is, in short, a procedure that repeatedly merges the $r = |\mathcal{C}|$ trees in $\mathcal{F}$ that have the r smallest weights until the whole family $\mathcal{F}$ reduces to a singleton set which then contains the decoding tree corresponding to a minimum redundancy code. We shall show in this paper that if we consider the vector $(W_L(X), W_I(X))$ as the weight of the tree X, then repeatedly merging the r trees in $\mathcal{F}$ that have the r lexicographically smallest weights will produce a decoding tree that corresponds to the minimum redundancy code with the minimum variance of the word length.

The main body of this paper is divided into three sections. In § 2, we restate the Huffman algorithm and show that although there are minimum redundancy codes that are not producible via Huffman's algorithm, it is sufficient to consider those codes produced by Huffman's algorithm in searching for the minimum redundancy code with the minimum variance. In § 3, we give a characterization for the minimum redundancy codes that have the minimum variances. We also show that, with respect to any given input, $\mathcal{S}$, $\mathcal{C}$ and p, if U and V are the decoding trees for two distinct minimum variance minimum redundancy codes respectively, then $WL(U)^+ = WL(V)^+$.

In other words, the minimum redundancy code with the minimum variance is unique in the sense that it will produce a unique lexicographically sorted list of weight-length pairs of those words of the code that have nonzero probabilities. In § 4, we give an algorithm to produce the minimum variance minimum redundancy code. We also show a strong property of the code in that it minimizes a general class of functions of the minimum redundancy codes as long as the functions are nondecreasing with respect to the path lengths from the roots to the internal nodes of the corresponding decoding trees.

2. Characterizations of Huffman trees. We now restate Huffman's algorithm. Without loss of generality, we shall introduce source symbols each with zero probability, to make $|\mathcal{S}| \bmod (|\mathcal{C}|-1)=1$ so that every internal node in the output decoding tree will have $|\mathcal{C}|$ sons [2].

ALGORITHM H.

INPUT: a set of source symbols, $\mathcal{S}=\{s_1, s_2, \cdots, s_q\}$; a set of code letters $\mathcal{C}=\{c_1, c_2, \cdots, c_r\}$; the discrete probability density function on $\mathcal{S}$, p: $\mathcal{S} \to \mathcal{R}$.

OUTPUT: a decoding tree corresponding to an instantaneous code with the minimum expected value of the word length.

Step 0. Initialize $\mathcal{F}$ to be an empty set.

Step 1. For $i=1,2,\cdots,q$, set $\mathcal{F} \leftarrow \mathcal{F} \cup \{T_i\}$, where T_i is the weighted tree of a single node associated with the source symbol s_i and labeled with weight $p(s_i)$.

Step 2. Repeat Step 3 and Step 4 until $|\mathcal{F}|=1$.

Step 3. Choose r elements in $\mathcal{F}$, $T_{u_1}, T_{u_2}, \cdots, T_{u_r}$, such that for any $T_k \in \mathcal{F}$ other than these r chosen weighted trees, $W_L(T_{u_t}) \leqq W_L(T_k)$ for all $t=1,2,\cdots,r$.

Step 4. Replace $T_{u_1}, T_{u_2}, \cdots, T_{u_r}$ in $\mathcal{F}$ by a single weighted tree X in which $T_{u_1}, T_{u_2}, \cdots, T_{u_r}$ are the r subtrees of the root and the root is labeled with the weight $\sum_{t=1}^{r} W_L(T_{u_t})$. The edges connecting the root to the r subtrees are systematically labeled with letters $c_1, c_2, \cdots, c_r$ respectively.

Step 5. Output the decoding tree in $\mathcal{F}$.

Notice that in Step 3 there would be more than one qualified choice of $T_{u_1}, T_{u_2}, \cdots, T_{u_r}$. It is understood that one would make a random selection among all qualified choices. Therefore, strictly speaking, Algorithm H is nondeterministic in nature.

DEFINITION 3.

$\mathcal{H}(\mathcal{S}, \mathcal{C}, p) \triangleq$ the set of all decoding trees, with respect to the input $\mathcal{S}$, $\mathcal{C}$ and p, each of which is producible by Algorithm H via certain selections of the r elements, $T_{u_1}, T_{u_2}, \cdots, T_{u_r}$, in Step 3.

It is conventional to call the decoding trees in $\mathcal{H}(\mathcal{S}, \mathcal{C}, p)$ the Huffman trees and the corresponding codes Huffman codes. The following are two fundamental facts about Huffman trees and/or decoding trees in general.

THEOREM 1. (Knuth [2]). *Let T be a decoding tree. Then* $\text{MEAN}(T)=WPL(T)=W_I(T)$.

THEOREM 2. (Glassey and Karp [4]). *$T \in \mathcal{H}(\mathcal{S}, \mathcal{C}, p)$ if and only if, for every nondecreasing concave function f: $[0,\infty) \to \mathcal{R}$ and every decoding tree T' corresponding*

to an instantaneous code with respect to $\mathcal{S}$, $\mathcal{C}$ *and* $\not{p}$, *we have* $\sum_{i\in I(T)} f(W_L(T(i))) \leqq \sum_{j\in I(T')} f(W_L(T'(j)))$.

By examining the two decoding trees, T and T', in Example 2, it is obvious that $T \in \mathcal{H}(\mathcal{S}, \mathcal{C}, \not{p})$, $T' \notin \mathcal{H}(\mathcal{S}, \mathcal{C}, \not{p})$ and $\mathrm{MEAN}(T) = \mathrm{MEAN}(T')$. We summarize this fact in the following lemma.

LEMMA 1. *There exist minimum redundancy codes that are not producible via Huffman's algorithm.*

DEFINITION 4.

$\mathcal{A}(\mathcal{S}, \mathcal{C}, \not{p}) \triangleq$ the set of all decoding trees corresponding to the set of all minimum redundancy codes with respect to the input $\mathcal{S}$, $\mathcal{C}$ and $\not{p}$.

In view of Lemma 1, we might suspect whether $\min\{\mathrm{VAR}(U) | U \in \mathcal{H}(\mathcal{S}, \mathcal{C}, \not{p})\}$ is equal to $\min\{\mathrm{VAR}(V) | V \in \mathcal{A}(\mathcal{S}, \mathcal{C}, \not{p})\}$. However, we shall show that, as far as the functions of the weights and lengths of the words are concerned, $\mathcal{H}(\mathcal{S}, \mathcal{C}, \not{p})$ contains sufficiently many trees for our consideration.

THEOREM 3. *Let T be a decoding tree for an instantaneous code with respect to* $\mathcal{S}$, $\mathcal{C}$ *and* $\not{p}$. *Then* $T \in \mathcal{A}(\mathcal{S}, \mathcal{C}, \not{p})$ *if and only if* $WL(T) \in \{WL(V) | V \in \mathcal{H}(\mathcal{S}, \mathcal{C}, \not{p})\}$.

Proof. The "if" part of the theorem is immediate. We shall give an inductive proof for the "only if" part of the theorem. Let m be the number of internal nodes of T. If $m = 0$, both $\mathcal{A}(\mathcal{S}, \mathcal{C}, \not{p})$ and $\mathcal{H}(\mathcal{S}, \mathcal{C}, \not{p})$ contain one and only one element, T. In this case, the "only if" part of the theorem is trivially true. Assume then the statement is true for all $m = 0, 1, 2, \cdots, k$ where $k \geqq 0$. For $n = k+1$, let node i be one of the internal nodes in T that has the longest path length from the root. Interchange some of the r sons of node i with other leaves in T, if necessary, to ensure that the weights labeled at the r sons of node i are the r smallest weights among all weights labeled at the leaves in T. Make the necessary changes of the weights at some of the internal nodes of T which are caused by the foregoing interchanging of leaves. Call this new tree X. Now construct a decoding tree Y from X by replacing $X(i)$ in X by a single leaf labeled with source symbol $s' \notin \mathcal{S}$ and weight $W_L(X(i))$. Let $\mathcal{S}'$ and $\not{p}'$ be the corresponding new set of source symbols and new discrete probability density function respectively. Observe the following.

(i) $T \in \mathcal{A}(\mathcal{S}, \mathcal{C}, \not{p})$ implies that, in producing X from T, only interchanges of leaves at the same level can occur and hence $WL(T) = WL(X)$.

(ii) X minimizes the mean of the corresponding random point l with respect to $\mathcal{S}$, $\mathcal{C}$ and $\not{p}$ if and only if Y minimizes the corresponding random point l' with respect to $\mathcal{S}'$, $\mathcal{C}$ and $\not{p}'$.

(iii) $WL(X) \in \{WL(V) | V \in \mathcal{H}(\mathcal{S}, \mathcal{C}, \not{p})\}$ if and only if $WL(Y) \in \{WL(Z) | Z \in \mathcal{H}(\mathcal{S}', \mathcal{C}, \not{p}')\}$.

The decoding tree Y has k internal nodes. By (i), (ii), (iii) and the induction hypothesis, the proof for the case when T has $k+1$ internal nodes follows. We thus complete our proof. □

Intuitively, one decoding tree in $\mathcal{H}(\mathcal{S}, \mathcal{C}, \not{p})$ can be transformed into another in $\mathcal{H}(\mathcal{S}, \mathcal{C}, \not{p})$ by appropriate subtree interchanges. We formalize this notion in the following.

DEFINITION 5. Let X, Y be two distinct decoding trees. $X \sim Y$ if and only if Y can be constructed from X by interchanging two disjoint subtrees,[1] $X(i)$ and $X(j)$, of X where either node i and node j have the same father or $W_L(X(i)) = W_L(X(j))$.

[1] Two subtrees are disjoint if one is not a subtree of the other.

DEFINITION 6. Let X, Y be two decoding trees. $X * Y$ if and only if there exists a sequence of decoding trees, $Z^1, Z^2, \cdots, Z^t$, for some $t \geqq 1$, such that $X = Z^1$, $Y = Z^t$ and, if $t > 1$, $Z^u \sim Z^{u+1}$ for all $u = 1, 2, \cdots, t-1$.

The following lemma is a direct consequence of Definition 5 and Algorithm H. We shall leave the details of the proof to the readers.

LEMMA 2. *Assume X and Y are two decoding trees such that $X \sim Y$. Then $X \in \mathcal{H}(\mathcal{S}, \mathcal{C}, p)$ if and only if $Y \in \mathcal{H}(\mathcal{S}, \mathcal{C}, p)$.*

It should be clear that $*$ is an equivalence relation over the set of all decoding trees. We shall show that $\mathcal{H}(\mathcal{S}, \mathcal{C}, p)$ is an equivalence block under $*$.

THEOREM 4. *Let $X \in \mathcal{H}(\mathcal{S}, \mathcal{C}, p)$. Then $Y \in \mathcal{H}(\mathcal{S}, \mathcal{C}, p)$ if and only if $X * Y$.*

Proof. The "if" part of the theorem follows immediately from Lemma 2. To prove the "only if" part of the theorem, we shall consider the sequence of subtrees brought into the set $\mathcal{F}$ during the execution of Algorithm H. For any $V \in \mathcal{H}(\mathcal{S}, \mathcal{C}, p)$, where V contains n nodes, let $V^1, V^2, \cdots, V^n$ be the sequence of subtrees, including V itself, in the order of their joining the membership of $\mathcal{F}$ during the execution of Algorithm H. The first $q = |\mathcal{S}|$ subtrees in the sequence correspond to the q leaves in V. We may specify the construction of V^i, $q < i \leqq n$, by identifying the index set $\{i(h) | 1 \leqq i(h) < i \text{ and } h = 1, 2, \cdots, r\}$ such that $V^{i(1)}, V^{i(2)}, \cdots, V^{i(r)}$ are the r subtrees to be selected to form V^i. Let s be the largest index such that, for all $i = 1, 2, \cdots, s$, $X^i = Y^i$. Such an index exists since for $j = 1, 2, \cdots, q$, $X^j = Y^j$. Let $\lambda = n - s$. We shall now proceed with the proof by induction on λ. If $\lambda = 0$, then $s = n$ and $X = X^n = Y^n = Y$ which implies $X * Y$ by the definition of $*$. Now assume the "only if" part of the theorem is true for $\lambda = 0, 1, \cdots, k$. For $\lambda = k+1$, we have $s = n-k-1$. Let $X^{a(1)}$, $X^{a(2)}, \cdots, X^{a(r)}$ be the r subtrees selected in Step 3 of Algorithm H when constructing X^{n-k} and $Y^{b(1)}, Y^{b(2)}, \cdots, Y^{b(r)}$ be the corresponding r subtrees selected when constructing Y^{n-k}. Notice the following.

(i) For all $h = 1, 2, \cdots, r$, $X^{a(h)}$ is a member of the subsequence $X^1, X^2, \cdots, X^{n-k-1}$ and $Y^{b(h)}$ is a member of the subsequence $Y^1, Y^2, \cdots, Y^{n-k-1}$. Since the two subsequences are identical, both $X^{a(h)}$ and $Y^{b(h)}$ are subtrees of X and subtrees of Y, for all $h = 1, 2, \cdots, r$.

(ii) If there exist subtrees E and F such that $E \in (\bigcup_{i=1}^r \{X^{a(i)}\} - \bigcup_{i=1}^r \{Y^{b(i)}\})$ and $F \in (\bigcup_{i=1}^r \{Y^{b(i)}\} - \bigcup_{i=1}^r \{X^{a(i)}\})$, then $W_L(E) = W_L(F)$.

If $(\bigcup_{i=1}^r \{X^{a(i)}\} - \bigcup_{i=1}^r \{Y^{b(i)}\}) \neq \emptyset$, let $\pi: (\bigcup_{i=1}^r \{a(i)\} - \bigcup_{i=1}^r \{b(i)\}) \to (\bigcup_{i=1}^r \{b(i)\} - \bigcup_{i=1}^r \{a(i)\})$ be a bijective mapping. Then (ii) suggests that we can construct a decoding tree $Z \in \mathcal{H}(\mathcal{S}, \mathcal{C}, p)$ where, for $\alpha = 1, 2, \cdots, n-k$, Z^α is constructed by using the index set $\{\alpha(h) | 1 \leqq \alpha(h) < \alpha \text{ and } h = 1, 2, \cdots, r\}$ which is the same as the index set used for the construction of X^α and, for $\beta = n-k+1, n-k+2, \cdots, n$, Z^β is constructed by using the index set $\{\beta(h) | 1 \leqq \beta(h) < \beta \text{ and } h = 1, 2, \cdots, r\}$ which is the same as the index set used for the construction of Y^β except that if for some j, $1 \leqq j \leqq r$, $\beta(j) \in (\bigcup_{i=1}^r \{a(i)\} - \bigcup_{i=1}^r \{b(i)\})$, we shall replace the index $\beta(j)$ by $\pi(\beta(j))$ during the construction. (i) and (ii) discussed above imply that $Y * Z$. Now applying the induction hypothesis on Z, we have $X * Z$. We conclude that $X * Y$ since $*$ is commutative and transitive. The proof is thus completed. □

3. Minimal variance Huffman codes. In this section, we shall give a characterization for minimum variance Huffman code. We rely on the following lemmas.

LEMMA 3. *Let $T \in \mathcal{H}(\mathcal{S}, \mathcal{C}, p)$, $T(i)$, $T(j)$ be two subtrees of T and α_i, α_j be the path lengths of node i and node j respectively from the root of T. Then* (i) $W_L(T(i)) > W_L(T(j))$ *implies $\alpha_i \leqq \alpha_j$;* (ii) *$T(i)$, $T(j)$ are disjoint and $W_L(T(i)) = W_L(T(j)) \neq 0$ imply $0 \leqq |\alpha_i - \alpha_j| \leqq 2$ where $|\alpha_i - \alpha_j| = 2$ only if $W_I T(i) = W_I T(j) = 0$.*

Proof. (i) Assume on the contrary that we have $W_L(T(i)) > W_L(T(j))$ and $\alpha_i > \alpha_j$. Let $i, i_1, i_2, \cdots, i_{\alpha_i}$ be the sequence of nodes specifying the path from node i to the root of T and $j, j_1, j_2, \cdots j_{\alpha_j}$ be the sequence of nodes specifying the path from node j to the root of T. The assumption that $W_L(T_i) > W_L(T_j)$ implies that $T(i_1)$ is constructed after $T(j_1)$, during the processing of Algorithm H and $W_L(T(i_1)) > W_L(T(j_1))$. The latter in turn implies that $T(i_2)$ is constructed after $T(j_2)$, during the processing of Algorithm H and $W_L(T(i_2)) > W_L(T(j_2))$. It follows that $W_L(T(i_{\alpha_j})) > W_L(T(j_{\alpha_j}))$. However, this contradicts the fact that $W_L(T(j_{\alpha_j})) = W_L(T)$.

(ii) Without loss of generality, let $\alpha_i \geqq \alpha_j$. Assume on the contrary that we have $T(i)$, $T(j)$ being disjoint, $W_L(T(i)) = W_L(T(j)) \neq 0$ and $\alpha_i - \alpha_j \geqq 3$. Let node i_1 be the father of node i and node i_2 be the father of node i_1. It is clear that $W_L(T(i_2)) \geqq W_L(T(i_1)) \geqq W_L(T(i)) \neq 0$. The path length from the root of T to node i_2 is greater than α_j. Part (i) of this lemma then implies $W_L(T(i_2)) \leqq W_L(T(j)) = W_L(T(i))$. Hence, $W_L(T(i)) = W_L(T(i_1)) = W_L(T(i_2)) \neq 0$. This implies that the node i_2 has a son k_2 such that $W_L(T(k_2)) = 0$. Then, according to Algorithm H, each of the r subtrees of node i_1, namely, $T(i_1(h))$, for $h = 1, 2, \cdots, r$, should have weight zero, i.e., $W_L(T(i_1(h)) = 0$. This contradicts the fact that $W_L(T(i_1)) \neq 0$. Therefore, $T(i)$, $T(j)$ are disjoint and $W_L(T(i)) = W_L(T(j)) \neq 0$ implies $0 \leqq |\alpha_i - \alpha_j| \leqq 2$. Now if $|\alpha_i - \alpha_j| = 2$ then the path length from the root of T to the node i_1 is greater than α_j. Using arguments similar to the above, we conclude that $W_L(T(i)) = W_L(T(i_1)) \neq 0$. Thus node i_1 has a son k_1 such that $W_L(T(k_1)) = 0$. If $W_I(T(i)) \neq 0$, i.e., node i is not a leaf, then using arguments similar to those given above, we would have $W_L(T(i)) = 0$, a contradiction. On the other hand, if $W_I(T(j)) \neq 0$, i.e., node j is not a leaf, then there are two cases. In the first case, we assume $T(j)$ is constructed before $T(i_1)$ is. Then $W_L(T(j)) \neq 0$ implies that $T(k_1) > 0$, a contradiction. In the second case, we assume $T(i_1)$ is constructed before $T(j)$ is. Then $W_L(T(i)) \leqq W_L(T(j(h)))$ for all $h = 1, 2, \cdots, r$, where $T(j(h))$ is a subtree of node j. This contradicts the fact that $W_L(T(i_1)) = W_L(T(i)) = W_L(T(j)) \neq 0$. Therefore $|\alpha_i - \alpha_j| = 2$ can take place only if $W_I(T(i)) = W_I(T(j)) = 0$. We thus complete our proof. □

LEMMA 4. *Let $T(i)$, $T(j)$ be two disjoint subtrees of $T \in \mathscr{H}(\mathscr{S}, \mathscr{C}, \not{p})$ such that $W_L(T(i)) = W_L(T(j))$ and α_i, α_j be the path lengths from the root of T to node i and node j respectively. If the decoding tree U is constructed from T by interchanging $T(i)$ and $T(j)$ in T, then* $\mathrm{VAR}(T) - \mathrm{VAR}(U) = 2\delta_{ij}(W_I(T(i)) - W_I(T(j)))$, *where*

$$\delta_{ij} = \begin{cases} +1 & \text{if } \alpha_i > \alpha_j, \\ 0 & \text{if } \alpha_i = \alpha_j, \\ -1 & \text{if } \alpha_i < \alpha_j. \end{cases}$$

Proof. A straightforward evaluation of $\mathrm{VAR}(T)$ and $\mathrm{VAR}(U)$ together with the application of Theorem 1 will yield $\mathrm{VAR}(T) - \mathrm{VAR}(U) = 2(\alpha_i - \alpha_j)(W_I(T(i)) - W_I(T(j)))$. This lemma then follows immediately from Lemma 3. We leave the details to the readers. □

LEMMA 5. *Let $T \in \mathscr{H}(\mathscr{S}, \mathscr{C}, \not{p})$ and* $\mathrm{VAR}(T) = \min\{\mathrm{VAR}(U) | U \in \mathscr{H}(\mathscr{S}, \mathscr{C}, \not{p})\}$. *Then, for all $X \in \mathscr{H}(\mathscr{S}, \mathscr{C}, \not{p})$ either $WL(X)^+ = WL(T)^+$ or there exists a sequence of decoding trees $Z^1, Z^2, \cdots, Z^t$, for some $t > 1$ such that*

(i) *$X = Z^1$ and, for all $h = 1, 2, \cdots, t-1$, $Z^h \sim Z^{h+1}$;*

(ii) *$WL(Z^t)^+ = WL(T)^+$;*

(iii) $\mathrm{VAR}(Z^1) > \mathrm{VAR}(Z^2) > \cdots > \mathrm{VAR}(Z^t)$;

(iv) *for all* $h = 1, 2, \cdots, t-1$, Z^{h+1} *can be constructed from* Z^h *by interchanging two subtrees in* Z^h *where one of the two subtrees consists of a single node only.*

Proof. We first point out that both X and T are members of $\mathcal{H}(\mathcal{S}, \mathcal{C}, p)$ and hence (1) both X and T have the same number of internal nodes and if we list the weights labeled at the internal nodes of X in nondecreasing order, the list would be the same as the corresponding list of the weights labeled at the internal nodes of T; (2) in X (or T), if node i and node j are two internal nodes labeled with the same weights then all the weights labeled at their sons are identical; (3) for each internal node u in T (or X) there exists an internal node v in X (or T) such that the weight labeled at node u is the same as the weight labeled at node v and the nondecreasing order list of the weights labeled at the sons of node u is the same as the nondecreasing order list of the weights labeled at the sons of node v. Recall the indexing scheme for the nodes of a decoding tree as defined in the first section of this paper. Let i be the largest index such that for each node $u = 1, 2, \cdots, i$ in T, there is a one-to-one correspondence, node v, in X where (i) node v is at the same level in X as node u is in T; (ii) node v and node u are labeled with the same weight; (iii) if the weights labeled at node u and v are not zero then node v is an internal node if and only if node u is. Such an i exists since the roots of X and T are in the same level, labeled with the same weight, 1, and one is an internal node if and only if the other is. Let n be the total number of nodes in T (or X) and let $\lambda = n - i$. We shall give a proof for our lemma by induction on λ. If $\lambda = 0$, then all nonzero weighted leaves in one decoding tree have their correspondences in the other decoding tree at the same level. Therefore, $WL(X)^+ = WL(T)^+$ and the lemma is clearly true. Assume the lemma is true for $\lambda = 1, 2, \cdots, k$. For $\lambda = k+1$, we have $i = n - \lambda = n - k - 1$. Let node $n-k$ in T be in the Lth level and is labeled with weight W. From (2) and (3) discussed above, we conclude that the only reason that we can not find a node in X corresponding to the node $n-k$ in T is that, although there are nodes eligible for consideration that satisfy condition (i) and (ii), they violate condition (iii). We claim that node $n-k$ in T is an internal node. For otherwise, node $n-k$ is a leaf in T. By (1) mentioned above, we would have an internal node z in T located at a level lower than the Lth level and labeled with the same weight $W \neq 0$. Interchanging subtrees $T(n-k)$ and $T(z)$ will result in a tree T^1. Theorem 4 and Lemma 4 indicate that $T^1 \in \mathcal{H}(\mathcal{S}, \mathcal{C}, p)$ and $\mathrm{VAR}(T) > \mathrm{VAR}(T^1)$, a contradiction. Now that node $n-k$ in T is an internal node, the implication of (2) and (3) is that there is a leaf y in X located at level L and labeled with the weight $W \neq 0$ and furthermore, by (1), there is an internal node z in X located at a lower level than level L also labeled with weight $W \neq 0$. Interchanging subtrees $X(y)$ and $X(z)$ in X will result in a decoding tree Z^2. Let $Z^1 = X$. Clearly, Z^2 is constructed from Z^1 by interchanging two subtrees where one is a single leaf and, by Lemma 4, $\mathrm{VAR}(Z^1) > \mathrm{VAR}(Z^2)$. Now apply the induction hypothesis on Z^2. It follows immediately that the lemma is true for $\lambda = k+1$. We thus complete our proof. □

We now introduce the main theorem for the characterization of the minimal variance Huffman code.

THEOREM 5. *Let* $T \in \mathcal{H}(\mathcal{S}, \mathcal{C}, p)$. *Then, the following statements are equivalent.*

(i) $\mathrm{VAR}(T) = \min \{\mathrm{VAR}(U) | U \in \mathcal{H}(\mathcal{S}, \mathcal{C}, p)\}$.

(ii) $\mathrm{VAR}(T) = \min \{\mathrm{VAR}(V) | V \in \mathcal{A}(\mathcal{S}, \mathcal{C}, p)\}$.

(iii) *For all decoding trees* $X \sim T$, $\mathrm{VAR}(X) \geqq \mathrm{VAR}(T)$.

(iv) *For all pairs of disjoint subtrees* $T(i)$ *and* $T(j)$ *in* T, $W_L(T(i)) = W_L(T(j))$ *and* $W_I(T(i)) < W_I(T(j))$ *imply* $\alpha_i \geqq \alpha_j$, *where* α_i, α_j *are the path lengths from the root of* T *to node i and node j respectively.*

Proof. (i) *implies* (ii). This is an immediate consequence of Theorem 3.

(ii) *implies* (iii). If $X \sim T$ then $X \in \mathscr{H}(\mathscr{S}, \mathscr{C}, \not{p})$. The proof follows.

(iii) *implies* (iv). This is an immediate consequence of Lemma 4.

(iv) *implies* (i). If (i) is not true then Lemma 5 implies that there exists a decoding tree Z such that $T \sim Z$ and $\mathrm{VAR}(T) > \mathrm{VAR}(Z)$. This implies, by Lemma 4, that (iv) is not true. Hence (iv) implies (i). □

Although a minimal variance Huffman code is not unique in the strict sense, the lexicographically sorted list of weight-length pairs for the leaves with nonzero weights in the minimal variance Huffman tree is unique. We state this formally in the following corollary.

COROLLARY 1. *Let* $X \in \mathscr{A}(\mathscr{S}, \mathscr{C}, \not{p})$, $Y \in \mathscr{A}(\mathscr{S}, \mathscr{C}, \not{p})$ *and* $\mathrm{VAR}(X) = \min\{\mathrm{VAR}(V) | V \in \mathscr{A}(\mathscr{S}, \mathscr{C}, \not{p})\}$. *Then* $\mathrm{VAR}(Y) = \mathrm{VAR}(X)$ *if and only if* $WL(Y)^+ = WL(X)^+$.

Proof. If $WL(Y)^+ = WL(X)^+$, then clearly $\mathrm{VAR}(Y) = \mathrm{VAR}(X)$. On the other hand, if $WL(Y)^+ \neq WL(X)^+$, then Lemma 5 and Theorem 5 imply that $\mathrm{VAR}(Y) \neq \min\{\mathrm{VAR}(V) | V \in \mathscr{A}(\mathscr{S}, \mathscr{C}, \not{p})\} = \mathrm{VAR}(X)$. □

Notice that, given the fixed value of the mean of the random point l, the variance of l is minimized if and only if the second moment of l is minimized. For any weighted tree T, let us call the quantity $\sum_{i \in L(T)} \not{p}(i) l(i)^k$ the kth moment of T, where $k \geqq 0$, $L(T)$ is the set of leaves in T, $\not{p}(i)$ is the weight labeled at node i and $l(i)$ is the path length from the root of T to the node i in T. Algorithm H repeatedly merges the r subtrees with the smallest 0th moment and finally produces a decoding tree with the minimum first moment. Algorithm K (below) repeatedly merges the r subtrees with the smallest 0th moment and breaks ties by choosing the subtrees with the smallest 1st moments and finally produces a decoding tree with the minimum second moment among all decoding trees with the minimum first moment. Corollary 1 implies that if T is the decoding tree produced by Algorithm K with respect to input $\mathscr{S}$, $\mathscr{C}$ and $\not{p}$ then $V \in \mathscr{A}(\mathscr{S}, \mathscr{C}, \not{p})$ has the same second moment as T has implies that V has the same kth moment as T has for all $k > 2$.

4. An algorithm to produce a minimum variance Huffman code. In this section, we shall give a simple algorithm to produce the minimum variance Huffman code.

ALGORITHM K.

INPUT: a set of source symbols, $\mathscr{S} = \{s_1, s_2, \cdots, s_q\}$; a set of letters, $\mathscr{C} = \{c_1, c_2, \cdots, c_r\}$; the discrete probability density function on $\mathscr{S}$, $\not{p}: \mathscr{S} \to \mathscr{R}$.

OUTPUT: a decoding tree corresponding to a minimal variance Huffman code.

Step 0. Initialize $\mathscr{F}$ to be an empty set.

Step 1. For $i = 1, 2, \cdots, q$ set $\mathscr{F} \leftarrow \mathscr{F} \cup \{T_i\}$ where T_i is the weighted tree of a single node associated with the source symbol s_i and labeled with weight $\not{p}(s_i)$.

Step 2. Repeat Step 3 and Step 4 until $|\mathscr{F}| = 1$.

Step 3. Choose r elements in $\mathscr{F}$, $T_{u_1}, T_{u_2}, \cdots, T_{u_r}$, such that for any $T_k \in \mathscr{F}$ other than these r chosen weighted trees, $(W_L(T_{u_t}), W_I(T_{u_t}))$ is lexicographically smaller or equal to $(W_L(T_k), W_I(T_k))$ for all $t = 1, 2, \cdots, r$.

Step 4. Replace $T_{u_1}, T_{u_2}, \cdots, T_{u_r}$ in $\mathscr{F}$ by a single weighted tree X in which $T_{u_1}, T_{u_2}, \cdots, T_{u_r}$ are the r subtrees of the root and the root is labeled with the weight $W_L(X)$. The edges connecting the root to the r subtrees are systematically labeled with letters, $c_1, c_2, \cdots, c_r$ respectively.

Step 5. Output the decoding tree in $\mathscr{F}$.

The lemma stated below gives an important characteristic of Algorithm K. Its proof follows from a similar argument as given for the proof of part (i) of Lemma 3. We state the lemma here without proof.

LEMMA 6. *Let T be a decoding tree produced by Algorithm* K *with inputs $\mathcal{S}$, $\mathcal{C}$ and p, $T(i)$ and $T(j)$ be two subtrees of T and α_i, α_j be the path lengths from the root of T to node i and node j respectively. Then $(W_L(T(i)), W_I(T(i)))$ is lexicographically greater than $(W_L(T(j)), W_I(T(j)))$ implies $\alpha_i \leqq \alpha_j$.*

To conclude this paper, we shall give some final remarks on the correctness of Algorithm K and the functions of the minimum redundancy codes that the corresponding code produced by Algorithm K minimizes. We need a few more definitions.

DEFINITION 7. For all $X \in \mathcal{H}(\mathcal{S}, \mathcal{C}, p)$, the vector of the sorted internal path lengths of X, $IPL(S)$, is defined as

$$IPL(X) \triangleq \begin{cases} \Theta & \text{if } X \text{ has no internal node,} \\ (a_1, a_2, \cdots, a_m) & \text{if } X \text{ has } m \geqq 1 \text{ internal nodes,} \end{cases}$$

where, for all $i = 1, 2, \cdots, m$ a_i is the ith longest path length from the root of X to an internal node of X.

DEFINITION 8. Let $X \in \mathcal{A}(\mathcal{S}, \mathcal{C}, p)$ and $Y \in \mathcal{A}(\mathcal{S}, \mathcal{C}, p)$. Then $IPL(X) < IPL(Y)$ if and only if both X and Y have no internal node or else $IPL(X) = (a_1, a_2, \cdots, a_m)$, $IPL(Y) = (b_1, b_2, \cdots, b_m)$, for some $m \geqq 1$, and $a_i \leqq b_i$ for all $i = 1, 2, \cdots, m$.

DEFINITION 9. A function, $f: \{WL(X)^+ | X \in \mathcal{A}(\mathcal{S}, \mathcal{C}, p)\} \to \mathcal{R}$, is nondecreasing with respect to $<$ if and only if for all $X \in \mathcal{A}(\mathcal{S}, \mathcal{C}, p)$ and $Y \in \mathcal{A}(\mathcal{S}, \mathcal{C}, p)$, $IPL(X) < IPL(Y)$ implies $f(WL(X)^+) \leqq f(WL(Y)^+)$.

We shall show that if T is a decoding tree produced by Algorithm K with respect to input $\mathcal{S}$, $\mathcal{C}$ and p, then $f(WL(T)^+) = \min \{f(WL(X)^+) | X \in \mathcal{A}(\mathcal{S}, \mathcal{C}, p)\}$ for any nondecreasing function $f: \{(WL(X)^+) | X \in \mathcal{A}(\mathcal{S}, \mathcal{C}, p)\} \to \mathcal{R}$ with respect to $<$. We first need the following lemma.

LEMMA 7. *Let $X \in \mathcal{A}(\mathcal{S}, \mathcal{C}, p)$ and $Y \in \mathcal{A}(\mathcal{S}, \mathcal{C}, p)$. Then $WL(X) = WL(Y)$ if and only if $IPL(X) = IPL(Y)$.*

Proof. X and Y have the same number of internal nodes. If $IPL(X) \neq IPL(Y)$, then $IPL(X) \neq \Theta \neq IPL(Y)$. Let s be the smallest index such that the sth component of $IPL(X)$ and the sth component of $IPL(Y)$ are different. Let h be equal to the smaller of the two components. It should be clear that $h > 0$. Let us count the levels of the decoding trees from the first level where the root is down to the bottom level. Thus an internal node with the path length from the root equal to h is situated at the $(h+1)$th level. It follows that at any level h', $1 \leqq h' \leqq h$, both X and Y have the same number of nodes and the same number of internal nodes. Therefore, at level $h+1$, both X and Y have the same number of nodes but one has at least one more internal node than the other one does. This implies $WL(X) \neq WL(Y)$. On the other hand, if $WL(X) \neq WL(Y)$, then both X and Y have at least one internal node. Let $WL(X) = ((p(s_{i_1}), l(s_{i_1})), (p(s_{i_2}), l(s_{i_2})), \cdots, (p(s_{i_q}), l(s_{i_q})))$, and $WL(Y) = ((p(s_{j_1}), l(s_{j_1})), (p(s_{j_2}), l(s_{j_2})), \cdots, (p(s_{j_q}), l(s_{j_q})))$. The elements of $WL(X)$ and $WL(Y)$ are sorted lexicographically. Hence, for all $u = 1, 2, \cdots, q$, $p(s_{i_u}) = p(s_{j_u})$. Since $X \in \mathcal{A}(\mathcal{S}, \mathcal{C}, p)$ and $Y \in \mathcal{A}(\mathcal{S}, \mathcal{C}, p)$, we must have $l(s_{i_1}) \geqq l(s_{i_2}) \geqq \cdots \geqq l(s_{i_q})$ and $l(s_{j_1}) \geqq l(s_{j_2}) \geqq \cdots \geqq l(s_{j_q})$. Let k be the largest index such that $(p(s_{i_k}), l(s_{i_k})) \neq (p(s_{j_k}), l(s_{j_k}))$ and let e equal to the smaller of $l(s_{i_k})$ and $l(s_{j_k})$. It is clear that $e > 0$. Thus, at any level e', $1 \leqq e' \leqq e$, both X and Y have the same number of nodes and the same number of leaves. Therefore, at level $e+1$, both X and Y have the same number of nodes but one has at least one more leaf than the other one has. This implies $IPL(X) \neq IPL(Y)$. The proof is thus completed. □

THEOREM 4. *Let T be a decoding tree produced by Algorithm* K *with input* $\mathcal{S}$, $\mathcal{C}$ *and* $\not{p}$. *Then*: (i) *for all* $X \in \mathcal{A}(\mathcal{S}, \mathcal{C}, \not{p})$, $\mathrm{VAR}(T) \leqq \mathrm{VAR}(X)$; (ii) *for all* $X \in \mathcal{A}(\mathcal{S}, \mathcal{C}, \not{p})$, $f(WL(T)^+) \leqq f(WL(X)^+)$, *where* $f: \{WL(X)^+ | X \in \mathcal{A}(\mathcal{S}, \mathcal{C}, \not{p})\} \rightarrow \mathcal{R}$ *is any function that is nondecreasing with respect to* $<$.

Proof. (i) Lemma 4 and Lemma 6 imply that, for all $Y \sim T$, $\mathrm{VAR}(T) \leqq \mathrm{VAR}(Y)$. Part (i) of the theorem then follows immediately from Theorem 5.

(ii) By Theorem 3, for any $X \in \mathcal{A}(\mathcal{S}, \mathcal{C}, \not{p})$, there exists $U \in \mathcal{H}(\mathcal{S}, \mathcal{C}, \not{p})$ such that $WL(U) = WL(X)$. By Lemma 7 we have $IPL(U) = IPL(X)$. It should be clear that $WL(T) \in \{WL(V) | V \in \mathcal{H}(\mathcal{S}, \mathcal{C}, \not{p})\}$. Lemma 5 implies that either $WL(U)^+ = WL(T)^+$, in which case $f(WL(T)^+) = f(WL(U)^+) = f(WL(X)^+)$, or there exists a sequence of decoding trees, $Z^1, Z^2, \cdots, Z^t$, for some $t > 1$ such that (i), (ii), (iii), (iv) of Lemma 5 are satisfied. Lemma 5 together with Lemma 4 then imply that, for all $h = 1, 2, \cdots, t-1$, Z^{h+1} is constructed from Z^h by interchanging a subtree $T(i)$ at a lower level in Z^h with a subtree $T(j)$ at a higher level in Z^h, where $W_L(T(i)) = W_L(T(j))$, $W_I(T(i)) \neq 0$ and $W_i(T(j)) = 0$. ($T(j)$ contains a single node only.) This implies that for all $h = 1, 2, \cdots, t-1$, $IPL(Z^{h+1}) < IPL(Z^h)$. The definition of f implies that $f(WL(T)^+) = f(WL(Z^t)^+) \leqq f(WL(Z^{t-1})^+) \leqq \cdots \leqq f(WL(Z^1)^+) = f(WL(U)^+) = f(WL(X)^+)$. $\square$

Acknowledgment. The author is in debt to Professor Richard Hamming for his encouragement and for his inspiring lecture on coding and information theory where this research work was originated. The author would also like to thank his colleague, George Markowsky, for many technical discussions. Lastly but not leastly, the author is grateful to the referees of this paper for their careful reviews and their pointing out several mistakes in the first draft of this paper.

REFERENCES

[1] D. A. HUFFMAN, *A method for the construction of minimum-redundancy codes*, Proc. IRE, 40 (1952), pp. 1098–1101.

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

[3] T. C. HU AND A. C. TUCKER, *Optimal computer search trees and variable-length alphabetical codes*, SIAM J. Appl. Math., 21 (1971), pp. 514–532.

[4] C. R. GLASSEY AND R. M. KARP, *On the optimality of Huffman trees*, SIAM J. Appl. Math., 31 (1976), pp. 368–372.

[5] M. R. GAREY, *Optimal binary search trees of restricted maximal depth*, this Journal, 3 (1974), pp. 101–110.

[6] J. VAN LEEUWEN, *On the construction of Huffman trees*, Proc. 3rd International Colloq. on Automata, Languages, and Programming, Edinburgh, July 1976, pp. 382–410.

[7] A. ITAI, *Optimal alphabetic trees*, this Journal, 5 (1976), pp. 9–18.

[8] D. S. PARKER, JR., *Conditions for optimality of the Huffman algorithm*, this Journal, 9 (1980), pp. 470–489.

[9] E. S. SCHWARTZ, *An optimal encoding with minimum longest code and total number of digits*, Inform. and Control, 7 (1964), pp. 37–44.

SIAM J. COMPUT.
Vol. 11, No. 1, February 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1101-0012 $01.0/0

POLYGON RETRIEVAL*

DAN E. WILLARD†

Abstract. Given a set of N points on the plane and an arbitrary polygon, we consider how to efficiently find the subset of these points lying inside this polygon. A data structure will be displayed that occupies $O(N)$ space and enables polygon retrieval to be performed in $O(N^{\log_6 4})$ worst-case execution time. This is the best currently known worst-case complexity.

Key words. Multidimensional retrieval, quad tree, K-d tree, augmented tree, K-fold tree, K-range, range tree, super-B-tree

1. Introduction. In this paper, S will denote a set of N points, Q a query requesting some subset of these points, and COUNT(Q) the number of records that are requested by Q. The locate-and-copy runtime of Q will be defined as the amount of execution time needed to find and copy those records specified by Q into the user's workspace. This concept is not especially useful because the degenerate case where COUNT(Q) $= O(N)$ forces most queries to have an $O(N)$ worst-case locate-and-copy runtime. In order to avoid these difficulties, a new criterion for measuring performance will be employed in this paper, called worst-case locate runtime. A query Q will be defined to have an $O[f(N)]$ *worst-case locate runtime* iff the worst conceivable set of cardinality N can be queried in execution time $O(f(N)+\text{COUNT}(Q))$. It is meaningful to measure an algorithm's worst-case locate runtime because this concept has been adjusted to preclude the trivial degeneration that results when COUNT(Q) $= O(N)$.

Notions similar to locate runtime have been used during the last three years to measure worst-case performance in several papers about multidimensional searching [4], [16], [17], [20], [21], [24] although most of these papers did not formally use this term. The phrase "worst-case locate runtime" first appeared in [21] because it seemed desirable to attach a special name to a concept that was repeatedly occurring in different contexts in several articles.

The main concern of most of the previous literature about multidimensional retrieval has been the efficient retrieval of records from orthogonally defined regions such as rectangles, boxes and their K-dimensional analogues of the form

$$a_1 \leqq \text{KEY}.1 \leqq b_1 \wedge a_2 \leqq \text{KEY}.2 \leqq b_2 \wedge \cdots \wedge a_K \leqq \text{KEY}.K \leqq b_K.$$

A variety of different retrieval runtimes have been obtained for this query, depending on the amount of memory that is needed by the associated data structure. The best known asymptotic retrieval times in $O(N)$, $O(N \log^{K-1} N)$ and $O(N^{1+\varepsilon})$ memory space are discussed respectively in [4], [21], and [4]. The data structure of the second of these articles is a modified version of the K-fold trees of [3], [7], [14], [15], [17], [20], [23], [25] which has a $\log N$ better runtime in many practical static and dynamic applications due to the use of one additional type of pointer. Two other interesting data structures are K-d and quad trees of [2], [6], [10], [16], [22], [24]; these data structures do not have quite as good an asymptotic retrieval time as the comparable $O(N)$ memory space data structures of [4]; however, they are attractive because their memory space utilization has a significantly better coefficient. General methods for manipulating the preceding data structures in a dynamic setting are discussed in [5], [14], [15], [18], [20],

* Received by the editors March 30, 1979, and in revised form April 13, 1981.

† Department of Computer Science, University of Iowa. New address, Bell Laboratories, Holmdel, New Jersey 07733.

[21], [22], [23], [24], [25]. Lower bounds on the complexity of dynamic worst-case orthogonal range queries are discussed in [11], [12], [13].

The main omission of the previous multidimensional literature was the study of more complex geometric regions such as, for instance, polygons. It will be shown in this article that polygon retrieval is much more complex than one might first expect and that all of the previous data structures are of no use when optimizing worst-case polygon retrieval time. The data structure of this article will thus be rather significant because it attains an $O(N^{\log_6 4})$ worst-case polygon locate time while occupying only $O(N)$ space. Recently, Fredman [11] has calculated a lower bound on dynamic polygon query time (which is of order $N^{1/3}$ in the notation of this article). The gap between this article's upper bound and Fredman's lower bound remains an open question.

The discussion in this paper is divided into three major parts. Section 2 defines our proposed polygon tree data structure and proves that near-ideal 3-way polygon trees have the claimed $O(N^{\log_6 4})$ worst-case locate runtime. Section 3 explains the significance of this theorem by showing that every set of N points can be represented as a near-ideal polygon tree; it also makes some meaningful comparison between polygon tree and the related quad and K-d tree data structures. Section 4 shows how near-ideal polygon trees can constructed in $O(N \log^2 N)$ expected and $O(N^2)$ worst-case time. Open questions for future research are raised at the end of this paper.

2. The data structure. A set of J lines, $L_1, L_2, \cdots, L_J$ will be said to form a J-way division of the x, y-plane if and only if the following three conditions are satisfied:

(i) L_1 and L_2 must be two distinct straight lines with infinite reach in both directions;

(ii) for $i \geqq 3$, each L_i must be a half line whose starting point is on one of lines $L_2 L_3 \cdots L_{i-1}$ and which lies fully to the right of line L_{i-1};

(iii) the L_1 line must intersect each of the $L_2 L_3 \cdots L_J$ lines.

An example of five lines that form such a partition of the x, y-plane is shown in Fig. 1.

Let Z denote a set of points in the x, y-plane and D a set of lines that form a J-way division of this plane. Note that the lines of division D form boundaries of exactly $2J$

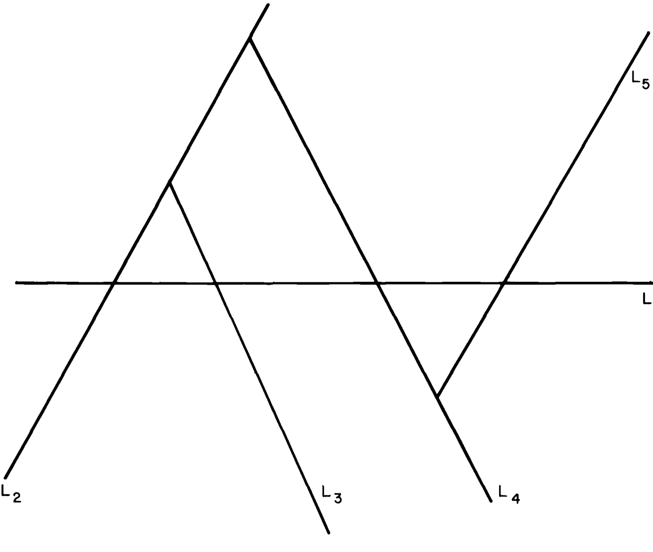


FIG. 1

open regions in the x, y-plane. The symbols $D(Z, 1)$, $D(Z, 2), \cdots, D(Z, 2J)$ will denote the respective subsets of Z that lie in these $2J$ open regions. Also, the symbols $D(Z, 2J+1), D(Z, 2J+2), \cdots, D(Z, 3J)$ will denote the subsets of Z whose points lie on the respective lines of $L_1L_2 \cdots L_J$.

The data structure proposed in this paper will be a tree that uses J-way divisions in the intuitively natural manner to define its underlying structure. This data structure is called a J-way polygon tree. In order to formally define it, we let v denote an internal node of this tree and Z_v the set of leaves descending from v; also assume that every internal node v contains stored information describing a J-way division, which we denote as D_v. *A J-way polygon tree will be said to represent the set Z of points* if and only if:

(1) there exists a one-to-one correspondence between the leaves of this tree and the elements of set Z;
(2) each internal node v will have one son S_i for every $D_v(Z_v, i)$ set that is nonempty;
(3) the set of leaves associated with the subtree rooted at such a son S_i will consist of precisely the $D_v(Z_v, i)$ set.

Every node in a polygon tree will also be said to have a *range*. This will be denoted as R_v and defined to equal:

(1) the whole x, y-plane when v is the root of the polygon tree;
(2) the set $D_f(R_f, i)$ when v is the "ith son" of a node f whose J-way division and ranges are respectively D_f and R_f (the phrase "ith son" was put in quotes to emphasize it means "ith" in the sense of part 2 of the definition of polygon trees).

Note that the range of any internal node v contains the physical positions of all leaves descending from it. A node in a J-way polygon tree will be said to be *linear* if and only if its range consists of a straight line, and *nonlinear* otherwise. Some typical ranges in a 2-way polygon tree are illustrated in Fig. 2.

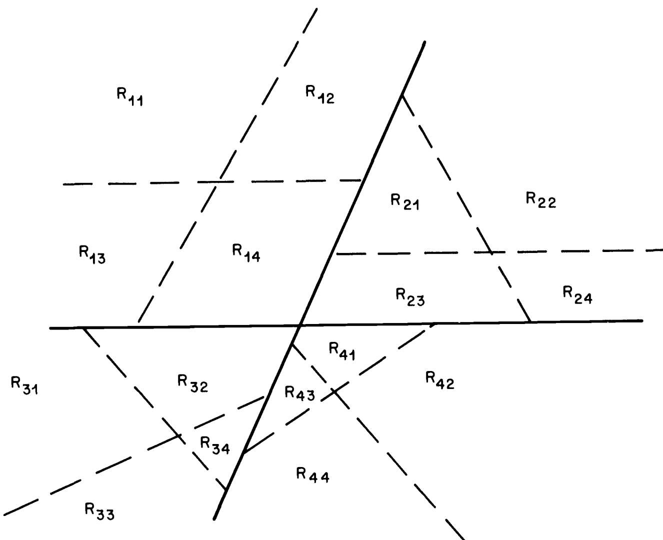


FIG. 2. *Each region R_{ij} represents the range of one of the root's nonlinear grandsons in a 2-way polygon tree. For any i, the union of regions R_{i1}, R_{i2}, R_{i3} and R_{i4} represents the range of the root's ith nonlinear son. The full and dashed lines in Fig. 2 represent, respectively, the ranges of the root's linear sons and grandsons.*

A J-way polygon tree will be said to be *ideal* iff the following two conditions are met:

(1) precisely the first $2J$ sons of each internal node are nonnull elements (note that this set corresponds to the sons with nonlinear range);

(2) all leaves of a polygon tree of height h have a depth equal to exactly h.

An example of an ideal 2-way polygon tree is shown in Fig. 3.

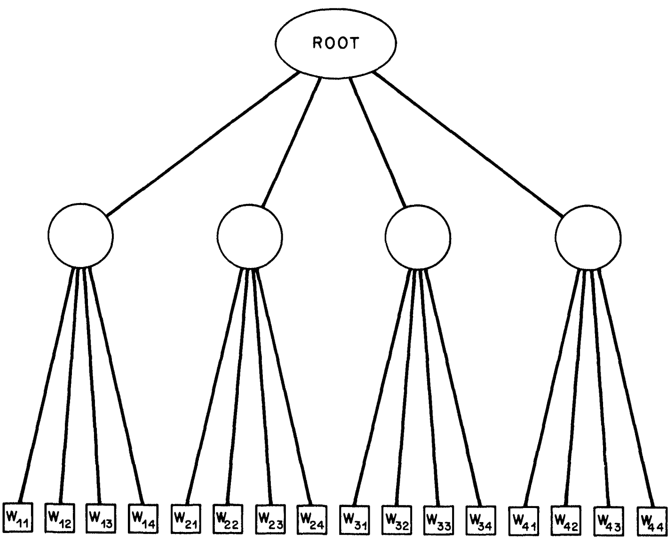


FIG. 3. *Assume that leaf W_{ij} lies in region R_{ij} of Fig. 2. Then the above data structure is an ideal 2-way tree based on the 2-way division of Fig. 2.*

Note that the definition of range in a J-way polygon tree implies $Z_v = Z \cap R_v$. The algorithm for performing retrievals in J-way polygon trees is essentially the natural procedure motivated by this equality and the definition of range. This algorithm will consist of a standard topdown tree walk that visits all nodes whose range overlaps with the region associated with the polygon (or other geometric shape) that is being requested. All leaves thus visited will have their coordinates tested for whether they correspond to a specific point lying in the designated region. Those leaves passing this test will be copied into the user's workspace, since they constitute his requested set of records.

The above algorithm is the polygon tree counterpart of earlier procedures that have performed orthogonal range queries for quad and K-d trees [2], [6], [10], [16], [22], [24]. This paper shows that certain unique structural characteristics of J-way polygon trees lead to more efficient polygon retrieval than is possible with these other data structures.

The following lemma will be related to our runtime analysis.

LEMMA 1. *Every straight line will intersect with the ranges of no more than $O(N^{\log_{2J}(J+1)})$ vertices of an ideal J-way polygon tree with N leaves.*

Proof. Let I_d denote the number of nodes of depth d whose ranges intersect with straight line L. Note that L cannot possibly intersect with more than $J+1$ of the $2J$ area regions associated with any J-way division D. This implies that $I_d \leqq (J+1)\cdot I_{d-1}$. This observation, the principle of induction, and the fact that $I_0 = 1$ jointly imply that $I_d \leqq (J+1)^d$ for all d. The maximum number of nodes whose range intersects with L in a

J-way polygon tree of height h will therefore be

$$(1) \qquad \sum_{d=0}^{h} (J+1)^d,$$

which equals

$$(2) \qquad \frac{(J+1)^{h+1}-1}{J}.$$

The latter quantity lies in $O(N^{\log_{2J}(J+1)})$ because an *ideal* polygon tree of height h will have exactly $N=(2J)^h$ leaves. Hence, the desired bound on the number of intersected vertices has been obtained. Q.E.D.

LEMMA 2. *Any polygon query Q can be performed in an ideal J-way polygon tree within $O(N^{\log_{2J}(J+1)})$ worst-case locate time.*

Proof. Note that the polygon retrieval algorithm will visit precisely those nodes whose "range" intersects with the geometric region associated with the user's specified query Q. The range of each such intersecting node must clearly either:

(i) intersect with both the geometric region of Q and its complement;
(ii) or lie completely contained within Q's region.

As the runtime of our retrieval algorithm is proportional to the number of visited nodes, it follows that this runtime can be calculated by simply counting the number of nodes belonging to categories (i) and (ii).

By Lemma 1, the number of nodes in category (i) for a query Q whose boundary consists of K straight lines must be $O(KN^{\log_{2J}(J+1)})$. Since the constant K is ignored in our asymptotic estimates, this quantity has an $O(N^{\log_{2J}(J+1)})$ magnitude.

Next, we wish to calculate the number of nodes belonging to category (ii). Its number of leaves is surely less than COUNT(Q) because each such leaf satisfies query Q. The number of *internal nodes* belonging to category (ii) can be seen to be less than its number of leaves by using the observations that:

(a) the set of nodes in category (ii) can be regarded as a forest;
(b) all internal nodes in each subtree of this forest will have at least two sons;[1]
(c) a trivial inductive argument shows that any tree, and therefore also forest, satisfying condition b, must have more leaves than internal nodes.

Hence, the total number of leaves plus internal nodes in category (ii) is less than $2 \cdot \text{COUNT}(Q)$.

Since the sum of the preceding two quantities lies in $O(N^{\log_{2J}(J+1)} + \text{COUNT}(Q))$, this quantity is a bound on the locate-and-copy time of the polygon retrieval algorithm. Hence, this algorithm has an $O(^{\log_{2J}(J+1)})$ worst-case locate time. Q.E.D.

At first, it may appear that the preceding lemma proves the worst-case runtime asserted in this paper since $N^{\log_{2J}(J+1)} = N^{\log_6 4}$ when $J=3$. However, we have, not yet actually proven our main assertion because many sets Z cannot be presented in the simple form of an "ideal" J-way polygon tree. One source of difficulty is that an ideal tree of height h has exactly $(2J)^h$ leaves and therefore can represent only sets of exactly this cardinality. Another difficulty is that sets with a large number of collinear points can not be represented in the ideal J-way form. In order to resolve these difficulties, we employ the concept of near-ideal J-way polygon trees, defined in the next paragraph. The most subtle part of this paper will be the proof that every set Z can be represented as a near-ideal J-way polygon tree.

---

[1] This is because all descendants of a node in category (ii) are also in category (ii).

In our definition of near-ideal trees, N_v denotes the number of leaves descending from node v. A J-way polygon tree will be said to be *near-ideal* iff the following two conditions are met:

(1) every *nonlinear* internal node v must have at least two sons and for each $i \leqq 2J$, there must be no more than $\lceil \bar{N}_v/2\bar{J} \rceil$ members associated with each corresponding $D_v(Z_v, i)$ set;

(2) every *linear* internal node must have exactly two sons, and these two sons must have either the exact same number of leaf-descendants or a number of leaf descendants which differs by at most one.

(An informal restatement of condition (2) is that the portion of a J-way polygon tree that descends from a linear node must be a binary tree whose balance is as close to ideal as possible.)

THEOREM 1. *Near-ideal J-way polygon trees have the same* $O(N^{\log_{2J}(J+1)})$ *worst-case polygon locate time as ideal polygon trees.*

The above proposition can be intuitively justified by observing that near-ideal trees are sufficiently similar to ideal polygon trees to ascertain that their worst-case runtimes differ only by a coefficient. A more formal proof is given in the rest of this section. Some readers may wish to skim this proof because the example of ideal polygon trees illustrates the intuition behind Theorem 1. The next section explains the significance of near-ideal trees by showing that every set Z can be represented in such a form.

The proof of Theorem 1 rests on the following preliminary lemma, which is the direct analogue for near-ideal trees of Lemma 1.

LEMMA 3. *For every near-ideal J-way polygon tree of N leaves and every straight line L (regardless of whether it be a line segment, a full line of infinite reach, or a half line), the number of nodes in tree T whose range intersects but is not contained within L will be bounded by* $O(N^{\log_{2J}(J+1)})$.

Proof. We will first prove Lemma 3 for the case where L is a full line of infinite reach and then consider the other cases of half lines and line segments.

For each nonlinear node v of polygon tree T, define $S_L(v)$ to be the subset of v's descendants satisfying the following two conditions:

(a) each such node must have a range that intersects but *is not* contained within L;

(b) each such node must either be one of v's linear sons or a descendant of such a son.

Also, define $\Pi_L(v)$ to be equal to:

(i) one plus the cardinality of $S_L(v)$ when v is a *nonlinear* node whose range intersects L;

(ii) zero, otherwise.

It can be readily verified that the definition of $\Pi_L(v)$ implies that the number of nodes which belong to polygon tree T and satisfy Lemma 3's intersection property will simply be

$$\sum_{v \in T} \Pi_L(v). \tag{3}$$

Therefore, the proof of Lemma 3 will rest on calculating a bound on the above sum.

Recall that N_v denotes the number of leaves descending from v. Let w denote a linear son of v, R_w the range of w, T_w the subtree descending from w, and $\Pi_L(w, v)$ the number of vertices in T_w that also belong to set $S_L(v)$. Our first claim is that for any *full line L*, the inequality $\Pi_L(w, v) \leqq \lceil \log_2 N_v \rceil$ will hold. In the cases where $R_w \cap L$ equals either the empty set or a set of more than one point, it is easy to verify $\Pi_L(w, v) = 0$ (this quantity vanishes in the first case because no node of T_w has a range intersecting L and

in the second case because all ranges are actually straight lines contained within L). The only remaining case is therefore the possibility that $R_w \cap L$ consists of precisely one point. In this case, no more than one node at each level of tree T_w can have a range intersecting L; this implies that $\Pi_L(w, v)$ can be no greater than the height of subtree T_w. Now, it is easy to conclude $\Pi_L(w, v) \leqq \lceil \log_2 N_v \rceil$ by using part 2 of the definition of near-ideal polygon trees to show that tree T_w has height bounded by $\lceil \log_2 N_v \rceil$.

The preceding analysis enables us to compute a bound on $\Pi_L(v)$: this quantity must be bounded by $J \lceil \log_2 N_v \rceil + 1$ because no more than J linear sons will be associated with any nonlinear node v. Now, we use the following three observations to calculate bounds on the value of N_v:

(i) Each *nonlinear* node at a depth d in a near-ideal polygon tree will have $N_v \leqq \lceil N/(2J)^d \rceil$;

(ii) no more than $(J+1)^d$ such nonlinear nodes at depth d will have ranges intersecting line L;

(iii) no nonlinear internal nodes can exist in a near-ideal tree at depth below $\lceil \log_{2J} N \rceil$.

Clearly, observation *i* together with our bound on $\Pi_L(v)$ implies that all nodes at depth d must satisfy

$$\Pi_L(v) \leqq 1 + J \cdot \log_2 (2J) \cdot (\lceil \log_{2J} N \rceil - d). \tag{4}$$

Applying (4) and observations (ii) and (iii) to equation (3), we obtain that this sum is no greater than

$$\sum_{d=0}^{\lceil \log_{2J} N \rceil} (J+1)^d [1 + J \cdot \log_2 (2J)(\lceil \log_{2J} N \rceil - d)]. \tag{5}$$

Since the latter sum lies in $O(N^{\log_{2J}(J+1)})$, our proof has verified that this quantity bounds the number of vertices whose ranges intersect but is not contained by a line L of infinite reach.

The same general reasoning is used to verify Lemma 3 for the alternate cases of half lines and line segments. The only difference in these two cases is that $\Pi_L(w, v)$ has respective bounds $\lceil \log_2 N_v \rceil$ and $2 \lceil \log_2 N_v \rceil$ when $R_w \cap L$ has more than one intersection point. This event affects no more than one son of every nonlinear internal node v; it therefore increases the final coefficient by a factor of less than $1 + 1/J$. Q.E.D.

We are now ready to explain how Theorem 1 can be proven. The proof rests on reasoning similar to the proof of Lemma 2 except that Lemma 3 rather than Lemma 1 must now be used in the step of this proof that counts the number of type (i) vertices. That is, the previous proof of Lemma 2 was (carefully) worded so that it will also prove Theorem 1 once its citation of Lemma 1 is changed to a reference to Lemma 3.

COROLLARY 1. *In a near-ideal J-way polygon tree, a request for either the points lying on a straight line or in a (nonpolygon) region whose boundary is determined by several straight lines can be performed in the same* $O(N^{\log_{2J}(J+1)})$ *worst-case locate retrieval time as polygon requests.*

Proof. It is easy to see the validity of Corollary 1 for any convex region: such a geometric shape with K boundary lines has a worst-case locate runtime proportional to $KN^{\log_{2J}(J+1)}$ by the exact same reasoning as was previously used to prove Lemma 2 and Theorem 1. One way to prove Corollary 1 for nonconvex regions is to first decompose an initial nonconvex region into a union of disjoint convex parts (the Chazelle–Dobkin algorithm [7] will produce such a decomposition for nonconvex polygons). The

nonconvex query is then performed by taking the union of the results from querying its convex parts. It is easy to see that such a retrieval algorithm runs in $O(N^{\log_{2J}(J+1)})$ locate runtime. Q.E.D.

Comment 1. Also, the results of this paper can be extended to geometric regions whose boundary consists of differentiable curves. This is because differentiable curves tend to be almost linear in small regions of space. As a result, differential curves will appear almost as straight lines within local regions of our polygon tree. This means that the efficiency of polygon trees can be heuristically justified in the context of differentiable curves.

Comment 2. Let f denote a function that maps the elements of set Z onto real numbers. A polygon tree will be said to be an aggregate representation of function f over set Z if and only if each internal node of that tree contains an additional field indicating the sum of the f-values of all the leaves descending from it. The theorems in this section can be modified to state that aggregate polygon trees make it possible to calculate the sum of the f-values lying inside any arbitrary polygon in $O(N^{\log_{2J}J+1})$ worst-case execution time. This time is somewhat different from our earlier results because it is a *strict* worst-case result rather than a worst-case *locate* measurement.

3. The significance of polygon trees. This section will prove that every conceivable set of N elements can be represented as a J-way near-ideal polygon tree. Our discussion of this existence theorem is significant because the same proposition does not hold when it is applied to other related data structures (such as the quad trees of [6], [10]).

By the end of this section, it will become apparent how polygon trees are more efficient than previously proposed data structures.

LEMMA 4. *Assume that four numbers n_i have been chosen so that $n_1+n_2+n_3+n_4=N$ and that Z is a set of cardinality N where n_1+n_2 (respectively n_3+n_4) of its elements lie above (below) L_1. Then it is possible to find a line L_2 (as shown in Fig. 4) so that the number of points lying in each region R_i is less than or equal to n_i. (The possibility of inequality is allowed in this lemma because some members of Z may lie on either the L_1 or L_2 line.)*

Proof. The lemma is trivial (and basically uninteresting) to verify in the degenerate case where at least one of the four n_i quantities equal zero. In our remaining discussion, it will therefore be assumed that each n_i is greater than or equal to 1.

Given a fixed point P on the L_1 line, let $f(P)$ denote the least possible value for θ in Diagram 4 such that the line L_2 through P at this angle causes there to be no more than respectively n_1 and n_2 points in the regions R_1 and R_2. Similarly, let $g(P)$ denote the least θ such that there are no more than respectively n_3 and n_4 points in the R_3 and R_4

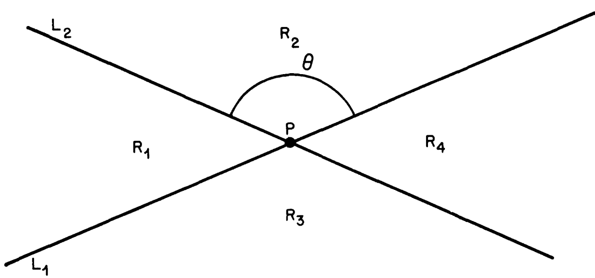


FIG. 4

regions. Note that the functions $f(P)$ and $g(P)$ are continuous as P is moved along the L_1 line. Also, the following limits must hold as P is moved along this line:

$$\lim_{p\to\infty} f(P)=\Pi, \tag{6}$$

$$\lim_{p\to-\infty} f(P)=0, \tag{7}$$

$$\lim_{p\to\infty} g(P)=0, \tag{8}$$

$$\lim_{p\to-\infty} g(P)=\Pi. \tag{9}$$

The combination of the above four limits and the continuity of f and g imply the existence of a point P with $f(P)=g(P)$. The line L_2 passing through this point with angle $f(P)$ will satisfy the conditions of the lemma. Q.E.D.

COROLLARY 2. *There will always exist some line L_2 which intersects at least two points of set Z and satisfies Lemma* 4.

Proof. It is clear that the line L_2 in the preceding proof satisfies this condition. Q.E.D.

LEMMA 5. *For every set Z of N points and every real number s, there exists a J-way division similar to Fig.* 1 *such that*:

(i) *there are no more than $\lceil N/2J \rceil$ points in any of the $2J$ open regions whose boundary is determined by this division*;

(ii) *the L_1 line has a slope of s.*

Proof. There clearly is no difficulty in selecting an L_1 line of slope s such that it separates the set Z of points into two subsets that each have less than or equal to $\lceil N/2 \rceil$ points. Our remaining proof rests on repeated applications of Lemma 4 to successively prove the existence of the $L_2L_3 \cdots L_J$ lines that are required by the present proposition.

More specifically, Lemma 4 is first utilized to prove the existence of a L_2 line such that there are no more than $\lceil N/2J \rceil$ points in each of the two leftmost regions of Diagram 1 and such that the remaining points to its right are divided by line L_1 into two subsets of no more than $\lceil (J-1)N/2J \rceil$ points.

For our second application of Lemma 4, we have Z_2 denote the subset of set Z that lies to the right of line L_2; this lemma immediately implies the existence of a line L_3 that divides Z_2 and therefore also Z in the desired manner.

The existence of the additional $L_4L_5 \cdots L_J$ lines is proved by further repeated applications of Lemma 4. Q.E.D.

THEOREM 2. *It is possible to represent every set Z as a near-ideal J-way polygon tree.*

Proof. The natural algorithm for constructing a near-ideal polygon tree is a topdown procedure that first builds the J-way division that is needed by the root and then recursively calls itself to build the portions of this tree which (this division indicates should) descend from each of the root's sons. In order to show that near-ideal trees can always be constructed with such a topdown algorithm, it is sufficient to verify that the root can always be made to satisfy the requirements of near-idealness. Note that the only nontrivial aspect of the definition of near-idealness is its first part and that Lemma 5 indicates that the root can always be made to satisfy this condition. Our theorem thus follows from the previous lemma. Q.E.D.

Comment 3. The combined implication of Theorems 1 and 2 is that every set of N points can be represented as a near-ideal J-way polygon tree with an $O(N^{\log_{2J} J+1})$ worst-case polygon locate time. This asymptotic time is minimized when J is chosen to have the value 3. Under these circumstances, the derived $O(N^{\log_6 4})$ runtime has an $O(N^{.77})$ magnitude.

Comment 4. A 2-way polygon tree will be said to be a quad tree if the L_1 and L_2 lines of every division D_v have been chosen so that they are respectively parallel to the x- and y-axes. The runtime characteristic of quad trees for finding the subset of set Z whose x and y components satisfy a query of the form $a_1 < x < b_1$ and $a_2 < y < b_2$ has been discussed in [6], [10], [16]. An examination of quad trees in the context of this article indicates that Theorem 1 holds for near-ideal quad trees but not Theorem 2. In other words, near-ideal quad trees do have an $O(N^{\log_4 3})$ worst-case polygon locate runtime but there is no guarantee that an arbitrary set of points can be represented in such a form. This shortcoming of quad trees is serious. The worst-case polygon locate runtime for quad trees lies in $O(N)$ because certain sets can not be represented in a reasonable manner by this data structure[2]. It is for this reason that polygon trees are more suitable in many applications.

Comment 5. Also, there are some interesting comparisons between K-d trees and polygon trees. K-d trees are discussed in [2], [16], [22], [24]. They have difficulty with polygon retrieval for just the opposite reason as quad trees. Thus, Theorem 2 but not 1 holds for K-d trees. In other words, every set of points in the plane can be represented as a near-ideal (two-dimensional) K-d tree, *but there is no guarantee these trees have efficient polygon retrieval time* (since analogs of Lemmas 1 and 3 do not hold for K-d trees). The intuitive idea that motivated much of this paper's research into polygon trees was that they would attain an efficient worst-case polygon locate time by combining the best characteristics of quad and K-d trees.

Comment 6. It is easy to develop counterexamples which demonstrate that all the other multidimensional data structures mentioned in the introduction of this paper, also have inefficient $\Omega(N)$ worst-case polygon locate times. We will not discuss the relevant details here because these structures were intended for a different type of application and have an underlying design which is unrelated to this article's polygon trees.

Comment 7. A polygon tree will be said to satisfy the *alternating* condition if the L_1 line of the D_v division of every node v of this tree is

(i) parallel to the x-axis when v has even depth;

(ii) parallel to the y-axis when v has odd depth.

Note that part (ii) of Lemma 5 indicated that complete freedom exists in choosing the slopes of L_1 lines of the J-way divisions. Consequently, Theorem 2 can be strengthened to read that every set Z can be represented as an *alternating* near-ideal J-way polygon tree. In most applications, users will desire to employ alternating rather than nonalternating polygon trees because the former have an improved runtime for the special case where the retrieval query is a request for those records lying in a rectangle with horizontal and vertical edges. Near-ideal alternating polygon trees permit this request to be processed in $O(N^{\log_{(4J^2)}(J^2+J)})$ worst-case locate time (because the alternating condition guarantees that no horizontal or vertical line can intersect the ranges of more than J^2+J of the $4J^2$ nonlinear grandsons of a given node). For the cases of J equals 2 and 3, the preceding quantities reduce respectively to $N^{.65}$ and $N^{.69}$ worst-case rectangle

[2] For an example, consider any finite set at points whose coordinates satisfy $y = x$. Such a set can never be represented by a near-ideal quad tree. Any nonideal quad tree representing this set will have an $\Omega(N)$ worst-case polygon locate runtime.

retrieval times. The central point is that the alternating concept improves the rectangle retrieval time for polygon trees without weakening any of the preceding results.

Comment 8. We should point out that quad and *K-d* trees have $\sqrt{N}$ worst-case runtime for rectangle queries; they thus outperform polygon trees for these special queries. Some readers may be surprised to learn how slightly changed assumptions make polygon trees asymptotically more efficient from one standpoint and less from another. There are actually many other surprising characteristics of multidimensional trees. For instance, a slightly imbalanced *K-d* tree, satisfying the AVL condition, can require $\Omega(N)$ orthogonal range query time if the query has dimension of at least four [24]. The same reference also shows that randomly generated *K-d* trees have a surprisingly inefficient expected retrieval time for partial match and orthogonal range queries. A major difference between one and multidimensional retrieval is that the latter has a runtime that is more sensitive to relatively minor changes in the data structure.

Comment 9. For the sake of simplicity, the emphasis of this paper has been on the optimization of CPU runtime rather than disc accesses. However, there is no difficulty in extending our results to the disc by designing special polygon trees that store nearby nodes on the same disc page. If M denotes the number of records stored on a page, then polygon trees will have $O[(N/M)^{\log_{J+1}(2J)}]$ worst-case number of page accesses in this context.

Comment 10. One of the most attractive aspects of polygon trees is their potential applications in data-bases. The nature of these applications can be understood once it is noted that many algebraically defined sets can be converted into geometric structures if they are mapped onto their representation in a Cartesion coordinate system. For instance, the set of points satisfying "$y>0$ AND $y<2x$ AND $y<1-2x$" forms an isosceles triangle in the x, y-plane. Obviously, a retrieval for this algebraic relation can be performed efficiently with a polygon tree. As many algebraic relations have geometric representations that consist of straight lines, polygons or regions bounded by several straight lines, it follows that these relations can be queried in efficient worst-case runtime with polygon trees. The clearly attractive feature of these trees is that they efficiently serve a very broad class of predicate retrievals while occupying only $O(N)$ space.

4. Construction of polygon trees. Several algorithms for constructing near-ideal polygon trees are outlined in this section. We show that a J-way polygon tree can be constructed in $O(N^2)$ worst-case and $O(N \log^2 N)$ expected runtime. The crux of our analysis will consist of showing that for any set Z, the L_2 line of Lemma 4 can be found in $O(N^2)$ worst-case and $O(N \log N)$ expected runtime.

Recall that Corollary 2 indicated that it is always possible to select a line L_2 which satisfies the requirements of Lemma 4 and additionally intersects at least two points of set Z. This observation suggests one very simple (although inefficient) algorithm for finding the L_2 line:

> Make an exhaustive scan over all $O(N^2)$ different lines generated by pairs of points from set Z and compare the position of each examined line to the N points of set Z until a line satisfying L_2's requirements is found.

The principal disadvantage of the above procedure is that it runs in $O(N^3)$ worst-case time. Most of the rest of this section will explain how a more efficient $O(N^2)$ worst-case complexity can be obtained with a modified algorithm.

In our discussion, W_0 will denote the set of points that lie on line L_1 and also lie on one of the lines generated by the pairs of points in set Z. We will now explain how a line L_2 satisfying requirements of Lemma 4 can be quickly found by using a special variation of binary search. This search will be based on the natural linear ordering that line L_1 imposes upon set W_0.

Let P denote a point on line L_1; also assume that L_1 satisfies the usual condition of dividing set Z so that there are n_1+n_2 (respectively n_3+n_4) points above (below) L_1. The *upper* (respectively *lower*) *angular set* of P will be defined as the set of angles θ such that the L_2 line passing through P at angle θ causes regions R_1 and R_2 (respectively R_3 and R_4) of Fig. 4 to contain no more than n_1 and n_2 (n_3 and n_4) of Z's points. The following lemmas motivate the form of binary search employed in this paper.

LEMMA 6. *Again assume that line L_1 divides set Z so that n_1+n_2 (respectively n_3+n_4) elements lie above (below) it. A necessary and sufficient condition for some line L_2 to pass through point P of line L_1 and divide set Z so that each region R_i (in Diagram 4) has no more than n_i points is that P's upper and lower angular sets have a nonempty intersection. If, on the other hand, all the angles of the lower angular set are less (respectively greater) than those of the upper angular set then such a division will be formed by some line L_2 intersecting line L_1 to the left (right) of P.*

Proof. The first half of Lemma 6 is an obvious consequence of the definition of angular sets, and the second follows from reasoning similar to the proof of Lemma 4. Q.E.D.

LEMMA 7. *The upper and lower angular sets can always be calculated in $O(N)$ runtime.*

Proof. For any two points X_1 and X_2, simple techniques from linear algebra can be used to determine in $O(1)$ runtime whether or not line segment $\overline{X_1P}$ forms a greater angle with line L_1 than $\overline{X_2P}$. In order to calculate the upper angular set, we must find the n_1th and (n_1+1)th largest angles among the lines that connect point P to the subset of Z lying above L_1. It is well known that if an individual greater-than comparison can be performed in $O(1)$ time, then the Kth largest of N elements can be found in $O(N)$ time (see for instance [1, pp. 97–99]). The desired calculation of the upper and lower angular sets is therefore obtained by applying this algorithm to find the n_1th and (n_1+1)th largest angles. Q.E.D.

We are now ready to explain how a line L_2 satisfying all the requirements of Lemma 4 can be found in $O(N^2)$ worst-cast time. Our search algorithm will consist of the following two steps:

(1) Recall that W_0 denotes the set of points on line L_1 that intersect some line that contains at least two points of set Z. This step will construct set W_0 by performing a straightforward exhaustive scan over all $O(N^2)$ pairs of points in set Z; it will then prepare for the execution of the next step by making W equal to W_0.

(2) This step will consist of an iterative procedure which repeatedly divides the set W in half until the desired line L_2 is found. The precise procedure of this step will consist of repetitions of the following three substeps:

(2a) Find the median member of set W in time proportional to the cardinality of W by using the median element search procedure of [1, pp. 97–99]. Let P denote this point; construct two sets, W^+ and W^-, which designate the respective subsets of W whose points lie to the right (left) of P along line L_1.

(2b) Use the Lemma 7 procedure to calculate P's upper and lower angular sets.

(2c) If these two angular sets have a nonempty intersection then the search is done: in this case, L_2 is set equal to any line passing through P with such an angle. Otherwise, go back to substep (2a) with W made equal to W^+ if the lower angular set is greater than the upper angular set and equal to W^- when this inequality is reversed.

THEOREM 3. *For any line L_1 that divides set Z into subsets of respectively n_1+n_2 and n_3+n_4 points, the above algorithm:*

(i) *correctly finds a L_2 line such that each R_i region of Fig. 4 has no more than n_i points;*

(ii) *performs this task in $O(N^2)$ worst-case runtime.*

Proof. The correctness of the cited algorithm is an immediate consequence of Corollary 2 and Lemma 6. Its $O(N^2)$ runtime is a consequence of the following observations:

(1) step 1 consumes $O(N^2)$ time and the ith iteration of substeps (2a), (2b) and (2c) consume respectively $O(N^2/2^i)$, $O(N)$ and $O(1)$ runtime;

(2) there are no more than $\lceil 2 \log N \rceil$ iterations of substeps (2a) through (2c) (because each iteration cuts W in half). Q.E.D.

We now describe an alternate algorithm for finding line L_2 that runs in $O(N \log N)$ expected time. This improved algorithm is based on the observation that the previous procedure was slow because it manipulated unwieldy sets, W and W_0, that had expensive cardinalities in $O(N^2)$. Rather than think of W as a large finite set, our revised algorithm will regard W as a closed interval connecting two points belonging to line L_1. The only other major change in our revised algorithm is that point P will designate the position midway between W's two endpoints, rather than the median element of a finite set. The intuitive reason for the added efficiency of this modified algorithm is that midway points can be calculated much more quickly than median elements.

Now more formally, we describe exactly how our revised algorithm differs from the previous procedure. Step 1 of the new algorithm will calculate the projection that set Z has on line L_1; it will then set W initially equal to the interval connecting the left and right most of these points. Substep (2a) of the new procedure will set P equal to the point that bisects W; it will then make W^- and W^+ denote the respective halves of interval W that lie to the left and right of P. These are the only parts of our revised algorithm that differ from the old procedure; other aspects of our new algorithm (specifically substeps (2b) and (2c)) are identical to the old procedure.

THEOREM 4. *Let Z denote a set of $N = n_1+n_2+n_3+n_4$ elements generated by a uniform distribution on any convex and bounded region R in the plane. Again let L_1 be some line such that n_1+n_2 (respectively n_3+n_4) of Z's points lie above (below) L_1. Then the above algorithm will find in $O(N \log N)$ expected time a line L_2 such that each region R_i in Fig. 4 has no more than n_i points.*

Proof sketch. Let $W^*(Z)$ denote the set of points P on line L_1 such that some line L_2 passing through P satisfies the above conditions. Since $W^*(Z)$ is a closed interval, it makes sense to use the symbol $l^*(Z)$ for denoting the length of this interval. Also let l denote the length of the part of line L_1 that intersects with region R. Our first claim is that the event $l^*(Z) < \lambda l$ will occur with a probability less than $2\lambda N$ when Z is generated by the uniform probability distribution.

We will only sketch the proof of this assertion since its formal verification is trivial and tedious and since the later parts of our analysis will not require the particular coefficient 2, associated with λN in this paragraph. Given one specific line L_2 which

divides region R so that n_i points from set Z lie in each subregion R_i, let x denote the length of that portion of line L_1 which lies to the left of L_2 in region R, L_2^- and L_2^+ two lines which are parallel to L_2 and which intersect L_1 at respective distances λx to the left and $\lambda(l-x)$ to the right of L_2's intersection point with L_1, and E_i the portion of region R_i that lies between lines L_2^- and L_2^+ (see Fig. 5). Since each region R_i is convex and bounded, the ratio of the areas of E_i to R_i must be less than 2λ, for each i. The regions E_i

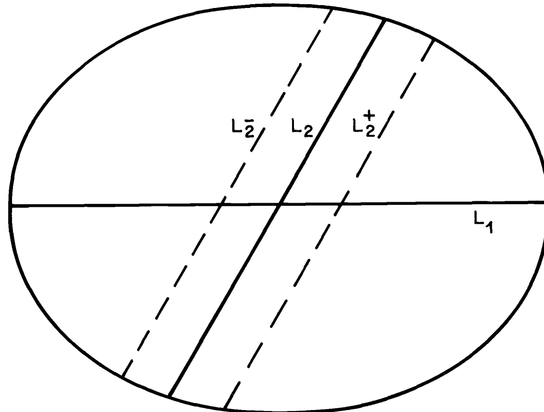


FIG. 5. *In this figure, R is an oval shaped region, R_i one of the regions bounded by the solid lines, and E_i the part of R_i between the parallel lines.*

are thus sufficiently small to assure that there is less than a probability $2\lambda N$ for one or more elements of set Z to lie in any of the regions E_i. But if no element of set Z lies between L_2^- and L_2^+ then $W^*(Z)$ will certainly include the entire portion of line L_1 between these two parallel lines. And the latter condition implies $l^*(Z) \geqq \lambda l$. Hence, there must be a probability less than $2\lambda N$ that $l^*(Z) < \lambda l$, for all positive λ.

The above observation implies that the expected value of $\log(l/l^*(Z))$ lies in $O(\log N)$. In turn, this implies that our search algorithm will find one qualifying line L_2 in a number of iterations of step (2) which is expected to be $O(\log N)$. Since each iteration of our revised search algorithm runs in $O(N)$ time, the total expected time of this algorithm is $O(N \log N)$. Q.E.D.

THEOREM 5. *A near-ideal J-way polygon tree can be constructed in*

(a) $O(N^2)$ *worst-case runtime for any set* Z;

(b) $O(N \log^2 N)$ *expected runtime for a set Z whose points are generated by the uniform distribution on a bounded convex region.*

Proof. We will prove only assertion (a) since assertion (b) has a similar verification. Note that for any slope s and set Z_v of cardinality N_v, it is possible in $O(N_v)$ runtime to find a line L_1 of slope s such that no more than $\lceil N_v/2 \rceil$ records lie on each side of L_1 (one way to find this line is to again invoke the median element selection procedure of [1, pp. 97–99]). Applying $J-1$ iterations of the Theorem 4 procedure to line L_1, we can obtain lines $L_2 L_3 \cdots L_J$ such that each of the open regions in Fig. 5 contain no more than $\lceil N_v/2J \rceil$ records. Since constant J does not appear in our estimates of asymptotic values, the preceding J-way division is constructed in $O[(N_v)^2]$ runtime. Now given this complexity, we wish to show that $O(N^2)$ is a bound on the total time to construct a near-ideal J-way tree.

Consider a topdown procedure for constructing J-way trees similar to the one discussed in the proof of Theorem 2. Let $V(d)$ denote the set of internal nodes at depth

d in a near-ideal tree. The last paragraph implies that the total time spent constructing the J-way divisions of depth d nodes is $O[\sum_{v \in V(d)} (N_v)^2]$. Note that every node at this depth in a near-ideal tree must satisfy $N_v \leqq \lceil N/2^{d-1} \rceil$; also, it is easily seen that $\sum_{v \in V(d)} N_v \leqq N$. Substituting the second and third terms into the first, we obtain that the total time spent building the J-way division of depth d nodes is bounded by $O(N^2/2^{d-1})$. Taking the sum of this quantity over all integers d, we derive $O(N^2)$ as the worst-case cost of building a near-ideal polygon tree. Q.E.D

Polygon trees can be efficiently manipulated in a dynamic environment as a forest with techniques similar to those proposed in [5], [22], [24] and more recently in [18]. Reasoning similar to that in [24] can be generalized to show that traditional tree balancing methods should not be applied to dynamically changing polygon trees.

5. Further research. Six open questions about the complexity of polygon retrieval are raised in this section. Both *upper* and *lower* bounds in answer to these questions would be interesting.

Question 1. Is it possible to construct near-ideal J-way polygon trees in better than worst-case time $O(N^2)$? The large gap between the two runtimes in Theorem 5 suggests that better runtime may be possible.

Question 2. Let A denote an open and bounded region on the plane, μ a probability density which is defined on A and has a bounded derivative, Z a set of N elements generated by probability density μ, T a J-way near-ideal polygon tree that satisfies the alternating condition of Comment 7, and B_d the set of boundary lines that divide the ranges of the nodes of depth d in T. This author conjectures that the intersection of B_d with set A will have an aggregate length of expected magnitude $O((2J)^{d/2})$. Is this conjecture true? If so, then it can be proven that polygon trees will have an expected locate query time $O(\sqrt{N})$ for such an input distribution.

Question 3. A class of trees will be said to be *universal* if and only if every set of elements can be represented as a tree of this form. In this terminology, Theorem 2 can be interpreted as stating that near-ideal polygon trees are universal and Comment 4 as indicating that near-ideal quad trees are not universal. An important open question is whether better polygon retrieval times can be obtained with other universal classes of data structures, perhaps occupying more memory space. Researchers investigating this question should keep in mind that one can easily be misled by alternate data structures that appear to have a better worst-case retrieval time than the N^{77} performance of 3-way polygon trees until it is realized that they are not universal. This author has an uneasy suspicion that it may be difficult to improve asymptotic retrieval time without a prohibitive increase in memory space. For instance, how much memory space is necessary to achieve $\log N$ worst-case polygon locate time?

Question 4. Are the natural three and four dimensional generalizations of polygon trees universal? This author has determined that their five-dimensional generalization is not![3] What universal classes of data structures will generalize the results of this paper in higher dimensions?

Question 5. Note that any algebraically defined set over the x, y-plane, generated by the primitives of addition, subtraction, and scalar multiplication, has linear boundaries and therefore can be efficiently queried with a polygon tree. For instance, the set of points satisfying "$y > 0$ AND $y < 2x$ AND $y < 1-2x$" is an isosceles triangle and

---

[3] This is because Lemma 4 does not generalize for dimension $k \geqq 5$. Note that this generalization would consist of using k distinct $(k-1)$-dimensional hyperplanes for dividing k-space into 2^k volumes. Since $2^k - 1 > k^2$ when $k \geqq 5$, there are more constraints than degrees of freedom when $k \geqq 5$. This means a division is not possible for arbitrary values of n_i.

therefore can be easily handled by this data structure. What can be said about more complex queries that employ multiplication of variables such as for instance "$xy > 1$ AND $x + y < 2$"? Does there exist a universal class of data structures that occupies a reasonable amount of memory space and serves these queries in better than $O(N)$ *worst-case* locate time?

Question 6. For an arbitrary data structure in a dynamic setting, what are the lower bounds on the tradeoffs between worst-case record insertion, deletion, and polygon retrieval times? This question was raised by Fredman [11], and Fredman also conducted the initial research in this topic.

Acknowledgment. I would like to thank the referee for having carefully read this article and for his several useful comments and observations.

REFERENCES

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

[2] J. L. BENTLEY, *Multidimensional binary search trees used for associative searching*, Comm. ACM, 18 (1975), pp. 509–517.

[3] ———, *Multidimensional binary search trees*, Comm. ACM, 23 (1980), pp. 214–228.

[4] J. L. BENTLEY AND H. A. MAURER, *Efficient worst-case data structures for range searching*, Acta Inform., 13 (1980), pp. 155–168.

[5] J. L. BENTLEY AND J. B. SAXE, *Decomposable searching problem #1: static to dynamic transformation*, J. Algorithms, 1 (1980), pp. 301–358.

[6] J. L. BENTLEY AND D. F. STANAT, *Analysis of range searches in quad trees*, Inform. Proc. Letters, 3 (1975), pp. 170–173.

[7] J. L. BENTLEY AND M. I. SHAMOS, *A problem in multi-variate statistics: algorithm, data structure, and applications*, Proc. 15th Allerton Conference on Communications, Control, and Computing, 1977, pp. 193–201.

[8] B. CHAZELLE AND D. DOBKIN, *Decomposing a polygon into its convex parts*, Proc. 11th. ACM Symposium on Theory of Computing, 1979, pp. 38–45.

[9] D. DOBKIN AND R. J. LIPTON, *Multidimensional searching problems*, this Journal, 5 (1976), pp. 181–186.

[10] R. A. FINKEL AND J. L. BENTLEY, *Quad trees: a data structure for retrieval on composite keys*, Acta Inform., 4 (1974), pp. 1–9.

[11] M. L. FREDMAN, *The inherent complexity of dynamic data structures which accommodate range queries*, Proc. 21st IEEE Symposium on Foundations of Computer Science, 1980, pp. 191–200.

[12] ———, *Lower bounds on the complexity of some optimal data structures*, this Journal, 10 (1981), pp. 1–11.

[13] ———, *A lower bound on the complexity of orthogonal range queries*, J. Assoc. Comput. Mach., to appear.

[14] G. S. LUEKER, *A transformation for adding range restriction capability to dynamic data structures for decomposable search problems*, Tech Rep #129, Department of Computer Science, Univ. of California at Irvine.

[15] G. S. LUEKER AND D. E. WILLARD, *A data structure for dynamic range queries*, forthcoming report.

[16] D. T. LEE AND C. K. WONG, *Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees*, Acta Inform., 9 (1977), pp. 23–29.

[17] ———, *Quintary trees: a file structure for multidimensional database systems*, ACM Trans. Database Systems, 5 (1980), pp. 339–353.

[18] M. H. OVERMARS AND J. VAN LEEUWEN, *Two general methods for dynamizing decomposable searching problems*, Computing, 26 (1981), pp. 155–166.

[19] R. L. RIVEST, *Partial match retrieval algorithms*, this Journal, 5 (1976), pp. 19–50.

[20] D. E. WILLARD, *Predicate-oriented database search algorithm*, Ph.D. thesis, Mathematics Department, Harvard Univ., Cambridge, MA, disseminated as one of volumes in Garland Publishing Company's series of "Outstanding Dissertations in Computer Science."

[21] ———, *New data structures for orthogonal range queries*, Tech. Rep. TR-22-78, Harvard Aiken Comp. Lab, Cambridge, MA, Nov., 1978.

[22] ———, *Balanced forests of K-d Trees as a dynamic data structure*, Tech. Rep. TR-23-78, Harvard Aiken Comp. Lab., Cambridge, MA, Nov., 1978.

[23] ———, *The super-B tree algorithm*, Tech. Rep. TR-79-03, Harvard Aiken Comp. Lab, Cambridge, MA.

[24] ———, *K-d Trees in a dynamic environment*, Tech. Rep. 80-1, Computer Science Department, University of Iowa, Iowa City, May, 1980.

[25] D. E. WILLARD AND G. S. LUEKER, *A transformation for adding range restriction capability to data structures*, forthcoming report.

SIAM J. COMPUT.
Vol. 11, No. 1, February 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1101-0013 $01.00/0

A POLYNOMIAL TIME ALGORITHM FOR DECIDING THE EQUIVALENCE PROBLEM FOR 2-TAPE DETERMINISTIC FINITE STATE ACCEPTORS*

E. P. FRIEDMAN† AND S. A. GREIBACH‡

Abstract. The equivalence problem for the class of one-way deterministic 2-tape finite state acceptors is the problem of deciding "$L(M_1) = L(M_2)$", where M_1 and M_2 are machines in this class. A new algorithm for deciding equivalence is provided, having time complexity proportional to $p(n)$, where p is a polynomial and n is the size of the machines. This improves upon the best previously known upper bound having order 2^{cn^6}, where c is a constant [C. Beeri, Theoret. Comput. Sci., 3 (1976), pp. 305–320].

Key words. deterministic, equivalence problem, finite state acceptors, multitape automata, polynomial time, 2-tape acceptors

1. Introduction. Consider the class of deterministic finite state machines with t tapes, each with its own read-only head moving from left to right. The equivalence problem for machines in this class has remained open for $t \geqq 3$ since it was originally posed [6]. Rabin and Scott [6] showed that equivalence is decidable for $t = 1$, and this problem was later proven to have time complexity of order $nG(n)$, where n is the size of the machines and $G(n)$ is a function which grows extremely slowly [1], [5]. Although the equivalence problem remains decidable for nondeterministic single-tape machines [6], the problem becomes undecidable for classes of nondeterministic machines with more than one tape [4], [7]. Hereafter we restrict our attention to deterministic devices.

Equivalence for the case $t = 2$ was first shown to be decidable by Bird [3]. Later, Valiant [8], [9] provided a different algorithm for determining equivalence of 2-tape acceptors, and his algorithm was modified by Beeri [2], yielding an upper bound on the time complexity of 2^{cn^6}, where c is some constant and n is the size of the machines. In this paper we provide another algorithm for determining equivalence of 2-tape acceptors, but with the time complexity reduced from an exponential bound to a polynomial one.

In § 2, we establish notation and define 2-tape deterministic finite state acceptors formally. In § 3, we provide an exponential time algorithm for determining equivalence between two states of a single 2-tape acceptor. In § 4 we build upon the ideas in this algorithm to obtain a polynomial time algorithm for deciding equivalence. The remarks in § 5 indicate how this algorithm can be improved to have time complexity of order n^4, where n is the size of the machines. All complexity bounds are given for execution on a RAM under the uniform cost criterion (cf. [1] for discussion of this concept).

2. Preliminaries. A *2-tape deterministic finite-state acceptor* (abbreviated 2-dfsa) is denoted by $M = (K_1, K_2, \Sigma, \delta, q_0, F)$, where K_1 and K_2 are disjoint finite sets of *states*, with K_1 controlling tape 1 and K_2 controlling tape 2, Σ is a finite set of *input symbols*, q_0 in $K_1 \cup K_2$ is the *initial state*, F a subset of $K_1 \cup K_2$ is the set of *accepting* or *final states*, and $\delta : ((K_1 \cup K_2) \times \Sigma) \rightarrow (K_1 \cup K_2)$ is the *transition function*.

* Received by the editors August 16, 1979, and in final form February 5, 1981. This research was supported in part by the National Science Foundation under Grant MCS-78-04725.

† Radar Systems Group, Hughes Aircraft Company, Bldg. R1, Mail Station B218, P.O. Box 92426, Los Angeles, California 90009 and Computer Science Department, University of California, Los Angeles, California 90024. This work was completed while this author was at the Software Engineering Laboratory, Radar Systems Group, Hughes Aircraft Company, El Segundo, California.

‡ Computer Science Department, University of California, Los Angeles, California 90024.

Let e denote the *empty word*, $|x|$, the *length of word* x, and $|(u, v)| = |u| + |v|$, the *length of pair* (u, v). We say that x is a *suffix* of y if $y = zx$ for some z; it is a *proper suffix* if $x \neq y$. We let $|S|$ denote the cardinality of finite set S.

We write $\delta(p, a) = q$ as p—$(a, e) \to q$ when p is in K_1 and as p—$(e, a) \to q$ when p is in K_2, and call it a *1-step computation*. We let p—$(e, e) \to p$ trivially for any p in $K_1 \cup K_2$; for any p, q, r in $K_1 \cup K_2$ and u, v, x, y in Σ^*, if p—$(u, v) \to q$ and q—$(x, y) \to r$, we write p—$(ux, vy) \to r$, and call it a *computation*; this computation is said to have *length* $|(ux, vy)|$. A state q is *accessible from a state* p if p—$(u, v) \to q$ for some u, v in Σ^*, and *accessible* if it is accessible from the initial state q_0.

The *language accepted from state* p is

$$L(p) = \{(u, v) \mid p\text{—}(u, v) \to q \text{ for some } q \text{ in } F\}$$

and the language accepted by M is $L(M) = L(q_0)$. A pair (u, v) is accepted by M if it is in $L(M)$; otherwise, it is rejected.

The *left quotient* of $L(p)$ by pair (x, y), denoted $(x, y)\backslash L(p)$, is the set

$$\{(u, v) \mid p\text{—}(xu, yv) \to r \text{ for some } r \text{ in } F\}.$$

Two states p and q are *equivalent* if $L(p) = L(q)$. Two machines are equivalent if their initial states are equivalent. If $p \neq q$ and (u, v) is in $(L(p) - L(q)) \cup (L(q) - L(p))$, then (u, v) *distinguishes* p and q. If L and L' are subsets of $\Sigma^* \times \Sigma^*$, $L \neq L'$, and (u, v) in in $(L - L') \cup (L' - L)$, then (u, v) *distinguishes* L and L'.

The *state equivalence problem for 2-dfsa's* is the problem of deciding "$L(p) = L(q)$" where p and q are states in a 2-dfsa. The *equivalence problem for 2-dfsa's* is the problem of deciding "$L(M_1) = L(M_2)$" where M_1 and M_2 are 2-dfsa's.

3. The simulating machine. In this section we develop the main ideas to be used in our algorithm for deciding equivalence of 2-dfsa's. The construction is illustrated by first describing a less efficient mechanism for deciding whether "$L(p) = L(q)$", for p and q two states in an arbitrary 2-dfsa M. In § 4 we build upon this construction to obtain the desired polynomial time algorithm for deciding equivalence. But first, our strategy is to simulate the two computations of M by a single 2-dfsa whose finite tate control encodes the information needed to carry out this simulation. This simulator rejects all input tape pairs if and only if $L(p) = L(q)$. As expected, our method uses some of the ideas first developed by Valiant [8] and later modified by Beeri [2].

Let $M = (K_1, K_2, \Sigma, \delta, q_0, F)$ be any 2-dfsa and p, q be any two states in $K_1 \cup K_2$. We would like to construct a simulating machine $S(M, p, q)$ to simulate at the same time two computations of M on the same input tape pair, one from state p and another from state q, and accept if and only if $L(p) \neq L(q)$. This will provide the required algorithm because emptiness is decidable for 2-dfsa [6].

Consider the operation of a "naive" simulator on input tape pair (xu, yv). If

$$p\text{—}(x, yv) \to r$$

and

$$q\text{—}(xu, y) \to s$$

with $|u| = |v|$ (both computations have read the same number of input symbols, namely $|x| + |yv| = |xu| + |y|$), then the naive simulator might record this in its finite state control as

$$[(r, u, e), (s, e, v)].$$

State $[(r, u, e), (s, e, v)]$ means that the simulated computation continuing from state r, hereafter called the LEFT computation, must simulate reading the string of stored symbols u that the other computation, called the RIGHT computation, has "read ahead" on tape 1; similarly, the RIGHT computation must simulate processing the string v on tape 2 that the LEFT computation has read ahead and stored. Therefore, this state represents testing whether $(u, e)\backslash L(r) = (e, v)\backslash L(s)$. But observe that the two computations being simulated may operate on the tapes in quite different manners. For example, the LEFT computation might search down tape 1 until it reads some designated symbol before starting to read tape 2, whereas the RIGHT one might search tape 2 first. Thus, the naive simulator discussed above will not work in general, since there is no a priori bound on the length of the strings u or v that it would need to store in state $[(r, u, e), (s, e, v)]$.

The first modification that we discuss is called the "semi-naive" simulator. This simulator keeps the two computations in synchronism on tape 1, but allows one of the computations to read ahead on tape 2 (and store the symbols read) while the other is waiting to read tape 1. For example, if

$$p\text{---}(x, yv) \to r$$

and

$$q\text{---}(x, y) \to s$$

for s in K_1, then the semi-naive simulator reads input (x, yv) and records this in its finite state control as

$$[(r, e), (s, v)].$$

State $[(r, e), (s, v)]$ means that the LEFT computation has read ahead on tape 2 and stored the string v of symbols read for the RIGHT computation to operate on later. Hence, this state represents testing whether $L(r) = (e, v)\backslash L(s)$. The LEFT computation keeps reading symbols from tape 2 and storing them until it needs to read tape 1. At that time, both the LEFT and RIGHT computations read a symbol from that tape. As before the RIGHT computation must perform actions on the stored string before processing real symbols from tape 2. We do not consider the problem of identifying when the RIGHT computation reads ahead on tape 2 instead of the LEFT one. This is not important to the discussion at hand. Even without these details, however, we can see that our semi-naive simulator is no better than the other; there is no bound on the length of v.

How do we get around this problem? Consider a computation of the semi-naive simulator that reaches a state

$$[(r, e), (s, v)]$$

for some r in K_2, s in K_1, after having previously reached a state

$$[(r, e), (t, w)]$$

for some t in K_1, and where either $t \neq s$ or $w \neq v$. The simulating machine $S(M, p, q)$ that we shall construct is able to recognize this situation. When in such a state $[(r, e), (s, v)]$, it no longer continues the simultaneous computations from states r and s, testing whether $L(r) = (e, v)\backslash L(s)$. Instead, it changes to a state of the form

$$[(t, w), (s, v)]$$

to test whether $(e, w)\backslash L(t) = (e, v)\backslash L(s)$. We call such an action a REPLACEMENT.

How can we justify this? Lemma 3.2(3) and Theorem 3.3 guarantee that such a state change is fail-safe. At the time such a change would occur, $S(M, p, q)$ is testing whether $L(r) = (e, v)\backslash L(s)$. If tape pair (x, y) distinguishes $L(r)$ and $(e, v)\backslash L(s)$, then either (x, y) distinguishes $L(r)$ and $(e, w)\backslash L(t)$, or (x, y) distinguishes $(e, v)\backslash L(s)$ and $(e, w)\backslash L(t)$. On the other hand, if $L(p) = L(q)$, then $(e, v)\backslash L(s) = L(r) = (e, w)\backslash L(t)$.

After such a REPLACEMENT has occurred, the LEFT computation must simulate reading stored string w before reading "real" input symbols on tape 2, and the RIGHT computation must simulate reading stored string v before reading "real" symbols on tape 2. Both computations keep reading real symbols from tape 1 in synchronism, as necessary. This process continues until one of the computations has read all of its stored symbols. Let us say that $S(M, p, q)$ gets into a state of the form

$$[(t', e), (s', v')]$$

for v' a suffix of v. Here, only the RIGHT computation still has stored information remaining, so now we can continue the operation as discussed before the REPLACEMENT.

But how do we handle the case where

$$[(t', w'), (s', e)]$$

occurs for w' a suffix of w, so that only the LEFT computation still has stored information? Since we are testing whether $(e, w')\backslash L(t') = (e, e)\backslash L(s')$, we can reverse the ordering of the computations and record this as

$$[(s', e), (t', w')].$$

So now, only the RIGHT computation has stored string w' it must preprocess, and the operation of the simulator can continue as before the REPLACEMENT.

There is one problem remaining with the implementation of the simulating machine $S(M, p, q)$. It must have some mechanism for remembering previously reached states in order to perform the REPLACEMENT action from state $[(r, e), (s, v)]$ to state $[(t, w), (s, v)]$. We shall see that this can be done with a finite amount of memory. We shall discuss the states as though they were composed of two segments. One segment, called CURRENT, has the form of the states we have been describing; e.g., $[(r, e), (s, v)]$ and $[(t, w), (s, v)]$. The other segment, called HISTORY, will contain the relevant past history of states visited by $S(M, p, q)$ which enables it to determine proper REPLACEMENTs when necessary. The actual structure of the HISTORY segment will be shown later.

The construction of simulating 2-dfsa $S(M, p, q)$ is sketched below. The operation follows the outline above until it recognizes that one of the computations would accept an input tape pair and the other would not accept this pair. When this occurs, $S(M, p, q)$ accepts, and this is the only situation in which $S(M, p, q)$ can accept an input tape pair. Thus, we can see that $L(p) = L(q)$ if and only if $L(S(M, p, q)) = \varnothing$.

Let $M = (K_1, K_2, \Sigma, \delta, q_0, F)$ be a 2-dfsa. Let $K = K_1 \cup K_2$, and let p, q be any two states in K. Simulating machine $S(M, p, q)$ will have a special state ACCEPT, which is the only final state, that it enters when it knows that $L(p) \neq L(q)$. Once in state ACCEPT the machine remains there, reading symbols from tape 1.

Machine $S(M, p, q)$ also has a set of nonaccepting states having two segments [HISTORY, CURRENT]. We call these latter types of states SIMULATING states.

The CURRENT segment of a SIMULATING state is of the form

$$[(r, u), (s, v)]$$

for r, s in K, u, v in Σ^* with $0 \leqq |u|, |v| \leqq |K_2|$. We call u the LEFT CURRENT word and v the RIGHT CURRENT word.

The HISTORY segment of a SIMULATING state is an n-tuple, $n = |K_2|$, recording the relevant past history of the LEFT and RIGHT computations, which enables it to determine proper REPLACEMENTS. Each component of this n-tuple is associated with a state in K_2, say r; we call this the r component of the HISTORY segment and denote it by HISTORY(r).

The r component of HISTORY has the form either

$$\text{NIL}$$

or

$$(s, w)$$

for w in Σ^*, $0 \leqq |w| < |K_2|$, and s in K_1.

When HISTORY(r) is NIL, then simulating machine $S(M, p, q)$ has never been in a state with CURRENT segment of the form $[(r, e), (t, u)]$, for any state t in K_1 or string u. Otherwise, when the r component is (s, w), then machine $S(M, p, q)$ has previously been in a state with CURRENT segment $[(r, e), (s, w)]$. Thus, HISTORY(r) records a string that the LEFT computation in state r has read ahead of the RIGHT computation on tape 2; HISTORY also records the state in K_1 that the RIGHT computation was in at that time.

Machine $S(M, p, q)$ starts in the SIMULATING state with CURRENT segment $[(p, e), (q, e)]$, and HISTORY segment having NIL for each component. This corresponds to the initial situation where both computations are in synchronism, with the LEFT in state p, the RIGHT in state q, and no past HISTORY.

Now we shall consider the operation of $S(M, p, q)$ in a SIMULATING state. To facilitate the presentation, we classify the action of a SIMULATING state as being one of two basic types: READ and PREPARATION. When $S(M, p, q)$ performs a READ type action, it simulates a single step of the LEFT computation on an input symbol of the tape indicated; a step of the RIGHT computation is also simulated on this symbol when the state of the RIGHT computation reads the same tape as the LEFT and no stored symbols remain to be processed on that tape. Before $S(M, p, q)$ can perform another READ type action, it may make several PREPARATION type actions. A PREPARATION type action adjusts the CURRENT and HISTORY segments, performs REPLACEMENTs and other actions necessary to put the simulation in proper form for performing another READ type action. We relax our definition of a 2-dfsa to allow PREPARATION type actions of $S(M, p, q)$ not to read any symbols from the input tapes. This will not affect our results as the machine will remain deterministic. When a rule is not defined for a SIMULATING state, then it is because this state is not accessible; the transition from such a state is irrelevant.

I. PREPARATION *type actions.* Suppose $S(M, p, q)$ attempts to execute a PREPARATION type rule when it has CURRENT segment

$$[(r, u), (s, v)]$$

where $0 \leqq |u|, |v| \leqq |K_2|$. $S(M, p, q)$ examines the possibilities in the order indicated below. If the CURRENT segment does not meet the requirements of any of the actions, it performs a READ type action, as specified in II below. Otherwise, PREPARATION type actions are performed on input (e, e). After each such action,

$S(M, p, q)$ tries to execute another PREPARATION type action (except when it goes to ACCEPT after action 4).

(1) UNPACK LEFT

If r is in K_2 and $u = au'$ for some a in Σ, then the CURRENT segment is changed to

$$[(\delta(r, a), u'), (s, v)].$$

(2) UNPACK RIGHT

If s is in K_2 and $v = av'$ for some a in Σ, then the CURRENT segment is changed to

$$[(r, u), (\delta(s, a), v')].$$

(3) SWITCH

If r is in K_1 and either (a) $u \neq e = v$, or (b) s is in K_2 and $u = e = v$, then the CURRENT segment is changed to

$$[(s, e), (r, u)].$$

(4) ACCEPT

Go to ACCEPT in the following two situations where there is some w such that (w, e) distinguishes $(e, u)\backslash L(r)$ and $(e, v)\backslash L(s)$:

(i) $w = e$ and (e, e) distinguishes $(e, u)\backslash L(r)$ and $(e, v)\backslash L(s)$,

(ii) $w \neq e$, state r is in K_2, $u = e$, state s is in K_1, and (w, v) is in $L(s)$.

(5) HISTORY-RECORDING

If r is in K_2, s is in K_1, $u = e$, and HISTORY $(r) =$ NIL, then change this HISTORY component to (s, v).

(6) REPLACEMENT

If r is in K_2, s is in K_1, and HISTORY $(r) = (t, x) \neq (s, v)$, then the CURRENT segment is changed to

$$[(t, x), (s, v)].$$

II. READ *type actions.* Simulating machine $S(M, p, q)$ performs a READ type action when the CURRENT segment does not satisfy the requirements for any PREPARATION type action.

During a READ type action, $S(M, p, q)$ simulates a 1-step computation of both the LEFT and RIGHT computations on the same input tape whenever possible. Otherwise, it simulates a 1-step computation of just the LEFT computation. After any such action, $S(M, p, q)$ performs as many PREPARATION type actions as possible.

There are three cases.

(1) BOTH READ TAPE 1

If r and s are both in K_1, then $S(M, p, q)$ reads a symbol, say a, from tape 1 and changes the CURRENT segment to

$$[(\delta(r, a), u), (\delta(s, a), v)].$$

(2) BOTH READ TAPE 2

If r and s are both in K_2 and $u = v = e$, then $S(M, p, q)$ reads a symbol, say a, from tape 2 and changes the CURRENT segment to

$$[(\delta(r, a), e), (\delta(s, a), e)].$$

(3) READ-AHEAD

If r is in K_2, s is in K_1, and $u = e$, then $S(M, p, q)$ reads a symbol, say a, from tape 2 and changes the CURRENT segment to

$$[(\delta(r, a), e), (s, va)].$$

This concludes the construction of $S(M, p, q)$. The next theorem shows that $S(M, p, q)$ is well defined. In particular, there exists a bound on the lengths of the strings stored in the CURRENT and HISTORY segments of a SIMULATION state. Because machine $S(M, p, q)$ is constructed only to illustrate the ideas behind the algorithm of the next section, we omit the proof.

THEOREM 3.1. *Let S be an accessible* SIMULATING *state of $S(M, p, q)$ with* CURRENT *segment* $[(r, u), (s, v)]$. *The following statements hold.*

(1) $|u| < |K_2|$.

(2) $|v| \leqq |K_2|$.

(3) *For each t in K_2, if* HISTORY (t) *has the form* (t', w), *then*

$$|w| < |K_2|.$$

The next lemma gives important properties of $S(M, p, q)$ needed to establish our desired result, that $L(S(M, p, q)) = \varnothing$ if and only if $L(p) = L(q)$. Each part verifies the validity of a rule of the simulating machine. The proof is straightforward and is therefore omitted.

LEMMA 3.2.

(1) UNPACKING TAPE 1

If r is in K_1 and $r—(a, e) \to r'$ for a in Σ, then for any u, v in Σ^*,

$$(au, v)\backslash L(r) = (u, v)\backslash L(r').$$

(2) UNPACKING TAPE 2

If s is in K_2 and $s—(e, a) \to s'$ for a in Σ, then for any u, v in Σ^*,

$$(u, av)\backslash L(s) = (u, v)\backslash L(s').$$

(3) REPLACEMENT

If r is in K_2, s, t are in K_1, u, v are in Σ^*, then

$$L(r) = (e, u)\backslash L(s) \text{ and } L(r) = (e, v)\backslash L(t) \text{ if and only if}$$

$$L(r) = (e, u)\backslash L(s) \text{ and } (e, u)\backslash L(s) = (e, v)\backslash L(t).$$

(4) STRAIGHT SIMULATION

(A) TAPE 1

If r and s are in K_1, then for all u, v in Σ^*, $(e, u)\backslash L(r) = (e, v)\backslash L(s)$ if and only if

(a) *(e, e) does not distinguish $(e, u)\backslash L(r)$ and $(e, v)\backslash L(s)$*

and

(b) *for all a in Σ, $(e, u)\backslash L(\delta(r, a)) = (e, v)\backslash L(\delta(s, a))$,*

(B) TAPE 2

If r and s are in K_2, then $L(r) = L(s)$ if and only if

(a) *(e, e) does not distinguish $L(r)$ and $L(s)$*

and

(b) *for all a in Σ, $L(\delta(r, a)) = L(\ \ (s, a))$.*

(5) READ-AHEAD SIMULATION

If r is in K_2, *s is in* K_1, *and v is in* Σ^*, *then* $L(r) = (e, v)\backslash L(s)$ *if and only if*

(A) (e, e) *does not distinguish* $L(r)$ *and* $(e, v)\backslash L(s)$,

(B) *for all u in* Σ^+, (u, v) *is not in* $L(s)$

and

(C) *for all a in* Σ, $L(\delta(r, a)) = (e, va)\backslash L(s)$.

We claim that if p and q are equivalent, then $S(M, p, q)$ never enters state ACCEPT for any input pair. Moreover, if p is not equivalent to q, then there is some input pair which takes $S(M, p, q)$ to state ACCEPT. This is captured in Theorem 3.3 below. Again, because $S(M, p, q)$ will not be used in our final algorithm, we omit the proof.

THEOREM 3.3. *Let* $M = (K_1, K_2, \Sigma, \delta, q_0, F)$ *be a 2-dfsa, let p, q be any two states in* $K_1 \cup K_2$, *and let* $S(M, p, q)$ *be constructed as described earlier. Then*

$$L(p) = L(q)$$

if and only if

$$L(S(M, p, q)) = \varnothing.$$

4. The equivalence algorithm. The SIMULATING machine $S(M, p, q)$ described in the last section provides an exponential time algorithm for testing equivalence of states in a 2-dfsa. In this section we improve this bound by the following argument.

Notice first that if S and S' are two accessible SIMULATING states, with respective CURRENT segments $[(r, e), (s, v)]$ and $[(r, e), (s', v')]$, then it would be valid to use (s, v) for a REPLACEMENT for S' into a state with CURRENT segment $[(s, v), (s', v')]$; S' need not be accessible from S. Second, a careful study of $S(M, p, q)$ would show that if $L(p) \neq L(q)$, then there is a path from S_0 to ACCEPT during which all states have distinct CURRENT segments. This suggests building the accessible submachine of $S(M, p, q)$ and simultaneously testing for emptiness, i.e., whether or not ACCEPT is accessible from S_0. Only the CURRENT segments of the SIMULATING states are stored, while a uniform HISTORY (the *same* for all SIMULATING states) is constructed with space for exactly $|K_2|$ entries. (In the notation of Valiant [9] and Beeri [2], the uniform HISTORY is a REPLACEMENT function constructed along with $S(M, p, q)$.) When a new accessible state S with CURRENT segment $[(r, e), (s, v)]$ is found, the r component of HISTORY is set to (s, v) if previously NIL; if the r component of HISTORY is $(t, u) \neq (s, v)$, then a SIMULATING state with CURRENT segment $[(t, u), (s, v)]$ is constructed. An overall list of CURRENT segments of accessible states is kept along with a pushdown store of CURRENT segments of accessible but unexamined states. Essentially the algorithm performs a depth-first search of accessible states looking for ACCEPT. If ACCEPT appears on the overall list, $L(p) \neq L(q)$; if the pushdown store is emptied before ACCEPT appears, then $L(p) = L(q)$. The details of the algorithm are presented below.

To determine whether two states in a 2-dfsa are equivalent, we apply procedure EQUIVALENCE to 2-dfsa $M = (K_1, K_2, \Sigma, \delta, q_0, F)$ and states p, q in $K_1 \cup K_2$, which are inputs to the program. Eventually EQUIVALENCE, or one of the procedures that it invokes, will HALT and print either "$L(p) = L(q)$" or "$L(p) \neq L(q)$", as appropriate. EQUIVALENCE and all other procedures in this section are written in a dialect of Pidgin ALGOL; the reader is referred to [1] for more details.

Three global data structures are central to the operation of the algorithm—LIST, STACK, and HISTORY. LIST is a list of CURRENT segments of states in machine

$S(M, p, q)$ that the algorithm has found to be accessible. Stack (i.e., pushdown store) structure STACK holds those elements of LIST whose immediate successors in $S(M, p, q)$ have not yet been examined. HISTORY is a $|K_2|$ element vector holding the relevant HISTORY components. To begin, both STACK and LIST contain the single element $[(p, e), (q, e)]$; each component of HISTORY is NIL.

EQUIVALENCE searches through the elements in STACK looking for ACCEPT. If STACK empties without finding ACCEPT to be accessible, then EQUIVALENCE halts and prints "$L(p)=L(q)$". Otherwise, EQUIVALENCE removes the elements from STACK, one by one, from the top. Suppose the topmost element is $S=[(r, u), (s, v)]$. Procedure ACCEPT(S) is invoked to determine if $S(M, p, q)$ would enter state ACCEPT (PREPARATION rule of type (4). If ACCEPT(S) returns value "YES", then EQUIVALENCE halts and prints "$L(p) \neq L(q)$". If the value returned is "NO", then the algorithm continues by placing those CURRENT segments in $S(M, p, q)$ that are immediate successors of S onto the STACK for later examination by EQUIVALENCE. This is accomplished by procedure EXAMINE(S) and the subroutines that it invokes.

First, procedure EXAMINE performs a HISTORY-RECORDING action (PREPARATION rule of type 5) when necessary. Second, if $S(M, p, q)$ could perform a READ type action when having CURRENT segment S, procedure EQUIVALENCE invokes subroutine NEXT. NEXT finds all possible CURRENT segments resulting from a single READ move (for each symbol in Σ), and stores them in both LIST and STACK if they are not already in LIST, that is they have not previously been found accessible. Otherwise, when S indicates that $S(M, p, q)$ could not perform a READ action, EXAMINE calls subroutine PREPARATION to see if either an UNPACK, SWITCH or REPLACEMENT type action would be indicated by the SIMULATING machine. The resulting CURRENT segment from such a move is placed on both STACK and LIST, if not already in LIST. The detailed definitions of procedures EQUIVALENCE, ACCEPT, NEXT, EXAMINE, and PREPARATION follow.

```
procedure EQUIVALENCE:
begin
    STACK ← [(p, e), (q, e)];
    LIST ← [(p, e), (q, e)];
    HISTORY ← NIL;
    while STACK not empty do
        begin
            S ← top element of STACK;
            pop STACK;
            if ACCEPT(S) = "YES" then
                begin
                    print "L(p) ≠ L(q)";
                    halt
                end
            else
                EXAMINE(S)
        end
    print "L(p) = L(q)";
    halt
end
```

```
procedure ACCEPT(S):
being
    comment Suppose that S has form [(r, u), (s, v)];
    if (e, e) distinguishes (e, u)\L(r) and (e, v)\L(s) then
        return ("YES")
    else
        if u = e, r is in K2, s is in K1,
            and there exists some w ≠ e such that
            (w, v) is in L(s) then
            return ("YES")
        else
            return ("NO")
end

procedure EXAMINE(S):
begin
    comment Suppose that S has form [(r, u), (s, v)];
    if r is in K2, s is in K1, u = e
        and HISTORY (r) = NIL, then
            HISTORY (r) ← (s, v);
    if r and s are in K1, or
        r and s are in K2, u = v = e, or
        r is in K2, s is in K1, u = e, HISTORY (r) = (s, v) then
            NEXT(S)
    else
            PREPARATION(S)
end

procedure NEXT(S):
begin
    comment Suppose that S has form [(r, u), (s, v)];
    for each a in Σ do
        begin
            comment BOTH READ TAPE 1;
            if r, s are in K1 then
                Sa ← [(δ(r, a), u), (δ(s, a), v)];
            comment BOTH READ TAPE 2;
            if r, s are in K2, u = v = e then
                Sa ← [(δ(r, a), e), (δ(s, a), e)];
            comment READ-AHEAD;
            if r is in K2, s is in K1, u = e, then
                Sa ← [(δ(r, a), e), (s, va)];
            comment ADD NEW ELEMENT TO STACK AND LIST;
            if Sa is not in LIST then
                begin
                    add Sa to LIST;
                    push Sa onto STACK
                end
        end
end
```

```
procedure PREPARATION(S):
begin
    comment Suppose that S has form [(r, u), (s, v)];
    comment UNPACK LEFT or RIGHT;
    if r is in K2, u = au', a is in Σ then
        SP ← [(δ(r, a), u'), (s, v)]
    else
        if s is in K2, v = av', a is in Σ then
            SP ← [(r, u), (δ(s, a), v')];
    comment SWITCH;
        if r is in K1, v = e, and either u ≠ e or s is in K2 then
            SP ← [(s, e), (r, u)];
    comment REPLACEMENT;
    if r is in K2, s is in K1, u = e,
        and HISTORY (r) = (s', v') ≠ (s, v) then
            SP ← [(s',v'), (s, v)];
    if SP is not on LIST then
        begin
            add SP to LIST;
            push SP onto STACK;
        end
end
```

This concludes the definition of our algorithm for determining equivalence of states in a 2-dfsa. The next several lemmas establish that our procedure is well-defined, in particular, that the lengths of the strings stored in elements of LIST, STACK, and HISTORY are bounded. We shall then show that the algorithm always halts, and gives the correct answer as to whether or not $L(p)=L(q)$ in time $O(n^{12})$, where n is the size of machine M. First we need some notation.

Let I denote the number of input symbols in M, so $I=|\Sigma|$. At time t during the execution of the algorithm we have the following values defined.

$$\text{HISTORYSIZE} = |\{r \mid \text{HISTORY}(r) \neq \text{NIL}\}|,$$
$$\text{HISTORYWORD} = \{v \mid \text{HISTORY}(r) = (s, v)\},$$
$$\text{HISTORYWORDSIZE} = \begin{cases} \text{MAX}\{|v| \mid v \text{ is in HISTORYWORD}\}, & \text{if HISTORYWORD} \neq \varnothing \\ -1, & \text{if HISTORYWORD} = \varnothing, \end{cases}$$
$$\text{SUFFIX} = \{x \mid x \text{ is a suffix of some word in HISTORYWORD}\},$$
$$\text{SUFFIXPLUS} = \{xa \mid x \text{ is in SUFFIX}, a \text{ is in } \Sigma\}.$$

HISTORYSIZE is the number of the components of the HISTORY vector that are not NIL at time t; HISTORYWORD contains all nonNIL strings stored in HISTORY and HISTORYWORDSIZE gives the maximum length of any such string. SUFFIX contains all suffixes of words stored in HISTORY, whereas words in SUFFIXPLUS have an additional input symbol concatenated on the right. We make the following observations:

$$|\text{SUFFIX}| \leqq (\text{HISTORYSIZE})(\text{HISTORYWORDSIZE}+1),$$
$$|\text{SUFFIXPLUS}| \leqq (I)(|\text{SUFFIX}|) \leqq (I)(\text{HISTORYSIZE})(\text{HISTORYWORDSIZE}+1),$$
$$\text{HISTORYSIZE} \leqq |K_2|.$$

The following lemma is immediate from the comments above.

LEMMA 4.1. *Upon any call of* EXAMINE(S) *from procedure* EQUIVALENCE, *the following conditions must hold.*

(1) $S_1 \neq S$ *is in* LIST *but not in* STACK *if and only if* EXAMINE(S_1) *has been called previously.*

(2) *If* HISTORY $(r) = (s, v)$, *then* $[(r, e), (s, v)]$ *is in* LIST *but not in* STACK; *so* $[(r, e), (s, v)]$ *was previously on the top of* STACK.

(3) S *has never been on top of* STACK *before.*

We can now state the Bounding Lemma needed to establish bounds on the size of the words stored in the data structures.

LEMMA 4.2 (Bounding lemma). *Upon any call of* EXAMINE(S) *from procedure* EQUIVALENCE, *the following two conditions hold.*

(1) HISTORYWORDSIZE < HISTORYSIZE.

(2) *If* $[(r, u), (s, v)]$ *appears in* LIST, *then either* $u = v = e$ *or both* u *is in* SUFFIX *and* v *is in* SUFFIX ∪ SUFFIXPLUS.

Proof. We proceed by induction on the number of times that EXAMINE has been called. Both (1) and (2) hold trivially initially.

Suppose that (1) and (2) hold upon the current call of EXAMINE (S), and $S = [(r, u), (s, v)]$. It suffices to show that (1) and (2) hold after the execution of EXAMINE (S), since when the ACCEPT subroutine returns value "YES" the algorithm terminates.

Either NEXT (S) or PREPARATION (S) is executed. Suppose that NEXT (S) is executed. Then for each a in Σ, a value of S_a is determined. There are two cases. If both r and s are in K_1 or both r and s are in K_2, then S_a is of the form $[(r', u), (s', v)]$, so (2) holds whether or not S_a is added to LIST. Also, HISTORY is unaffected so (1) still holds. If r is in K_2, s is in K_1, $u = e$ and at the time NEXT (S) is executed HISTORY $(r) = (s, v)$, then $S_a = [(\delta(r, s), e), (s, va)]$. Now if HISTORY (r) had been (s, v) at the start of EXAMINE (S), then Lemma 4.1(2) insures that $[(r, e), (s, v)] = S$ would have been on the top of STACK previously, a contradiction to Lemma 4.1(3). So at the start of EXAMINE (S), HISTORY (r) was NIL and it was reset to (s, v). Thus HISTORYSIZE is increased by 1. At the start of EXAMINE (S), v is in SUFFIX ∪ SUFFIXPLUS ∪ $\{e\}$, so $|v| \leqq$ HISTORYWORDSIZE + 1. Hence HISTORYWORDSIZE is either unaffected or increased by 1. So after EXAMINE (S) is executed, (1) holds and since v is now in HISTORY, va is in SUFFIXPLUS, and (2) holds also.

On the other hand, suppose PREPARATION (S) is executed. If upon call of EXAMINE (S), r is in K_2, s is in K_1, $u = e$, and HISTORY (r) = NIL, then NEXT (S) would have been executed. Hence HISTORY is unaffected by PREPARATION (S), so (1) holds afterwards. There are four cases. If an UNPACK is applied, S' is of the form $[(r', u'), (s', v')]$, where u' is a suffix of u and v' is a suffix of v, so (2) holds for S'. If SWITCH is applied, $v = e$ and $S' = [(s, e), (r, u)]$, u in SUFFIX, so again (2) holds for S'. In the remaining case (REPLACEMENT), r is in K_2, s is in K_1, $u = e$, HISTORY $(r) = (s', v') \neq (s, v)$, and $S' = [(s', v'), (s, v)]$. We still have v in SUFFIX ∪ SUFFIXPLUS. By definition, v' is in HISTORYWORD ⊆ SUFFIX. Hence (2) holds after PREPARATION (S) in this case too. □

Now we can establish upper bounds on the lengths of the words stored in entries of LIST, STACK and HISTORY.

THEOREM 4.3. *The following conditions must hold throughout the execution of the algorithm.*

(1) *If* HISTORY $(r) = (s, w)$, *then* $|w| < |K_2|$.

(2) *If* $[(r', u), (s', v)]$ *appears in either* LIST *or* STACK, *then* $|u| < |K_2|$ and $|v| \leqq |K_2|$.

Proof. Statement (1) follows from the bounding lemma and the fact that HISTORYSIZE $\leqq |K_2|$. Then (2) follows from (1) since either $u = v = e$ or $|u| \leqq$ HISTORYWORDSIZE and $|v| \leqq$ HISTORYWORDSIZE $+ 1$. □

We shall now establish an upper bound on the time needed to execute the algorithm. Observe that before subroutine NEXT or PREPARATION can add an entry to LIST, a search must first be made through LIST to determine whether this entry was previously added. The length of LIST clearly affects the timing of the algorithm, and the next theorem provides a bound on this length.

THEOREM 4.4. *Let* $I = |\Sigma|$, $k = |K_2|$, $N = |K_1| + |K_2|$. *Then the number of entries in* LIST *is bounded by*

$$(I+1)N^2k^2(k+1)^2 \leqq (I+1)N^6.$$

Proof. By the bounding lemma (1), throughout execution of the algorithm

$$\text{HISTORYWORDSIZE} < k$$

and

$$\text{HISTORYSIZE} \leqq k,$$

so

$$|\text{SUFFIX}| \leqq k^2$$

and

$$|\text{SUFFIXPLUS}| \leqq Ik^2.$$

Any entry in LIST is of the form $[(r, u), (s, v)]$, with r, s in $K_1 \cup K_2$. By the bounding lemma, either $u = v = e$, or u is in SUFFIX and v is in SUFFIX $\cup$ SUFFIXPLUS. There are N possibilities each for r and s, k^2 possibilities for u, and $(I+1)k^2$ possibilities for v, yielding a bound of

$$N^2(I+1)k^4$$

which is bounded by

$$(I+1)N^6.$$

COROLLARY 4.5. *The time complexity of the algorithm is*

$$O(I^3N^{12})$$

or

$$O(n^{12})$$

taking machine size as $n = IN$ (= *size of the transition table*).

Proof. First consider the time needed to execute procedure ACCEPT (S), where $S = [(r, u), (s, v)]$. The test for whether pair (e, e) distinguishes $(e, u)\backslash L(r)$ and $(e, v)\backslash L(s)$ can clearly be done in time $O(|u| + |v|)$. On the other hand, the **else** clause is applicable if and only if $S = [(r, e), (s, v)]$ for some r in K_2, s in K_1, $0 \leqq |v| \leqq |K_2|$ and $(\Sigma^+ \times \{v\}) \cap L(s) = \varnothing$. This test can be completed within time $O(IN(|v|+1))$. Therefore, ACCEPT (S) has time complexity $O(1) + O(|u| + |v|) + O(IN(|v|+1))$, and since the bounding lemma shows that $|u| + |v| \leqq 2N - 1$, the time complexity is $O(IN^2)$.

Next we consider procedures PREPARATION (S) and NEXT (S). The time to execute PREPARATION quickly becomes dominated by the time required to search through LIST. Since the size of LIST is bounded by $(I+1)N^6$, the time needed to complete NEXT is $O(1)+O((I+1)N^6) \leqq O((I+1)N^6)$. Similar arguments show that each iteration of the **for** loop in NEXT requires this same amount of time, and since the **for** loop is executed I times, the time needed to complete NEXT is $O(I(I+1)N^6) \leqq O(I^2N^6)$.

Procedure EXAMINE uses some fixed amount of time to set HISTORY, when necessary, and then the time to execute either NEXT or PREPARATION. So EXAMINE has time bound $O(I^2N^6)$.

Finally we come to procedure EQUIVALENCE. Every execution of the **while** loop removes an entry from STACK, and no two executions can have the same topmost entry. Since each entry in STACK must also be in LIST, and there are at most $(I+1)N^6$ elements in LIST, the **while** loop can be executed at most $(I+1)N^6$ times. Each such execution may call both ACCEPT and EXAMINE. So the time to execute EQUIVALENCE is bounded by $O(1)+(I+1)(O(IN^2)+O(I^2N^6)) \leq O(I^3N^{12})$. □

We wish to show that if $L(p)=L(q)$, then the algorithm never halts and prints "$L(p) \neq L(q)$". Lemma 4.6 below establishes this claim.

Let $S_0 = [(p, e), (q, e)]$. Define

$$\text{EQUIV} = \{[(r, u), (s, v)] \mid (e, u)\backslash L(r) = (e, v)\backslash L(s)\}.$$

Observe that S_0 is in EQUIV if and only if $L(p)=L(q)$.

LEMMA 4.6 (Preserve equivalence). *If S_0 is in* EQUIV, *then every S which appears on* LIST *is in* EQUIV.

Proof. We proceed by induction on the numer of times the **while** loop in procedure EQUIVALENCE is executed. The lemma holds trivially initially.

Suppose that the lemma holds at the start of the current execution of the loop. It suffices to show that if S is on top of STACK, then ACCEPT (S) returns "NO" and any entry added to STACK (and hence LIST) by subroutine NEXT or PREPARATION is in EQUIV.

Let $S = [(r, u), (s, v)]$. ACCEPT (S) returns "YES" if either (e, e) distinguishes $(e, u)\backslash L(r)$ and $(e, v)\backslash L(s)$ or $u = e$, r is in K_2, s is in K_1, and for some $w \neq e$, (w, v) is in $L(s)$. In the first case S is obviously not in EQUIV. In the second case, (w, e) is in $(e, v)\backslash L(s)$ but not in $(u, e)\backslash L(r) = L(r)$ (since r is in K_2 and $w \neq e$), so S is not in EQUIV. Hence if S is in EQUIV, ACCEPT (S) returns value "NO".

Case analysis shows easily that if S is in EQUIV and NEXT (S) adds S_a to STACK, then S_a is in EQUIV. Suppose PREPARATION (S) applies, and adds S' to STACK. IF an UNPACK or SWITCH applies, S' codes the same sets as S and so S' is in EQUIV.

Consider when a REPLACEMENT applies. In this case $S' = [(r, e), (s, v)]$, and HISTORY $(r) = (s', v')$. Since S is in EQUIV, $L(r) = (e, v)\backslash L(s)$. At a previous time, $S'' = [(r, e), (s', v')]$ was on top of STACK. So if all members of LIST are in EQUIV, $L(r) = (e, v')\backslash L(s')$, so $(e, v)\backslash L(s) = (e, v')\backslash L(s')$. Hence S' is in EQUIV. This covers all cases. □

From Lemma 4.6 we can conclude that if p and q are equivalent, then the procedure EQUIVALENCE never halts and prints "$L(p) \neq L(q)$". But what if p and q are not equivalent? Lemma 4.7 below establishes the desired result: EQUIVALENCE eventually prints "$L(p) \neq L(q)$".

Given any element $S=[(r, u), (s, v)]$ in LIST or STACK, we say that pair (x, y) *distinguishes* S if (x, y) distinguishes sets $(e, u)\backslash L(r)$ and $(e, v)\backslash L(s)$. If (x, y) distinguishes S, then S is $(|(x, y)|)$-*distinguishable*. We say that (x, y) distinguishes LIST if it distinguishes some element in LIST; in such a case, LIST is $(|(x, y)|)$-distinguishable.

Consider the **while** loop located in procedure EQUIVALENCE. We write LIST_t to denote the value of LIST just prior to the tth iteration through the loop; similarly, STACK_t denotes the value of STACK and TOP_t the value of the topmost element of STACK_t at this time. When the tth execution of the loop finds value "YES" returned from procedure ACCEPT, then we say that $\text{LIST}_{t+1}=\text{ACCEPT}$.

LEMMA 4.7 (Find accept). *If* LIST_t *is n-distinguishable, then the following two statements hold.*

(1) *There exists some m for which either*

(A) $\text{LIST}_m=\text{ACCEPT}$

or

(B) LIST_m *is* $(n-1)$-*distinguishable.*

(2) *There exists some p with*

$$\text{LIST}_p=\text{ACCEPT}.$$

Proof. We first show (1) by examination of procedures NEXT and PREPARATION, and then obtain (2) from (1) by induction on n.

First consider (1). We can assume that TOP_t is n-distinguishable for the following reasons. Suppose TOP_t is not n-distinguishable, and S is some n-distinguishable entry in LIST_t. If S is in LIST_t but not STACK_t, then at the start of some previous execution of the **while** loop S was on top of STACK. Otherwise, if S is in STACK_t, then during future executions of the loop entries above S in STACK are popped off unless ACCEPT returns "YES" before this can happen.

Suppose $S=\text{TOP}_t=[(r, u), (s, v)]$ and (x, y) distinguishes S, $|x|+|y|=n$. If $n=0$, then ACCEPT (S) returns "YES", and if ACCEPT (S) returns "YES", then (A) holds. So assume that $n\geqq 1$ and that ACCEPT (S) returns "NO". First suppose that NEXT (S) is executed. There are three cases.

(i) r, s are both in K_1. Then $x=ax'$ for some a in Σ (or else (x, y) would be in neither $(e, u)\backslash L(r)$ nor $(e, v)\backslash L(s)$). Hence NEXT (S) finds value $S_a=[(\delta(r, a), u), (\delta(s, a), v)]$, and clearly (x', y) distinguishes S_a, with $|x'|+|y|=n-1$. Either S_a is already on LIST_t or is added to it by NEXT. So S_a is on LIST_{t+1}, and (B) holds.

(ii) r, s are both in K_2, and $u=v=e$. Then $y=ay'$ for some a in Σ. Hence NEXT (S) finds $S_a=[(\delta(r, a), e), (\delta(s, a), e)]$, which is distinguished by (x', y) and so is $(n-1)$-distinguishable. Thus S_a is on LIST_{t+1} and (B) holds.

(iii) r is in K_2, s is in K_1, $u=e$, and HISTORY $(r)=(s, v)$. If we had $y=e$, $x\neq e$, then we would have $(x, e)=(x, y)$ in $(e, v)\backslash L(s)$ (since (x, e) cannot be in $L(r)$ for r in K_2) and hence ACCEPT (S) would return "YES". So $y=ay'$, for some a in Σ. Now NEXT (S) finds $S_a=[(\delta(r, a), e), (s, va)]$ which is distinguished by (x, y') and so is $(n-1)$-distinguishable. Since S_a is on LIST_{t+1}, (B) holds.

Now we consider the cases in which PREPARATIONS (S) is executed.

(iv) First suppose that a REPLACEMENT rule applies. This means that r is in K_2, s is in K_1, $u=e\neq v$, HISTORY $(r)=(s', v')\neq(s, v)$, for s' in K_1. We have entry $S_1=[(r, e), (s', v')]$ also in LIST_t. So PREPARATION (S) finds value $S_P=[(s', v'), (s, v)]$, which it ensures is on LIST_{t+1}. There are two cases. Either (x, y) distinguishes $(e, v')\backslash L(s')$ and $(e, v)\backslash L(s)$ or (x, y) distinguishes $(e, v')\backslash L(s')$ and $L(r)$.

In the first case, (x, y) distinguishes S_P. Since S_P is in LIST_{t+1}, similar arguments to those seen earlier show that either S_P was at the top of STACK before some previous execution of the loop, or subsequent executions of the loop will make S_P the top element (unless ACCEPT returns "YES" during some execution, at which point (A) holds). So assume that $S_p = \mathrm{TOP}_m$ for some m. If ACCEPT (S_P) returns "YES", (A) holds; otherwise NEXT (S_P) is executed and the arguments of Case (i) apply to S_P (s, s' in K_1).

Now suppose on the contrary that (x, y) distinguishes $(e, v')\backslash L(s')$ and $L(r)$. Since HISTORY $(r) = (s', v')$, at some previous time $S_1 = [(r, e), (s', v')]$ was on top of STACK. So $S_1 = \mathrm{TOP}_m$ for some $m < t$. The arguments of Case (iii) apply to S_1 (r in K_2 and s' in K_1).

(v) Finally we consider the other cases in which PREPARATION (S) is executed. These cases (UNPACK or SWITCH) find a value for S_P encoding the same sets $\{(e, u)\backslash L(r), (e, v)\backslash L(s)\}$; thus (x, y) distinguishes S_P and ACCEPT (S_P) returns "NO". Arguments similar to those seen earlier show that unless ACCEPT returns "YES", there is some execution of the loop that has S_P on top of STACK. In the worst case, $|u|+|v|$ UNPACK rules, and possibly one SWITCH rule can apply. So PREPARATION rules other than REPLACEMENTS can apply at most $|u|+|v|+1$ times until one of Cases (i)–(iv) must occur and the appropriate arguments apply.

This establishes (1). Let us consider (2)—i.e., Case (A) of (1) always holds eventually. We proceed by induction on n.

If $n = 0$, then S is 0-distinguishable. Hence ACCEPT (S) returns "YES" and so (2) holds. Suppose (2) holds whenever $n' < n$, for $n \geqq 1$.

By (1), either (a) holds and thus (2), or else STACK_m is $(n-1)$-distinguishable for some m. By the induction hypothesis, (2) holds. □

Corollary 4.5 and Lemmas 4.6 and 4.7 yield the desired result.

THEOREM 4.8. *The equivalence problem for 2-dfsa's is decidable in time $O(n^{12})$, where n is the size of the machines involved.*

5. Remarks. The UNION-FIND algorithm has been used to improve the upper bound for determining equivalence of deterministic one tape finite state acceptors from $O(n^2)$ to $O(nG(n))$, where $G(n)$ grows very slowly [1]. We can use it in a similar fashion to improve the time complexity of our algorithm from $O(n^{12})$ to $O(n^4)$.

In a typical application of UNION-FIND, one starts with a collection of pairwise disjoint sets. An execution of UNION(A, A', B) joints the sets named A and A' and names the new set B. An execution of FIND(x) locates the name of the set to which element x currently belongs. For any constant c, another constant c' can be found (depending only on c) such that a series of up to cn UNION-FIND operations can be performed on n elements in time $c'nG(n)$. The reader is referred to [1] for details. We sketch briefly below the ideas behind the use of the UNION-FIND algorithm to improve our algorithm.

We wish to compute equivalence among m sets of the form $(e, v)\backslash L(r)$—henceforth denoted (r, v)—for v in SUFFIX ∪ SUFFIXPLUS, with $m \leqq (I+1)N^3$. Initially, we hypothesize that "$L(p) = L(q)$". If an entry $[(r, u), (s, v)]$ appears in our algorithm, that means that if $L(p) = L(q)$, then we must also have $(e, u)\backslash L(r) = (e, v)\backslash L(s)$. Equivalence is transitive so if we now hypothesize that "$(r, u) = (s, v)$", all the languages to which (r, u) or (s, v) is equivalent must be pairwise equivalent. The ACCEPT subroutine described above gives the conditions under which a proposed equivalence is contradictory and hence $L(p) \neq L(q)$. If all equivalences are established without finding a contradiction (ACCEPT does not return "YES"), then $L(p) = L(q)$.

The Improved Algorithm starts with each (r, v) in a set by itself. A STACK gives a list of proposed equivalences to test; initially it contains only $[(p, e), (q, e)]$ for the hypothesis "$L(p) = L(q)$". Suppose $S = [(r, u), (s, v)]$ is on the top of the STACK. It is removed and the following tasks are performed. If (e, e) distinguishes (r, u) and (s, v), then the algorithm HALTs with the answer "$L(p) \neq L(q)$". If r is in K_2, s is in K_1, $u = e$ and HISTORY (r) = NIL, then HISTORY (r) is changed to (s, v), and the algorithm determines whether $(e, v)\backslash L(s) \cap [\Sigma^+ \times \{e\}] = \varnothing$. If the answer is no, it HALTs with the answer "$L(p) \neq L(q)$". The next step in the algorithm has FIND((r, u)) and FIND((s, v)) locate the sets A and A' to which (r, u) and (s, v) belong. If $A = A'$, no further action is taken and the next element on the STACK is examined. If $A \neq A'$, UNION (A, A', A) joints A and A'. Then the appropriate NEXT or PREPARATION subroutine is executed as in the original algorithm. Either a sequence of I (for NEXT) entries or one entry (for PREPARATION) is made on the STACK. Again, when the STACK empties the algorithm HALTs with the answer "$L(p) = L(q)$".

Now the UNION operation can be performed at most $m - 1$ times, since the algorithm starts with m unit sets. Each execution of the UNION algorithm adds at most I entries to the STACK. Hence there are at most $I(m - 1)$ entries on the STACK altogether and so at most $I(m - 1)$ executions of the FIND operation. Hence this part of the algorithm has cost at most $O(ImG(m))$ or $O(I^2N^3G(IN))$ [1]. Each execution of UNION or FIND is associated with other subroutines. The NEXT or PREPARATION subroutines following a UNION can be assigned cost no more than $O(IN)$, for a total of $O(mIN)$ or $O(I^2N^4)$. Before each FIND operation, there is a part of the ACCEPT subroutine (does (e, e) distinguish (r, u) and (s, v)?) which can be assigned cost $O(N)$, for a total of $O(ImN)$ or $O(I^2N^4)$. Finally, in the up to $|K_2|$ steps which change HISTORY, the expensive part of the ACCEPT subroutine (does $(e, v)\backslash L(s)$ contain a word (w, e), $w \neq e$?) is performed. As we saw, this can be done in time $O(IN^2)$, for a total cost of $O(IN^3)$. The dominant term in this analysis in $O(I^2N^4)$, so altogether we get $O(n^4)$ for $n = IN$.

There is one difficulty with the algorithm outlined above. Initially we do not know which strings will be in HISTORY and thus in T = SUFFIX ∪ SUFFIXPLUS. Hence, we cannot predict which m sets will be merged eventually, and so we cannot foresee which m sets need initialization. However, we do know that strings in T will be bounded in length by k for $k = |K_2|$, and that for each i, $0 \leqq i \leqq k$, there will be at most MIN $(I^i, (I + 1)k)$ distinct strings of size i in T. So initially we set up an array WORD (i, j) for $0 \leqq i \leqq k$, $0 \leqq j \leqq$ Min $(I^i; (I + 1)k)$, with entries NIL (except for WORD $(0, 1) = e$). When a new entry in HISTORY appears, we update the appropriate entries WORD (i, j). This setup and updating can be considered to have a one-time cost of $O(I^2N^3)$ (there are up to N updates, each involves taking at most $(I + 1)N$ possible members of T and then determining whether a candidate v already fills one of the up to $(I + 1)N$ slots for WORD $(|v|, j)$). A language $(e, v)\backslash L(r)$ is denoted (r, i, j) where v = WORD (i, j). The updating precedes each READ-AHEAD step, so when $[(r, u), (s, v)]$ is to be added to STACK, u and v are already in the WORD array. Their proper identifiers (i.e., $i = |u|$, $i' = |v|$, j, j' with WORD $(i, j) = u$ and WORD $(i', j') = v$) can be found in time $O(IN)$. This cost can be assigned to a preceding UNION operation, and we have already assigned a cost of $O(IN)$ to the subroutines following each UNION, and so these considerations do not affect the order of the algorithm.

Thus, the time of the algorithm can be improved to $O(n^4)$ on a RAM under the uniform cost criterion.

We believe that the results of this paper strongly support the conjecture that the equivalence problem is decidable for general multitape deterministic finite state acceptors, and, perhaps, polynomially decidable.

REFERENCES

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

[2] C. BEERI, *An improvement on Valiant's decision procedure for equivalence of deterministic finite turn pushdown machines*, Theoret. Comput. Sci., 3 (1976), pp. 305–320.

[3] M. BIRD, *The equivalence problem for deterministic two-tape automata*, J. Comput. Syst. Sci., 7 (1973), pp. 218–236.

[4] P. C. FISCHER AND A. L. ROSENBERG, *Multitape one-way nonwriting automata*, J. Comput. Syst. Sci., 2 (1968), pp. 88–101.

[5] J. E. HOPCROFT AND R. M. KARP, *An algorithm for testing the equivalence of finite automata*, TR-71-114, Department of Computer Science, Cornell University, Ithaca, NY, 1971.

[6] M. O. RABIN AND D. SCOTT, *Finite automata and their decision problems*, IBM J. Res. Develop., 3 (1959), pp. 114–125.

[7] A. L. ROSENBERG, *Nonwriting extensions of finite aütomata*, Ph.D. thesis, Harvard University, Cambridge, MA, August, 1965.

[8] L. G. VALIANT, *The equivalence problem for deterministic finite-turn pushdown automata*, Inf. Control, 25 (1974), pp. 123–133.

[9] ———, *Decision procedures for families of deterministic pushdown automata*, Ph.D. thesis, Theory of computation – Report No. 1, Dept. of Computer Science, University of Warwick, Coventry, England, August, 1973.

SIAM J. COMPUT.
Vol. 11, No. 1, February 1982
© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1101-0014 $01.00/0

COMPLEXITY OF MATROID PROPERTY ALGORITHMS*

PER M. JENSEN† AND BERNHARD KORTE‡

Abstract. A general theorem is proved which can be used to show that for a large number of matroid properties there is no good algorithm of a certain type for determining whether these properties hold for general matroids. Specifically, there exists no algorithm in which the matroid is represented by an independence test oracle (or an oracle polynomially related to an independence test oracle) and which solves the problem in question after a number of calls on the oracle which is bounded by a polynomial in the number of elements of the ground set of the matroid.

Key words. Computational complexity, lower bounds, oracle algorithms, matroid properties.

Introduction. We will denote by M a matroid $(E, \mathcal{F})$ on a finite set E with $\mathcal{F} \subseteq 2^E$ the appropriate independence system. All notation and definitions used in this paper which are not explained are standard in matroid theory (cf. Welsh [15]). We shall consider questions concerning a single matroid M, which have a well-defined unique answer and furthermore the same answer if M is replaced by a matroid isomorphic to M. Such a question defines a *matroid property* Π, which is a mapping of the class of all matroids into some set (the set of truth values, the set of integers, etc.) such that any two isomorphic matroids have the same image under this mapping. In other words a matroid property is a partition of the set of all matroids whose classes are closed under isomorphism.

A matroid property is considered to be *easy* if there is a polynomial algorithm for determining it, i.e., an algorithm which decides about the property in a number of steps which can be expressed as a polynomial in $|E|$. So far we know only a few easy matroid properties, among which are maximum weighted basis, k-disjoint bases, k-connectivity (for a fixed and finite k), and the property of being graphic. The first mentioned property is an algorithmic equivalent to the definition of matroids and can be discovered by the famous greedy algorithm. For the other three properties we are aware of good algorithms by J. Edmonds [5], R. E. Bixby, W. H. Cunningham and J. Edmonds [2], [4] and P. D. Seymour [14].

When studying algorithms for deciding about matroid properties one question immediately comes up: How is the matroid described to the algorithm? The number of nonisomorphic matroids on a set E (with $n := |E|$) is at least $2^{2^{n-(3/2)\log n + O(\log\log n)}}$. Thus, even if we could number the classes of isomorphic matroids and specify a matroid to the algorithm just by this isomorphism class number, then reading the digits in the binary expansion of this number would require a number of operations which grows exponentially with n.

One possible solution to this problem is to study *oracle algorithms*. This approach was first used in Hausmann and Korte [7], [8], [9] to derive lower bounds on the worst-case complexity of algorithms for optimization problems over independence systems or fixed point approximation problems. This paper is a straightforward extension of the techniques developed there. L. Lovász [11], G. C. Robinson and D. J. A. Welsh [13] and P. D. Seymour [14] have also derived results about the complexity of matroid properties by using oracle approaches.

* Received by the editors December 4, 1979, and in final revised form April 8, 1981.
† The Technical University of Denmark, Lyngby, Denmark.
‡ University of Bonn, Bonn, Germany.
The work of this author was supported by Sonderforschungsbereich 21 (DFG), Institut für Ökonometrie und Operations Research, University of Bonn.

An oracle can be defined as a mapping from 2^E into some finite set of size $\leqq |E|$. We consider here only mappings into sets of size 2; that is, the possible oracle answers are "yes" and "no". The oracle mapping can then be specified relative to the different matroid definitions. A basic matroid oracle is the *independence test oracle* which says "yes" for $F \subseteq E$ if and only if $F \in \mathcal{F}$. An oracle algorithm is then an algorithm in which one of the possible operations is the calling of a "subroutine", the oracle, which answers questions of a certain type concerning the matroid (e.g., whether a set $F \subseteq E$ is independent or not). Calling the oracle is the only way in which the algorithm can obtain information about the matroid. Other than this the algorithm is not in any way restricted. The operation of calling the oracle is assumed to take constant time (or an amount of time which is bounded by a polynomial in the size of the ground set of the matroid). Thus an oracle algorithm is considered polynomial if the number of calls on the oracle and the number of other operations in the algorithm are both bounded by a polynomial in the size of the ground set of M. This verbal definition and explanation of oracles and oracle algorithms should be sufficient for the reader of this paper. A more precise definition of oracle algorithms can be found in Hausmann and Korte [8], [9].

We should state that all the above-mentioned good algorithms for matroid properties, like the greedy algorithm, are precisely independence test oracle algorithms. On the other hand it can be said that for all important special cases of matroids which occur in "real life", e.g., graphic matroids, transversal matroids, gammoids, representable matroids, it is possible to make a polynomial "realization" of the independence test oracle. Also for all common matroid operations (direct sum, sum, truncation, dualization, etc.) the independence test oracles for the original matroids can be combined by a polynomial algorithm into a realization of the independence test oracle of the resulting matroid.

A lower bound on the complexity of matroid algorithms. We will now state a general lemma from which we can easily derive statements about the complexity of special matroid properties.

LEMMA. *Let Π be a matroid property, and suppose two matroids M_n and M'_n are given on a set E of size n, which are different with respect to Π and for which Q_1, $Q_2, \cdots, Q_{q(n)} \subseteq E$ are dependent in one of the two matroids and independent in the other one. Then every independence test oracle algorithm which decides Π has complexity at least*

$$\frac{f(n)}{\sum_{i=1}^{q(n)} \rho(Q_i)},$$

where $f(n)$ is the number of automorphisms on M_n and $\rho(Q_i)$ is the number of automorphisms on M_n which map Q_i onto itself, $i = 1, \cdots, q(n)$.

Proof. Suppose we have an algorithm for the property. If the algorithm is executed on M_n it calls the independence test oracle for some sets $P_1, P_2, \cdots, P_p$. For a given one of these sets P_j and a given set Q_i on which the matroids differ, the number of automorphisms of M_n which map Q_i onto P_j is at most $\rho(Q_i)$, so there are at most $p \sum_{i=1}^{q(n)} \rho(Q_i)$ automorphisms of M_n which map any of the Q_i's onto any of the P_j's. Suppose $f(n) > p \sum_{i=1}^{q(n)} \rho(Q_i)$. Then there exists an automorphism ϕ of M_n which does not map any Q_i onto any P_j. Let $\phi(M'_n)$ be the matroid isomorphic to M'_n which is the image of M'_n under the mapping ϕ. The matroids $\phi(M'_n)$ and $\phi(M_n) = M_n$ differ only on the sets $\phi(Q_1), \phi(Q_2), \cdots, \phi(Q_{q(n)})$ and since none of the P_j's are images of a Q_i under ϕ, the algorithm, when run on $\phi(M'_n)$, must behave exactly as when it

was applied to M_n and thus give the same answer, a contradiction. So

$$f(n) \leqq p \sum_{i=1}^{q(n)} \rho(Q_i) \quad \text{and} \quad p \geqq \frac{f(n)}{\sum_{i=1}^{q(n)} \rho(Q_i)}. \qquad \square$$

The most important special case of this lemma is the following:

COROLLARY 1. *Let Π be a matroid property, and suppose a matroid M'_n with rank r on a set E of size n is given which is different with respect to Π from $U_{n,r}$ (the uniform matroid of rank r or a ground set of size n). If the rank of M'_n is bounded by*

$$\alpha n \leqq r \leqq (1-\alpha)n \quad \textit{with } 0<\alpha<1,$$

and M'_n has at most polynomially many (in n) dependent sets of size $\leqq r$, then there is no polynomial independence test oracle algorithm which decides Π.

Proof. We apply the theorem with $M_n = U_{n,r}$. The number of automorphisms of $U_{n,r}$ is $f(n) = n!$. The sets on which the matroids differ are the dependent sets of M'_n of size $\leqq r$. Since the number of these sets is bounded by the polynomial, say $q(n)$, and all supersets of a dependent set are dependent, the size of a dependent set of M'_n must be at least $r-k \geqq \alpha n - k$, where k is the degree of the polynomial $q(n)$. If Q_i is a dependent set of M'_n of size $\leqq r$, then the number of automorphisms of $U_{n,r}$ mapping Q_i onto itself is $\rho(Q_i) = (|Q_i|)!(|E \backslash Q_i|)!$ Furthermore $|Q_i| \geqq \alpha n - k$ and $|E\backslash Q_i| \geqq n - r \geqq \alpha n \geqq \alpha n - k$. For fixed n the function $x!(n-x)!$ increases if either x or $n-x$ decreases from $n/2$. Therefore we have

$$\rho(Q_i) \leqq (\alpha n - k)!(n-(\alpha n - k))!$$

Thus an independence test oracle algorithm deciding about Π must have complexity at least

$$\frac{n!}{\sum_{i=1}^{q(n)} (\alpha n - k)!(n-(\alpha n - k))!} = \frac{1}{q(n)}\binom{n}{\alpha n - k},$$

which grows exponentially with n. $\square$

Corollary 1 requires "nearly uniform" matroids which can be constructed as paving matroids by use of the following proposition which is a straightforward application of matroid definition.

PROPOSITION. *Let $\mathcal{P}_r(E)$ be the set of all r-subsets of E and let $\mathcal{C} \subseteq \mathcal{P}_r(E)$ be such that $\mathcal{C}$ does not contain two sets C_1, C_2 with $|C_1C_2| = 1$. Then $\mathcal{P}_r(E)\backslash\mathcal{C}$ is the set of bases of a matroid whose circuits of size $\leqq r$ are the sets in $\mathcal{C}$.*

With this we construct now some matroids to be used with Corollary 1 ($\dot{\cup}$ denotes disjoint union):

(i) $M^1_{2r,r}$ is the matroid on a $2r$-set with bases all r-sets except one.

(ii) $M^2_{2r,r}$ is the matroid on a $2r$-set with bases all r-sets except two complementary sets.

(iii) Let r be even and $A_0 \dot{\cup} A_1 \dot{\cup} A_2 \dot{\cup} A_3$ be a $2r$-set with $|A_i| = r/2$, $i = 0, 1, 2, 3$.
$M^3_{2r,r}$ is the matroid on $\cup A_i$ with bases all r-sets except $A_0 \cup A_1$, $A_0 \cup A_2$, $A_0 \cup A_3$.

(iv) Let r be even and $A_0 \dot{\cup} A_1 \dot{\cup} \cdots \dot{\cup} A_{r-1}$ be a $2r$-set with $|A_i| = 2$, $i = 0, 1, \cdots, r-1$.
$M^4_{2r,r}$ is the matroid on $\cup A_i$ with bases all r-sets except $A_0 \cup A_2 \cup \cdots \cup A_{r-2}$ and $A_k \cup A_{k+1} \cup \cdots \cup A_{k+r/2-1}$, $k = 0, 1, \cdots, r-1$ (indices modulo r).

(v) Let $r \geqq 4$ and $A_0 \dot{\cup} A_1 \dot{\cup} A_2 \dot{\cup} A_3 \dot{\cup} B_0 \dot{\cup} B_1$ be a $2r$-set with $|A_i| = 2$ and $|B_i| = r-4$.
$M^5_{2r,r}$ is the matroid on $(\cup A_i) \cup B_0 \cup B_1$ with bases all r-sets except $B_0 \cup (A_0 \cup A_1)$, $B_0 \cup (A_0 \cup A_2)$, $B_0 \cup (A_0 \cup A_3)$, $B_0 \cup (A_1 \cup A_2)$, $B_0 \cup (A_1 \cup A_3)$.

(vi) Let $r \geqq 4$ and $A_0 \dot{\cup} A_1 \dot{\cup} A_2 \dot{\cup} A_3 \dot{\cup} B_0 \dot{\cup} B_1$ be a $2r$-set with $A_i = \{a_i, a'_i\}$, $0 \leqq i < 3$ and $|B_i| = r-4$.
$M^6_{2r,r}$ is the matroid on $(\cup A_i) \cup B_0 \cup B_1$ with bases all r-sets except $B_0 \cup (A_i \cup A_j)$, $0 \leqq i < j \leqq 3$ and $B_0 \cup \{a_0, a_1, a_2, a'_3\}$, $B_0 \cup \{a_0, a_1, a'_2, a_3\}$, $B_0 \cup \{a_0, a'_1, a_2, a_3\}$, $B_0 \cup \{a'_0, a_1, a_2, a_3\}$, $B_0 \cup \{a'_0, a'_1, a'_2, a'_3\}$.

(vii) $M^7_{2r,r}$ is the matroid on a $2r$-set with bases all r-sets except the supersets of a given $(r-1)$-set. (Here we cannot use the above proposition but one can easily verify that $M^7_{2r,r}$ is a matroid.)
(We thank Rick Giles [6] for the idea of the constructions (v) and (vi).)

With these preparations are are now able to prove the following:

THEOREM 1. *There exists no polynomial independence test oracle algorithm for any of the following matroid properties (formulated as questions or commands):*

(1) *Is M uniform?*
(2) *Find the girth of M (minimum cardinality circuits)!*
(3) *Find the number of circuits of M!*
(4) *Find the number of bases of M!*
(5) *Find the number of hyperplanes of M!*
(6) *Find a hyperplane of maximum size!*
(7) *Find the number of flats!*
(8) *Is the automorphism group of M transitive?*
(9) *Is M self-dual (isomorphic to its dual)?*
(10) *Given that M is self-dual, is M identically self-dual (equal to its dual)?*
(11) *Is M transversal?*
(12) *Given that M is transversal, is M fundamental transversal (cf. Welsh [14, Exercise 3, p. 245])?*
(13) *Find the connectivity of M!*
(14) *Find the Tutte polynomial of M!*
(15) *Find the Crapo invariant $\beta(M)$ (cf. Welsh [14, p. 269])!*
(16) *Is M representable?*
(17) *Is M orientable (as defined by Bland and Las Vergnas [3], which is more general than the definition of Welsh [9])?*
(18) *Is M a paving matroid?*
(19) *Is M bipartite?*
(20) *Is M Eulerian?*

Proof.
(1) $U_{2r,r}$ is uniform, $M^1_{2r,r}$ is not.
(2) $U_{2r,r}$ has girth $r+1$, $M^1_{2r,r}$ has girth r.
(3) $U_{2r,r}$ has $\binom{2r}{r+1}$ circuits, $M^1_{2r,r}$ has $\binom{2r}{r+1}+1-r$ circuits.
(4) $U_{2r,r}$ has $\binom{2r}{r}$ bases, $M^1_{2r,r}$ has $\binom{2r}{r}-1$ bases.
(5) $U_{2r,r}$ has $\binom{2r}{r-1}$ hyperplanes, $M^1_{2r,r}$ has $\binom{2r}{r-1}+1-r$ hyperplanes.
(6) The maximum hyperplane in $U_{2r,r}$ is an $r-1$ set, in $M^1_{2r,r}$ it is an r-set.
(7) The number of flats in $U_{2r,r}$ differs from the number of flats in $M^1_{2r,r}$ by r.
(8) $U_{2r,r}$ has a transitive automorphism group, $M^1_{2r,r}$ not.
(9), (10) $U_{2r,r}$ is identically self-dual, $M^1_{2r,r}$ is self-dual, but not identically self-dual, $M^3_{2r,r}$ is not even self-dual.

(11), (12) $U_{2r,r}$ is fundamental transversal, $M^2_{2r,r}$ is transversal, but not fundamental transversal, $M^4_{2r,r}$ is not even transversal (Theorem of Mason, cf. Welsh [15, p. 245]).

(13) $U_{2r,r}$ has connectivity ∞, $M^1_{2r,r}$ has connectivity r. Note that this result does not conflict with the above-mentioned polynomial algorithm of Edmonds, Bixby and Cunningham for deciding about k-connectivity (k fixed and finite).

(14) The Tutte polynomial of $U_{2r,r}$ has no mixed terms, the Tutte polynomial of $M^1_{2r,r}$ has a term xy.

(15) $\beta(U_{2r,r}) = \beta(M^1_{2r,r}) + 1$.

(16) $U_{2r,r}$ is representable, $M^5_{2r,r}$ is not, since $M^5_{2r,r}|(E\backslash B_1) \cdot \bigcup A_i$ is the Vamos matroid.

(17) $U_{2r,r}$ is orientable, $M^6_{2r,r}$ is not, since $M^6_{2r,r}|(E\backslash B_1) \cdot \bigcup A_i = M^6_{8,4}$ is not orientable (see the example in Bland and Las Vergnas [3, p. 112]).

(18) $U_{2r,r}$ is a paving matroid, $M^7_{2r,r}$ is not.

(19) Let r be odd, $U_{2r,r}$ is bipartite, $M^1_{2r,r}$ is not.

(20) $U_{2r,r}$ is not Eulerian, $M^2_{2r,r}$ is Eulerian. □

What is really shown in (16) and (17) above is that there is no polynomial algorithm for deciding whether a matroid has a minor isomorphic to respectively the Vamos matroid and the matroid $M^6_{8,4}$, or whether it is uniform (and has only uniform minors). The idea behind the construction of the matroids $M^5_{2r,r}$ and $M^6_{2r,r}$ can be extended to show for a quite large class of matroids that they cannot be detected as minors by a polynomial algorithm.

Let $M^0_{n,r}$ be a nonuniform matroid of rank r on a set A of size n satisfying the condition that it has no circuit of size less than r and no cocircuit of size less than $n-r$. Let $E = A \dot\cup B_0 \dot\cup B_1$, $|B_i| = k$. By taking as circuits all sets of the form $C \cup B_0$, where C is a circuit of $M^0_{n,r}$ of size r, and all subsets of E of size $k+r+1$ which do not contain any of the former sets as a subset, we obtain a matroid $M^0_{2k+n,k+r}$ with $M^0_{n,r}$ as a minor ($M^0_{2k+n,k+r}|(E\backslash B_1 \cdot A = M^0_{n,r}$). Furthermore the number $q(n)$ in Corollary 1 is constant (= the number of circuits of $M^0_{n,r}$ of size r). Therefore Corollary 1 gives us directly:

COROLLARY 2. *Let $M^0_{n,r}$ be a nonuniform matroid with no circuit of size less than the rank of the matroid and no cocircuit of size less than its corank. Then there is no polynomial independence test oracle algorithm which will determine whether an arbitrary matroid has a minor isomorphic to $M^0_{n,r}$.*

Some results require a nonuniform M_n and cannot be obtained from Corollary 1, but rather by direct application of the lemma:

THEOREM 2. *There exists no polynomial independence test oracle algorithm for the following matroid properties:*

(21) *Is M binary?*

(22) *Is M Hamiltonian (i.e., does M have a circuit of size greater than the rank of M)?*

(23) *Given a partition of the ground set of M into pairs $A_0, A_1, \cdots, A_{n/2-1}$, $|A_i| = 2$, what is the maximum size of an independent set being the union of some of the sets A_i?*

((23) is the 2-parity problem. (22) is the natural matroid generalization of the Hamiltonian graph problem.)

The result for property (21) was first proved by Seymour [14]. Our proof is a reformulation of his, one which demonstrates that it also fits into the framework of the general lemma. L. Lovász [11] proved the result for property (23) independently

by using polymatroids. Moreover he gave a polynomial algorithm for the 2-parity problem in the case a representation of the matroid is known [12].

There is a slight complication in the case of property (23). Here there is, apart from the matroid, another "parameter" to the problem, namely the partition of the ground set. In such a case the word "automorphism" in the lemma should be read as "automorphism under which the extra parameters of the problem are invariant". In the above case all automorphisms of $M^{10}_{2r,r}$ satisfy this condition.

Proof of Theorem 2. For r odd let $M^8_{2r,r}$ be the binary matroid having a standard representative matrix over $GF(2)$ of the form $(I \mid J-I)$, where I is an $r \times r$ identity matrix and J an $r \times r$ matrix of all 1's. If Q denotes the set corresponding to the $J-I$ part of the representative matrix, then it can be shown that adding Q to the collection of bases of $M^8_{2r,r}$ produces a new matroid $M^9_{2r,r}$, and that this matroid is nonbinary. Also it can be shown that $M^8_{2r,r}$ does not have circuits of size $r+1$ whereas $M^9_{2r,r}$ does. The lemma gives the following bound on the complexity of an algorithm for (21) or (22):

$$\frac{2^{r-1}r!}{\sum_{i=1}^{1} r!} = 2^{r-1}.$$

For r even let $E = A_0 \dot{\cup} A_1 \dot{\cup} \cdots \dot{\cup} A_{r-1}$, $|A_i| = 2$ and let $M^{10}_{2r,r}$ be the matroid on E with bases all r-sets except those that are unions of $r/2$ of the sets A_i. $M^{11}_{2r,r}$ is the matroid on E whose bases are the bases of $M^{10}_{2r,r}$ and the set $A_0 \cup A_1 \cup \cdots \cup A_{r/2-1}$. Clearly the solution to (23) for $M^{10}_{2r,r}$ is $r-2$, and for $M^{11}_{2r,r}$ it is r. The theorem gives the following bound:

$$\frac{2^r r!}{\sum_{i=1}^{1} 2^r \left(\frac{r}{2}\right)! \left(\frac{r}{2}\right)!} = \binom{r}{\frac{r}{2}}. \qquad \square$$

Conclusion. Almost all results in this paper were derived from the general theorem in such a way that one of the two matroids considered with respect to a certain property was chosen to be the uniform matroid. This was only done in order to have a general framework for *many* matroid properties against the *same* property of being uniform. Indeed, Robinson and Welsh [12] have proved that uniformity is among all matroid properties the hardest one to decide by an independence test oracle. This might have been the intuitive reason for our approach.

Another comment should be made on the choice of the oracle. We have explained why we have chosen the independence test oracle, but one can also think of other matroid "questions" for a possible oracle, such as: basis, circuit, flat, rank, closure, girth, etc. They could be classified due to their different power and difficulty of realization. For example, let us define the *girth of a set* $F \subseteq E$ of a matroid on E to be 0 if F is independent and the minimum cardinality of a circuit contained in F otherwise. It is trivial to make a polynomial realization of an independence test oracle using an oracle which finds the girth of a set. On the other hand the girth oracle cannot be polynomially simulated by the independence test oracle. Thus, there are oracles which are strictly more powerful than the independence test. However, they are also more difficult to realize. For graphic matroids one can still find a polynomial realization of the girth oracle (with higher complexity than the independence test oracle), but for general binary matroids the realization of the girth oracle is equivalent to finding the minimum distance of a binary error-correcting code, a problem which has been proved to be NP-complete (Berlekamp, McEliece, and van Tilborg [1]).

So instead of just a single standard type of oracle what is needed is a stock of more or less powerful oracles. The objective is then to find, for each class of matroids, polynomial realizations of oracles which are as powerful as possible, and for each matroid property (hopefully polynomial) algorithms which solve the problem using oracles which are as simple as possible. Studies of relations among different types of oracles can be found in Hausmann and Korte [10] and Robinson and Welsh [13].

REFERENCES

[1] E. R. BERLEKAMP, R. J. MCELIECE AND H. C. A. VAN TILBORG, *On the inherent intractability of certain coding problems*, IEEE Trans. Inform. Theory, IT-24 (1978), pp. 384–386.

[2] R. E. BIXBY AND W. H. CUNNINGHAM, *Matroids, graphs and 3-connectivity*, in Graph Theory and Related Topics, J. A. Bondy and U. S. R. Murty, eds., Academic Press, New York, 1978, pp. 91–103.

[3] R. G. BLAND AND M. LAS VERGNAS, *Orientability of matroids*, J. Combin. Theory, Ser. B, 24 (1978), pp. 94–123.

[4] W. H. CUNNINGHAM AND J. EDMONDS, *Decomposition of linear systems*, to appear.

[5] J. EDMONDS, *Minimum partition of a matroid into independent subsets*, J. Res. National Bureau of Standards, 69B (1965), pp. 67–72.

[6] F. R. GILES, private communication.

[7] D. HAUSMANN AND B. KORTE, *Lower bounds on the worst-case complexity of some oracle algorithms*, Discrete Mathematics, 24 (1978), pp. 261–276.

[8] ———, *Oracle algorithms for fixed point problems—an axiomatic approach*, in Optimization and Operations Research, R. Henn, B. Korte and W. Oettli, eds., Proc. Workshop in Bad Honnef, Springer-Verlag, Berlin-Heidelberg-New York, 1978, pp. 137–156.

[9] ———, *Worst-case analysis for a class of combinatorial optimization algorithms*, in Optimization Techniques, Part 2, J. Stoer, ed., Proc. 8th IFIP Conference on Optimization Techniques, Würzburg, 1977, Springer-Verlag, Berlin-Heidelberg-New York, 1978, pp. 216–224.

[10] ———, *Algorithmic versus axiomatic definitions of matroids*, Math. Prog. Study, 14 (1981), pp. 98–111.

[11] L. LOVÁSZ, *The matroid matching problem*, in Algebraic Methods in Graph Theory, L. Lovász and V. T. Sós, eds., Proc. Coll. Math. Soc. J. Bolyai, North-Holland, Amsterdam, 1981, to appear.

[12] ———, *Matroid matching and some applications*, J. Combin. Theory Ser. B, 28 (1980), pp. 208–236.

[13] G. C. ROBINSON AND D. J. A. WELSH, *The computational complexity of matroid properties*, Math. Proc. Comb. Phil. Soc., 87 (1980), pp. 29–45.

[14] P. D. SEYMOUR, *Recognizing graphic matroids*, Combinatorica, 1 (1981), pp. 75–78.

[15] D. J. A. WELSH, *Matroid Theory*, Academic Press, London-New York-San Francisco, 1976.

SIAM J. COMPUT
Vol. 11, No. 1, February 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397 / 82 / 1101-0015 $01.00 / 0

DOMINATING SETS IN CHORDAL GRAPHS*

KELLOGG S. BOOTH† AND J. HOWARD JOHNSON†

Abstract. A set of vertices D is a dominating set for a graph if every vertex is either in D or adjacent to a vertex which is in D. We show that the problem of finding a minimum dominating set in a chordal graph is NP-complete, even when restricted to undirected path graphs, but exhibit a linear time greedy algorithm for the problem further restricted to directed path graphs. Streamlined to handle only trees, our algorithm becomes the algorithm of Cockayne, Goodman and Hedetniemi. An interesting parallel with graph isomorphism is pointed out.

Key words. chordal graph, directed path graph, dominating set, graph isomorphism, interval graph, minimum dominating set, undirected path graph

1. Introduction. A set of vertices D is a *dominating set* for a graph $G = (V,E)$ if every vertex is either in D or adjacent to a vertex which is in D. A smallest such set is a *minimum dominating set*. For arbitrary graphs the problem of finding a minimum dominating set is NP-complete [6]. For the special case of trees there are algorithms which run in linear time [3], [15]. We improve upon both of these results by showing that the problem remains NP-complete when restricted to undirected path graphs but that a further restriction to directed path graphs admits a linear time solution.

Our method is a simple greedy algorithm which walks a tree representation of a directed path graph. It utilizes a linear time recognition algorithm for directed path graphs devised by Dietz, Furst and Hopcroft [4]. If the graph is a tree (trees are a subfamily of the directed path graphs) our algorithm can be streamlined to the original procedure of Cockayne, Goodman and Hedetniemi [3].

The remainder of this section is devoted to notation and a brief review of chordal graphs. Further details are available in the cited references.

We make heavy use of the term clique. A *clique* is any maximal set of vertices which are all mutually adjacent. Some authors do not insist on maximality; our usage follows Harary [11].

An edge is a *chord* of a cycle if it connects two vertices of the cycle but is not itself an edge within the cycle. A graph is *chordal* if and only if every cycle of length greater than three has a *chord*. For our purposes a more useful definition of chordality is the characterization proven by Gavril. His result can be used to classify a hierarchy of chordal graphs in terms of intersection models.

A graph is an *intersection graph* if there is a correspondence between its vertices and a family of sets (the *intersection model*) such that two vertices are adjacent in the graph if and only if their two corresponding sets have a nonempty intersection. Restricting the sets to subtrees of a tree determines the class of chordal graphs [8].

*Received by the editors July 24, 1980, and in final form August 4, 1981. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada under grant A4307. This paper was typeset using software developed at the University of Waterloo.

†Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1.

If the intersection model is further restricted, so that each subtree must be a path, a proper subclass called the *undirected path graphs* results [10]. Further restricting the model to rooted trees with paths directed away from the root yields the *directed path graphs* [9]. Requiring that the tree itself be a path defines the class of *interval graphs* [5], [13].

As Gavril has shown in his papers, the intersection models can always be chosen so that the nodes of the tree are the cliques of the original graph. Each vertex then corresponds to the subtree comprised of exactly those cliques to which it belongs. We call such an intersection model a *clique tree* for the graph. We want to extract as much structural information as possible from the graph; thus we insist that the clique tree have each vertex correspond to a subtree, undirected path, directed path or subpath depending upon whether the original graph is a chordal graph, undirected path graph, directed path graph or interval graph.

The hierarchy of chordal graphs is illustrated in Fig. 1 which shows (a) a chordal graph, (b) an undirected path graph, (c) a directed path graph and (d) an interval graph. Each belongs to its subclass but does not belong to the next more restrictive subclass. These four graphs are closely related. Each successive graph is obtained from the previous graph by vertex or edge deletion where the deleted elements are indicated with dashed lines. The examples are drawn from Gavril [8], [9] and Lekkerkerker and Boland [13].

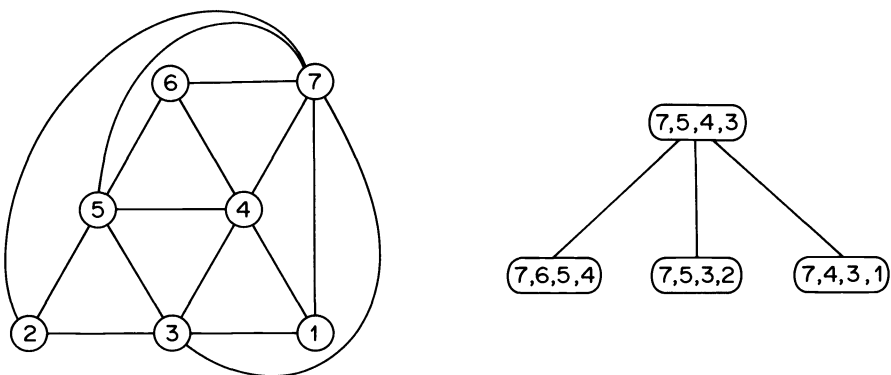


FIG. 1(a). *A chordal graph with its clique tree.*

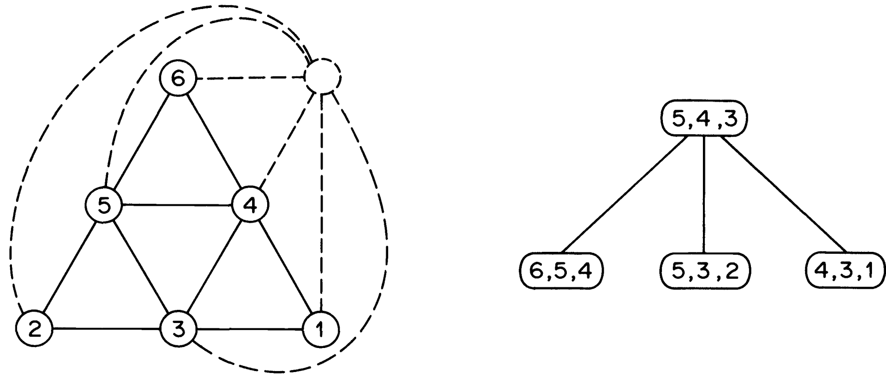


FIG. 1(b). *An undirected path graph with its clique tree.*

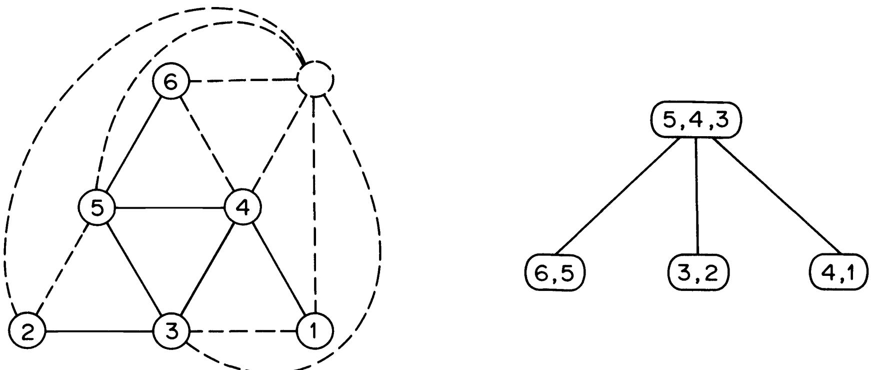


FIG. 1(c). *A directed path graph with its clique tree.*

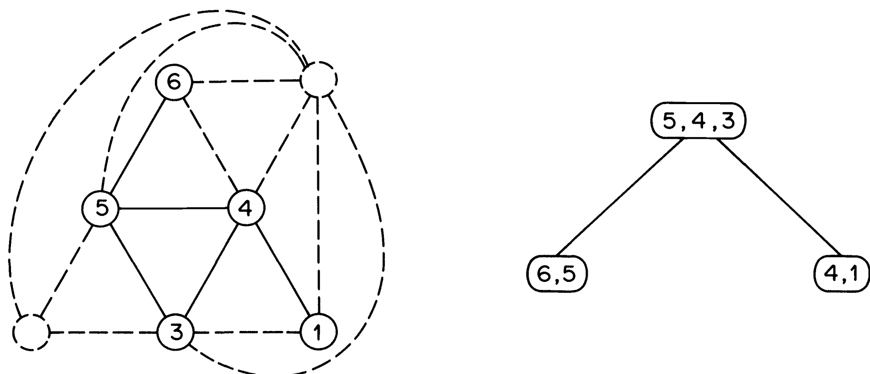


FIG. 1(d). *An interval graph with its clique tree.*

A polynomial time algorithm for constructing the clique tree of an interval graph is easily obtained from an algorithm of Fulkerson and Gross [5]. Polynomial time algorithms for the other three classes of chordal graphs were first given by Gavril [8], [9], 10]. Linear time algorithms are now known for all but one of these classes. Lueker, Rose and Tarjan [17] have an algorithm for chordal graphs. Dietz, Furst and Hopcroft have a recognition algorithm for directed path graphs [4] which can be modified to produce a clique tree. Booth and Lueker [2] have a recognition algorithm for interval graphs which already produces the equivalent of a clique tree. To our knowledge there is no published linear time algorithm which builds a clique tree for undirected path graphs.

The notion of a path graph has been around for more than ten years. Renz introduced the idea in 1970 when he gave a partial characterization for undirected path graphs [16]. Dietz, Furst and Hopcroft base their work on a more recent paper by Truszczynski [4], [18]. He refers to the problem as the generalized consecutive retrieval problem (Booth and Lueker [2] provide references to this and related concepts).

2. NP-completeness for chordal graphs. No one has produced a polynomial time algorithm for finding a minimum dominating set of an arbitrary graph. The problem is NP-hard. As is usual for combinatorial minimization problems, the NP-complete version is stated as a recognition problem: "Given a graph G and an

integer k, is there a dominating set of size k for G?" We will use the latter formulation of the problem throughout this section.

The general dominating set problem was shown NP-complete using a reduction from the vertex cover problem [6, p. 190]. A slight variation of that reduction suffices to prove that even for the restricted case of chordal graphs the dominating set problem is NP-complete. We prove a stronger result using a reduction from the 3-dimensional matching problem [6, pp. 50-53].

THEOREM 1. *The dominating set problem for undirected path graphs is* NP-*complete.*

Proof. Let W, X, and Y be three disjoint sets each of cardinality q and let M be a subset of $W \times X \times Y$ having cardinality p. We use the following notation of Garey and Johnson [6].

$$W = \{w_j \mid 1 \leq j \leq q\}$$
$$X = \{x_k \mid 1 \leq k \leq q\}$$
$$Y = \{y_l \mid 1 \leq l \leq q\}$$
$$M = \{m_i = \langle w_j, x_k, y_l \rangle \mid w_j \in W,\ x_k \in X,\ y_l \in Y,\ 1 \leq i \leq p\}.$$

The *3-dimensional matching problem* is to find a subset M' of M having cardinality exactly q such that each $w_j \in W$, $x_k \in X$ and $y_l \in Y$ occurs precisely once in a triple of M'.

For a 3-dimensional matching problem with triples M we may assume that each element of W, X and Y occurs in at least two triples since otherwise the single triple must occur in any solution so we could reduce the problem to a smaller one. We construct a tree having $6p + 3q + 1$ cliques from which we will obtain an undirected path graph. The cliques of the tree are explained below.

For each triple m_i in M there are six cliques whose vertices depend only upon the triple itself and not upon the elements within the triple. These six cliques form the subtree corresponding to m_i, which is illustrated in Fig. 2,

$$\{A_i, B_i, C_i, D_i\}$$
$$\{A_i, B_i, D_i, F_i\}$$
$$\{C_i, D_i, G_i\}$$
$$\{A_i, B_i, E_i\}$$
$$\{A_i, E_i, H_i\}$$
$$\{B_i, E_i, I_i\} \qquad \text{for } 1 \leq i \leq p.$$

Next, there is a clique for each element of W, X and Y which depends upon the triples of M to which the element belongs.

$$\{J_j\} \cup \{A_i \mid w_j \in m_i\} \qquad \text{for } 1 \leq j \leq q,$$
$$\{K_k\} \cup \{B_i \mid x_k \in m_i\} \qquad \text{for } 1 \leq k \leq q,$$
$$\{L_l\} \cup \{C_i \mid y_l \in m_i\} \qquad \text{for } 1 \leq l \leq q.$$

And finally there is one large clique, the root of the tree, which contains vertices for each of the triples

$$\{A_i, B_i, C_i \mid 1 \leq i \leq p\}.$$

We see that the sets are cliques by verifying that no set is properly contained within another. We check that each element is contained only in a family of cliques which form an undirected path within the tree; it is then easy to see that there is only one way in which the cliques can be connected into a tree so that these conditions hold. This is the arrangement shown in Fig. 2. We thus know that the graph G whose cliques were built from the 3-dimensional matching problem is an undirected path graph and the clique tree is unique.

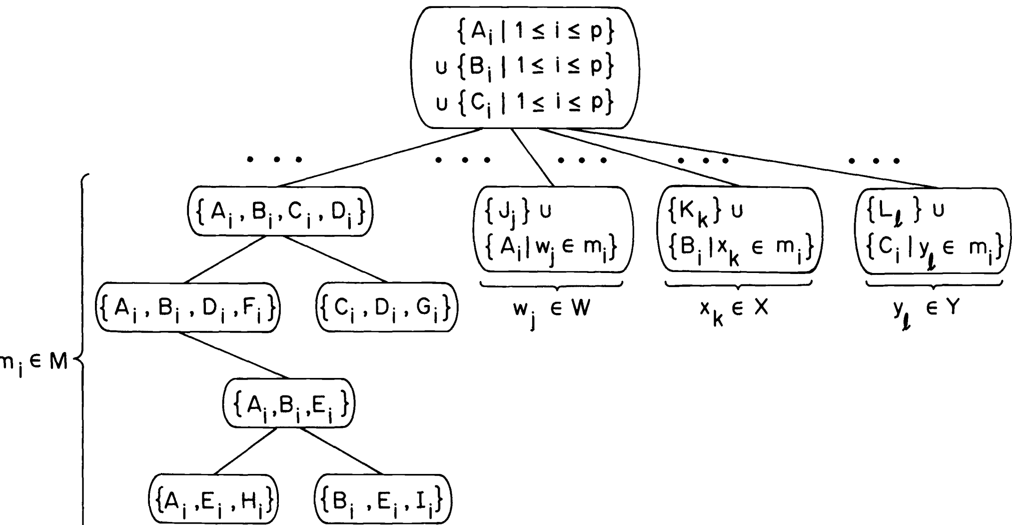


FIG. 2. *The clique tree for the undirected path graph G corresponding to a 3-dimensional matching problem.*

We next claim that the undirected path graph G has a dominating set of size $2p+q$ if and only if the 3-dimensional matching problem has a solution.

Verifying one direction of the claim is easy. If the 3-dimensional matching problem has a solution M' we simply choose for each m_i in M' all of the vertices A_i, B_i and C_i corresponding to that m_i. There are precisely $3q$ of these. For all other m_i not in M' we choose the corresponding vertices D_i and E_i. There are $2p-2q$ of these. Altogether we have chosen $2p+q$ vertices which form a dominating set for G.

Conversely, given a dominating set for G we can assume without loss of generality that for each i either all three of A_i, B_i and C_i or else both of D_i and E_i have been included. This follows from the observation that the only way to dominate the subtree corresponding to m_i with two vertices is to choose D_i and E_i, and that any larger dominating set might just as well consist of A_i, B_i and C_i since none of the other possible vertices dominate any vertex outside of the subtree.

The proof is completed by noting that if there are t of the m_i for which A_i, B_i and C_i are chosen in a dominating set of size $2p+q$, these account for $3t$ vertices and the remaining E_i and D_i account for $2p-2t$ vertices. It must be the case that $t=q$. Picking the q triples m_i for which A_i, B_i and C_i were chosen yields a solution to the 3-dimensional matching problem. □

3. A linear algorithm for directed path graphs. This section describes a linear time algorithm for finding a minimum dominating set in a directed path graph. We will not give all of the details for our algorithm. Instead we will explain how to build it from pieces of other algorithms.

First we assume that a clique tree has been constructed for the directed path graph $G = (V,E)$. This requires only linear time using an easy modification to the recognition algorithm of Dietz, Furst and Hopcroft. The result is a rooted tree each of whose nodes is a clique of G. It has the property that the cliques containing any vertex form a contiguous path directed away from the root.

We next collect the following information, all in linear time, walking (or while constructing) the clique tree and using the original graph G.

[1] For each clique C find the integer $depth[C]$ which is the length of the path from the root of the clique tree to C. A list of all vertices contained in the clique is also generated.

[2] For each vertex v find the integer $high[v]$ which is the depth of the highest clique to which v belongs. A list of all adjacent vertices is also generated.

The rest of the algorithm is straightforward. Initialize the dominating set D to be empty, mark as "undominated" every vertex, then walk the clique tree in preorder (or any order in which all children are visited before their parent) performing the following operation.

[3] If there is a vertex v in the current clique C marked "undominated" and $\text{high}[v] = C$ then choose a vertex x in C having the smallest value $\text{high}[x]$ among all vertices in C. Add x to D and mark as "dominated" x and all vertices adjacent to x.

A linear running time for the algorithm is easily established. The *size* of a graph G is the sum of its number of vertices and its number of edges, $n + e$. The sum of the sizes of all of the cliques in a chordal graph is $O(n + e)$ because each vertex in a clique can be paired off against either a vertex of G or against an edge of G [7], [8]. Each of the major steps is linear either in the size of G, in the size of the clique tree (the sum of the sizes of the cliques), or in the size of a clique. Hence the entire algorithm is linear in the size of G.

With the exception of the clique tree building, which uses the Dietz, Furst and Hopcroft algorithm, our solution is built entirely of standard algorithms from graph theory. Each of these is almost trivially linear time. Their algorithm is a bit more complicated and the proofs of linearity and correctness are quite involved. We remark that even without their algorithm we can substitute the polynomial time algorithm of Gavril to obtain a guaranteed polynomial running time for our algorithm.

Correctness of the algorithm follows from the next theorem. We introduce the notation DOM (u) to denote the set of vertices dominated by a vertex u and DOM (U) to denote the set of vertices dominated by a set U of vertices. A vertex dominates itself and any adjacent vertex.

THEOREM 2. *At the termination of the dominating set algorithm* DOM $(D) = V$ *and* $|D|$ *is minimum along all dominating sets for G.*

Proof. We show by induction on the number of cliques visited that there is a minimum dominating set D' which contains D and that DOM(D) contains at least those vertices which appear only in visited cliques.

The basis is trivial. After zero iterations D is empty and clearly contained within any minimum dominating set. There are no vertices which D is required to dominate.

For the inductive step, if no v satisfies the condition of Step [3] in the

algorithm the hypothesis continues to hold. Otherwise suppose that x is added to D while visiting C in the clique tree. By the induction hypothesis $D - \{x\}$ is contained within some minimum dominating set D'. If x is in D' we are done. Otherwise let v be the vertex of C which is not dominated by $D - \{x\}$.

The condition on v is that high$[v]$ = depth$[C]$. Hence we know that v occurs only in cliques within the subtree rooted at C. Let y be any vertex in D' which dominates v. Consider the set $D'' = D' - \{y\} \cup \{x\}$. We claim that D'' is a dominating set for G.

Assume to the contrary that there is a vertex z not dominated by D''. Note that z appears in at least one clique which is not below C in the clique tree or it would be in DOM(D) and hence in DOM(D''). Similarly z cannot be in C or it would be in DOM(x) and hence in DOM(D''). The cliques containing z form a path so z appears in no clique of the subtree rooted at C.

We conclude that y must appear in a clique of the subtree rooted at C because it dominates v but that it must also appear in a clique not in that subtree because it dominates z. The requirement that the cliques containing y form a path forces us to conclude that y actually appears in C.

This produces the desired contradiction because the choice of x dictates that high$[x] \geq$ high$[y]$ which in turn implies that for vertices appearing in C or in cliques above C, x can substitute for y as a dominator, whereas for vertices appearing only in cliques below C other vertices of D can be substituted. The path constraint on cliques containing z guarantees that these are the only two cases which arise. The induction is now complete. D'' is a dominating set for G, $|D'| = |D''|$ so D'' is a minimum dominating set, and D dominates every vertex appearing only in cliques which have been visited. □

Cockayne, Goodman and Hedetniemi actually solved a more general problem for trees. They allowed a set R of *required vertices* and a set F of *free vertices* to be specified. Required vertices must always be in D while free vertices are not required to be dominated by D. Initializing D to be R and marking all vertices in F plus all those in or adjacent to R as "dominated" converts our algorithm to this purpose. This can be accomplished in a linear pass over the graph G. Step [3] of the algorithm is then executed as before.

If G is an interval graph we can certainly apply the directed path graph dominating set algorithm, but there is a substantially simpler method. Instead of building a directed path model, which uses the Booth-Lueker interval graph test as a subroutine, we can simply run the interval graph test and use the data structure it builds to directly find a linear arrangement of the cliques [2]. The bottom-up walk of the clique tree becomes a left-to-right scan of the cliques. In practice we would expect this algorithm to be easier to implement and more efficient at run time than the general directed path graph algorithm, although both are of course asymptotically linear in the size of G.

Finally, it should be noted that every tree is a directed path graph so our algorithm applies, but there is obviously no need to explicitly compute a clique tree. The cliques of a tree are its edges. Visiting the edges in a depth-first order corresponds to our algorithm and in fact is the original algorithm of Cockayne, Goodman and Hedetniemi.

4. A possible relationship with graph isomorphism. An alternate construction, different from the one used in Theorem 1, which reduces the general vertex cover problem to the dominating set problem for chordal graphs is the same construction

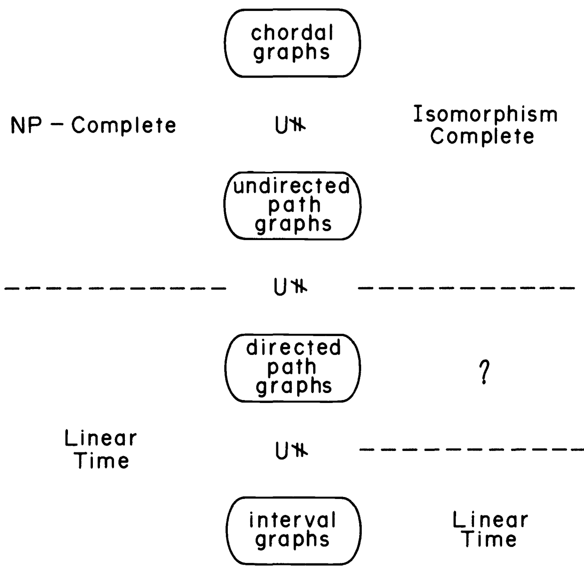


FIG. 3. *The current status of the dominating set and graph isomorphism problems for chordal graphs*

used to prove that isomorphism testing of chordal graphs is complete with respect to graph isomorphism [2, Theorem 5]. This suggests a possible parallel between the dominating set problem and the graph isomorphism problem.

The situation is summarized in Fig. 3. For the dominating set problem we have just shown that the problem for chordal and undirected path graphs is NP-complete and that directed path and interval graphs have linear time algorithms. For graph isomorphism both the chordal and undirected path graphs are known to be isomorphism-complete [1, p. 13] whereas interval graphs have a linear time algorithm [2], [14]. Only the isomorphism question for directed path graphs remains open.

One possibility is that the two problems are related and that the isomorphism problem for directed path graphs has a linear time (or at least polynomial time) solution. We have not been able to prove or disprove this conjecture, but the following observations may be useful.

The linear time isomorphism test for interval graphs follows from the fact that a canonical representation of all possible clique trees for an interval graph can be built in linear time; the algorithm of Dietz, Furst and Hopcroft does construct a clique tree but not a canonical one. The same is true of Gavril's recognition algorithm. Thus we currently see no obvious way of obtaining a polynomial time isomorphism algorithm for directed path graphs, although Paul Dietz is investigating ways of building a canonical clique tree as part of his directed path graph recognition algorithm.

The other possibility, that directed path graphs are isomorphism complete, does not seem to follow from the techniques used for chordal and undirected path graphs because the proof that isomorphism of undirected path graphs is complete with respect to graph isomorphism hinges upon the fact that an arbitrary graph can be represented by a clique tree in which some of the paths travel up one branch of the tree and down another. This behaviour is not allowed in a directed path graph so some other technique will have to be found if we are to prove isomorphism completeness for directed path graphs.

Acknowledgements. The problem of finding dominating sets for chordal graphs was first suggested by Sandra and Steve Hedetniemi. They were also kind enough to point out a number of references in the literature. Conversations with Merrick Furst and John Hopcroft concerning this work have also been very helpful.

REFERENCES

[1] K. S. Booth and C. J. Colbourn, *Problems polynomially equivalent to graph isomorphism,* Department of Computer Science, University of Waterloo, CS-77-04 (1979).

[2] K. S. Booth and G. S. Lueker, *Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms,* J. Comput. System Sci. 13(1976), pp. 335-379.

[3] E. Cockayne, S. Goodman and S. Hedetniemi, *A linear algorithm for the domination number of a tree,* Information Processing Letters 4(1975), pp. 41-44.

[4] P. Dietz, M. Furst and J. Hopcroft, *A linear time algorithm for the generalized consecutive retrieval problem,* Department of Computer Science, Cornell University, TR 79-386 (1979).

[5] D. R. Fulkerson and O. A. Gross, *Incidence matrices and interval graphs,* Pacific J. Math. 15(1965), pp. 835-855.

[6] M. R. Garey and D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness,* W.H. Freeman and Company, San Francisco, 1979.

[7] F. Gavril, *Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph,* this Journal 1(1972), pp. 180-187.

[8] F. Gavril, *The intersection graphs of subtrees in trees are exactly the chordal graphs,*. J. Combin. Theory 16(1974), pp. 47-56.

[9] F. Gavril, *A recognition algorithm for the intersection graphs of directed paths in directed trees,* Discrete Mathematics 13(1975), pp. 237-249.

[10] F. Gavril, *A recognition algorithm for the intersection graphs of paths in trees,* Discrete Mathematics 23(1978), pp. 221-227.

[11] F. Harary, *Graph Theory,* Addison-Wesley, Reading, Massachusetts, 1969.

[12] B. L. Hartnell, *Some problems on minimum dominating sets,* in *Proceedings of the Eighth Southeastern Conference on Combinatorics, Graph Theory and Computing,* Utilitas Mathematica, Winnipeg, Manitoba, (1977), pp. 317-320.

[13] C. G. Lekkerkerker and J. Ch. Boland, *Representation of a finite graph by a set of intervals on the real line,* Fundamenta Mathematicae 51(1962), pp. 45-64.

[14] G. S. Lueker and K. S. Booth, *A linear time algorithm for deciding interval graph isomorphism,* J. Assoc. Comput. Mach. 26(1979), pp. 183-195.

[15] K. S. Natarajan and L. J. White, *Optimum domination in weighted trees,* Information Processing Letters 7(1978), pp. 261-265.

[16] P. L. Renz, *Intersection representation of graphs by arcs,* Pacific J. Math. 34(1970), pp. 501-510.

[17] D. J. Rose, R. E. Tarjan and G. S. Lueker, *Algorithmic aspects of vertex elimination on graphs,* this Journal 5(1976), pp. 266-283.

[18] M. Truszczynski, *The theorem characterizing the acyclic families of sets,* ICS Polish Academy of Sciences Reports 314 (1978).

SIAM J. COMPUT.
Vol. 11, No. 2, May 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1102-0001 $01.00/0

CONTINUOUS DATA TYPES*

M. R. LEVY† AND T. S. E. MAIBAUM‡

Abstract. Data types can be elegantly characterized as the algebraic quotient of the initial algebra in the appropriate class of algebras. In this paper, data types whose domain is continuous (continuous data types) are defined and studied. It is shown that an algebraic quotient of the appropriate initial continuous algebra can be used to characterize continuous data types when certain conditions are satisfied. Two well-known computer science examples are presented to illustrate the results. These types are lists, including infinite lists and control structures considered as operators of a data type.

Key words. abstract data types, initial algebra semantics, continuous domains, lists

1. Introduction. Data types play a central role in programming and it is therefore important to find ways of giving semantic characterizations for them. Some authors have suggested that data types are (many-sorted) algebras (ADJ [1], Guttag), and ADJ [1] have shown that a data type may be characterized as a quotient algebra which is initial in the class of algebras satisfying a set of equations. This algebra is found by factoring a "term algebra" T_Σ by an appropriate congruence q and is denoted T_Σ/q.

A particular class of data types which is of additional interest is the class of data types whose operators are continuous and whose set of objects is a complete partial order or complete lattice (Scott, ADJ [2]). These data types arise when one is considering any types with infinite objects. Circular lists, for example, can be treated as infinite objects of a continuous type (Reynolds). It has been shown (ADJ [2]) that the class of all such data types (hereafter called continuous data types) has an initial algebra denoted CT_Σ which is, intuitively, the algebra of finite *and* infinite terms. It is natural to ask whether the elegant characterization of data types in terms of quotients given by ADJ [1] extends simply to continuous data types. In this paper we show that the quotient CT_Σ/q—where q is obtained from a set of equations in the usual way (ADJ [1])—is sometimes but not always initial in the class of continuous algebras satisfying the equations. Firstly, we show that in general the quotient CT_Σ/q does not admit a partial order which is consistent with the partial order on CT_Σ. Thus, even though CT_Σ/q is a Σ-algebra, it is not a member of the class of continuous Σ-algebras and hence cannot be initial in this class. We then define a function nf, called a normalizer, which is a continuous function that selects a normal form from each class in a congruence q. In order for such a function to exist, the congruence will have to have a property of continuity, namely that the congruence respects limits—that is, if two directed sets are pairwise congruent their least upper bounds must be congruent. It is shown that, given any set of equations, there exists a unique least continuous congruence containing these equations. CT_Σ/q is then "made" into a partial order by defining a partial order relation on CT_Σ/q in terms of the relationship between normal forms. It is then possible to establish the main result of the paper, namely that if q is a continuous congruence generated by a set of equations and a normalizer exists, then CT_Σ/q is initial in the class of continuous Σ-algebras satisfying the equations. Hence continuous data types can be characterized as initial quotients of

* Received by the editors March 21, 1979, and in final revised form December 15, 1980. This work was supported by grants from the Canadian Natural Science and Engineering Research Council and the University of Waterloo.

† Department of Computer Science, University of Victoria, Victoria, British Columbia, Canada V8W 2Y2.

‡ Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1.

CT_Σ just as data types were characterized as quotients of T_Σ by giving a set of equations for the type and by finding a normalizer function for the type.

It is sometimes easier to find an algebra of normal or canonical terms for a data type than to find the normalizer function directly. We show that if such a normal algebra exists, then a normalizer function exists, and thus CT_Σ/q is initial in the appropriate class. The characterization is illustrated with two well-known computer science constructs, namely circular lists and while loops. It is shown that with the appropriate equation for while it is possible to define a quotient for while in terms of if then else. This quotient forms the basis of a model for an equational logic of while programs.

2. Mathematical preliminaries. A data type is viewed here as a many-sorted algebra. (For a discussion of algebras, see Cohn or Grätzer.) This view was put forward previously by ADJ [1], Guttag and also Levy [1]. The notation and results in the section are adopted from ADJ [1], [2]. We assume some familiarity with the definitions and results of ADJ [1], [2], but provide intuitive explanations of the main results.

DEFINITION 1. Let $\mathcal{S}$ be a set whose elements are called *sorts*. An $\mathcal{S}$-sorted operator domain Σ is a family of sets $\Sigma_{w,s}$ of symbols, for $s \in \mathcal{S}$ and $w \in \mathcal{S}^*$, where $\mathcal{S}^*$ is the free monoid on $\mathcal{S}$. $\Sigma_{w,s}$ is the set of *operator symbols of type* $\langle w, s\rangle$, *arity* w and *sort* s.

A Σ-algebra consists of a family $\langle A_s\rangle_{s\in\mathcal{S}}$ of sets called the *carrier* of A, and for each $\langle w, s\rangle \in \mathcal{S}^* \times \mathcal{S}$ and each $\sigma \in \Sigma_{w,s}$ a function

$$\sigma_A : A_{s_1} \times A_{s_2} \times \cdots \times A_{s_n} \to A_s$$

(where $w = s_1 s_2 \cdots s_n$) called the *operation of* A *named by* σ. (If $w = s_1 s_2 \cdots s_n$, then let A^w denote $A_{s_1} \times A_{s_2} \times \cdots \times A_{s_n}$.)

We use $\langle x_s\rangle_{s\in\mathcal{S}}$ to denote a family of objects x_s indexed by s, such that there is exactly one object x_s for each $s \in \mathcal{S}$. The subscript $s \in \mathcal{S}$ will be omitted when the index set $\mathcal{S}$ can be determined from the context. For $\sigma \in \Sigma_{\lambda,s}$ where λ is the empty string, $\sigma_A \in A_s$ (also written $\sigma_A : \to A_s$). These operators are called *constants* of A of sort s. If $s \in \mathcal{S}$, we usually denote the set A_s by s. If $\mathcal{S}$ has only one element then we get the standard definition of a (one-sorted) Σ-algebra. In this case let Σ be a family of sets $\Sigma = \Sigma_0 \cup \Sigma_1 \cup \Sigma_2 \cup \cdots$ such that, for each $\sigma \in \Sigma_n$, there is a function

$$\sigma_A : A \times A \times \cdots \times A \to A.$$

From this point on, all definitions and results in the paper will be for the one-sorted case, although they could be generalized to many-sorted algebras. An example of an algebra is a group which has a single binary operation (the "group operation"), a single unary operation (the inverse operation), and a single constant operation (the group identity).

DEFINITION 2. If A and A' are both Σ-algebras, then a Σ-*homomorphism* is a function

$$h : A \to A'$$

such that if $\sigma \in \Sigma_n$ and $\langle a_1, \cdots, a_n\rangle \in A^n$ then $h(\sigma_A(a_1, \cdots, a_n)) = \sigma_{A'}(h(a_1), \cdots, h(a_n))$.

DEFINITION 3. A Σ-algebra A in a class $\mathbf{C}$ of Σ-algebras is said to be *initial* in $\mathbf{C}$ if and only if for every B in $\mathbf{C}$ there exists a unique homomorphism $h : A \to B$.

THEOREM 1. *The class of all Σ-algebras has an initial algebra called T_Σ. It also has an algebra $T_\Sigma(X)$, called the free algebra on X in the class, such that for any function $f : X \to A$, where A is a Σ-algebra, there is a unique homomorphism $\bar{f} : T_\Sigma(X) \to A$ extending f.*

Intuitively, T_Σ is the algebra of finite terms and $T_\Sigma(X)$ is the algebra of finite terms with variables. The unique homomorphism to an algebra A can be thought of as the evaluation of the terms of T_Σ in A.

DEFINITION 4. A Σ-*equation* is a pair $e = \langle L, R\rangle$ where $L, R \in T_\Sigma(X)$. A Σ-algebra A *satisfies* e if

$$\bar{\theta}(L) = \bar{\theta}(R)$$

for *all* assignments $\theta: X \to A$. If ε is a set of Σ-equations, then A satisfies ε if and only if A satisfies each $e \in \varepsilon$.

Thus a set of equations ε can be viewed as a set of axioms whose free variables are implicitly universally quantified. The class of Σ-algebras which satisfy ε is denoted $\mathbf{Alg}_{\Sigma,\varepsilon}$.

THEOREM 2. $\mathbf{Alg}_{\Sigma,\varepsilon}$ *has an initial algebra called* $T_{\Sigma,\varepsilon}$.

The structure of $T_{\Sigma,\varepsilon}$ can be characterized as an algebraic quotient of T_Σ where intuitively two elements of T_Σ are equivalent if and only if one can be derived from the other by using the equations. That is, $T_{\Sigma,\varepsilon}$ groups together all equivalent terms.

An important concept in the theory of abstract data types is the idea of quotients mentioned above. A quotient partitions the carrier of an algebra, and when this quotient is over T_Σ, it can be interpreted as a way of equating syntactic terms over the alphabet of the type. The importance of such "equations" is that they provide a means for expressing the difficult concept of abstraction. Furthermore, quotients are defined in terms of equivalence relations which are congruences; intuitively, terms that have been equated must behave in the same way with respect to the operators of the type (referential transparency).

DEFINITION 5. A Σ-*congruence* $\equiv$ on a Σ-algebra A is an equivalence relation on A such that, if $\sigma \in \Sigma_n$ and for $1 \leqq i \leqq n$ if $a_i, a'_i \in A$ and $a_i \equiv a'_i$, then

$$\sigma_A(a_1, \cdots, a_n) \equiv \sigma_A(a'_1, \cdots, a'_n).$$

If A is a Σ-algebra and $\equiv$ is a Σ-congruence on A, let $A/\equiv$ be the set of $\equiv$-equivalence classes of A. For $a \in A$ let $[a]$ denote the $\equiv$-class containing a. It is possible to make $A/\equiv$ into a Σ-algebra by defining the operations $\sigma_{A/\equiv}$ as follows.

(i) If $\sigma \in \Sigma_0$, then $\sigma_{A/\equiv} = [\sigma_A]$.

(ii) If $\sigma \in \Sigma_n$ and $[a_i] \in A/\equiv$ for $1 \leqq i \leqq n$, then $\sigma_{A/\equiv}([a_1], \cdots, [a_n]) = [\sigma_A(a_1, \cdots, a_n)]$.

Then it can be shown that $A/\equiv$ is a Σ-algebra called the *quotient* of A by $\equiv$. (The property of $\equiv$ being a congruence ensures that $\sigma_{A/\equiv}$ is well defined.)

A set of Σ-equations $\varepsilon = \{\langle t, t'\rangle \mid t, t' \in T_\Sigma(X)\}$ generates a binary relation $R \subseteq A \times A$ called *the relation generated by* ε. This relation is the set of all pairs $\{\langle\bar{\theta}(t), \bar{\theta}(t')\rangle \mid \theta$ is an assignment$\}$.

THEOREM 3. *If A is a Σ-algebra and R is a relation of A, then there exists a least Σ-congruence relation on A containing R; it is called the congruence relation generated by R on A. (The ordering on Σ-congruences is the subset ordering.)*

THEOREM 4. *If ε is a set of Σ-equations generating a congruence q on T_Σ, then T_Σ/q is initial in* $\mathbf{Alg}_{\Sigma,\varepsilon}$.

The importance of the above theorems is that any set of Σ-equations (axioms) "automatically" defines an algebra which can be regarded as the symbolic model of the object being defined. This model can be used to answer such questions as "Do the axioms characterize some particular model of the type?" and "Is a given implementation of the type correct?".

DEFINITION 6. A *partially ordered set* (*poset*) $(P, \leqq)$ is a set P together with a binary relation $\leqq$ which is reflexive, transitive and antisymmetric.

All posets are here assumed to have a minimum element denoted $\perp$ ("bottom" or undefined) such that $\perp \leqq p$ for any $p \in P$.

DEFINITION 7. A subset S of P is said to be *directed* if and only if every finite subset of S has an upper bound in S. A function $f: P \to P'$ is said to be *monotonic* if and only if for all $p_1 \leqq p_2$ in P, $f(p_1) \leqq f(p_2)$ in P'. Such a function is said to be *continuous* if it preserves all least upper bounds of directed sets that exist in P. That is, f is continuous if and only if

$$f\Big(\bigsqcup_{i \in I} p_i\Big) = \bigsqcup_{i \in I} f(p_i),$$

where $\langle p_i \rangle_{i \in I}$ is a directed set in P and $\bigsqcup_{i \in I} p_i$ denotes the least upper bound of $\langle p_i \rangle_{i \in I}$ if it exists. A poset P is *complete* if and only if all directed sets have least upper bounds in P.

DEFINITION 8. A Σ-algebra is *continuous* if and only if its carrier is *strict* (has a minimum element $\perp$), is complete, and if its operations are continuous. A data type is said to be *continuous* if it is continuous as an algebra. A function $f: A \to B$ is *strict* if $f(\perp_A) = \perp_B$.

The following important result is proved in ADJ [2].

THEOREM 5. *The class of continuous Σ-algebras with strict continuous Σ-homomorphisms, called* $\mathbf{CAlg}_\Sigma$, *has an initial algebra called* CT_Σ.

As before with T_Σ, we let $CT_\Sigma(X_n)$ denote the free Σ-algebra in $\mathbf{CAlg}_\Sigma$ generated by X_n. An element $x_i \in X_n$ is called a *variable*. Intuitively, CT_Σ is the algebra of finite *and* infinite terms, rather than just finite terms as in T_Σ. So, for example, it is possible to express least fixed point solutions of recursive equations as infinite terms in CT_Σ. See ADJ [2] for more details.

DEFINITION 9. The class of all continuous Σ-algebras that satisfy ε together with continuous Σ-homomorphisms between them is denoted $\mathbf{CAlg}_{\Sigma,\varepsilon}$.

We now investigate whether $\mathbf{CAlg}_{\Sigma,\varepsilon}$ has an initial algebra which can be expressed as a quotient of CT_Σ.

3. Normal forms and initiality. Suppose q is a congruence generated by some set of equations. When taking the quotient of CT_Σ by this congruence of q, it will not always be the case that an appropriate partial order relation on CT_Σ/q exists, and hence clearly CT_Σ/q will not be initial in $\mathbf{CAlg}_{\Sigma,\varepsilon}$.

For example, let Σ be defined as

$$\Sigma_0 = \{\perp, a, b\}, \qquad \Sigma_1 = \{\sigma_1, \sigma_2\}, \qquad \Sigma_i = \varnothing, \quad i \geqq 2,$$

and consider the set of equations

$$\sigma_1(\perp) = \sigma_2(b), \qquad \sigma_1(a) = \sigma_2(\perp).$$

Let the function $f: CT_\Sigma \to \{0, 1, 2, 3, 4, 5, 6\}$ be defined as

$$\begin{aligned} f(\perp) &= 4, & f(\sigma_1(b)) &= 3,\\ f(a) &= 5, & f(\sigma_2(\perp)) &= 2,\\ f(b) &= 6, & f(\sigma_2(a)) &= 3,\\ f(\sigma_1(\perp)) &= 1, & f(\sigma_2(b)) &= 1,\\ f(\sigma_1(a)) &= 2, & f(t) &= 3 \quad \text{all other } t. \end{aligned}$$

Ker (f) is an equivalence relation on CT_Σ and is clearly in fact a congruence by construction of f.

Also, by construction, $(\sigma_1(\perp), \sigma_2(b)) \in \text{Ker}(f)$, $(\sigma_1(a), \sigma_2(\perp)) \in \text{Ker}(f)$ so Ker (f) is a congruence in CT_Σ containing the above equations. Now suppose that q is the least congruence on CT_Σ generated by the above equations (thus $q \subseteq \text{Ker}(f)$), and assume that it is possible to define a partial order $\sqsubseteq$ on CT_Σ/q that is consistent with the partial order in CT_Σ, that is $t_1 \sqsubseteq_{CT_\Sigma} t_2 \Rightarrow [t_1] \sqsubseteq [t_2]$.

Then, it must be the case that

$$(\sigma_1(a), \sigma_2(b)) \in q,$$

since

$$\begin{aligned}
[\sigma_1(a)] &= [\sigma_2(\perp)] && \text{by the equations,}\\
&\sqsubseteq [\sigma_2(b)] && \text{since } \sigma_2(\perp) \sqsubseteq_{CT_\Sigma} \sigma_2(b),\\
&= [\sigma_1(\perp)] && \text{again by the equations,}\\
&\sqsubseteq [\sigma_1(a)] && \text{since } \sigma_1(\perp) \sqsubseteq_{CT_\Sigma} \sigma_1(a).
\end{aligned}$$

Hence, $(\sigma_1(a), \sigma_2(b)) \in q$ as required. But if this is the case, then $(\sigma_1(a), \sigma_2(b)) \in$ Ker (f) which is not true, and hence $\sqsubseteq$ cannot exist.

Intuitively, the least congruence generated by the equations is defined independently of any order structure on CT_Σ. It is possible to use different congruences and a different quotient structure, but the simplicity of least congruences suggests that it is worthwhile to try and use them when possible.

For practical reasons, it is often useful to try to characterize a class of values which are equivalent in some equivalence relation by a single representative of the class. We call such a representative a *normal form* of the class. This practical consideration in fact leads to a sufficient condition for guaranteeing the initiality of CT_Σ/q.

DEFINITION 10. Suppose there exists a function

$$\text{nf}: CT_\Sigma \to CT_\Sigma$$

such that, for $t, t_1, t_2 \in CT_\Sigma$ and any congruence q,

(1) $[t_1] = [t_2] \Rightarrow \text{nf}(t_1) = \text{nf}(t_2)$;

(2) $[\text{nf}(t)] = [t]$;

(3) nf is continuous (in the usual ordering on CT_Σ).

Here $[t] = \theta(t)$, where θ is the natural homomorphism induced by the congruence q. nf is called a *normalizer function* (or *normalizer*) for CT_Σ/q.

LEMMA 5. (i) $\text{nf}(\text{nf}(t)) = \text{nf}(t)$. *That is,* nf *is idempotent.*

(ii) $\text{nf}(t_1) = \text{nf}(t_2) \Rightarrow [t_1] = [t_2]$. *That is, two normal forms are equal only if the corresponding terms are equivalent.*

Proof. (i) From property 2 we get $[\text{nf}(t)] = [t]$, so $\text{nf}(\text{nf}(t)) = \text{nf}(t)$ from property 1.

(ii) Suppose $\text{nf}(t_1) = \text{nf}(t_2)$. Then $[t_1] = [\text{nf}(t_1)] = [\text{nf}(t_2)] = [t_2]$. □

Until now, given a set of equations ε, we have considered the least congruence q generated by these equations. However, consider two directed sets $\langle t_i \rangle$, $\langle t_i' \rangle$ in CT_Σ such that

$$t = \bigsqcup_i t_i \quad \text{and} \quad t' = \bigsqcup_i t_i',$$

and such that for each $i \in I$, $(t_i, t_i') \in q$. In general, it would not be true that $(t, t') \in q$,

but this condition is necessary for nf to exist. This is because if nf exists, then

$$\begin{aligned} \mathrm{nf}\left(\bigsqcup_i t_i\right) &= \bigsqcup_i \mathrm{nf}\,(t_i) && \text{by continuity of nf,} \\ &= \bigsqcup_i \mathrm{nf}\,(t_i') && \text{since } (t_i, t_i') \in q \text{ and by property 1 of nf,} \\ &= \mathrm{nf}\left(\bigsqcup_i t_i'\right) && \text{again by continuity.} \end{aligned}$$

Hence, $(t, t') \in q$ by Lemma 5. This motivates a concept called continuous congruence which is defined below. We show that a least continuous congruence exists for an arbitrary set of equations, and that continuous congruences have some desirable properties.

DEFINITION 11. A Σ-congruence q on a continuous Σ-algebra A is said to be *continuous* if whenever there exist two directed sets $\langle t_i \rangle_{i\in I}$ and $\langle t_i' \rangle_{i\in I}$ in A such that for all $i \in I$, $(t_i, t_i') \in q$, then $(\bigsqcup_i t_i, \bigsqcup_i t_i') \in q$.

We now generalize Theorem 3 to continuous Σ-algebras.

THEOREM 6. *If A is a continuous Σ-algebra and R is a relation on A, then there exists a least continuous Σ-congruence on A containing R, called the "continuous congruence relation on A generated by R".*

Proof. Let $\mathscr{K}(R)$ be the class of all continuous Σ-congruence relations on A that contain R. $\mathscr{K}(R) \neq \emptyset$ since

$$U = \langle u_s = A_s \times A_s | s \in \mathscr{S} \rangle$$

is in $\mathscr{K}(R)$, and is continuous since $(a_1, a_2) \in U$ for any $a_1, a_2 \in A$. Let $\equiv_R = \bigcap \mathscr{K}(R)$. It is shown in ADJ [1] that $\equiv_R$ is a Σ-congruence relation. We show that $\equiv_R$ is continuous. Suppose that $\langle a_i \rangle_{i\in I}$ and $\langle a_i' \rangle_{i\in I}$ are directed sets in A with

$$a = \bigsqcup_i a_i, \qquad a' = \bigsqcup_i a_i'.$$

Also, suppose that

$$\langle a_i, a_i' \rangle \in \equiv_R \quad \text{for each } i \in I.$$

Hence,

$$\langle a_i, a_i' \rangle \in K \quad \text{for each } i \in I, \text{ and each } K \in \mathscr{K}(R)$$

by definition of $\equiv_R$. But each $K \in \mathscr{K}(R)$ is continuous, so

$$\langle a, a' \rangle \in K \quad \text{for each } K \in \mathscr{K}(R);$$

thus $\langle a, a' \rangle \in \equiv_R$ as required. □

DEFINITION 12. The *kernel* of a Σ-homomorphism $h: A \to B$ is the relation $\mathrm{Ker}\,(h) = \{\langle a, a' \rangle | a, a' \in A \text{ and } h(a) = h(a')\}$.

It is well known that the kernel of a homomorphism is a congruence.

LEMMA 7. *If A and B are continuous Σ-algebras and $h: A \to B$ is a continuous Σ-homomorphism, then* Ker (h) *is a continuous Σ-congruence.*

Proof. We know Ker (h) is a Σ-congruence and we must prove continuity. Let $\langle a_i \rangle_{i\in I}$ and $\langle a_i' \rangle_{i\in I}$ be directed sets in A such that $(a_i, a_i') \in \mathrm{Ker}\,(h)$ for all $i \in I$. Now

$$\begin{aligned} h\left(\bigsqcup_i a_i\right) &= \bigsqcup_i h(a_i) && \text{since } h \text{ is continuous,} \\ &= \bigsqcup_i h(a_i') && \text{since } h(a_i) = h(a_i') \text{ for all } i \in I, \\ &= h\left(\bigsqcup_i a_i'\right) && \text{since } h \text{ is continuous.} \end{aligned}$$

Thus $(\bigsqcup_i a_i, \bigsqcup_i b_i) \in \mathrm{Ker}\,(h)$. □

We now define a partial order on CT_Σ/q, where q is a continuous congruence, in the case that a normalizer nf exists for q.

DEFINITION 13. Suppose $[t_1], [t_2] \in CT_\Sigma/q$. Then define a partial order relation $\sqsubseteq$ on CT_Σ/q by

$$[t_1] \sqsubseteq [t_2] \quad \text{iff} \quad \mathrm{nf}\,(t_1) \leqq \mathrm{nf}\,(t_2),$$

where $\leqq$ is the partial order relation on CT_Σ.

Note that nf need not be unique, and so the order relation $\sqsubseteq$ also depends on nf. See the corollary of Theorem 11 for clarification.

Consider briefly the following examples of normal forms. For the type linear list with carrier ⟨Atom, List⟩ and operators

$$\mathrm{tl}: \mathrm{List} \to \mathrm{List},$$

$$\mathrm{hd}: \mathrm{List} \to \mathrm{Atom},$$

$$\mathrm{cons}: \mathrm{Atom} \times \mathrm{List} \to \mathrm{List},$$

$$\mathrm{nil}: \to \mathrm{List},$$

the (noncontinuous) normal form of a finite term is an equivalent term with no occurrence of hd or tl. For example,

$$\mathrm{cons}\,(\mathrm{hd}\,(\mathrm{cons}\,(a, \mathrm{nil})), \mathrm{cons}\,(b, \mathrm{nil}))$$

has normal form

$$\mathrm{cons}\,(a, \mathrm{cons}\,(b, \mathrm{nil})).$$

If we regard circular lists as being infinite terms in this algebra and say $t = \bigsqcup_i t_i$ is such an infinite term and each t_i is finite, the normal form of t will be

$$\bigsqcup_i \mathrm{nf}\,(t_i),$$

where nf on finite terms is defined as outlined above. This example is worked out in detail in § 4. (Other important examples of normal forms occur where least fixed point solutions to recursive equations are used as normal forms.)

Sometimes the normalizer will exist but be arbitrary. For example, consider the data type with one sort and equation

$$x + (y + z) = (x + y) + z.$$

Here we can arbitrarily define

$$\mathrm{nf}\,((t_1 + t_2) + t_3) = \mathrm{nf}\,(t_1 + (t_2 + t_3)),$$

$$\mathrm{nf}\,(c + (t_2 + t_3)) = c + \mathrm{nf}\,(t_2 + t_3) \quad \text{for each constant } c.$$

Finally, we exhibit a rather unnatural type where no normal form exists. This type has two nullaries,

$$a, \bot : \to S,$$

and a countable set of unary operators

$$\sigma_i : S \to S, \qquad i \geqq 0.$$

The equations are

$$(*) \qquad \sigma_i(a) = \sigma_{i+1}(\bot), \qquad i \geqq 0.$$

Define a function

$$f: CT_\Sigma \to \mathcal{N} + \{a\},$$

where $\mathcal{N}$ is the set of natural numbers and + is a disjoint union operator as follows:

$$f(\perp) = 0, \qquad f(a) = a,$$

$$f(\sigma_i(t)) = \begin{cases} i+2, & t = a, \\ i+1, & \text{otherwise.} \end{cases}$$

Notice that this definition is complete since any term $t \in CT_\Sigma$ may be written as $t = \bigsqcup_i t_i$ for a directed set $\langle t_i \rangle_{i \in I}$ in CT_Σ, and if t is not finite then

$$t = \bigsqcup_i \sigma_j(t'_i) \qquad \text{for some } \sigma_j,$$

$$= \sigma_j\Big(\bigsqcup_i (t'_i)\Big) \quad \text{by continuity of } CT_\Sigma.$$

Clearly Ker (f) is an equivalence relation on CT_Σ, and it is in fact easily seen to be a congruence, since if $f(t_1) = f(t_2)$ and $t_1 = \sigma_i(t'_1)$, $t_2 = \sigma_i(t'_2)$ for some i and if $t'_1 = a$ then $t'_2 = a$ since $f(t_2) = f(t_1)$.

Also, Ker (f) is a continuous congruence, since if $\langle t_i \rangle$, $\langle t'_i \rangle$ are two infinite directed sets in CT_Σ such that $f(t_i) = f(t'_i)$ for each i, then it must be the case that $f(t_i) = f(t'_i) = j$ since each t_i, t'_i must be of the form $\sigma_{j-2}(t)$. Hence, $f(\bigsqcup_i t_i) = f(\bigsqcup_i t'_i) = j$.

Finally, the equations are in Ker (f) since

$$f(\sigma_i(a)) = i+2 \quad \text{and} \quad f(\sigma_{i+1}(\perp)) = i+1+1 = i+2.$$

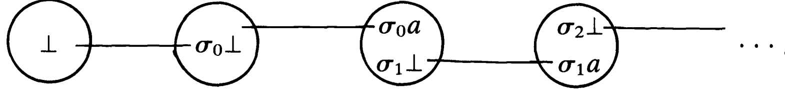
Now, consider the chain

$$0 \leqq 1 \leqq 2 \leqq \cdots,$$

derived by f from

$$\perp \leqq \sigma_0 \perp \leqq \sigma_0 a = \sigma_1 \perp \leqq \sigma_1 a \leqq \cdots,$$

diagrammatically



This chain clearly has no upper bound, since if it did it would have to be one of the portions $0, 1, 2, \cdots, a$, say k. But there is no element in k greater than any of the elements in $k+1$, and so no upper bound exists.

Now suppose that q is the least continuous congruence containing the equations (*) above. It can hence be shown, since $q \subseteq \text{Ker}(f)$, that with the above equations CT_Σ/q cannot be complete with respect to any ordering that is consistent with the ordering $\leqq$ on CT_Σ. Thus CT_Σ/q is not in the class of continuous algebras satisfying the equations (*) and is thus not initial. This also illustrates, since q is continuous, that continuity is not sufficient to guarantee initiality. In the remaining discussion it is assumed that nf does exist.

LEMMA 8. $\sqsubseteq$ *is a partial order on* CT_Σ/q.

Proof. Obvious, since $\leqq$ is a partial order on CT_Σ. □

LEMMA 9. *Let* $\langle t_i \rangle_{i \in I}$ *be a set directed in* CT_Σ, *and* q *a continuous congruence, and suppose a normalizer* nf *exists. Then* $\langle [t_i] \rangle$ *is directed in* CT_Σ/q, *and it has a least upper bound denoted* $\bigsqcup_i [t_i]$ such that $\bigsqcup_i [t_i] = [\bigsqcup_i t_i]$.

Proof. Let $t = \bigsqcup_i t_i$, which exists since CT_Σ is complete. By the monotonicity of nf and definition of $\sqsubseteq$, $\langle [t_1] \rangle$ is directed. Now for all $i \in I$, $t_i \leqq t$ since t is the least upper

bound of $\langle t_i \rangle_{i\in I}$. Hence for all i nf $(t_i) \leqq$ nf (t) so for all i $[t_i] \sqsubseteq [t]$ by definition of $\leqq$. So $[t]$ is an upper bound of $\langle [t_i] \rangle_{i\in I}$. Now suppose that for some t', for all $i \in I$, $[t_i] \sqsubseteq [t']$. Then for all $i \in I$, nf $(t_i) \leqq$ nf (t'). But since nf is continuous and $\langle t_i \rangle_{i\in I}$ is directed, $\langle \text{nf}(t_i) \rangle_{i\in I}$ is directed and

$$\bigsqcup_i \text{nf}(t_i) \leqq \text{nf}(t') \quad \text{by continuity of nf,}$$

so

$$\text{nf}\left(\bigsqcup_i t_i\right) \leqq \text{nf}(t') \quad \text{by definition of } \leqq,$$

and

$$\left[\bigsqcup_i t_i\right] \sqsubseteq [t'].$$

Hence, $[\bigsqcup_i t_i]$ is the least upper bound of $\langle [t_i] \rangle_{i\in I}$. That is,

$$\bigsqcup_i [t_i] = \left[\bigsqcup_i t_i\right]. \qquad \square$$

LEMMA 11. *If B is a continuous Σ-algebra satisfying ε, q is the continuous congruence generated by ε, $t_1, t_2 \in CT_\Sigma$ and $(t_1, t_2) \in q$ and*

$$h_B : CT_\Sigma \to B$$

is the unique homomorphism guaranteed to exist by the initiality of CT_Σ, then $h_B(t_1) = h_B(t_2)$.

Proof. Let Ker (h_B), the kernel of h_B, be defined as before. We know that Ker (h_B) is a continuous congruence and, moreover, $\varepsilon(B) \subseteq$ Ker (h_B) where $\varepsilon(B)$ is the relation on B generated by the set of equations ε. This is because for each assignment $\theta: \mathrm{X} \to \mathrm{B}$ and each $\langle L, R\rangle \in \varepsilon$, $\bar{\theta}(L) = \bar{\theta}(R)$. Now $\bar{\theta} = h_B$ by uniqueness of h_B and hence $h_B(L) = h_B(R)$. But q is the least continuous congruence satisfying ε (containing $\varepsilon(B)$) and so $q \subseteq$ Ker (h_B). Thus $(t_1, t_2) \in$ Ker (h_B) and hence $h_B(t_1) = h_B(t_2)$ as required. $\square$

Much of the power of considering abstract data types as many-sorted algebras centers around the property of isomorphism. An isomorphism is a homomorphism that is injective and surjective, that is, 1-to-1 and onto. If there exists an isomorphism $h: A \to B$, we write $A \cong B$ and say that A is isomorphic to B. Different implementations of the same data type can be considered members of a class of isomorphic algebras. In order to characterize this class precisely, the concept of initial algebra is used. The initial algebra in a class of algebras contains in some sense the least amount of information needed to specify a member of the class. Thus we would like to say that a particular abstract data type *is* the initial algebra in a class of algebras satisfying the specifications. Initiality ensures that the operators do no more than required by the specification. ADJ [1] shows that T_Σ/q is initial in the class of algebras satisfying the equations which generate the congruence q. It is natural to ask whether or not CT_Σ/q is initial, and we show that if the normalizer nf exists, then indeed CT_Σ/q is initial (where q now is the least *continuous* Σ-congruence generated by the equations).

THEOREM 11. *If a normalizer* nf *exists for q, then CT_Σ/q is initial in* $\mathbf{CAlg}_{\Sigma,\varepsilon}$.

Proof. We must find a unique

$$h_B : CT_\Sigma/q \to B,$$

for any $B \in \mathbf{CAlg}_{\Sigma,\varepsilon}$. By Theorem 5, $h_1 : CT_\Sigma \to B$ exists, and is unique. Now define

$$h_B([t]) = h_1(\text{nf}(t)).$$

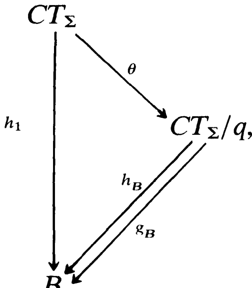
(i) h_B is a Σ-homomorphism. We must show that

$$h_B([\sigma(t_1, \cdots, t_n)]) = \sigma(h_B([t_1]), \cdots, h_B([t_n]))$$

for any $\sigma \in \Sigma_n$.

$$\begin{aligned}
&h_B([\sigma(t_1, \cdots, t_n)]) \\
&\quad = h_1(\mathrm{nf}\,(\sigma(t_1, \cdots, t_n))) && \text{by definition of } h_B, \\
&\quad = h_1(\sigma(t_1, \cdots, t_n)) && \text{by Lemma 10 and Property 2 of nf}, \\
&\quad = \sigma(h_1(t_1), \cdots, h_1(t_n)) && \text{since } h_1 \text{ is a homomorphism}, \\
&\quad = \sigma(h_1(\mathrm{nf}\,(t_1)), \cdots, h_1(\mathrm{nf}\,(t_n))) && \text{again by Lemma 10 and Property 2 of nf}, \\
&\quad = \sigma(h_B([t_1]), \cdots, h_B([t_n])) && \text{by definition of } h_B.
\end{aligned}$$

(ii) h_B is unique. Suppose there is a $g_B: CT_\Sigma/q \to B$ such that g_B is a homomorphism. Now consider the diagram



where θ is the natural homomorphism induced by a. If g_B exists, then we must have $\theta \circ h_B = \theta \circ g_B = 1$ since $\theta \circ h_B$ and $\theta \circ g_B$ are both homomorphisms into B. But θ is onto, hence $h_B = g_B$.

(iii) CT_Σ/q is complete. Let $\langle [t_i] \rangle_{i \in I}$ be directed in CT_Σ/q. Then, by the definition of $\leqq$, $\langle \mathrm{nf}\,(t_i) \rangle_{i \in I}$ is directed in CT_Σ. Applying Lemma 9, and since $[t] = [\mathrm{nf}\,(t)]$ for any $t \in CT_\Sigma$, we get

$$\bigsqcup_i [t_i] = \bigsqcup_i [\mathrm{nf}\,(t_i)] = \left[\bigsqcup_i \mathrm{nf}\,(t_i) \right] = [t],$$

where $t = \bigsqcup_i \mathrm{nf}\,(t_i)$ exists since CT_Σ is complete.

(iv) CT_Σ/q is continuous. By (iii) CT_Σ/q is complete. $[\perp]$ is the minimum element of CT_Σ/q since nf is continuous, and hence $\mathrm{nf}\,(\perp) = \perp$. Now we must show that for each $\sigma \in \Sigma_n$, and each $1 \leqq j \leqq n$, $\sigma([t_1], \cdots, \bigsqcup_i [t_j^i], \cdots, [t_n]) = \bigsqcup_i \sigma([t_1], \cdots, [t_j^i]; \cdots, [t_n])$; there results

$$\begin{aligned}
&\sigma([t_1], \cdots, \bigsqcup_i [t_j^i], \cdots, [t_n]) \\
&\quad = \sigma\left([t_1], \cdots, \left[\bigsqcup_i t_j^i\right], \cdots, [t_n]\right) && \text{by Lemma 9}, \\
&\quad = \left[\bigsqcup_i (\sigma(t_1, \cdots, t_j^i, \cdots, t_n))\right] && \text{by definition and continuity of } \sigma, \\
&\quad = \bigsqcup_i [\sigma(t_1, \cdots, t_j^i, \cdots, t_n)] && \text{by Lemma 9}, \\
&\quad = \bigsqcup_i (\sigma([t_1], \cdots, [t_j^i], \cdots, [t_n])) && \text{by definition of } \sigma.
\end{aligned}$$

(v) h_B is continuous. We must show that if $[t]$ is the least upper bound of a directed set $\langle [t_i] \rangle_{i \in I}$ in CT_Σ/q then

$$h_B([t]) = \bigsqcup_i h_B([t_i]).$$

By Lemma 9 and part (iii) above we know that if $\langle [t_i] \rangle_{i \in I}$ is directed, and if $t = \bigsqcup_i \text{nf}(t_i)$, then

$$\bigsqcup_i [t_i] = \bigsqcup_i [\text{nf}(t_i)] = [t] = \left[\bigsqcup_i t_i\right] = \left[\bigsqcup_i \text{nf}(t_i)\right].$$

Now

$$\begin{aligned} h_B([t]) &= h_1(\text{nf}(t)) && \text{by definition of } h_B, \\ &= h_1\left(\text{nf}\left(\bigsqcup_i \text{nf}(t_i)\right)\right) && \text{by definition of } t, \\ &= h_1\left(\bigsqcup_i \text{nf}(t_i)\right) && \text{since nf is continuous and idempotent,} \\ &= \bigsqcup_i h_1(\text{nf}(t_i)) && \text{since } h_1 \text{ is continuous,} \\ &= \bigsqcup_i h_B([t_i]) && \text{by definition of } h_B \text{ as required.} \end{aligned}$$

COROLLARY 12. *The particular normalizer chosen will not affect the ordering on CT_Σ/q, because any two initial algebras must be isomorphic and have the same structure.*

In practice, an algebra of normal forms is useful for establishing properties about an abstract data type, and this leads us to another definition. (This will be a generalization of the concept of canonical term algebra in ADJ [1].)

DEFINITION 14. A continuous Σ-algebra $\mathscr{L}_\Sigma$ is called a *normal term algebra* for q if

(i) The carrier of $\mathscr{L}_\Sigma$ is a subset of the carrier of CT_Σ, where if $l_1, l_2 \in \mathscr{L}_\Sigma$ then $l_1, \sqsubseteq_{\mathscr{L}_\Sigma} l_2$ if and only if $l_1 \sqsubseteq_{CT_\Sigma} l_2$;

(ii) $\mathscr{L}_\Sigma \cong CT_\Sigma/q$;

and

(iii) $h: CT_\Sigma \to \mathscr{L}_\Sigma$ is a normalizer, where h is the unique homomorphism guaranteed to exist by initiality of CT_Σ.

If a normalizer exists, then it is possible to construct a normal term algebra.

THEOREM 13. *Let* $\text{nf}: CT_\Sigma \to CT_\Sigma$ *be a normalizer, and define a Σ-algebra $\mathscr{L}_\Sigma$ as*:

(i) *The carrier is $\mathscr{L}$, $\mathscr{L} = \{\text{nf}(t) | t \in CT_\Sigma\}$.*

(ii) *For each $\sigma \in \Sigma$, $\sigma_{\mathscr{L}}(\text{nf}(t_1), \cdots, \text{nf}(t_n)) = \text{nf}(\sigma(t_1, \cdots, t_n))$.*

Then $\mathscr{L}_\Sigma$ is a normal term algebra.

Proof. Let $g: \mathscr{L}_\Sigma \to CT_\Sigma/q$ be defined as the restriction of the natural homomorphism θ to $\mathscr{L}_\Sigma$. Then;

(i) g is a homomorphism, since θ is.

(ii) g is surjective, since if $[t] \in CT_\Sigma/q$, then $[t] = [\text{nf}(t)]$ since nf is a normalizer, and so $g(\text{nf}(t)) = [t]$ by definition of g.

(iii) g is injective, since if $g(\text{nf}(t_1)) = g(\text{nf}(t_2))$ then $[\text{nf}(t_1)] = [\text{nf}(t_2)]$ by definition of g, and so $\text{nf}(t_1) = \text{nf}(t_2)$ since nf is a normalizer.

Finally, since nf is a homomorphism, $\text{nf} = h$ the unique homomorphism from CT_Σ to $\mathscr{L}_\Sigma$. □

The converse of this theorem, namely that if a normal algebra exists then there is a normalizer, is a trivial consequence of conditions (iii) of Definition 14.

4. Examples.

4.1. Finite and infinite lists. Let $S = \langle \text{List}, \text{Atom} \rangle$ be a set of sorts with

$$\begin{aligned} a &: \to \text{Atom} \quad \text{for each } a \in \text{Atom}, \\ \bot &: \to \text{List}, \\ \text{nil} &: \to \text{List}, \\ \text{cons} &: \text{Atom} \times \text{List} \to \text{List}, \\ \text{hd} &: \text{List} \to \text{Atom}, \\ \text{tl} &: \text{List} \to \text{List} \end{aligned}$$

being a set of operators. Let

$$\Sigma' = \{\text{cons}, \bot, \text{nil}\} \cup \text{Atom}, \qquad \Sigma = \{\text{cons}, \text{hd}, \text{tl}, \bot, \text{nil}\} \cup \text{Atom}.$$

Let ε be

$$\text{hd}\,(\text{cons}\,(a, l)) = a, \qquad \text{hd}\,(\bot) = \text{tl}\,(\bot) = \text{hd}\,(\text{nil}) = \bot,$$

$$\text{tl}\,(\text{cons}\,(a, l)) = l, \qquad \text{tl}\,(\text{nil}) = \bot.$$

We show that there is a normal term algebra for the congruence q generated by the above axioms.

(i) Let the carrier $\mathscr{L}$ of $\mathscr{L}_\Sigma$ be the carrier of $CT_{\Sigma'}$.

(ii) Define

$$a_{\mathscr{L}} = a, \qquad \text{tl}_{\mathscr{L}}\,(\text{cons}\,(a, l)) = l,$$

$$\text{hd}_{\mathscr{L}}\,(l) = \bot \text{ for } l = \bot \text{ or } l = \text{nil}, \qquad \text{tl}_{\mathscr{L}}\,(\text{nil}) = \bot,$$

$$\text{hd}_{\mathscr{L}}\,(\text{cons}\,(a, l)) = a, \qquad \text{cons}_{\mathscr{L}}\,(a, l) = \text{cons}\,(a, l).$$

Since CT_Σ is initial in the class of all continuous Σ-algebras there exists a unique homomorphism $h: CT_\Sigma \to \mathscr{L}_\Sigma$ ($\mathscr{L}_\Sigma$ is clearly continuous).

The kernel of a continuous homomorphism is a continuous congruence. Furthermore, it is easy to see that $\varepsilon(R) \subseteq \text{Ker}\,(h)$. (For example, $h(\text{hd}\,(\text{cons}\,(a, l))) = \text{hd}_{\mathscr{L}}\,(\text{cons}_{\mathscr{L}}\,(a, l)) = a_{\mathscr{L}} = h(a)$.) Thus, by the minimality of q, $q \subseteq \text{Ker}\,(h)$. Now:

(1) If $(t_1, t_2) \in q$, then $(t_1, t_2) \in \text{Ker}\,(h)$ by the above, so $h(t_1) = h(t_2)$ as required.

(2) We must show that $(h(t), t) \in q$.

(i) If t is finite then:

(a) If t is an atom or $\bot$, $h(t) = t$ and the result follows. Else

(b) Suppose that for any t of depth n $(h(t), t) \in q$.
Then $h(t)$ must be of the form cons (a, l).
Now consider the term hd (t),

$$h(\text{hd}\,(t)) = \text{hd}_{\mathscr{L}}(h(t)) = \text{hd}_{\mathscr{L}}\,(\text{cons}\,(a, l)) = a.$$

But, $(h(t), t) \in q$ (by induction assumption),
so $(\text{cons}\,(a, l), t) \in q$,
so $(\text{hd}\,(\text{cons}\,(a, l)), \text{hd}\,(t)) \in q$ since q is a congruence,
hence, $(a, \text{hd}\,(t)) \in q$
and $(h(\text{hd}\,(t)), \text{hd}\,(t)) \in q$ as required.

The arguments for tl and cons go through similarly.

(ii) Suppose that t is infinite. Then, $t = \bigsqcup_i t_i$ where each t_i is finite.
Now, $h(t) = h(\bigsqcup_i t_i) = \bigsqcup_i h(t_i)$ by continuity of h.
By (i) above, $(t_i, h(t_i)) \in q$, and since q is continuous, $(t, \bigsqcup_i, h(t_i)) \in q$, so $(t, h(t)) \in q$ as required.

(3) h is continuous by definition, and is thus a normalizer.

We have thus shown that $\mathscr{L}_\Sigma$ is a normal term algebra for the above axioms, and that CT_Σ/q is initial in the class of all algebras satisfying ε. Note that minimality of q is used in part (1) of the above proof and continuity in Part (2ii). Establishing the existence of normal forms for this type is both natural and straightforward. Reasoning about the type uses the usual tools of equational logic, but with an additional "continuity property", namely that if we prove $t_i = t_i'$ for two directed sets $\langle t_i \rangle$ and $\langle t_i' \rangle$, then we can prove $\bigsqcup t_i = \bigsqcup t_i'$.

4.2. Recursive functions. An important continuous data type is the algebraic characterization of recursive equations.

Sorts: Bool, Command.
Operations: ifthenelse: Bool × Command × Command → Command
whiledo: Bool × Command → Command
; : Command × Command → Command
null: → Command
$\perp$: → Bool
b_i: → Bool for $i \geqq 0$.

Now let t^{inf} denote the least fixed point solution in Bool × Command → Command of the definition:

$$w \Leftarrow \lambda b \lambda s, \text{ if } b \text{ then } s; w(b, s) \text{ else null.}$$

We introduce the single equation,

$$\text{while } b \text{ do } s = t^{\text{inf}}(b, s).$$

PROPOSITION 15. *There exists a normalizer for the above algebra.*

Proof. Let nf: Bool → Bool be the identity. Define nf: Command → Command as follows:

$$\begin{aligned}
&\text{nf (if } b \text{ then } s_1 \text{ else } s_2) = \text{if } b \text{ then nf }(s_1) \text{ else nf }(s_2),\\
&\text{nf }(s_1; s_2) = \text{nf }(s_1); \text{nf }(s_2),\\
&\text{nf (null)} = \text{null},\\
&\text{nf (while } b \text{ do } s) = t^{\text{inf}}(b, \text{nf }(s)).
\end{aligned}$$

A simple inductive argument will show that $[\text{nf }(t)] = [t]$. Secondly, we can show that Ker (nf) is a continuous congruence since nf is a homomorphism and it contains the congruence generated by the equation. Hence, since q is minimal,

$$[t_1] = [t_2] \Rightarrow (t_1, t_2) \in \text{Ker (nf)} \Rightarrow \text{nf }(t_1) = \text{nf }(t_2) \quad \text{as required.}$$

nf is easily seen to be continuous. □

We thus have a model for this flow-diagram-like language. It can be seen, for example, that

$$\text{while } b \text{ do } s = \text{if } b \text{ then } s; \text{while } b \text{ do } s \text{ else null,}$$

by substituting t^{inf} on both sides of this equation. In general, in fact, any recursive function can be treated in a similar way, thus giving rise to a logic for reasoning about recursive functions.

This reasoning could then be applied, for example, to formalize the (informal) logic of systems of recursive definitions outlined in Burstall and thus give a formal basis for their program transformation system. Similarly, the establishment of a library of pairs of equivalent recursive functions, as suggested in Burstall and Courcelle [3], could be based on such a logic.

5. Relation to other work. Several authors have studied quotient algebras in some form (ADJ [1], [3], Courcelle [1], Lehmann and Hennessy). ADJ [1] is concerned with the class of all Σ-algebras (rather than of continuous Σ-algebras), and it is the main results of ADJ [1] that have been generalized here, using the notion of normal forms. In Courcelle [1], Courcelle and Nivat investigate quotients of Σ-algebras taken from congruences that have been defined in terms of pre-orders (rather than simply the least congruence generated by a set of equations), but they do not examine the initiality of CT_Σ/q. Hennessy has shown, independently of the present work, that the completion of T_Σ/q is initial in the class of Σ-algebras satisfying q, where q is the congruence obtained using Courcelle and Nivat's construction on pre-orders and the class of algebras of interest is expressed in terms of a set of *in*equalities rather than with equations. As a consequence of the main theorem of this paper (Theorem 12), the initial algebra of Hennessy will be isomorphic to CT_Σ/q when normal forms exist. (Note that a set of equations $\{t_1 = t_1', t_2 = t_2', \cdots, t_n = t_n'\}$ may be regarded as the set of inequalities $\{t_1 \leqq t_1', t_1' \leqq t_1, t_2 \leqq t_2', t_2' \leqq t_2, \cdots, t_n \leqq t_n', t_n' \leqq t_n\}$.)

Lehmann has also investigated independently the initial algebra in a continuous equational class using a categorical framework, and has shown that the completion of T_Σ/q is initial in this class. This result is essentially the same as that of Hennessy. ADJ [3] have investigated quotients in so-called rational algebraic theories. In Courcelle [2], Courcelle has independently shown the initiality of CT_Σ/q also using normal forms. The major difference between Courcelle [2] and this paper is in the characterization of the congruence q and in the notational framework. We take q to be the least (continuous) congruence containing some relation where Courcelle defines q in terms of a preorder derived from the preorder on CT_Σ and the given relation. In this paper we illustrate the usefulness of using minimal continuous congruences, which are simply and elegantly characterized. Furthermore the idea of continuous congruences is described independently of normal forms, this concept being natural and interesting in its own right. For example, it is easy to see that the kernel of a continuous homomorphism is a continuous congruence and that the class of continuous congruences on a given continuous algebra is a complete lattice. We believe that using minimal congruences provides a simpler mechanism for defining quotients than the preorder cum completion construction used in Courcelle [1], [2].

The results in this paper were strongly motivated by the consideration of types where either it would be desirable for normal forms to exist or it was clear that they did exist (see, for example, Levy [1], [2]). Normal forms are also important for expressing simply the "value" of a computation or when considering the problem of decidability of two expressions. Huet has investigated the existence of normal forms in a noncontinuous framework, and Berry and Courcelle (in Berry) have investigated classes of interpretations where normal forms (called canonical terms by them) exist. In addition, Courcelle (in Courcelle [2], [3]) has studied conditions under which (what are essentially our) normal forms exist for an equationally specified continuous class of algebras.

The present paper thus provides a simple extension of ADJ [1], avoiding the more complex constructions of Lehmann, Hennessy, Courcelle [2], [3] and ADJ [3]

in the useful case where normal forms exist. In practice, the biggest advantage of this approach is that the congruence q considered is just the "usual" *least* congruence containing a set of equations, or possibly the least continuous congruence containing a set of equations. Further, the algebra CT_Σ/q is just the quotient of CT_Σ by q in the usual algebraic sense rather than being a more complex completion. This minimality of q is an extremely useful fact that can be used for proving various properties of continuous data types, a property in general absent from congruences derived from completions of pre-orders. (See ADJ [1] and Levy [1], [2] for some more uses of minimality of congruences in proofs.) Thus the main thrust of this paper differs from the other papers cited in that the concern is not so much "Does initial algebra exist in a continuous equational class?" but "Is CT_Σ/q initial in this class?".

6. Conclusions. Continuous data types arise naturally in many settings when one is studying the semantics of programs and data. Data types are elegantly characterized by universal algebras, where one of the most powerful tools used in the study of universal algebra is the construction of quotient algebras. It would have been useful to be able to use this technique in the more restricted domain of continuous algebras. This, however, turned out to be impossible as the quotient of a continuous algebra by an arbitrary congruence may not yield a continuous algebra (as the quotient set may not admit a partial order or, even if it does, the partial order may not be complete).

Our purpose in this report was to characterize continuous data types by finding conditions under which the quotient of the initial continuous algebra by some congruence *would* yield a continuous quotient algebra. We were particularly interested in the case where the congruence was generated by a set of equations (axioms). Two simply stated conditions suffice for this purpose. Firstly, the congruence has to have a special property called continuity. A continuous congruence is one which relates (puts in the same congruence class) upper bounds of directed sets if the elements of the directed sets are pairwise related by the congruence. This seems to be a highly desirable and natural property of congruences. For example, we showed that the kernel of a continuous homomorphism between continuous algebras is a continuous congruence.

Secondly, the congruence has to be such that from each congruence class a unique representative can be chosen by using a (continuous) map called a normalizer. The normal form (image under the normalizer) of an expression is a generalization of the canonical form of an expression introduced in the study of abstract data types (see ADJ [1]). (In contrast to canonical forms, however, normalizers do not always exist.) Since our motivation for studying quotients was to generalize the work on abstract data types to the setting of continuous algebras, this was a natural place to start. Moreover, continuous data types which did not have normal forms would be impossible to represent (even aside from problems of representing infinite objects by finite ones).

As indicated above, the motivation for this work was the need to generalize the work on abstract data types to the continuous setting in order to handle "naturally infinite" objects (such as lists represented by equations of the form $x = \text{cons}\,(a, x)$). The "list" represented by this equation is clearly $\text{cons}\,(a, \text{cons}\,(a, \text{cons}\,(a, \cdots)\cdots))$. Such objects do not exist in the usual algebras of finite objects studied in the theory of abstract data types. Moreover, it turns out that concepts such as sharing and circularity in the definition of structures are best handled by resorting to continuous data types. These ideas are developed further in Levy [1] and Levy [2].

The second example in the preceding section illustrates how the results in this paper may be applied to define an equational logic for recursive function definitions.

REFERENCES

ADJ [1]. J. A. GOGUEN, J. W. THATCHER, E. G. WAGNER AND J. B. WRIGHT, *An initial algebra approach to the specification, correctness and implementation of abstract data types*, Current Trends in Programming Methodology, Vol. IV, R. T. Yeh, ed., Prentice-Hall, Englewood Cliffs, NJ, 1978.

ADJ [2]. ———, *Initial algebra semantics and continuous algebras*, J. Assoc. Comput. Mach., 24 (1977), pp. 68–95.

ADJ [3]. ———, *Rational algebraic theories and fixed point solutions*, Proc. 17th IEEE Symposium on Foundations of Computer Science, Houston, 1976, pp. 147–158.

Berry. G. BERRY AND B. COURCELLE, *Program equivalence and canonical forms in stable discrete interpretations*, Proc. 3rd Colloquium on Automata, Languages and Programming, University of Edinburgh, 1976.

Burstall. R. M. BURSTALL AND J. DARLINGTON, *Some transformations for developing recursive programs*, Proc. International Conference on Reliable Software, SIGPLAN Notices (10) (1975), pp. 465–472.

Cohn. P. M. COHN, *Universal Algebra*, Harper and Row, New York, 1965.

Courcelle [1]. B. COURCELLE AND M. NIVAT, *Algebraic families of interpretations*, Proc. 17th IEEE Symposium on Foundations of Computer Science, Houston, 1976, pp. 137–146.

Courcelle [2]. B. COURCELLE, *On recursive equations having a unique solution*, Proc. 19th IEEE Symp. on Foundations of Comp. Sci., Ann Arbor, 1978.

Courcelle [3]. ———, *Infinite trees in normal form and recursive equations having a unique solution*, Mathematical Systems Theory 13 (1979), U. de Bordeaux I, 1979, pp. 131–180.

Grätzer. G. GRÄTZER, *Universal Algebra*, D. Van Nostrand, New York, 1968.

Guttag. J. GUTTAG, *Abstract data types and the development of software*, Comm-ACM, 20 (1977), pp. 396–404.

Hennessy. M. C. HENNESSY, *Initial algebras and Herbrand interpretations*, Technical Report, University of Pernambuco, Recife, Brazil, 1978.

Lehman. D. J. LEHMAN, *On the algebra of order*, Proc. 19th IEEE Symposium on Foundations of Computer Science, Ann Arbor, 1978.

Levy [1]. M. R. LEVY, *Data types with sharing and circularity*, Ph.D. thesis, Technical Report CS-78-26, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1978.

Levy [2]. M. R. LEVY AND T. S. E. MAIBAUM, in preparation.

Reynolds. J. C. REYNOLDS, *Notes on a lattice theoretic approach to the theory of computation*, Course Notes, Syracuse University, Syracuse, NY 1972.

Scott. D. S. SCOTT, *The lattice of flow diagrams*, Symposium on Semantics of Algorithmic Languages, Lecture Notes in Mathematics, 188, Springer-Verlag, New York, 1971.

SIAM J. COMPUT.
Vol. 11, No. 2, May 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1102-0002 $01.00/0

GEOMETRIC PROBLEMS WITH APPLICATION TO HASHING*

DOUGLAS COMER† AND MICHAEL J. O'DONNELL†‡

Abstract. Efficient algorithms are presented for two geometric problems. Both problems involve finding the best projection of a set of points from two-space onto a line, with two different notions of "best". The key technique is to identify critical angles in between which the functions to be optimized have nice trigonometric forms that can be solved exactly. Applications to hashing arise when we look for the best linear combination of two hashing functions.

Key words. hashing, geometry problems, geometric complexity, geometric algorithms

1. Introduction. Recently researchers have found efficient algorithms for several geometric problems [GRAH72], [SHAM75], [SHAM75a], [JARV73], [PREP77]. These include convex hull in 2- and 3-dimensions as well as nearest neighbor problems. This paper presents algorithms for two new geometric problems. Both problems have applications to the choice of good hashing functions.

The first problem is motivated by Sprugnoli [SPRU77], who studies perfect hashing functions for a static set of keys, proposing solutions which seem to require large amounts of space. No analysis of the space requirements is given, however, so a quantitative assessment is not available. Sprugnoli's work suggests the following problem: given a fixed set of keys S, and two functions h_1, h_2 which map S to Z^+, find constants c_1, c_2 and c_3 such that the hash function $h(x)=|c_1h_1(x)+c_2h_2(x)+c_3|$ produces a minimum table size perfect hashing of S.

A related, practically motivated problem raised in [COME79] also concerns finding an optimum linear combination of two functions. In this problem the hashing function need not be perfect; however, we allow only b buckets in the hash table (fixed b) and minimize cost based on the bucket sizes. Typical cost functions might measure the maximum bucket size, the number of empty buckets or the uniformity of distribution.

Section 2 defines the geometric problems underlying the hashing problems above; §§ 3 and 4 present the basic algorithms and their analysis; and § 5 discusses the applications in greater detail. Finally, § 6 concludes with a summary of open problems.

2. Definitions.

DEFINITION 1. Let S be a finite subset of $R\times R$, and let $\theta\in[0,\pi)$ be an *angle of projection*. The *projection* of S at angle θ is

$$P_\theta^S=\{x\cos\theta+y\sin\theta\,|\,(x,y)\in S\}.$$

Where S is understood, we write P_θ^S as P_θ.
The *span* of projection P_θ is

$$\text{span}\,(P_\theta)=\max\,(\{|u-v|\,|\,u,v\in P_\theta \text{ and } u\neq v\}).$$

The *resolution* of projection P_θ is

$$\text{res}\,(P_\theta)=\min\,(\{|u-v|\,|\,u,v\in P_\theta \text{ and } u\neq v\}),$$

* Received by the editors July 3, 1979, and in final revised form June 8, 1981.
† Computer Science Department, Purdue University, West Lafayette, Indiana 47907.
‡ The work of this author was supported by the National Science Foundation under grant MCS 7801812.

and the *length* of projection P_θ is

$$\text{len}(P_\theta) = \frac{\text{span}(P_\theta)}{\text{res}(P_\theta)}.$$

Intuitively, we think of S as a set of n points in 2-dimensional space projected onto a line at angle θ. The resolution of a projection gives the minimum distance between projected points along the line; the length gives the distance between endpoints after the resolution has been normalized to unity.

Problem 1. Given S, a finite subset of $R \times R$, find $\theta \in [0, \pi)$ which minimizes len (P_θ).

In the second problem we think of a finite set of points in 2-dimensional space projected onto a line. Using the minimum and maximum projected points to determine a line segment, mark off b equal size buckets $0, 1, \cdots, b-1$ such that bucket 0 starts with the minimum projected value and bucket $b-1$ ends with the maximum projected value. Some number of projected points lie within each bucket; this number is called the size of the bucket. The problem then is to find an angle of projection which minimizes the cost of the resulting distribution of bucket sizes according to a given cost function C. For example, a typical cost function might be the maximum number of objects in a bucket or the number of nonempty buckets. In any case, $T(b)$ denotes the complexity of computing the cost function given the b bucket sizes. Usually $T(b)$ is small.

DEFINITION 2. Let S be a finite subset of $R \times R$; let $\theta \in [0, \pi)$ be an angle of projection; and let $b \in Z^+$. Using min (P_θ) and max (P_θ) to denote the minimum and maximum elements in P_θ, the *scale* of P_θ with b buckets is

$$\text{scale}(P_\theta, b) = \frac{\max(P_\theta) - \min(P_\theta)}{b}.$$

The *size* of bucket i under projection P_θ scaled to b is size $(P_\theta, b, i) = |P_\theta^{b,i}|$, where

$$P_\theta^{b,i} = \{s \mid s \in P_\theta \text{ and } s \in [\min(P_\theta) + (i)\,\text{scale}(P_\theta, b), \min(P_\theta) + (i+1)\,\text{scale}(P_\theta, b))\} \quad \text{for } 0 \leqq i < b-1,$$

$$P_\theta^{b,b-1} = \{s \mid s \in P_\theta \text{ and } s \in [\min(P_\theta) + (b-1)\,\text{scale}(P_\theta, b), \min(P_\theta) + (b)\,\text{scale}(P_\theta, b)]\}.$$

Note that size $(P_\theta, b, i) > 0$ only if $0 \leqq i \leqq b$. All of the buckets are half-open intervals except the last, which is closed. The *distribution of* projection P_θ into b buckets is

$$\text{distr}(P_\theta, b) = (\text{size}(P_\theta, b, 0), \cdots, \text{size}(P_\theta, b, b-1)).$$

Let $C: Z^b \to Z^+$ be a cost function. Then the *cost* of a distribution is

$$\text{cost}(C, P_\theta, b) = C(\text{distr}(P_\theta, b)).$$

By convention, $T(b)$ will denote the time complexity of computing $C(\text{distr}(P_\theta, b))$, given that distr (P_θ, b) has been computed.

Problem 2. Given S, a finite subset of $R \times R$, $b \in Z^+$ and a cost function $C: Z^b \to Z^+$, find $\theta \in [0, \pi)$ which minimizes cost (C, P_θ, b).

3. Finding minimum length projections. This section presents an efficient algorithm for Problem 1. First, we give an overview of the algorithm and data structures. Then we discuss each piece in more detail. Finally, we show a simple lemma needed for correctness and conclude with a discussion of the algorithm's complexity.

Basically, our algorithm forms two lists, SPANLIST and RESLIST, corresponding to span (P_θ) and res (P_θ). Elements in these lists, ordered by increasing angle θ, consist of an angular interval $[\alpha, \beta)$[1] and a pair $(s_1, s_2) \in S \times S$, with the interpretation that for $\theta \in [\alpha, \beta)$ the projections of s_1 and s_2 determine span (P_θ) and res (P_θ), respectively. From SPANLIST and RESLIST, the algorithm forms a single list, LENLIST, which is, again, ordered by angle. LENLIST contains enough information to compute span (P_θ) and res (P_θ) for each angle $\theta \in [0, \pi)$, from which len (P_θ) can be determined.

Throughout the development, we note the time complexity of each step, explaining the analysis later and summarizing at the end.

ALGORITHM 1.
input: S a finite subset of $R \times R$
output: $\theta \in [0, \pi)$ and len (P_θ) such that len (P_θ) is minimum
method:
1. compute SPANLIST $O(n \log n)$
2. compute RESLIST $O(n^2 \log n)$
3. merge SPANLIST and RESLIST to form LENLIST $O(n^2)$
4. find an element of LENLIST for which len (P_θ) is minimum and output it $O(n^2)$

SPANLIST can be formed by first extracting a convex hull of the points in S and ordering the points of the hull counterclockwise, with respect to an interior point. Starting from an arbitrary point, search for another point of maximum separation to get two points which appear in SPANLIST. From these first two points, walk the hull counterclockwise to determine the angle over which each pair of points dominates. Finding the hull and ordering it requires $O(n \log n)$ time [GRAH72]; walking takes $O(n)$ time and produces a SPANLIST of $O(n)$ entries.

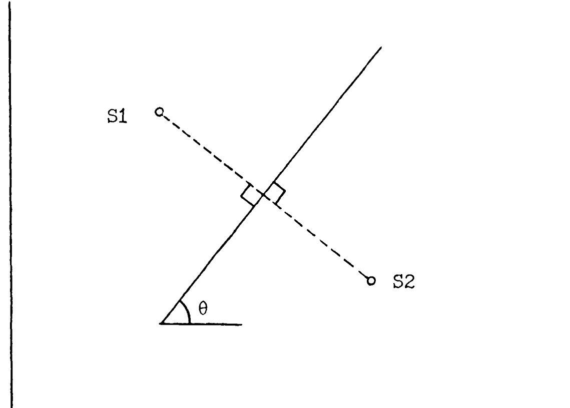


FIG. 1. *The angle θ such that the projections of s_1 and s_2 coincide. There is such an angle for each pair of points.*

Finding RESLIST is a bit trickier and requires some explanation. The key to understanding the algorithm lies in the following observation. Let L be the set of all possible unordered pairs of distinct points from S, and consider a particular pair $(s_1, s_2) \in L$. For some angle $\theta_z \in [0, \pi)$, s_1 and s_2 project to the same point as shown in Fig. 1. We call the angle for which s_1 and s_2 project to the same point the *zero angle* for the pair.

[1] The interval $[\alpha, \beta)$ is taken clockwise from α to β. Since len $(P_\theta) = $ len $(P_{\theta+\pi})$, we consider angles in $[0, \pi)$, instead of the usual $[0, 2\pi)$, and all angle arithmetic is performed mod π.

If we think of the distance between projections of two points s_1 and s_2 as the angle of projection increases from 0 to π, we see that it is a reflected sine wave of period π and amplitude equal to the distance from s_1 to s_2, as shown in Fig. 2. When another pair of points $(s_3, s_4) \in L$ with larger separation is added, the distance of their projection forms a reflected sine wave with period π, greater amplitude and possibly different phase. The distance between projections of s_3 and s_4 will be less than the distance between projections of s_1 and s_2 only around the zero angle of (s_3, s_4). This fact, expressed in Lemma 1, is the basis for computing RESLIST.

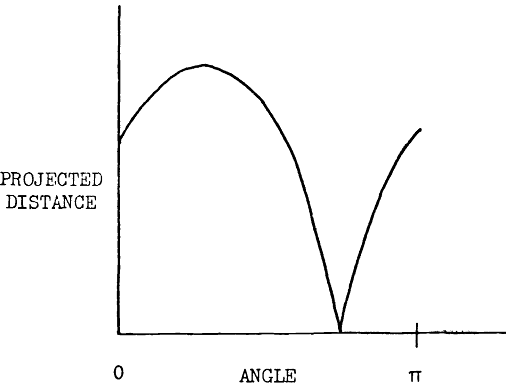


FIG. 2. *The distance between the projection of two points as a function of the angle of projection. The curve is a reflected sine wave of period π and amplitude $\|s_1 - s_2\|$.*

LEMMA 1. *Let (s_1, s_2) be an element of a finite set $S \subseteq (R \times R) \times (R \times R)$, $|S| \geqq 2$, $(s_1, s_2) \in S$ with $\|s_1 - s_2\|$ maximum, and let θ_z be the zero angle for (s_1, s_2). Then the angles θ for which the distance $\text{res}\,(P_\theta^{\{s_1, s_2\}})$ between projections of s_1 and s_2 is strictly minimum form an open interval (α, β) containing θ_z, with $\beta - \alpha \leqq \pi/2$.*

Proof. First, let (s_1, s_2) and (s_3, s_4) be two pairs of points with

$$d_{12} = \|s_1 - s_2\| \geqq \|s_3 - s_4\| = d_{34}.$$

Let θ_{12}, θ_{34} be the angles of the segments $\overline{s_1 s_2}$, $\overline{s_3 s_4}$, respectively. Then

$$\text{res}\,(P_\theta^{\{s_1, s_2\}}) = |d_{12} \cos(\theta - \theta_{12})|,$$

$$\text{res}\,(P_\theta^{\{s_3, s_4\}}) = |d_{34} \cos(\theta - \theta_{34})|.$$

Since the lengths of the projections of $\overline{s_1 s_2}$ and $\overline{s_3 s_4}$ depend only on the lengths and directions of these segments, not their positions, we can consider instead the projections of the corresponding vectors $\overline{s_1 s_2}$ and $\overline{s_3 s_4}$ positioned at the origin. The angles for which $\text{res}\,(P_\theta^{\{s_1, s_2\}}) = \text{res}\,(P_\theta^{\{s_3, s_4\}})$ are the angles θ_e, $\theta'_e \in [0, \pi)$, perpendicular to the vectors $\overline{s_1 s_2} - \overline{s_3 s_4}$ and $\overline{s_1 s_2} - \overline{s_4 s_3}$ (see Fig. 3).

Order θ_e, θ'_e so that $\text{res}\,(P_\theta^{\{s_1, s_2\}}) < \text{res}\,(P_\theta^{\{s_3, s_4\}})$ for $\theta \in (\theta_e, \theta'_e)$. Of course, $\theta_z \in (\theta_e, \theta'_e)$. $\theta'_e - \theta_e \leqq \pi/2$ may be proved geometrically from Fig. 3. Intuitively the longer segment $\overline{s_1 s_2}$ must produce the longer projection for at least one half of the angles in $[0, \pi)$.

Now the region over which $\text{res}\,(P_\theta^{\{s_1, s_2\}})$ is minimum is the intersection of all the intervals (θ_e, θ'_e) for all choices of (s_3, s_4). The intersection of open intervals containing θ_z of length $\leqq \pi/2$ must itself be such an interval. (The fact that the interval lengths are no greater than $\pi/2$ prevents two intervals from wrapping around the circle to intersect twice.)

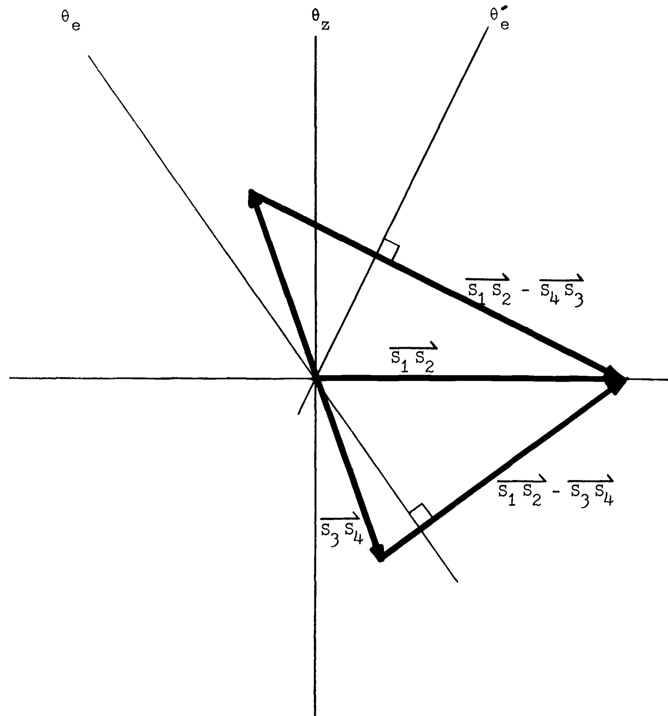


FIG. 3. *Angles θ_e and θ_e' where $\overline{s_1s_2}$ and $\overline{s_3s_4}$ project to equal lengths. θ_z is the zero angle for (s_1, s_2).*

From Lemma 1, we can form a procedure for computing RESLIST. Order the set L of all pairs of points in S by increasing distance between the points in a pair. To initialize, select a minimum distance pair, making its zero angle the origin for measuring angles. Place the pair on RESLIST with interval $[0, \pi)$.

Then insert each pair (s_1, s_2) from L into RESLIST by locating the interval which includes the zero angle of (s_1, s_2) and updating RESLIST. The updating of RESLIST involves creating a new entry to represent the interval $[\alpha, \beta)$ in which s_1 and s_2 determine the resolution, possibly splitting an existing interval containing $[\alpha, \beta)$ as a subinterval into two pieces, possibly reducing existing intervals which overlap $[\alpha, \beta)$ on either end and possibly deleting existing intervals which lie entirely within $[\alpha, \beta)$. All of these operations are handled together by starting with the existing interval containing the zero angle of (s_1, s_2), checking to the left until an existing interval is found which contains α, checking to the right to locate β, deleting all intervals which fall entirely between α and β and shortening those which overlap. The interval $[\alpha', \beta')$, with associated points s_1' and s_2' containing α, may be identified by the fact that (s_1s_2) has a shorter projection than (s_1', s_2') at angle β', but a longer projection at angle α'. Then the exact value of α is determined analytically as the angle for which the two projections have the same length. β is located in a similar fashion.

Observe that each new pair of points increases the length of RESLIST by at most two, in the case that the added interval splits an existing interval. In other cases, the length of RESLIST may increase by one, remain the same or even decrease because of deletions. Thus, RESLIST contains at most $O(n^2)$ entries corresponding to $O(n^2)$ pairs in L. By keeping the entries in the leaves of a balanced tree, such as a 2-3 tree, and linking the leaves in a list, we can find an interval including a zero angle in $O(\log n)$ time. Then, moving right and left in the list, we can determine how many

existing entries to delete. Note that while deletion costs $O(\log n)$, each entry will be added and deleted at most once, so we charge it both the cost of its insertion and deletion. Therefore, the running time is $O(\log(n^2)) = O(\log n)$ per entry.

procedure compute RESLIST

1. Generate L, a list of all unordered pairs of distinct points in S ordered by increasing distance of separation between points in the pair — $O(n^2 \log n)$
2. **for** each pair $(s_1, s_2) \in L$ **do** — $O(n^2)$ iterations
3. find interval in RESLIST containing the zero angle of (s_1, s_2) using a balanced tree — $O(\log n)$
4. update RESLIST possibly removing old intervals that are subsumed and updating the balanced tree — (see note in text)

Merging SPANLIST and RESLIST into LENLIST is straightforward and requires at most $O(n^2)$ time since there are at most $O(n^2)$ entries in RESLIST and $O(n)$ entries in SPANLIST. Once LENLIST is known, the minimum value for len (P_θ) may be found in $O(n^2)$ time by checking only the endpoints of intervals in LENLIST, as shown below. The following lemma shows that len (P_θ) must be minimized at the endpoint of some interval in LENLIST.

LEMMA 2. *Within each interval of* LENLIST, len (P_θ) *achieves its minimum at one of the endpoints of the interval.*

Proof. Notice that, within each interval $[\alpha, \beta)$ of LENLIST, the span and resolution of all projections is determined by the same two pairs of points. Let the pair of points (s_1, s_2) determining the span be joined by a line segment of length d_1 at angle α_1, and let the pair (t_1, t_2) determining the resolution be joined by a segment of length d_2 at angle α_2. Then within the given interval,

$$\text{len}(P_\theta) = \text{span}(P_\theta)/\text{res}(P_\theta) = |d_1 \cos(\alpha_1 - \theta)/d_2 \cos(\alpha_2 - \theta)|.$$

The minimum value for len (P_θ) will be found at either an endpoint of the interval (α or β) or at a zero of the derivative or at an angle with no derivative. len (P_θ) is nondifferentiable when span $(P_\theta) = 0$ (because of the reflection by the absolute value operation) and when res $(P_\theta) = 0$ (because of the division by 0); span $(P_\theta) = 0$ only at the zero angle of (s_1, s_2). This zero angle may not be in the interval $[\alpha, \beta)$ because when s_1 and s_2 project to the same point they do not determine the span. When res $(P_\theta) = 0$, len (P_θ) is infinite, so the minimum cannot occur there.

The derivative of len (P_θ) with respect to θ is

$$\pm \frac{\cos(\alpha_1 - \theta)\sin(\alpha_2 - \theta) - \cos(\alpha_2 - \theta)\sin(\alpha_1 - \theta)}{\cos(\alpha_2 - \theta)^2}.$$

This derivative is zero if and only if $\tan(\alpha_1 - \theta) = \tan(\alpha_2 - \theta)$. Two angles have the same tangent if and only if their difference is a multiple of π. So we either have no zeros of the derivative (when $\alpha_1 \neq \alpha_2 \bmod \pi$) or the derivative is everywhere zero (when $\alpha_1 = \alpha_2 \bmod \pi$), and len (P_θ) is a constant. In either case, the minimum value of len (P_θ) is found at an endpoint of one of the intervals in LENLIST.

The analysis above, with Lemmas 1 and 2, allows us to conclude the correctness and time complexity of Algorithm 1.

THEOREM 1. *Algorithm* 1 *solves Problem* 1 *in* $O(n^2 \log n)$ *time.*

Proof. Immediate from the discussion above.

4. Finding minimum cost distribution into buckets. This section presents an efficient algorithm for finding an angle of projection which minimizes the cost of a

distribution. It relies heavily on the reader's knowledge and intuition from the previous section, concentrating on differences between the two algorithms. As before, we present an overview of the solution first, followed by a more detailed discussion of each piece. Also as before, we note the complexity of each section as we present it, justifying the claims later.

Our algorithm for finding a minimum cost distribution begins, like Algorithm 1, by computing SPANLIST. Recall that SPANLIST, ordered by angle, contains angular intervals $[\alpha, \beta)$ and pairs of points $(s_1, s_2) \in S$ that define $\min(P_\theta)$ and $\max(P_\theta)$ for $\theta \in [\alpha, \beta)$. In terms of the distribution, s_1 is the start of bucket 0 and s_2 is the end of bucket $b-1$ for $\theta \in [\alpha, \beta)$. Assuming that the special case of 3 or more collinear points has been taken care of, the algorithm proceeds as follows:

ALGORITHM 2. $O(n^2 b \log(nb) + n^2 bT(b))$

input: S, a finite subset of $R \times R$, an integer $b > 0$ and a cost function $C: Z^b \to Z^+$

output: an angle of projection $\theta \in [0, \pi)$ and cost (C, P_θ, b) such that cost (C, P_θ, b) is minimum

method:

1.	form SPANLIST	$O(n \log n)$
2.	mincost $\leftarrow \infty$	
3.	**for** each $([\alpha, \beta), (s_1, s_2)) \in$ SPANLIST **do**	$O(n)$ iterations
4.	mincost $\leftarrow$ min (mincost, cost (C, P_α, b))	$O(T(b)+n)$
5.	find all distributions in $[\alpha, \beta)$	$O(nb \log(nb))$
6.	**for** each distribution in $[\alpha, \beta)$ **do**	$O(nb)$ iterations
7.	mincost $\leftarrow$ min (mincost, cost (C, P_θ, b))	$O(T(b))$
8.	output mincost and angle giving that cost	$O(1)$

Steps 5–7 each require further explanation; we begin with step 5.

The key to finding all distributions in an interval $[\alpha, \beta)$ lies in thinking of a line of projection with b buckets marked off rotating from α to β as shown in Fig. 4. One can easily construct such a line at angle α by projecting all points and marking off b equally spaced intervals between the smallest and largest.

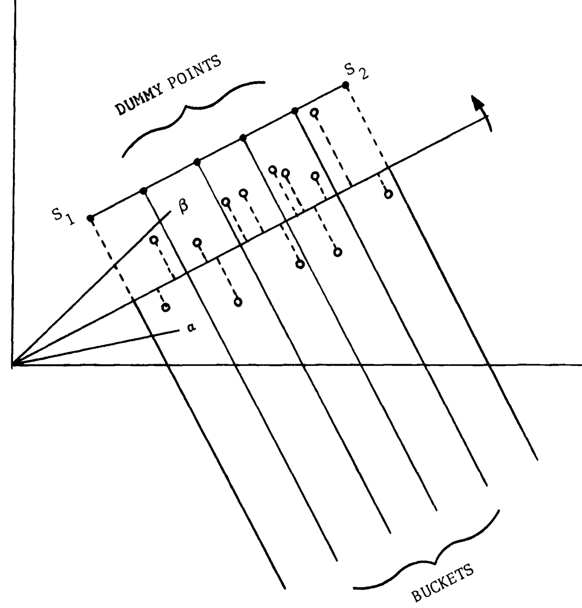


FIG. 4. *A line l rotating from angle α to β with b buckets marked off.*

Think of the b interval marks as the projection of a set of $b-1$ equally spaced dummy points along a line from s_1 to s_2 (s_1 and s_2 mark the left end of the first and the right end of the last bucket). As the imaginary line of projection rotates from α to β, the distribution changes whenever the projection of an element from S crosses a projection of one of the $b-1$ dummy points (i.e., a bucket mark). Thus, if D denotes the set of $b-1$ dummy points, a crossing corresponds to a zero angle of a pair from $D \times S$. To be more precise, our definition of buckets as half-oened intervals means that crossings may either occur exactly at the zero angle or just beyond it, depending on the direction of the crossing. The algorithm simply forms a list, CROSSLIST, consisting of bn triples (θ, d, s), where θ is the zero angle for $(d, s) \in D \times S$, marking those triples that correspond to exact crossings and those that correspond to crossings just beyond the zero angle. Of course, only those triples with $\theta \in [\alpha, \beta)$ are saved. Sorting CROSSLIST by θ value requires $O(nb \log (nb))$ time and produces a list of all angles in $[\alpha, \beta)$ where the distribution changes, as well as a record of which point changes buckets at that angle.

We summarize the procedure for step 5:

procedure step 5: find all distributions in $[\alpha, \beta)$

1. Form D, a set of $b-1$ equally spaced dummy points on the line segment $\overline{s_1 s_2}$ where s_1 and s_2 determine $\min(P_\theta)$ and $\max(P_\theta)$ for $\theta \in [\alpha, \beta)$ — $O(b)$
2. Find size (P_α, b, i) for $0 \leqq i < b$ — $O(n)$
3. Form CROSSLIST by finding all zero angles $\theta_z \in [\alpha, \beta)$ for pairs in $D \times S$ and sorting — $O(nb \log (nb))$

Given CROSSLIST, steps 6–7 become simple: remove the next element or elements (θ, d, s) with exact crossing at angle θ from CROSSLIST, update the appropriate bucket counts, compute the cost of the new distribution and record it in case the new cost is lower than the minimum found so far. Do the same for crossings that occur just beyond angle θ. The $O(nb)$ elements on CROSSLIST each require $O(T(b))$ time to process yielding a bound of $O(nbT(b))$. The loop in step 3 iterates steps 5–7 for each of the $O(n)$ items in SPANLIST, however, so the total processing in steps 5–7 requires $O(n^2bT(b))$. Similarly, step 5 requires a total of $O(n^2b \log (nb))$ because it is repeated $O(n)$ times. Thus, the total running time of Algorithm 2 is $O(n^2b \log (nb) + n^2bT(b))$.

THEOREM 2. *Algorithm 2 solves Problem 2 in* $O(n^2b \log (nb) + n^2bT(b))$ *time.*

Proof. Immediate from the discussion above.

In many cases, a cost function may be updated after a point crosses from one bucket to the next much more quickly than $O(T(b))$, the time to recompute the cost from scratch. For example, if the cost of a distribution is the sum of some fast (i.e., $O(1)$) function of the bucket sizes, then $T(b) = O(b)$. But, we may update the cost in time $O(1)$ by merely adding in the changes in function values of the two bucket sizes which have changed. In general, if $U(b)$ is the time required to update the cost function after one bucket crossing, Algorithm 2 may easily be improved to run in time $O(n^2b \log (nb) + n^2bU(b) + T(b))$.

5. Applications to hashing. This section describes how Algorithms 1 and 2 apply to the hashing problem mentioned in the Introduction. The first problem, concerned with finding a minimum table size perfect hashing function, can be defined as:

Problem H1. Given a set S of n keys and two functions h_1 and h_2 which map S to Z^+, find constants $c_1, c_2, c_3 \in R$ such that for $h(x) = |c_1h_1(x) + c_2h_2(x) + c_3|$ the following hold:

(1) $k_1, k_2 \in S$, $h(k_1) = h(k_2)$ if and only if $k_1 = k_2$.
(2) $\min(\{h(k) \mid k \in S\}) = 0$.
(3) $\max(\{h(k) \mid k \in S\})$ is minimized.

Property 1 guarantees that h is a perfect hashing, while Properties 2 and 3 assure a minimum table size.

Problem H1 is similar to the geometric problem of finding an angle of projection such that the length of projection is minimized when each pair of projected points is separated by an integer. Note, however, that Algorithm 1 does not always produce such a minimum projection. Instead it minimizes the length of projection while simultaneously placing the closest pair of projected points distance 1 apart.

To see the difference between the unit distance stipulation imposed by Algorithm 1 and the distinct cell stipulation given in the problem statement, consider three collinear points as shown in Fig. 5. Trouble arises when an integer separates two

FIG. 5. *A case where Algorithm 1 can produce a projection that is arbitrarily far from the optimum hash table size. The points p_1, p_2 and p_3 are collinear.*

projected points p_1 and p_2. Using real arithmetic, one could squeeze p_1 and p_2 arbitrarily close together, forcing p_3 to move close to p_2. Clearly the optimum solution requires only 3 cells in a hash table. Algorithm 1, which places p_1 and p_2 unit distance apart, requires more than 3 cells. Even using floating point hardware to approximate the real number solution may still lead to anomalies, like the one in Fig. 5, if the floating point approximations for p_1 and p_2 happen to lie on either side of an integer.

We can define the second hashing problem as:

Problem H2. Given S, a set of n keys, $b \in Z^+$, h_1 and h_2 which map S to Z^+ and a cost function $C: Z^b \to Z^+$, find constants c_1, c_2, $c_3 \in R$, such that for $h(x) = |c_1 h_1(x) + c_2 h_2(x) + c_3|$ the following hold:

(1) $\min(\{h(k) \mid k \in S\}) = 0$.
(2) $\max(\{h(k) \mid k \in S\}) = m - 1$.
(3) The cost of h as given by C is minimum.

Algorithm 2 solves Problem H2 exactly.

In particular, the problem formulated in [COME79] requires finding a projection that distributes keys as uniformly as possible into buckets. In the particular problem described, the cost of searching a bucket with t entries is $O(\log(t))$. Assuming that the complexity of computing $\log i$ for small integer i is constant, so $T(b) = b$; this yields a complexity of $O(n^2 b \log(nb)) + n^2 b^2)$ for finding an optimum projection. We can do slightly better by noticing that when only two bucket sizes change, the cost may be updated in constant time ($U(b) = O(1)$); so Algorithm 2 may easily be modified to find the optimum projection in time $O(n^2 b \log(nb) + n^2 b)$.

6. Conclusions and further research. In order to be used in graphics applications, the algorithms of this paper must be adapted to find optimal projections from 3 dimensions to 2 dimensions, instead of 2 to 1. A major component of the adapted algorithm, a solution to the 3-dimensional convex hull problem, is already known [PREP77]. The major remaining difficulty is to find a higher-dimensional analogue to the search tree representation of RESLIST.

Given two hashing functions h_1 and h_2 and a fixed set of keys, the algorithm of this paper may be used to choose a good hashing function of the form $h(x)=|c_1h_1(x)+c_2h_2(x)+c_3|$. The question of how good is the best hashing function in such a class is open. Very little is known about hashing with predetermined, static sets of keys; the only treatment of this problem seems to be Sprugnoli's [SPRU77]. A combinatorial analysis should at least settle the question of how large a class of hash functions is needed to guarantee a given level of performance for any set of keys.

Acknowledgments. The two anonymous referees were especially helpful in improving the style and the content of this paper. In particular, they corrected our errors in Lemma 1 and suggested the improvement to Algorithm 1 using fast updates to the cost function.

REFERENCES

[COME79] D. COMER AND V. Y. SHEN, *Hash-binary search. A fast technique for searching an English spelling dictionary*, Tech. Rep. TR-CSD-304, Dept. of Computer Science, Purdue University, West Lafayette, IN, 1979.

[GRAH72] R. L. GRAHAM, *An efficient algorithm for determining the convex hull of a finite planar set*, Inform. Proc. Letters, 1 (1972), pp. 132–133.

[JARV73] R. JARVIS, *On the identification of the convex hull of a finite set of points in the plane*, Inform Proc. Letters, 2 (1973), pp. 18–21.

[PREP77] F. PREPARATA AND S. HONG, *Convex hulls of finite sets of points in two and three dimensions*, Comm. ACM, 20 (1977), pp. 87–93.

[SHAM75] M. SHAMOS, *Geometric complexity*, Proc. 7th Annual ACM Symposium on Theory of Computing, 1975, pp. 224–233.

[SHAM75a] M. SHAMOS AND D. HOEY, *Closest point problems*, Proc. 16th Annual IEEE Symposium on Foundations of Computer Science, IEEE, New York, 1973, pp. 151–162.

[SPRU77] R. SPRUGNOLI, *Perfect hashing functions: A single probe retrieving method for static sets*, Comm. ACM, 20 (1977), pp. 841–850.

SIAM J. COMPUT.
Vol. 11, No. 2, May 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1102-0003 $01.00/0

GRAPHS THAT ARE ALMOST BINARY TREES*

JIA-WEI HONG† AND ARNOLD L. ROSENBERG‡

Abstract. This paper studies embeddings of graphs in binary trees. The *cost* of such an embedding is the maximum distance in the binary tree between images of adjacent graph vertices. Several techniques for bounding the costs of such embeddings from above are derived; notable among these is an algorithm for embedding any outerplanar graph in a binary tree with a cost that is within a factor of 3 of optimal. A number of techniques for bounding the costs of such embeddings from below are developed; notable here are two techniques for inferring the presence of large separators in graphs. Finally, a number of characterizations are established of those families of graphs that are *almost* binary trees, in the sense that every graph in the family is embeddable in a binary tree within bounded cost.

Key words. graph embeddings, binary trees, graph separators, proximity preservation, similarity of graphs, reticulated graphs, stratified graphs

1. Introduction.

Motivation. A large variety of computational problems can be formulated mathematically as graph embedding problems. Included here are problems concerning representing data structures in computer storage [1], [5]–[11], organizing computations on networks of processors [4], studying the relative efficiencies of program control structures [5], and laying out circuits in standard formats [12]. In all of these problem areas, one has a source graph that represents well the structure of the computation one wants to perform, and a target graph that is better suited to implementation or manipulation or analysis in the computer environment at hand. One's task is to embed the source graph in the target graph in a way that preserves adjacencies as well as possible. In all of the cited problem areas, one often seeks a target graph that is tree-like in structure, since trees are so easily implemented in both hardware and software, and because trees are relatively easy to analyze and to manipulate (in software). But the desire for efficient, adjacency-preserving embeddings often makes true trees (apparently) unsuitable target graphs, so one seeks target graphs that are augmented trees, i.e., trees with auxiliary edges that augment the set of adjacencies. Consider three examples aimed at simplifying tree traversals (cf. [3, § 2.3.2]): (1) if one anticipates repeated *preorder* tree traversals, one might add auxiliary edges connecting each leaf node $x12^n$ ($x \in \{1, 2\}^*$, $n \geqq 0$) with node $x2$, as in Fig. 1 (thereby making preorder traversal a Hamiltonian path); (2) if one anticipates repeated *inorder* tree traversals, one might add auxiliary edges connecting each pair of adjacent leaves (in the natural left-to-right ordering) to their least common ancestor, as in Fig. 2; (3) if one anticipates repeated *breadth-first* tree traversals, one might add auxiliary edges going across each level of the tree, as in Fig. 3. A basic issue raised by this scenario is: Which embellishments of trees make a substantive improvement in adjacency-preservation and which make only inconsequential improvements? This paper is devoted to studying this issue via the two basic questions: How far is a graph G from being a tree? How "well" can a graph G be embedded in a tree, where we measure

* Received by the editors July 14, 1980 and in revised form May 21, 1981. A portion of this paper was presented at the 13th ACM Symposium on Theory of Computing, Milwaukee, Wisconsin, May 11–13, 1981.

† Peking Municipal Computing Centre, Peking, China.

‡ Mathematical Sciences Department, IBM T. J. Watson Research Center, Yorktown Heights, New York 10598. Present address: Department of Computer Science, Duke University, Durham, North Carolina 27706.

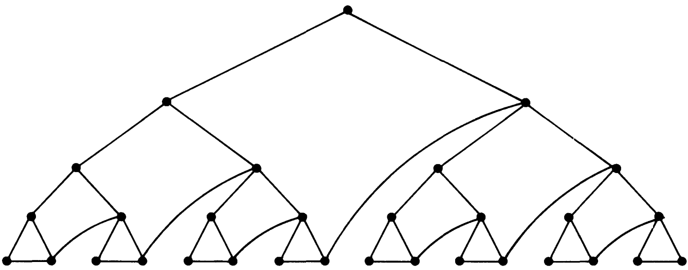


FIG. 1. *The depth-4 complete preorder tree.*

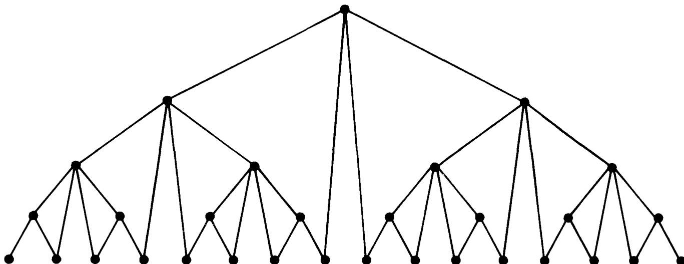


FIG. 2. *The depth-4 complete inorder tree.*

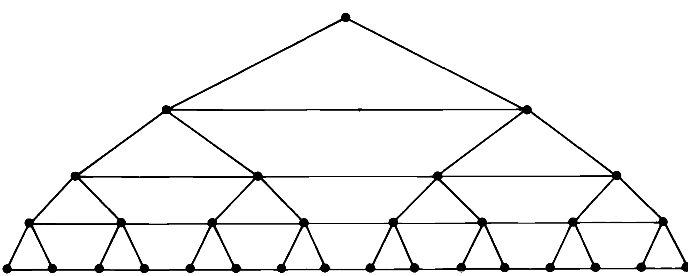


FIG. 3. *The depth-4 complete breadth-first tree.*

the quality of an embedding by the extent to which the images of adjacent vertices of G are close together in the tree? Our major results answer these questions for a variety of families of graphs. In particular we show that preorder and inorder trees are both equivalent to trees! Breadth-first trees are strictly more efficient than true trees, but by exponentially less than one might expect.

The formal setting. The vehicle for our investigation is a simple notion of a graph embedding and its cost. An *embedding* of a graph G in a graph H is a one-to-one association of the vertices of G with vertices of H. The *cost* of the embedding is the largest distance in H between images of adjacent vertices of G. We say that one proposed target graph H' *can simulate* another one H *to within the factor* $c>0$ if the cost of embedding any graph G in H' is at most c times the cost of embedding G in H. This condition is easily shown to be equivalent to the ability to embed H in H' within cost c [8]. We extend the notion of simulation to families of graphs in the obvious way: the family $\mathcal{H}'$ *can simulate* the family $\mathcal{H}$ (equivalently, $\mathcal{H}$ *is almost* $\mathcal{H}'$), if there is a constant $c>0$ such that each $H \in \mathcal{H}$ can be simulated by some $H' \in \mathcal{H}'$

to within the factor c. The families $\mathscr{H}$ and $\mathscr{H}'$ are *similar* if each can simulate the other (this is our notion of equivalence). One verifies easily that our notions behave well: costs of composed embeddings multiply; "can simulate" and "is almost" are transitive; and "is similar" is an equivalence relation [8]. (None of these facts need be true when some average measure of the cost of an embedding is used.) The relations we have just defined are very basic ones: just as algebra studies mappings that preserve structure, and topology studies mappings that preserve neighborhoods, a prime theme in graph theory should be mappings that preserve "closeness." In fact, when the simulation constant $c = 1$, our notion of similarity is just graph isomorphism.

Synopsis. Our main results describe (1) techniques for bounding from above the factor to within which binary trees can simulate other families of graphs (these techniques yield bounds that are within a small factor of optimal); (2) techniques for bounding from below the factor to within which a large variety of families of graphs can be simulated by binary trees; (3) a number of necessary-and-sufficient conditions for a family of graphs to be almost a family of binary trees. One of our main results, for example, can be stated:

> *The bounded-degree family of graphs $\mathscr{G}$ is almost a family of binary trees if and only if $\mathscr{G}$ is almost outerplanar.*

Applications. Although our main goal here is to understand better what makes a graph tree-like, our positive results do have application to certain of the motivating problems. For example, with regard to circuit layout, the embedding techniques of Valiant [12] require one to find a small separator for the graph to be laid out, then for the induced subgraphs, and so on. If one's graph is irregular in structure, finding these separators may be difficult. If, however, one's graph can be simulated by a binary tree (e.g., if the graph is outerplanar), then one can use our results to embed the graph in a tree and then use Valiant's techniques to lay the tree (which now has fat edges) out in a grid. Similar considerations apply to the problem of laying out large VLSI systems over several chips which have to be interconnected.

Related work. The already cited sources and work cited in them trace the history of the use of graph embeddings to model computational situations. The most relevant precursors of this work are the studies in [1], [5] [6], [8]–[11] of embeddings of graphs in trees. Among these, only [5], [10] study the present cost measure; all others study some notion of the average cost of an embedding. Only [6], [10] study general criteria for bounding costs; all others study embeddings of specific families of graphs in trees. [10] concentrates on general criteria for bounding from below the costs of embeddings of arbitrary G in arbitrary H; hence the results there cannot be as sharp as ours. Section 6 of [6] characterizes families of graphs that can be embedded efficiently in trees relative to an average-edge notion of adjacency preservation (with all edges weighted equally). Indeed, certain of our results can be viewed as doing for the worst-case cost of embeddings of graphs in trees what [6, § 6] does for the average-case cost. It appears that the worst-case cost is more difficult to treat definitively; and we feel that it is also the more basic notion in terms of the motivating computational problems.

2. Upper bounds on simulation costs. We derive in this section a variety of techniques for efficiently embedding a graph G in a binary tree, and we illustrate these techniques on sample graphs. Results in § 3 will show that these techniques yield embeddings that are close to optimal in cost. We stress that we are not supplying an efficient procedure that will announce, given an arbitrary graph G, whether or not

G is efficiently embeddable in a binary tree. It remains an open problem whether or not such an efficient decision procedure exists.

Several results in this section can be stated most succinctly with the aid of the following formalism. A *graph* G comprises a set of vertices and a set of two-element sets of vertices called *edges*. (Note that we prohibit self-loops here.) We denote by $|G|$ the size of the set Vertices(G). An *n-ary tree* is a graph whose vertices are a *prefix-closed* set of strings over the alphabet $[n] =_{\text{def}} \{1, 2, \cdots, n\}$ (so that string x is in the set whenever any extension $x\sigma$ of x is), and whose edges connect all vertices x and $x\sigma$. A 2-ary tree is termed *binary*.

A. *Techniques based on planarity considerations.* Our first family of embedding strategies is applicable in a wide variety of situations.

The graph G is *outerplanar* if it has a planar embedding that places all vertices in one face. If a graph is outerplanar, then (and only then) it can be embedded in a convex polygon in the plane so that the vertices of the graph are the vertices of the polygon, and the edges of the graph are either edges or noncrossing chords of the polygon.

THEOREM 2.1. (One-way outerplanar theorem) *If G is an outerplanar graph, then there is an embedding ε of G in a binary tree with* $\text{Cost}(\varepsilon) \leqq 3 \cdot \lceil \log_2 (2 \text{ maxdegree}(G)) \rceil$.

Proof. We shall show how to embed any outerplanar graph G in a $2d$-ary tree, where $d = \text{maxdegree}(G)$, within Cost 3. The Theorem will then follow from the obvious embedding of a $2d$-ary tree in a binary tree.

We assume that the graph G is presented to us embedded in its polygon; and we view the vertices of the polygon (hence the vertices of G) as oriented in a clockwise fashion. (This is the only nonconstructive part of the proof.) We begin our embedding process by labelling G's vertices with distinct strings over the alphabet $[2d]$ in a way that in fact embeds G in a $2d$-ary tree. We proceed as follows (see Fig. 4):

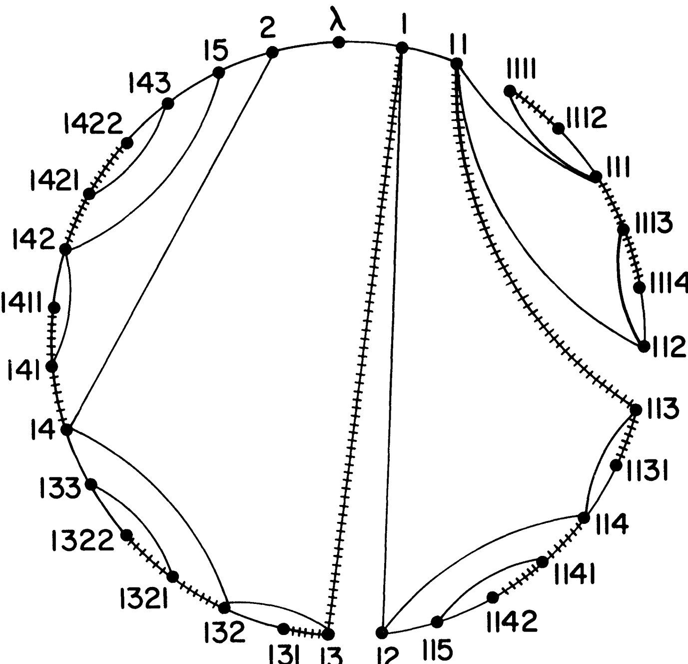


FIG. 4. *The depth-4 complete preorder tree labeled by the procedure of Theorem 2.1. Auxiliary (= nontree) edges are cross-hatched; the root of the preorder tree is labeled λ.*

We select some vertex of G to be the patriarch of the graph, and we label this vertex with the unique length-0 string λ. The neighbors of the patriarch will be his sons and will be labelled with the length-1 strings $1, 2, 3, \cdots$ in a clockwise direction; the unlabelled neighbors of each son will be the patriarch's grandsons and will be labelled with length-2 strings; and so on. A vertex's seniority in the graph G is determined by its proximity to the patriarch in G; a vertex's seniority among its brothers is determined by its proximity to the patriarch in the clockwise direction around the polygon. Before specifying the sought labelling, we must transfer some sons to new fathers. The rules of this transfer are specified by the Chinese Adoption Algorithm:

In ancient China there was a custom that a younger brother's sons would be adopted by an elder brother. The Chinese Adoption Algorithm codifies this custom by demanding that, in our graph,

> *a son that precedes its father in clockwise order should be adopted by its father's next more senior brother.*

Once the prescribed adoptions have taken place, the remaining sons of the vertex labelled x are labelled $x1, x2, x3, \cdots$ in clockwise order. Formally the algorithm is as follows.

Step 0. Label one vertex of G with the empty (null) string λ.

Step 1. Label the neighbors of vertex λ with 1, 2, 3, etc., *in clockwise order.*

Step $i+1$. Label all unlabelled neighbors of length-i vertices (=vertices having length-i labels) with length-$(i+1)$ strings in such a way that: (a) the length-$(i+1)$ strings are in lexicographic order when read clockwise; (b) the following rule is maintained (where x denotes a string and σ, α denote symbols in $[2d]$):

> a neighbor u of vertex $v = x\sigma$ has a label of the form $x(\sigma-1)\alpha$ or of the form $x\sigma\alpha$, the former obtaining if and only if *both* u precedes v in clockwise order *and* $\sigma > 1$.

(c) if k vertices get labels of the form $x\sigma$, then their labels are $x1, x2, x3, \cdots, xk$.

One verifies easily that the labelling produced by this algorithm is, in fact, an embedding of G in a $2d$-ary tree. We show now that this embedding has Cost at most 3.

Let v and w be vertices of G such that (i) there is an edge between v and w; (ii) v precedes w in clockwise order. Clearly v and w differ in seniority by at most 1, since we are looking at a planar embedding of G. Three possibilities arise.

(1) v is w's father. Since w succeeds its father in clockwise order, v remains w's father (i.e., no adoption takes place); hence v and w are distance 1 apart in the tree.

(2) w is v's father. Since w precedes its father in clockwise order, w is adopted by v's elder brother should one exist; hence v and w are distance 1 or distance 3 apart in the tree.

(3) v and w have the same seniority in G. Then we find their least common ancestor u. In the course of finding u, we find a polygon with vertices (in clockwise order)

$$u - v_k - v_{k-1} - \cdots - v_1 - v - w - w_1 - \cdots - w_{k-1} - w_k$$

having an odd number of edges. Now, v_k and w_k are brothers; after one adoption, v_{k-1} and w_{k-1} become brothers; and eventually, after a chain of adoptions, v and w become brothers. When they are labelled (i.e., embedded in the tree), therefore, v and w are distance 2 apart in the tree.

In any case, v and w are placed at distance at most 3 apart in the tree.

We now compose this Cost-3 embedding of G in a $2d$-ary tree with the natural Cost-$\lceil \log_2 2d \rceil$ embedding of the $2d$-ary tree in a binary tree. Since the Costs of composed embeddings multiply, this completes the proof. □

Example 2.1. *Preorder trees.* Preorder trees are outerplanar graphs; hence, we can use Theorem 2.1 directly to embed any preorder tree in a binary tree with Cost $\leqq 9$. (This is the embedding indicated in Fig. 4.)

Example 2.2. *Inorder trees.* Inorder trees are outerplanar graphs also; we can again use Theorem 2.1 directly to embed any inorder tree in a binary tree with Cost $\leqq 6$.

The constant in Example 2.2 is somewhat smaller than the conservative estimate of Theorem 2.1 would suggest. We feel it was not clear a priori that binary trees could simulate these embellished trees to within *any* factor independent of the depth of the tree.

We look now at a number of consequences of Theorem 2.1. We begin with an easy one.

COROLLARY 1. *If the graph G is embeddable in the outerplanar graph H with* Cost $\leqq c$, *then G is embeddable in a binary tree with* Cost $\leqq 3c \cdot \lceil \log_2 (2 \text{ maxdegree}(H)) \rceil$.

Proof. Immediate from Theorem 2.1 since the Costs of composed embeddings multiply. □

The next result is oriented more directly to the question that motivated this study initially: How much do auxiliary edges added to trees help? We answer the question by considering how efficiently the embellished trees can be simulated by true trees.

Any maxdegree-n graph G can be viewed as an n-ary tree (one of its spanning trees) augmented with nontree "auxiliary" edges. From this vantage point, G's vertex-set is a (prefix-closed) subset of $[n]^*$. When G has a binary spanning tree, we can build on this observation to find an often efficient embedding of G in a binary tree.

Say that the graph G has *deviation* (*from binary tree-hood*) (c, q), where $c, q \geqq 0$ are integers, if G has a spanning tree T such that:

(a) T is binary, so each vertex is in the set $[2]^*$.

(b) If the vertices v and w of G are ends of an auxiliary edge of G, then the path in T from the least common ancestor of v and w to either v or w changes direction (from "1" to "2" or from "2" to "1") at most $c-1$ times. Symbolically, this is equivalent to the assertion that v is of the form $v = xy$ and w is of the form $w = xz$ where, letting $A = 1^* + 2^*$, we have $y \in A^d$ and $z \in A^e$ for some $d, e \leqq c$.

For $v \in xA^d$, we call the longest prefix of v that is in xA the *head of v starting at x*.

(c) For all vertices x of G, there are at most q vertices that are heads of some termini of auxiliary edges of G starting at x.

COROLLARY 2. *If the graph G has deviation (c, q), then G is embeddable in a binary tree with* Cost $\leqq 6c \cdot \lceil \log_2 2(q+2) \rceil$.

Proof. A graph G with deviation (c, q) can be embedded in a maxdegree-$(q+2)$ outerplanar graph with Cost $\leqq 2c$. The embedding can be viewed as turning G into an outerplanar graph by adding edges to G in a way that replaces each auxiliary edge by a path of length $\leqq 2c$. This outerplanar graph is then embedded in a binary tree via the algorithm of Theorem 2.1. We describe very informally the method of adding new edges to make the original graph outerplanar. Take the spanning tree of G and add to it the following edges. (In what follows, x is an arbitrary string in $[2]^*$.) (a) Each vertex of the form $x1$ gets new edges connecting it to all vertices of the form $x12^a$, $a > 1$; (b) each vertex of the form $x2$ gets new edges connecting it to all vertices

of the form $x21^a$, $a>1$; and (c) vertex λ gets new edges connecting it to all vertices of the form 2^a or 1^a, $a>1$. This shaggy tree is clearly outerplanar; and it contains, for each auxiliary edge of G, a path of length $\leqq 2c$ connecting the endpoints of that edge (a consequence of the direction-changing properties of G's auxiliary edges). If one now prunes this shaggy tree, removing all of its auxiliary edges that do not enter into any of these length-($\leqq 2c$) paths, one is left with an outerplanar graph of maxdegree $\leqq q+2$ (a consequence of the "head" properties of G's auxiliary edges). Moreover, we have pruned this shaggy tree in such a way as to retain G's embeddability within Cost $2c$. It is thus the desired outerplanar graph. The details of our construction should become clearer from the following detailed example. □

Example 2.3. *Leap trees.* The depth-d *leap tree* is obtained from the depth-d complete binary tree by adding auxiliary edges connecting each node of the form 2^a1^b—$a\in\{0\}\cup[d-1]$, $b\in[d]$, $a+b\leqq d$—to node $2^{a+1}1^{b-1}$; see Fig. 5. The least common ancestor of the auxiliary edge $(2^a1^b, 2^{a+1}1^{b-1})$ is node 2^a, so the deviation constant $c=2$; and every node in the set 1^* is the head of some auxiliary edge, so the deviation constant $q=d$. Therefore the graph has deviation $(2, d)$ and so, by Corollary 2, is embeddable in a binary tree with Cost $\leqq 12\cdot\lceil\log_2 2(d+2)\rceil$. (The constant here is at least a factor of 2 too large.) The embedding into a binary tree proceeds as in the proof of Corollary 2: we embed the leap tree in a structurally similar outerplanar graph, and then embed this outerplanar graph in a binary tree. The first embedding can be viewed as replacing each auxiliary edge $(2^a1^b, 2^{a+1}1^{b-1})$ with the pair of edges $(2^a1^b, 2^a)$ and $(2^{a+1}, 2^{a+1}1^{b-1})$, thereby converting the graph into a degree-$(d+2)$ outerplanar graph. Since the auxiliary edge termini 2^a1^b and $2^{a+1}1^{b-1}$ are distance 3 apart in the outerplanar graph (via the path 2^a1^b, 2^a, 2^{a+1}, $2^{a+1}1^{b-1}$), it follows that the Cost of the indicated embedding is 3, whence the result.

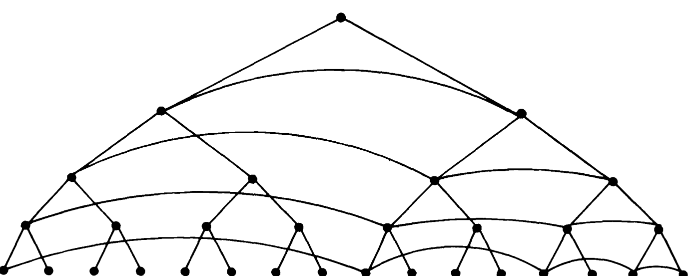


FIG. 5. *The depth-4 leap tree.*

Our final result in this family of techniques is the main result of this section. It employs yet another strategy to embed graphs in trees by first embedding them in outerplanar graphs and then employing the algorithm of Theorem 2.1.

Let G be an arbitrary graph. Embed G in the plane. Now, G may not be planar, so there may be crossovers among G's edges. In this contingency, replace G by the planar graph G^* obtained by adding a *pseudo-vertex* (*p-vertex*, for short) at each crossover point in the embedding: if in G's embedding the edges (v_1, v_2) and (v_3, v_4) crossed, G^* would have a p-vertex v and edges (v_1, v), (v, v_2), (v_3, v), (v, v_4); and edges (v_1, v_2) and (v_3, v_4) would be deleted. The *planarity number P* of this embedding of G is the largest number of p-vertices added along any single edge of G; and the *fanout number F* of the embedding is maxdegree(G^*). Although the graph G^* is

planar, it may not be outerplanar. We make it outerplanar as follows. Each interior (in the embedding) vertex of G^* must cross some number of edges of G^* in order to reach the exterior. We consider only *independent* "escape" routes for the interior vertices of G^*, that is, routes that never cross each other. Having chosen some independent set of escape routes, let E be the *escape number* of the set of routes, i.e., the maximum, over all edges e of G^*, of the number of interior vertices of G^* whose escape routes cross edge e. Dually, let D be the maximum, over all vertices v of G^*, of the number of edges that v crosses to reach the exterior. Say that vertex v (possibly a p-vertex) must cross edges $(v_1, v_2), (v_3, v_4), \cdots, (v_{2k-1}, v_{2k})$ in order to reach the exterior. We replace these k edges by the $2k$ edges $(v_{2i-1}, v), (v, v_{2i})$ for $1 \leqq i \leqq k$. Vertex v is now an exterior vertex of the altered graph. We repeat this process for every interior vertex of G^*, thereby obtaining an outerplanar graph G^{**}, which we embed in a binary tree via Theorem 2.1.

In essence, we have described here an embedding ε of G in the outerplanar graph G^{**}, via an embedding ε^* of G in the planar graph G^*. We note the following facts about these embeddings.

(a) The Cost of the embedding ε^* of G in G^* is $P+1$ (P = planarity number).

(b) The Cost of the embedding ε^{**} of G^* in G^{**} is $E+1$ (E = escape number).

(c) The graph G^{**} has maxdegree $\leqq F+2D$ (F = fanout number, D = dual escape number).

These properties of the graphs and embeddings we have been discussing yield our most general embedding result.

THEOREM 2.2. (one-way planar embedding theorem) *If the graph G is embeddable in the plane with planarity number P, fanout number F, escape number E, and dual escape number D, then there is an embedding ε of G in a binary tree with*

$$\text{Cost}(\varepsilon) \leqq 3(P+1)(E+1) \cdot \lceil \log_2 2(F+2D) \rceil.$$

Proof. We obtain ε as the composition of the embeddings $\varepsilon^*: G \to G^*$, $\varepsilon^{**}: G^* \to G^{**}$, and $\varepsilon': G^{**} \to$ Binary Tree, the last embedding coming from Theorem 2.1. The Cost of ε is obtained by multiplying the Costs of its constituent embeddings. □

We have chosen to derive Theorem 2.2 from Theorem 2.1 so as to separate into two parts the relatively complicated algorithms for the embeddings ε' and $\varepsilon^*\varepsilon^{**}$. (Also, Theorem 2.1 on its own has the two interesting Corollaries we have noted.) We could, of course, have opted to make Theorem 2.2 the focus of the section; in this case, Theorem 2.1 would have followed by noting that when G is outerplanar, it can be embedded in the plane with $P = E = D = 0$.

Example 2.4. *Breadth-first trees.* Let us consider a natural (hence planar) embedding of the depth-d breadth-first tree in the plane, such as is depicted in Fig. 3. For this embedding, the planarity number $P = 0$, and the fanout number $F = 5$. In order to determine the escape and dual escape numbers, imagine that the breadth-first tree is held vertically by the root so that all of its interior vertices fall to the ground; clearly, these gravity-induced escape routes are all independent. Each horizontal edge of the tree is crossed by just one escaping vertex, and no slanted edge is crossed at all; hence the escape number of the embedding is $E = 1$. Each interior vertex at level l of the tree (the root being at level 0) falls through $d-l$ horizontal edges in its escape; hence the dual escape number of the embedding is $D = d-2$. It follows from Theorem 2.2, then, that the depth-d breadth-first tree is embeddable in a binary tree with Cost $\leqq 6 \cdot \lceil \log_2 2(2d+3) \rceil$ (which is $O(\log \log n)$ where n is the size of the tree).

B. *Techniques based on graph splitting.* Our second family of techniques mirrors somewhat more closely one's notion of how trees are constructed.

A *split* σ of a graph G is a (vertex-) partition of G into two disjoint subgraphs, call them $G(L;\sigma)$ and $G(R;\sigma)$. Two splits σ_1 and σ_2 of G are *consistent* if one of the sets $G(L;\sigma_1)\cap G(L;\sigma_2)$, $G(L;\sigma_1)\cap G(R;\sigma_2)$, $G(R;\sigma_1)\cap G(L;\sigma_2)$, $G(R;\sigma_1)\cap G(R;\sigma_2)$, is empty. An edge e of G *belongs* to the split σ of G if e must be "cut" in order to effect the partition.

A *splitting* of the graph G is a set of mutually consistent splits of G. If the splitting comprises n splits, then it can be viewed as partitioning G into some number $\leqq 2^n$ of subgraphs. The splitting is *total* if each of these subgraphs has only one vertex. The *rank* of a splitting of G is the largest number of splits that any one edge of G belongs to. Each subgraph created by the splitting can be represented in a natural way as an intersection $\bigcap_i G(D_i;\sigma_i)$, $D_i\in\{L,R\}$, over some subset of the n splits. The *dimensionality* of the splitting is the smallest number d such that each created subgraph is the intersection of d or fewer $G(D;\sigma)$'s.

THEOREM 2.3. (one-way splitting theorem) *If the graph G admits a total splitting of rank r and dimensionality d, then there is an embedding ε of G in a binary tree with* Cost $(\varepsilon)\leqq r\log d$.

Proof. Any set Σ of splits of G gives rise in a natural way to a graph $\Sigma(G)$ whose vertex-set is the set of pieces of G created by the splits, and whose adjacency structure is given by the "distance" function

$$\delta(P,Q)=\text{the number of distinct splits that separate the pieces } P \text{ and } Q \text{ of } G;$$

by convention, $\delta(P,P)=0$. One verifies easily the following properties of this function:

(1) $\delta(P,Q)=0$ iff $P=Q$,

(2) $\delta(P,Q)=\delta(Q,P)$,

(3) $\delta(P,R)\leqq\delta(P,Q)+\delta(Q,R)$,

so δ is a valid notion of distance. The graph $\Sigma(G)$ connects two parts of G [=vertices of $\Sigma(G)$] just when $\delta(P,Q)=1$. There is a natural (possibly many-one) embedding of G in $\Sigma(G)$, with Cost $=r=$ the rank of the set of splits. To see this last point, note that if the edge $e=(v,v')$ of G belongs to r splits, then $\delta(P,Q)\geqq r$, where $v\in P$ and $v'\in Q$; hence, the edge e is replaced under the embedding by a path of length $\geqq r$ in $\Sigma(G)$.

When the set of splits Σ is a splitting of G, then the split-graph $\Sigma(G)$ is a tree each of whose nodes has degree $\leqq d=$ the dimensionality of the splitting. Moreover, in this case the embedding of G in $\Sigma(G)$ is a true embedding, since the pieces of G are of unit size. (By the previous paragraph, we will then have access to an embedding of G in a binary tree, with the right Cost, by composing this embedding with the obvious embedding of a d-ary tree in a binary tree.) These consequences of the consistency of splits in a splitting follow by induction on the number of splits in Σ. If there is only one split, then $\Sigma(G)$ is a two-node tree. Assume for induction that any n-split splitting gives rise to an $(n+1)$-node tree. Say that we have an n-split splitting $\sigma_1,\sigma_2,\cdots,\sigma_n$, and we add to it a nonredundant split σ that is consistent with all the σ_i. If we look at the action of σ on the $n+1$ parts $P_1,P_2,\cdots,P_n$ of G produced by the n splits, we find that σ partitions precisely one of them into two parts, say P_1 into P_{1L} and P_{1R}: since σ is nonredundant it must split at least one P_i, and since it is consistent with all the σ_i, it cannot split more than one. Therefore, σ partitions the sets P_i, $i>1$, into two groups: say $P_2,\cdots,P_m$ reside in $G(L;\sigma)$ together with P_{1L}; and $P_{m+1},\cdots,P_{n+1}$ reside in $G(R;\sigma)$ together with P_{1R}. Since each σ_i is consistent with σ, it follows that each σ_i splits either the set $G(L;\sigma)$ (and is a *left* split) or the

set $G(R;\sigma)$ (and is a *right* split), but not both. Thus σ divides the splitting $\sigma_1, \sigma_2, \cdots, \sigma_n$ into a left sub-splitting and a right sub-splitting. By induction the left sub-splitting gives rise to a split-graph that is a tree with nodes $P_{1L}, P_2, \cdots, P_m$, and the right sub-splitting gives rise to a split-graph that is a tree with nodes $P_{1R}, P_{m+1}, \cdots, P_{n+1}$. We claim that the split-graph for the entire splitting, $\sigma_1, \sigma_2, \cdots, \sigma_n, \sigma$, is obtained from these two trees by joining nodes P_{1L} and P_{1R} with an edge. This new edge is justified since the split σ is the only one in the entire splitting that separates P_{1L} from P_{1R}, so $\delta(P_{1L}, P_{1R}) = 1$. No additional edges are justified since any other (left node)-(right node) pair is separated by both σ and at least one σ_i, and hence has distance at least 2 in the split-graph. Thus the split-graph is a tree with $n+2$ nodes, so the induction is extended.

Finally we note that no vertex of the split-graph can have degree exceeding d, the dimensionality of the splitting: one verifies easily that if $\delta(P, Q) = 1$, and σ is the one split that separates P from Q, then σ must enter into any expression for P as an intersection of the $G(D;\sigma_i)$'s. Details are left to the reader.

By our previous remarks, the theorem follows. □

The main purpose of Theorem 2.3 is to facilitate the proof of the more easily applied Theorem 2.4.

To *reduce* a graph G:

(a) If G has at most three vertices, it is reduced.

(b) Otherwise, replace G by two graphs G_1 and G_2, as follows.

By cutting edges of G, partition it into two induced subgraphs G_1' and G_2'. Adjoin to G_1' a new vertex that is adjacent to each vertex of G_1' that was (in G) adjacent to a vertex in G_2'; the resulting graph is G_1. Do the same to G_2' to obtain G_2.

(c) Reduce G_1 and G_2.

If this process terminates, we call the resulting collection of graphs a *reduction* of G. The family of graphs produced in this process is graded in a natural way: G is a grade-0 graph, and the graphs produced in step (b) from grade-i graphs are grade-$(i+1)$ graphs. An edge e_i of a grade-$(i>0)$ graph *represents* an edge e of G if either (1) edge e_i replaces edge e in step (b) (by replacing a $G_1 \leftrightarrow G_2$ adjacency by a $G_j \leftrightarrow$(new vertex) adjacency); or (2) edge e_i replaces a grade-$(i-1)$ edge that represents edge e.

THEOREM 2.4. (one-way graph reduction theorem) *If the graph G admits a reduction for which no edge of G has more than r representatives, then there is an embedding ε of G in a binary tree with* $\text{Cost}(\varepsilon) \leqq 2r$.

Proof. Every execution of Step (b) in the reduction procedure effects a split of the guest graph G; and the ensemble of such executions effects a splitting of G—with the easily patched defect that the final parts in the splitting may contain as many as three vertices each, a mere technicality whose resolution is left to the reader. The rank r of the splitting is easily seen to be the maximum number of representatives of any edge of G. The dimensionality d of the splitting is easily verified to be the largest number of "new" vertices (from Step (b)) that any vertex ends up adjacent to in the completely reduced version of G; but easily, $d \leqq 3$ since the reduction procedure doesn't stop until each component of G has at most three vertices. Thus, Theorem 2.3 would lead us to believe that G is embeddable in a binary tree with Cost $\leqq r$. In fact, we must double this estimate in order to resolve the technical problem that each part of G in the final reduction can have as many as three vertices. □

Example 2.5. *Thick trees.* A *k-tree*, $k \geqq 1$, is defined recursively as follows. A k-tree on k nodes is just a k-clique (= a copy of the complete graph on k vertices). A k-tree on $n+1$ nodes is obtained from a k-tree on n nodes by connecting the $(n+1)$th node to some $k-1$ mutually adjacent nodes in the n-node k-tree. It is easy

to reduce a k-tree by successively cutting off the last-added node v, together with the latest added node among those that are adjacent to v. Since the reduction procedure adds one node to replace the two just cut off, we have effectively cut off only one node. Clearly, then, the number of representatives of an edge of the k-tree G is at most maxdegree(G). Thus, the Cost of the derived embedding of G in a binary tree will be proportional to log maxdegree(G).

This completes our study of techniques for bounding from above the Costs of embeddings of graphs in binary trees. It is now time to turn to the complementary study of techniques for bounding these Costs from below.

3. Lower bounds on simulation costs.

A. *Techniques for lower bounds.* Two basic techniques are known for bounding from below the costs of embeddings in trees. First, we have the vertex-degree bound, which follows from consideration of the sizes of "balls" in binary trees.

THEOREM 3.1. (vertex-degree bound) *If ε embeds the graph G in a binary tree, then*

$$\mathrm{Cost}(\varepsilon) \geqq \log \tfrac{1}{3}(\mathrm{maxdegree}(G)+3).$$

Proof. Given any node v of a binary tree and any distance $d>0$, at most $3\cdot 2^d-3$ nodes of the tree lie within distance d of v. □

The second technique uses the sizes of G's separators to bound the costs of embeddings of G in binary trees.

Let $s(x)$ be any function from the real interval $[0, 1]$ into the set of positive integers. The graph G has an *$s(x)$-separator* if for all $\alpha \in [0, 1]$, any partition of G into subgraphs of sizes $\alpha|G|$ and $(1-\alpha)|G|$ must cut $s(\alpha)$ edges of G.

The following generalizes a result of [10].

THEOREM 3.2. (separator bound) *Say that the graph G has an $s(x)$-separator. For any embedding ε of G in a binary tree: for each $\alpha \in [0, \frac{1}{2})$, there is a $\beta \in [\alpha, 2\alpha)$ such that*

$$\mathrm{Cost}(\varepsilon) \geqq \tfrac{1}{2} \log s(\beta).$$

Proof. Let G be embedded in the binary tree T. By a now-standard argument [5], [10], one verifies that, for each $\alpha \in [0, \frac{1}{2})$, there is a $\beta \in [\alpha, 2\alpha)$ and a subtree T' of T such that T' holds $\beta|G|$ images of G's vertices under the embedding. By definition of separator, at least $s(\beta)$ edges connect those vertices of G whose images reside in T' with those vertices of G whose images reside in $T-T'$. One shows easily that these "cross-edges" have at least $\sqrt{s(\beta)}$ distinct sources in either T or T'. The size of "balls" in binary trees forces at least one source vertex to have an image residing at distance $\geqq \frac{1}{2} \log s(\beta)$ from the root of T', hence at least this distance from the image of its other end. □

Of course, Theorem 3.2 can be used only in conjunction with a technique for bounding from below the separator functions of the graphs in question. This bounding problem is likely to be computationally infeasible in general, since the problem: given G, k, α, to decide if G can be cut into the proportions $\alpha:(1-\alpha)$ by cutting at most k edges, is NP-complete [2]. But we can prove that certain properties of G guarantee big separators (though these conditions may themselves be hard to detect). We now present two such structural properties.

The graph G is *n-reticulated* if Vertices(G) contains two disjoint n element subsets A and B, called the *inputs* and *outputs* of G, such that: for all $k \leqq n$, for all choices of a k-element subset A' of A and a k-element subset B' of B, there exist k

edge-disjoint paths in G connecting A' and B'. (Note that if we change the phrase "edge-disjoint" here to "vertex-disjoint," then we would be defining an n-superconcentrator.)

Examples of n-reticulated graphs are: side-$(n+1)$ grids, side-$2n$ pyramids, any graph containing a *homeomorph* of such a grid, hence depth-$2n$ breadth-first trees, depth-$2n$ leap trees, and so on; see Fig. 6.

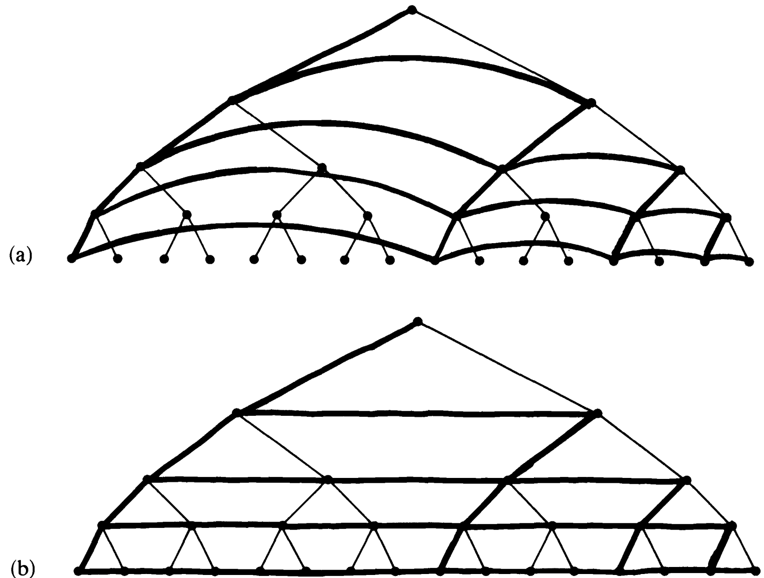


FIG. 6. (a) *The leap tree and* (b) *the breadth-first tree redrawn to emphasize their reticulation.*

THEOREM 3.3. (reticulation bound) *If G is n-reticulated, then any embedding ε of G in a binary tree has* $\mathrm{Cost}(\varepsilon) \geqq \frac{1}{2}\log n - 1$.

Proof. One can prove that any partition of an n-reticulated graph G into two subgraphs containing m $(\leqq n/2)$ and $n-m$ of G's inputs, respectively, must "cut" at least m edges of G; and one can prove the Theorem as a consequence. We opt instead for the simpler (in this case) course of proceeding from first principles.

When one lays G out in a binary tree T, there must be a subtree T' of T that receives between $n/4$ and $n/2$ outputs of G. (This follows from the same considerations from [5], [10] cited in the proof of Theorem 3.2.) This subtree effectively separates at least $n/4$ inputs of G from a like number of outputs, in the sense that one of the sets resides in the subtree, while the other set resides outside the subtree. But since G is n-reticulated, at least $n/4$ edge-disjoint paths in G connect the isolated inputs with the isolated outputs. Hence, there are at least $n/4$ edges of G connecting vertices whose images lie within the subtree T' with vertices whose images lie outside that subtree. The bound now follows from the existence of these "crossing edges" just as in the proof of Theorem 3.2. □

We use our second property to bound explicitly the sizes of G's separators.

The graph G is *stratified* if its vertices can be partitioned into *strata* $S_1, S_2, \cdots, S_m$ in such a way that

(1) each $|S_i| \leqq |S_{i+1}|$;
(2) the induced graph on each S_i is connected;
(3) the only edges between strata are between adjacent ones (S_i and $S_{i\pm1}$).

Associate with each vertex of each stratum S_i a finite family of subgraphs of G in any way so that for each $v \in S_i$ and its associated graph G_v:

(4) $v \in \text{Vertices}(G_v)$;

(5) G_v is connected;

(6) the vertex-sets of the graphs G_v partition the set $\bigcup_{j \geqq i} S_j$.

The integer m is called the *number of levels* in the stratification. As an example, let each stratum of the depth-d breadth-first tree be one of its (tree) levels, and let each graph G_v be the sub-breadth-first tree rooted at v. This specifies a $(d+1)$-level stratification of the tree.

Call any partition of G into strata plus an association of graphs G_v with each vertex v, a *stratification* of G. A stratification of G is *progressive* if the size $g(i)$ of the largest G_v with $v \in S_i$ decreases as i increases. The indicated stratification of the breadth-first tree is clearly progressive.

THEOREM 3.4. (stratification bound) *For every progressive m-level stratification that the graph G admits: if G has an $s(x)$-separator, then for all $\alpha \in (0, 1)$ and all strata S_i with $i \leqq m - s(\alpha)$,*

$$s(\alpha) \geqq \text{Min}\left\{m, \frac{\alpha|G| - i|S_{i+s(\alpha)}|}{g(i) + |S_{i+s(\alpha)}|}\right\}.$$

Proof. Say that we want to cut the graph G into subgraphs G_1 and G_2 whose sizes are in the proportion $\alpha:(1-\alpha)$ for some $\alpha \in (0, \frac{1}{2}]$. Assume that $s(\alpha) < m$. Look at any consecutive $s(\alpha)+1$ strata of G, say strata $i, i+1, \cdots, i+s(\alpha)$. By definition of separator and by the connectivity of strata, some one of these strata, say S_j, lies entirely in one of the G_i, say in G_2. But now, the only way we can have vertices from strata S_k, $k > j$, go into G_1 is to have edge-"cuts" below stratum j. Since these cuts cannot exceed $s(\alpha)$ in number, it follows that

$$\alpha|G| =_{\text{def}} |G_1| \leqq (i + s(\alpha))|S_{i+s(\alpha)}| + s(\alpha)g(i).$$

The first summand places all strata $k \leqq i + s(\alpha)$ into G_1 (perhaps overestimating by assuming that all of these strata have maximum possible population); the second summand places into G_1 $s(\alpha)$ copies of the biggest G_v with $v \in S_i$. This very conservative accounting clearly can only overestimate the size of G_1, whence the theorem. □

Remark. The same proof technique proves that if each $|S_i| \geqq |S_{i+1}|$, then

$$s(\alpha) \geqq \text{Min}\left\{m, \frac{\alpha|G| - i|S_i|}{g(i) + |S_i|}\right\}.$$

B. *Applications.*

Application 3.1. The construction of Theorem 2.1 is within a factor of 3 of being optimal.

Proof. Theorem 3.1. □

Application 3.2. For any embedding ε of the depth-d leap tree in a binary tree,

$$\text{Cost}(\varepsilon) \geqq \tfrac{1}{2} \log d - \tfrac{3}{2}.$$

Proof. The depth-d leap tree contains as a subgraph the side-$(d+1)$ pyramid: the horizontal edges of the pyramid are the horizontal edges of the tree; and the vertical edges of the pyramid all have the form $(2^a 1^b, 2^a 1^{b+1})$ in the tree [see Fig. 6(a)]. Thus the leap tree is $d/2$-reticulated. □

Application 3.3. For any embedding ε of the depth-d breadth-first tree in a binary tree,

$$\text{Cost}(\varepsilon) \geqq \tfrac{1}{2} \log d - \tfrac{3}{2}.$$

Proof. This application follows from two of our bounding techniques.

First, the depth-d breadth-first tree contains as a subgraph a homeomorph of the side-$(d+1)$ pyramid: the horizontal edges of the pyramid are horizontal paths in the tree; and the vertical edges of the pyramid all have the form $(2^a 1^b, 2^a 1^{b+1})$ in the tree [see Fig. 6(b)]. Thus the breadth-first tree is $d/2$-reticulated.

Second, as we noted earlier, the depth-d breadth-first tree admits a progressive d-level stratification: each stratum is a level of the tree, and each G_v is the sub-breadth-first tree rooted at v. Assume that the tree has an $s(x)$-separator, where $s(\alpha) < (1-\delta)d$ for some $\alpha \in (\frac{1}{4}, \frac{1}{2}]$ and $\delta > 0$. Then, invoking Theorem 3.4 for level $k = d/2$, we have

$$\begin{aligned} s(\alpha) &\geqq \frac{\alpha 2^{d+1} - \alpha - \left(\frac{\delta d}{2}\right) 2^{(1-\delta/2)d}}{2^{(1-\delta/2)d+1} - 1 + 2^{(1-\delta/2)d}} \\ &> (\text{const}) 2^{\delta d/2} \\ &> (1-\delta)d \end{aligned}$$

for sufficiently large d. This contradicts our assumed inequality on $s(\alpha)$. Since δ was arbitrary, we can now conclude that any sufficiently deep breadth-first tree has no $s(\alpha)$-separator smaller than its depth for any $\alpha \in (\frac{1}{4}, \frac{1}{2}]$. (This is easily shown to be true for any α bounded away from 0 and 1.) An appeal to Theorem 3.2 completes the proof. □

4. Graphs that are almost binary trees. Let $\mathscr{G}$ and $\mathscr{H}$ be families of graphs. Recall that the family $\mathscr{G}$ is *almost* $\mathscr{H}$ if $\mathscr{G}$ can be simulated by $\mathscr{H}$. We build on the results in § 2 to establish "two-way" versions of the one-way results of that section. These results all yield to extensions of the proofs of the "one-way" results of that section.

THE OUTERPLANAR THEOREM. *The bounded-degree family of graphs $\mathscr{G}$ is almost a family of binary trees iff it is almost outerplanar.*

Proof. Sufficiency follows from Theorem 2.1, and necessity from the fact that trees are outerplanar graphs. □

Example 4.1. The following three families of graphs are similar: Binary trees, Preorder trees, and Inorder trees.

THE PLANAR EMBEDDING THEOREM. *The family of graphs $\mathscr{G}$ is almost a family of binary trees iff there is a constant k such that every $G \in \mathscr{G}$ is embeddable in the plane with planarity number, fanout number, escape number, and dual escape number all less than or equal to k.*

Proof. Sufficiency. Theorem 2.2.

Necessity. We merely sketch the proof. Suppose the graph G can be embedded in a binary tree within Cost c. For each edge e of G, we draw the image e' in the binary tree in such a way that (1) the image e' is very close to the tree-edges, and (2) any two images e_1' and e_2' cross at no more than one point; see Fig. 7. In fact, this drawing is the planar embedding needed. It is easy to verify that (a) every image e' has a bounded number of crossing points, because the number of edges that cross any given point is bounded, so the planarity number is bounded; (b) every inner point (including crossing points) can escape to the exterior by crossing a bounded number of edges, so the escape number is bounded, and, dually, the dual escape number is bounded; and (c) the fanout number is bounded. □

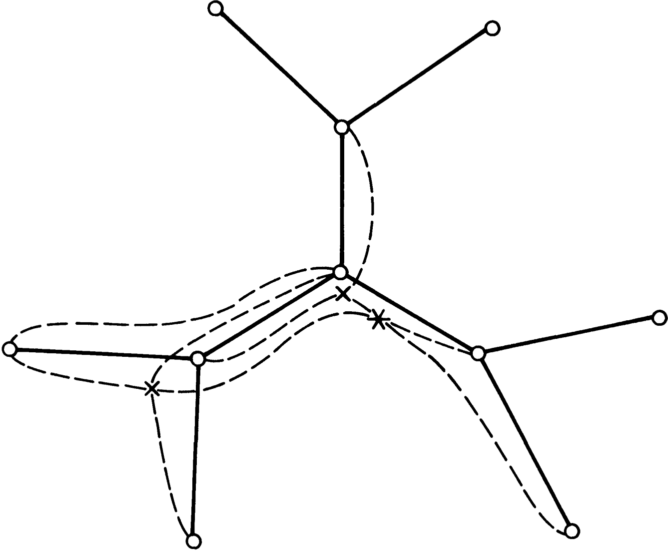


FIG. 7. *Illustrating the proof of the planar embedding theorem.*

Example 5.2. The family of triple binary trees, and its higher-index relatives, like quadruple binary trees, etc., are all similar to the family of binary trees.

THE GRAPH SPLITTING THEOREM. *The bounded-degree family of graphs $\mathcal{G}$ is almost a family of binary trees iff there is a constant k such that every $G \in \mathcal{G}$ admits a total splitting of rank $r \leqq k$ and dimensionality $d \leqq k$.*

Proof. Sufficiency. Theorem 2.3.

Necessity. By cutting every edge of a host binary tree, we obtain a total splitting of any graph G that is embedded in the tree. The dimensionality of the splitting is at most 3; and its rank is not more than the Cost of the embedding. □

THE GRAPH REDUCTION THEOREM. *The bounded-degree family of graphs $\mathcal{G}$ is almost a family of binary trees iff there is a constant k such that every $G \in \mathcal{G}$ admits a total reduction for which no edge of G has more than k representatives.*

Proof. Sufficiency. Theorem 2.4.

Necessity. By cutting in a host tree every edge that is not incident to a leaf, we obtain in a natural way a series of reductions of any graph G that is embedded in the tree. The resulting graphs have at most 4 nodes each; and each edge has at most c representatives, where c is the Cost of the embedding. It is a simple matter to reduce the resulting 4-node graphs to obtain a total reduction of G. □

Acknowledgments. The major portion of the research reported here was done while the authors were visiting the Department of Computer Science, University of Toronto, Toronto, Canada. A portion of the second author's research was done while visiting the Department of Mathematical Sciences, University of Tel-Aviv, Tel-Aviv, Israel. The authors are grateful to Romas Aleliunas for several stimulating conversations on the topics reported here.

REFERENCES

[1] R. A. DEMILLO, S. C. EISENSTAT AND R. J. LIPTON, *Preserving average proximity in arrays*, Comm. ACM, 21 (1978), pp. 228–231.

[2] M. R. GAREY, D. S. JOHNSON AND L. J. STOCKMEYER, *Some simplified NP-complete graph problems*, Theoret. Comput. Sci., 1 (1976), pp. 237–267.

[3] D. E. KNUTH, *The Art of Computer Programming I: Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1968.

[4] H. T. KUNG AND D. STEVENSON, *A software technique for reducing the routing time on a parallel computer with a fixed interconnection network*, in High Speed Computer and Algorithm Optimization, Academic Press, New York, 1977, pp. 423–433.

[5] R. J. LIPTON, S. C. EISENSTAT AND R. A. DEMILLO, *Space and time hierarchies for classes of control structures and data structures*, J. Assoc. Comput. Mach., 23 (1976), pp. 720–732.

[6] R. J. LIPTON AND R. E. TARJAN, *Applications of a planar separator theorem*, Proc. 18th IEEE Symposium on Foundations of Computer Science, (1977), pp. 162–170.

[7] A. L. ROSENBERG, *Preserving proximity in arrays*, this Journal, 4 (1975), pp. 443–460.

[8] ———, *Data encodings and their costs*, Acta Inform., 9 (1978), pp. 273–292.

[9] ———, *Encoding data structures in trees*, J. Assoc. Comput. Mach., 26 (1979), pp. 668–689.

[10] A. L. ROSENBERG AND L. SNYDER, *Bounds on the costs of data encodings*, Math. Syst. Theory, 12 (1978), pp. 9–39.

[11] A. L. ROSENBERG, D. WOOD AND Z. GALIL, *Storage representations for tree-like data structures*, Math. Syst. Theory, 13 (1980), pp. 105–130.

[12] L. G. VALIANT, *Universality considerations in VLSI circuits*, IEEE Trans. Elec. Comp., C-30 (1981), pp. 135–140.

SIAM J. COMPUT.
Vol. 11, No. 2, May 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1102-0004 $01.00/0

AN ASYMPTOTICALLY OPTIMAL ALGORITHM FOR THE DUTCH NATIONAL FLAG PROBLEM*

JAMES R. BITNER†

I dedicate this paper to my father, Richard Bitner (1922–1981).

Abstract. We develop an algorithm for the Dutch National Flag problem that has an adjustable integer parameter smax $\geqq 0$ allowing a time/space tradeoff. (Let n be the length of the input to be ordered.) The space required is proportional to smax (but independent of n), and the average number of swaps required is $((6\,\text{smax}+10)/(18\,\text{smax}+27))\,n+o(n)$. We show $\frac{1}{3}n+o(n)$ is a lower bound. Hence as smax $\to\infty$, the performance can be made arbitrarily close to the lower bound. This is a significant improvement over previous algorithms due to the use of a different invariant. We show that an algorithm using Dijkstra's invariant must use at least $\frac{5}{9}n$ swaps. We also study the problem of more than three colors.

Key words. Dutch National Flag problem, eulerian digraphs, algorithm design, loop invariant, stepwise refinement, probabilistic analysis of algorithms.

1. Introduction. In the Dutch National Flag problem, we are given a sequence of n pebbles, each of which is either red, white, or blue, and we are to rearrange them such that all the red pebbles occur first in the sequence, followed by all the whites, followed by all the blues. We are restricted to using the following two primitive functions in accessing the sequence: buck(i) which gives the color of the ith pebble in the sequence, and swap(i, j), which interchanges the ith and jth pebbles. Buck is deemed a very expensive operation and hence may be applied *only once* to each pebble in the sequence. Our objective is to find an algorithm to solve the problem which uses as few swaps as possible in the average case, where each of the 3^n initial sequences is equally likely. Also, the algorithm must operate using a *constant amount of space*, independent of the length of the sequence.

Notation. Through this paper n will denote the length of the sequence to be ordered and c will denote the number of possible colors (unless otherwise noted, $c=3$).

This problem was originally posed by Dijkstra [1] as an example of how the technique of refinement leads logically from a naive solution to a more efficient one. McMaster [3] analyzed both of Dijkstra's solutions and showed that the original naive solution required $\frac{2}{3}n$ swaps, while the refined solution required $\frac{2}{3}n-\frac{1}{4}+\frac{1}{4}(-\frac{1}{3})^n$ swaps. Thus, the improvement, bought by considerably complicating the algorithm, is asymptotically only $\frac{1}{4}$ of a swap. McMaster's result stresses that gains from refinements may be deceptive and that careful analysis is required.

An additional solution was later given by Meyer [2] which avoided swapping uninspected pebbles. Although not immediately apparent, Meyer's solution can also be derived from Dijkstra's by a sequence of refinements. McMaster also analyzed Meyer's solution and found it required $\frac{5}{9}n$ swaps.

In this paper, we will solve the problem using a different invariant which will not be naive, but carefully chosen based on an analysis of the problem. Our refinements will not be used to make major improvements in the efficiency, but to simplify our solution. This will lead to an algorithm asymptotically[1] requiring $\frac{1}{3}n$ swaps, which is proven to be optimal.

---

* Received by the editors January 17, 1980, and in final revised form April 1, 1981. This work was supported in part by the National Science Foundation under grant MCS 77-02705.

† Department of Computer Science, University of Texas, Austin, Texas 78712.

[1] See below for an exact definition.

Another of our results shows the limitation of refining an "obvious" but naive invariant in this problem. We show in § 6 that the figure of $\frac{5}{9}n$ achieved by Meyer is a *lower bound* on the number of swaps required by *any* solution using Dijkstra's invariant. Thus, no matter how long or cleverly one refines a solution using this invariant, it *must* use 66% more swaps than is required.

The paper is organized as follows:
In § 2, we develop a correspondence between sequences of pebbles and eulerian digraphs, which is then used to give a lower bound on the number of swaps required to order any given sequence for arbitrary c. We study the expected value of this lower bound to get a lower bound of $((c-1)/2c)n$ on the average number of swaps required to order a sequence. In § 3 we study algorithms to solve the problem for arbitrary c which use more than a constant amount of space. One is the *shortest cycle first algorithm*, which for $c \leqq 5$, orders any given sequence using the minimum number of swaps. For $c > 5$, we prove the *average* number of swaps used by this algorithm is asymptotically optimal as $n \to \infty$. We also study worst case bounds for $c > 5$.

In § 4, we use some ideas from the proof of the lower bound to develop an algorithm which solves the problem using constant space. The algorithm has a parameter, smax, that allows a time/space tradeoff. Increasing smax will reduce the number of swaps at the expense of requiring more space. It should be stressed that smax does not depend on n; therefore no matter what the value of smax, the algorithm still uses constant space (an amount proportional to smax). What we actually have is an infinite sequence of algorithms, each requiring constant space and each giving better and better performance.

Section 5 analyzes the algorithm. Asymptotically, for any smax $\geqq 0$, $((6\,\text{smax}+10)/(18\,\text{smax}+27))n$ swaps are required on the average. Even for smax $=0$, the average of $\frac{10}{27}n$ swaps is better than Meyer's algorithm, and as smax $\to \infty$, the average number of swaps can be made arbitrarily close to the lower bound and, hence, is asymptotically optimal in some sense.

Finally, § 6 shows that $\frac{5}{9}n$ is a lower bound on the number of swaps used by any algorithm which uses Dijkstra's invariant.

2. Lower bounds. In this section we develop a correspondence between sequences of pebbles and eulerian digraphs (see below for definition). We first use this correspondence to derive a lower bound on the number of swaps required to order any given sequence. We also use this correspondence to study the number of swaps required in the average case.

DEFINITION. An *eulerian digraph* is a directed graph in which every vertex has its indegree equal to its outdegree. We allow a digraph to have multiple edges and self-loops. (Unless otherwise noted all our digraphs will be eulerian.) For an eulerian digraph G, let $e(G)$ be the number of edges in G, and index(G) be $e(G)-M(G)$, where $M(G)$ is the number of cycles in a maximal decomposition of G into edge disjoint cycles. A *cycle* is a path in the digraph whose initial and final vertices are identical. It may pass through a vertex more than once. If a cycle passes through each vertex only once, it is a *simple cycle*. It is simplest to consider decompositions of digraphs into cycles and not require that the cycles be simple. (A maximal decomposition will, however, consist solely of simple cycles.)

To develop a correspondence between sequences of pebbles and eulerian digraphs, we first divide a sequence into "regions." If there are a total of x_i pebbles of color i in the sequence, let the first x_1 positions be region 1, the next x_2 in region 2, and so on. The digraph corresponding to this sequence has c vertices and is created by adding

one edge from vertex i to vertex j for every pebble of color i in region j. Note that a sequence is completely ordered if and only if for all i, all pebbles of color i are in region i. Hence, the digraph corresponding to the completely ordered sequence consists solely of self-loops.

THEOREM 2.1. *A digraph constructed from a sequence as described above is a eulerian digraph.*

Proof. For any i, the indegree of vertex i is the number of pebbles in region i, and the outdegree is the number of pebbles of color i. These quantities are equal by the definition of the regions. □

To order a sequence, given any decomposition of the corresponding digraph (call it G) into edge-disjoint cycles[2], we ignore the self-loops and sequence through the remaining cycles in any order. If the current cycle is $v_{i_1}, v_{i_2}, \cdots, v_{i_k}$, there must be a pebble of color i_j in region i_{j+1} for $j = 1, \cdots, k-1$ and a pebble of color i_k in region i_1. Clearly, $k-1$ swaps can be used to put each of these k pebbles in the correct region. Also note that after processing all the cycles, the sequence will be ordered. If the decomposition has k cycles with the ith having length L_i, the total number of swaps done is $\sum_{i=1}^{k} [L_i - 1] = e(G) - k$. Clearly, this quantity is minimized by using a maximal decomposition.

We now show that using the above procedure with the maximal decomposition uses the minimum number of swaps over *all* possible algorithms, not just those using this strategy. We use an "entropy argument", where index (G) is the entropy function. (G is the digraph corresponding to the current sequence in the execution of some algorithm.) Index (G) must be decreased using swaps from its initial value down to zero. (A sequence is ordered if and only if every pebble is in the correct region, i.e., its corresponding digraph has index equal to zero.) Lemma 2.1 shows that a swap cannot decrease index (G) by very much.

LEMMA 2.1. *Let S be any sequence and S' be any sequence obtained from S by doing one swap. Let G and G' be the digraphs corresponding to, respectively, S and S'. Then* index $(G') \geqq$ index $(G) - 1$.

Proof. The effect on G of swapping a color i pebble in region j with a color k pebble in region l is shown in Fig. 2.1 (i, j, k, and l are not necessarily distinct). Edges e_1 and e_2 in G are replaced by e_1' and e_2' to form G'. Since $e(G) = e(G')$, proving $M(G') \leqq M(G) + 1$ will prove the theorem. Suppose $M(G') > M(G) + 1$ and let a maximal decomposition of G' be $C_1, \cdots, C_m$ (where $m = M(G')$).

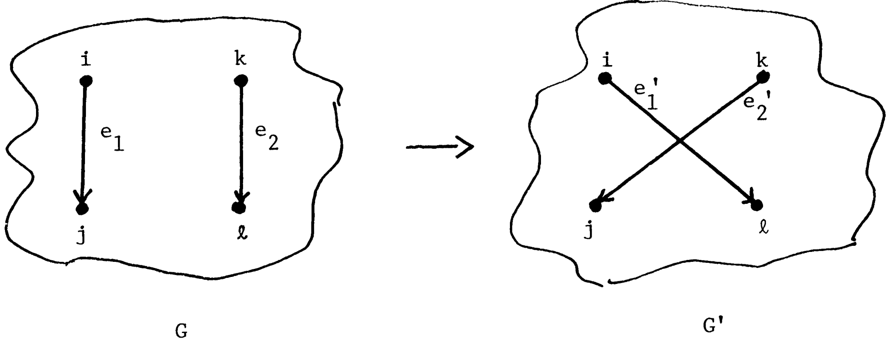


FIG. 2.1. *The effect of a swap on the corresponding digraphs.*

[2] It is easy to show that every eulerian digraph has such a decomposition, see for example [4].

We consider two cases.

Case 1 (e_1' and e_2' are on the same C_h). Let P_{lk} be the portion of C_h from l to k and P_{ji} the portion from j to i. Form two cycles in G: C_h', which consists of e_1 and P_{ji}, and C_h'', which consists of e_2 and P_{lk}. Hence $C_1, \cdots, C_{h-1}, C_h', C_h'', C_{h+1}, \cdots, C_m$ is a decomposition of G into $m+1$ cycles, a contradiction.

Case 2 (e_1' and e_2' are on different C_h's, say, C_1 and C_2). Let P_{li} be the portion of C_1 from l to i and P_{jk} be the portion of C_2 from j to k. Form a cycle C_1' in G consisting of e_1, P_{jk}, e_2, P_{li}. Then $C_1', C_3, \cdots, C_m$ forms a decomposition of G into $m-1$ cycles, again, a contradiction. Hence $M(G') \leqq M(G)+1$ and the lemma is proved. □

THEOREM 2.2. *Given a sequence S_0, let G_0 be the corresponding digraph. Then at least index (G_0) swaps must be used in order S_0.*

Proof. Suppose k swaps are used. Let S_i be the sequence after i swaps and G_i be the corresponding digraph. Since S_k is ordered, G_k consists solely of self-loops, and index $(G_k) = 0$.

By Lemma 2.1, index $(G_i) \geqq$ index $(G_{i-i}) - 1$ and clearly $k \geqq$ index (G_0). □

COROLLARY 2.1. *Using a maximal decomposition in the manner previously described orders any given sequence in the minimal number of swaps.*

Theorem 2.2 provides a lower bound and Corollary 2.1 shows it is achievable (though not necessarily by an algorithm using constant space). We will discuss the problem of actually finding a maximal decomposition in the next section.

Before considering the average value of index (G) in order to obtain a lower bound on the average number of swaps, we introduce some notation and prove several lemmas.

DEFINITION. For any $n \geqq 1$ and $c \geqq 1$ and $1 \leqq i, j \leqq c$ let the random variables $I_{ij}^{(n)}$ be the number of edges from i to j in the digraph corresponding to a random arrangement of n pebbles with c possible colors. (Note that $I_{ij}^{(n)}$ also depends on c, but we omit this to simplify the notation.)

DEFINITION. By the phrase *number of* 2*-cycles in a digraph*, we mean the maximum number of 2-cycles possible in any decomposition of that digraph.

This quantity can be simply expressed in terms of the $I_{ij}^{(n)}$'s. The key is the following lemma.

LEMMA 2.2. *Given an eulerian digraph G, there exists a maximal decomposition such that for any edge $e = (u, v)$ with $u \neq v$, if the cycle containing e in the decomposition has length greater than* 2, *then every edge of form (v, u) is contained in a* 2*-cycle in the decomposition.*

Proof. Suppose the lemma does not hold for some digraph. Choose any maximal decomposition of G having the smallest number of edges which violate the lemma. For this decomposition, there exist edges $e_1 = (u, v)$ and $e_2 = (v, u)$, each contained in a (distinct) cycle of length greater than 2. (See Fig. 2.2). Then a new decomposition can be formed by replacing C_1 and C_2 by the 2-cycle consisting of e_1 and e_2 and the cycle consisting of the remaining edges in C_1 and C_2. Since this decomposition has at least as many cycles as the original, it too is maximal, and then, since it has fewer edges which violate the lemma, it provides a contradiction. □

COROLLARY 2.2. *The number of* 2*-cycles in a digraph equals $\sum_{i<j} \min (I_{ij}^{(n)}, I_{ji}^{(n)})$.*

COROLLARY 2.3. *Given an eulerian digraph G, there exists a maximal decomposition which contains all the self-loops. In addition, the number of* 2*-cycles in the decomposition equals the number of* 2*-cycles in G.*

We now begin our analysis of the average case, which relies heavily on the law of large numbers. For three colors, we expect with high probability, a nearly equal

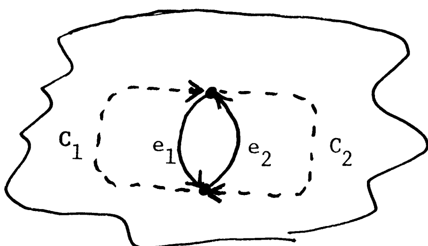


FIG. 2.2. *The 2-cycle is not in this maximal decomposition. C_1 and C_2 are the two cycles in the decomposition which contain the 2-cycle's edges.*

number of red, white and blue pebbles. Hence, we expect the three regions to be of nearly equal length and to contain approximately the same number of red, white and blue pebbles. For c colors, we have c regions of nearly equal length, with the fraction of pebbles of each color in each region nearly equal to $1/c$. This idea is formalized in Theorem 2.3. Note that this theorem is *not* just a trivial application of the law of large numbers, since the size of each region is unknown and, even worse, depends on the number of pebbles of each color. After proving Theorem 2.3 it is easy to prove that there exists a maximal decomposition of the digraph which consists nearly exclusively of self-loops and 2-cycles (Lemma 2.5), and hence, the required number of swaps is easily analyzed (Theorem 2.4).

Before proceeding, we need 2 easy lemmas whose proofs are omitted.

LEMMA 2.3. *Let $E_1, E_2, \cdots$ and $F_1, F_2, \cdots$ be sequences of events such that* Prob $(E_n) \to 1$ *and* Prob $(F_n) \to 1$. *Then* Prob $(E_n$ *and* $F_n) \to 1$.

LEMMA 2.4. *Let $A_1, A_2, \cdots$ and $B_1, B_2, \cdots$ be sequences of random variables such that A_n/n and B_n/n are bounded above and below (by constants) for all n. Further, suppose, for all $\varepsilon > 0$,*

$$\text{Prob}\left(\left|\frac{A_n}{n}-L\right|<\varepsilon\right)\to 1 \quad \text{and} \quad \text{Prob}\left(\left|\frac{B_n}{n}-L\right|<\varepsilon\right)\to 1.$$

Then

$$E\left(\min\left(\frac{A_n}{n}, \frac{B_n}{n}\right)\right)\to L.$$

THEOREM 2.3. *For any $n \geqq 1$, $c \geqq 1$, $\varepsilon > 0$, and $1 \leqq i, j \leqq c$,*

$$\text{Prob}\left(\left|\frac{I_{ij}^{(n)}}{n}-\frac{1}{c^2}\right|<\varepsilon\right)\to 1.$$

Proof. Define the following random variables:

$J_{ij}^{(n)}$—the number of pebbles of color i in positions $(j-1)n/c$ through jn/c,
$B_L^{(n)}$—the total number of pebbles of colors $1, \cdots, i-1$,
$B_R^{(n)}$—the total number of pebbles of colors $1, \cdots, i$,
$U_L^{(n)} = |B_L^{(n)} - (j-1)n/c|$, and $U_R^{(n)} = |B_R^{(n)} - jn/c|$.

$J_{ij}^{(n)}$ is our estimate of $I_{ij}^{(n)}$. $U_L^{(n)}$ and $U_R^{(n)}$ measure how near the actual boundaries of region i ($B_L^{(n)}$ and $B_R^{(n)}$) are to their means, giving a measure of how accurately $J_{ij}^{(n)}$ measures $I_{ij}^{(n)}$. Note that $J_{ij}^{(n)}$, $B_L^{(n)}$ and $B_R^{(n)}$ are all sums of a *fixed number* of Bernoulli random variables, hence the law of large numbers applies to them. Choose

any $\varepsilon > 0$. By the law of large numbers, we have

$$\text{(1)} \qquad \operatorname{Prob}\left(\left|\frac{J_{ij}^{(n)}}{n}-\frac{1}{c^2}\right|<\varepsilon\right)\to 1,$$

$$\text{(2)} \qquad \operatorname{Prob}\left(\left|\frac{B_L^{(n)}}{n}-\frac{(j-1)n}{c}\right|<\varepsilon\right)\to 1 \quad \text{so } \operatorname{Prob}\left(\frac{U_L^{(n)}}{n}<\varepsilon\right)\to 1,$$

$$\text{(3)} \qquad \operatorname{Prob}\left(\left|\frac{B_R^{(n)}}{n}-\frac{j}{c}\right|<\varepsilon\right)\to 1 \quad \text{so } \operatorname{Prob}\left(\frac{U_R^{(n)}}{n}<\varepsilon\right)\to 1.$$

From the definition of our random variables, it is clear that

$$J_{ij}^{(n)}-U_L^{(n)}-U_R^{(n)} \leqq I_{ij}^{(n)} \leqq J_{ij}^{(n)}+U_L^{(n)}+U_R^{(n)}$$

or, equivalently,

$$\text{(4)} \qquad \frac{J_{ij}^{(n)}}{n}-\frac{1}{c^2}-\frac{U_L^{(n)}}{n}-\frac{U_R^{(n)}}{n} \leqq \frac{I_{ij}^{(n)}}{n}-\frac{1}{c^2} \leqq \frac{J_{ij}^{(n)}}{n}-\frac{1}{c^2}+\frac{U_L^{(n)}}{n}+\frac{U_R^{(n)}}{n}.$$

We define two sequences of events: E_n occurs if and only if

$$\left|\frac{J_{ij}^{(n)}}{n}-\frac{1}{c^2}\right|<\frac{\varepsilon}{2} \quad \text{and} \quad \frac{U_L^{(n)}}{n}<\frac{\varepsilon}{4} \quad \text{and} \quad \frac{U_R^{(n)}}{n}<\frac{\varepsilon}{4}.$$

F_n occurs if and only if $|I_{ij}^{(n)}/n-1/c^2|<\varepsilon$.

Clearly, by (4), if E_n occurs, F_n must also occur. Hence $\operatorname{Prob}(F_n) \geqq \operatorname{Prob}(E_n)$. Since $\operatorname{Prob}(E_n) \to 1$ (by (1), (2), (3) and two applications of Lemma 2.3), we have $\operatorname{Prob}(F_n) \to 1$, proving the theorem. □

Notation. By $o(n)$ we mean some function $f(n)$ such that $\lim_{n\to\infty} f(n)/n = 0$.

DEFINITION. For a given n and c, let G be the digraph corresponding to a random arrangement of n pebbles of c colors. (G is actually a random variable.) Also define the functions:

$S(G)$—the number of self-loops in G,

$T(G)$—the number of 2-cycles in G,

$U(G)$—the maximum number of cycles in a decomposition of G', the graph resulting from G after removing all self-loops and 2-cycles.

(Note: Usually we will simply write "$E(S)$" instead of "$E(S(G))$.")

LEMMA 2.5. $E(S)=n/c+o(n)$, $E(T)=(c-1)n/2c+o(n)$, $E(U)=o(n)$.

Proof. By definition, $E(S)=E(\sum_{i=1}^{c} I_{ii}^{(n)})$. Then, since Theorem 2.3 shows $E(I_{ii}^{(n)})=n/c^2+o(n)$, $E(S)=n/c+o(n)$ as claimed. By Corollary 2.2, $E(T)=\sum_{i<j} E(\min(I_{ij}^{(n)}, I_{ji}^{(n)}))$. By Theorem 2.3 and Lemma 2.4, $E(\min(I_{ij}^{(n)}, I_{ji}^{(n)}))= n/c^2+o(n)$. Therefore $E(T)=c(c-1)/2\times[n/c^2+o(n)]=(c-1)n/2c+o(n)$ as claimed. The self-loops and 2-cycles account for $n+o(n)$ edges, leaving at most $o(n)$ edges in G after the self-loops and 2-cycles have been removed. Hence $E(U)= o(n)$. □

THEOREM 2.4. *The average number of swaps required to order an arrangement with c colors is* $((c-1)/2c)n+o(n)$.

Proof. For any eulerian digraph G

$$0\cdot S(G)+1\cdot T(G) \leqq \operatorname{index}(G) \leqq 0\cdot S(G)+1\cdot T(G)+(c-1)\cdot U(G).$$

By Lemma 2.5, $E(\operatorname{index}(G))=(c-1)n/2c+o(n)$. By Corollary 2.1 this gives the average number of swaps required to order the arrangement. □

COROLLARY 2.4. *The average number of swaps required to order an arrangement with* 3 *colors is* $n/3+o(n)$.

3. Algorithms for finding a maximal decomposition. In this section, we discuss algorithms to solve the Dutch National Flag Problem where an arbitrary number of colors are allowed. We will drop the constant space assumption and study algorithms that first count the number of pebbles of each color, then construct the digraph associated with the given sequence and decompose it in some manner into cycles, then sequence through the cycles and perform the appropriate sequence of swaps to "remove" each cycle. Hence, the problem reduces to one of finding a maximal or near-maximal decomposition. Though finding a maximal decomposition for arbitrary c appears to be a hard problem, we present a simple, "greedy" algorithm, the *shortest cycle first algorithm*, which, at each step, arbitrarily removes any of the shortest cycles remaining in the digraph. (The algorithm can either be thought of as nondeterministic, or as specifying a class of algorithms.) We prove that if the digraph has at most five vertices (i.e., $c \leqq 5$), the shortest cycle first algorithm finds a maximal decomposition. If $6 \leqq c \leqq 8$, the shortest cycle first algorithm might find a maximal decomposition (i.e., *some* "shortest cycle first" decomposition exists for every digraph). We prove a very general class of algorithms gives an asymptotically optimal decomposition in the average case and conclude with a worst case bound on the performance of this class.

THEOREM 3.1. *For an eulerian digraph with at most five vertices (i.e., $c \leqq 5$) the shortest cycle first algorithm finds a maximal decomposition.*

Proof. Consider any digraph G. If $M(G)=0$, the shortest cycle first algorithm will trivially be successful. We proceed by induction on $M(G)$. Suppose C is the first cycle chosen by the shortest cycle first algorithm and let x be the length of C.

We first show that $M(G-C)=M(G)-1$ (i.e., C was not a bad first choice). This is trivially true if C is contained in some maximal decomposition, so suppose no maximal decomposition contains C. Choose any one. Though it does not contain C, every edge in C must be included in one of its cycles. Suppose y cycles in this decomposition have edges in common with C. Consider a new decomposition (see Fig. 3.1) where we choose C, then the long cycle consisting of the remainder of the y cycles (call this C') and then choose the remainder of the original decomposition. Since each of the y original cycles has length at least x (the length of the shortest cycle), C' must have length at least $xy-x$ (x must be subtracted because C has been removed). Hence there must be a vertex occurring at least $\lceil (xy-x)/c \rceil$ times on C', and C' can be split into at least $\lceil (xy-x)/c \rceil$ simple cycles. The new decomposition has at least $1+\lceil (xy-x)/c \rceil$ cycles in place of the y of the original decomposition. Since it is easy to verify that $1+\lceil (xy-x)/c \rceil = y$ for $2 \leqq y \leqq x \leqq c \leqq 5$, the new decomposition has as many cycles as the original, providing a maximal decomposition containing C, a contradiction. Hence $M(G-C)=M(G)-1$.

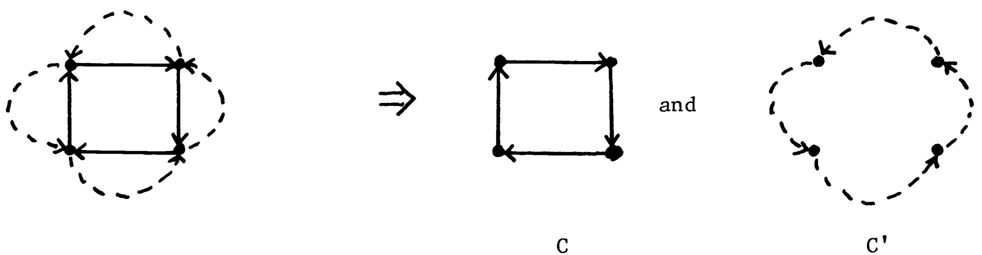


FIG. 3.1. *A cycle not in the decomposition and the cycles in the decomposition which contain its edges. The situation is not necessarily this simple; the dotted cycles may contain more than one edge of the other cycle and these edges need not be adjacent. However, C' is a cycle in any case.*

By induction, applying the shortest cycle first algorithm to $G-C$ will correctly give a decomposition into $M(G-C)=M(G)-1$ cycles. This added to C gives a decomposition into $M(G)$ cycles for G, proving the theorem. □

If $c=6$, more care is required. If we arbitrarily choose from among the shortest cycles, an optimal decomposition might not result (see Fig. 3.2). However, we can prove the following, weaker, result.

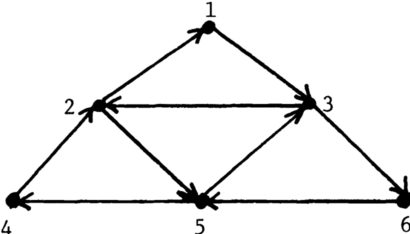


FIG. 3.2. *A digraph for which the shortest cycle first algorithm might not find a maximal decomposition. The algorithm might choose cycle* 2-5-3 *first and obtain a decomposition into two cycles. However, a decomposition into three cycles* (2-5-4, 1-3-2, 3-6-5) *is possible.*

THEOREM 3.2. *Given any eulerian digraph with* 6, 7, *or* 8 *vertices, there exists a maximal decomposition whose cycles can be ordered, such that, if the cycles are removed in that order from the digraph, a shortest cycle is removed at each step* (*i.e.*, *there is some "shortest cycle first" decomposition*).

Proof. (Similar to Theorem 3.1). Consider one of the shortest cycles in a given digraph. Call it C and suppose it has x nodes. If C does not occur in any maximal-decomposition, consider the y cycles that intersect C. If any has length x, it can be chosen, and the theorem can be proven by induction as Theorem 3.1. If none do, then we proceed as in Theorem 3.1, forming C', which now has $y(x+1)-x$ vertices and must form $\lceil (y(x+1)-x)/c \rceil$ cycles. Verifying that

$$1+\left\lceil \frac{y(x+1)-x}{c} \right\rceil \geqq y \quad \text{for } 2 \leqq y \leqq x \leqq c \leqq 7$$

proves the theorem for $c=6$ and 7.

The above proof "almost works" for $c=8$; the inequality is violated only when $x=y=4$. We analyze this situation as a special case. Suppose the theorem is false. Then there exists a digraph (G) whose shortest cycle, $C=(v_1, v_2, v_3, v_4)$, has length 4, and C is not in any maximal decomposition. Further, since $y=4$, each edge of C is in a different cycle in the optimal decomposition. The length of each of these four cycles is at least 5.

Consider the graph (G') consisting of these four cycles with the edges in C removed. Note that G' is a cycle. We repeatedly use the fact that G' can be partitioned into at most 2 cycles. (Otherwise its decomposition plus C would give at least 4 cycles and choosing C would provide a maximal decomposition.) We number the vertices in G' $x_1, x_2, \cdots, x_k$ by tracing the path from v_1 to v_4 to v_3 to v_2 to v_1. (Since $c=8$ the x's are not all distinct.) Since each of the original cycles intersecting C had length at least 5, $k \geqq 16$. However, we cannot have $k>16$ because then a vertex would occur at least three times in $x_1, \cdots, x_k$, providing a decomposition of G' into at least three cycles. Also, there cannot be a cycle in G' of length less than 8. Otherwise, removing this cycle would result in a cycle with at least 9 vertices. Hence one must occur twice, giving two cycles for a total of three.

Therefore, G' consists of two cycles of length 8; its form must be as shown in Fig. 3.3. Assume without loss of generality that v_1 is the vertex marked x. Then v_4 must be the vertex marked y since it is at distance 4 from v_1 along the cycle, and v_3 must be the vertex marked x since it is at distance 4 from v_4. However, this implies $v_1 = v_3$, a contradiction. Hence, the conjectured digraph cannot exist, and the theorem is proved. □

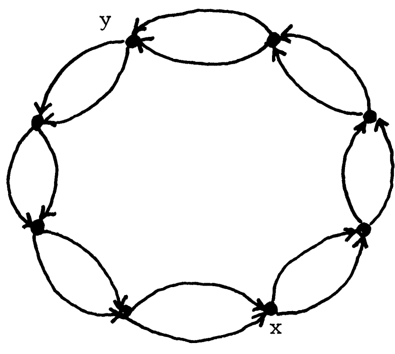


FIG. 3.3.

For $c \geqq 9$, no such theorem can be proved. Figure 3.4 provides a digraph for which no shortest cycle first decomposition exists.

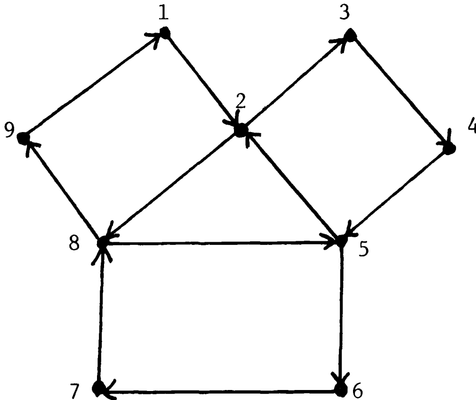


FIG. 3.4. *A digraph for which the shortest cycle first algorithm does not find a maximal decomposition.*

If we consider the average case, the analysis is especially simple since the associated digraph is nearly all 2-cycles. We consider the *all* 2*-cycles first algorithm*, which first removes all self-loops and 2-cycles then *arbitrarily* decomposes the digraph.

THEOREM 3.3. *The all* 2*-cycles first algorithm is asymptotically optimal in the average case.*

Proof. Immediate from Corollary 2.3 and Lemma 2.5. □

COROLLARY 3.1. *The shortest cycle first algorithm is asymptotically optimal in the average case.*

We now consider worst case performance. Rather than study the size of the decomposition, we study the number of swaps required in ordering a sequence based on this decomposition (i.e., $e(G)-k$ where k is the number of cycles in the decomposition).

THEOREM 3.4. *Given any sequence, let* COST *be the number of swaps required using the all* 2*-cycles first algorithm and let* $\mathrm{COST_{OPT}}$ *be the optimal number of swaps, then* $\mathrm{COST}/\mathrm{COST_{OPT}} \leqq \frac{3}{2}$. *Further,* $\frac{3}{2}$ *is the smallest possible constant bound.*

Proof. Let S and T be the number of self-loops and 2-cycles in the digraph. The all 2-cycles first algorithm results in a decomposition of at least $S+T$ cycles. Hence $\mathrm{COST} \leqq n-S-T$. Using Corollary 2.3, there is a maximal decomposition with $S+T+U$ cycles, where U is the number of cycles of length greater than two. Clearly, $U \leqq (n-S-T)/3$ and $\mathrm{COST_{OPT}} = n-S-T-U \geqq n-S-T-(n-S-T)/3 = \frac{2}{3}[n-S-T]$. Hence $\mathrm{COST}/\mathrm{COST_{OPT}} \leqq \frac{3}{2}$.

To prove this bound is tight, consider the eulerian digraph in Fig. 3.5, an n-cycle where each edge of the n-cycle is the base of a triangle. The digraph has $3n$ edges, and the optimal decomposition is clearly the n triangles. However, the all 2-cycles first algorithm might decompose the digraph into two cycles, the n-cycle and the cycle consisting of the remaining $2n$ edges. Hence $\mathrm{COST}/\mathrm{COST_{OPT}} = (3n-2)/(3n-n)$, which approaches $\frac{3}{2}$ as $n \to \infty$. □

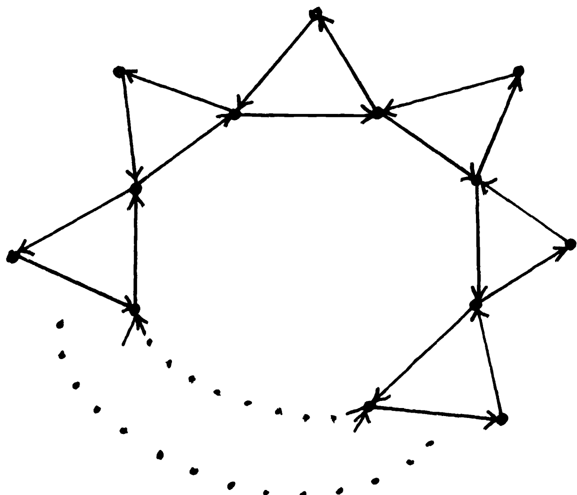


FIG. 3.5. *A worst case digraph for all 2-cycles first algorithm.*

This bound does not appear to be tight for the shortest cycle first algorithm. The next theorem gives the worst example found so far, which would require a bound of $\frac{5}{4}$.

THEOREM 3.5. *Let* $\mathrm{COST_{SCF}}$ *be the number of swaps using the shortest cycle first algorithm. Then for any* $n \geqq 1$ *there is an eulerian digraph* (*and hence a sequence*) *where* $\mathrm{COST_{SCF}}/\mathrm{COST_{OPT}} = (5n-3)/(4n-2)$.

Proof. Consider the eulerian digraph in Fig. 3.6, which has $6n-3$ edges. Its maximal decomposition is into the $2n-1$ triangles pointing down. However, a possible shortest cycle first decomposition is into the $n-1$ triangles pointing up, and the long cycle along the perimeter. In this case,

$$\frac{\mathrm{COST_{SCF}}}{\mathrm{COST_{OPT}}} = \frac{6n-3-n}{6n-3-(2n-1)} = \frac{5n-3}{4n-2}. \qquad \square$$

4. A constant space algorithm. In this section we describe a constant space algorithm to solve the Dutch National Flag problem. (A PASCAL implementation of the algorithm can be found in the Appendix.) The algorithm has an adjustable

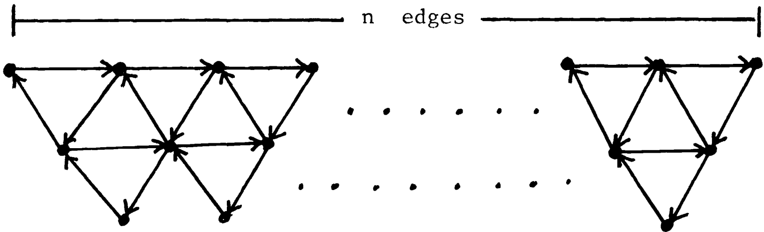


FIG. 3.6. *The worst case digraph for the shortest cycle first algorithm found so far.*

integer parameter *smax*, which may be any nonnegative integer. For any value of smax, the algorithm uses space proportional to smax (and *independent* of n). We show in the next section that a time/space tradeoff results; as smax $\to \infty$, the average number of swaps required $\to \frac{1}{3}n$, the optimum. Since we can get arbitrarily close to the lower bound by choosing smax sufficiently large, the algorithm is "asymptotically optimal" in some sense.

The algorithm has two phases. In the first, three pointers, r, w, and b are used to delimit regions consisting solely of, respectively, red, white, and blue pebbles according to the invariant:

"($1 \leqq i < r \to$ buck (i) = red) and
($n/3 \leqq i < w \to$ buck (i) = white) and
($2n/3 \leqq i < b \to$ buck (i) = blue) and
buck (r) $\neq$ red and buck (w) $\neq$ white and buck (b) $\neq$ blue."

Initially, $r = 1$, $w = n/3$, and $b = 2n/3$. Then r is successively incremented until buck (r) $\neq$ red. (We refer to this operation as "*advancing* r".) w and b are advanced in a similar manner. The algorithm then iterates, advancing the pointers and performing swaps when necessary to preserve the invariant. The first phase ends when $r = n/3$ or $w = 2n/3$ or $b = n+1$. The second phase merges the unexamined regions (from r to $n/3-1$, w to $2n/3-1$ and b to n) into the red, white, and blue regions, and possibly shifts the white region left or right. By the analysis in Theorem 2.3, the average size of the unexamined regions and the average distance the white region must be shifted is $o(n)$. Thus, any algorithm (such as a straightforward extension of Dijkstra's [1] algorithm or Meyer's algorithm [2]) can be used in the second phase without affecting the asymptotic performance of the algorithm. Hence, for simplicity, we examine only the first phase.

Two important ideas are used to reduce the number of swaps. First, w is initially $n/3$, not n as in the algorithms of Dijkstra and Meyer. Hence the white region will be from $n/3$ to w instead of from w to $b-1$. By Theorem 2.3, the final positions of the white pebbles will, with probability approaching one, be approximately from $n/3$ to $2n/3$. Hence white pebbles moved into the white region have, most likely, come to their final resting place.

Before discussing the second idea, we describe the only two situations in which the algorithm swaps pebbles. If buck (r) = white and buck (w) = red, then swap (r, w) is called a "single swap" (and similarly for the other two pairs of pointers). On the other hand, if buck (r) = white, buck (w) = blue, and buck (b) = red, then the sequence swap (r, b); swap (w, b) is called a "double swap" (and similarly if buck (r) = blue, buck (w) = red and buck (b) = white). The second idea then is to avoid double swaps as much as possible, preferring a single swap, which puts two pebbles in place at the cost of one swap, to a double swap, which puts three pebbles in place at the cost of two swaps. Clearly, this is motivated by the idea of finding a maximal decomposition.

The general strategy is as follows: A single swap is done on each interation, if one is possible. Otherwise, a "scout" is advanced from r in an attempt to find a pebble that will permit a single swap. (This scout is referred to as the "lead scout".) If such a pebble is found, a single swap with the lead scout will be performed on the next iteration. Otherwise, each successive iteration will advance a scout from the lead scout (with this new scout becoming the lead scout), until either a single swap becomes possible, or the maximum number of scouts has been reached and there is no alternative to doing a double swap. Now, instead of preserving the relation "$(1 \leqq i < r) \rightarrow$ buck $(i) =$ red" we preserve "$((1 \leqq i < ls)$ and i is not a scout$) \rightarrow$ buck $(i) =$ red)" where ls is the lead scout. Using the scouts in this manner avoids multiple calls to buck, as we only need to store the color of each scout (requiring only constant space) to know the color of all the pebbles in the region $1 \cdots ls$.

We now informally motivate the actual invariant. The following terminology is used: "smax" is the maximum number of scouts, and "s" is the number of currently active scouts. The scouts are kept in an array "scout", and "scout$[s]$" is the lead scout. It is convenient to store the value of "r" in scout [0]. Finally, *rcolor*, *scolor*, *wcolor*, and *bcolor* record the color of the pebble pointed to by, respectively, r, the lead scout, w and b.

Two other terms can be added to the previously stated invariant which simplify the program. The first is "$0 \leqq i < s \rightarrow$ buck (scout $[i]$) $=$ buck (r)". Basically, the reason this relation is preserved is as follows: When a single swap is not possible, we repeatedly advance scouts. If the lead scout has the same color as r, a single swap is still not possible, and another scout is advanced from this scout. However, once a different color is found, a swap is possible, and we immediately discontinue advancing scouts and swap the lead scout. Thus, all the scouts, except perhaps the lead scout, have the same color as r. This means we only need to store the color of r and the lead scout to know the color of all the scouts. Also, we only need to consider r and the lead scout in determining whether a single swap is possible.

Another term which is added to the invariant is:

"((rcolor = white and scolor = blue) $\rightarrow$ (wcolor = blue and bcolor = red))

and

((rcolor = blue and scolor = white) $\rightarrow$ (wcolor = red and bcolor = white))"

This is another consequence of advancing a scout *only* when a single swap is not possible and of swapping the lead scout immediately when a swap becomes possible. We will see below how this simplifies the procedure for performing single swaps.

The preceeding discussion gives the general strategy of the algorithm. To aid in understanding the program a brief, informal description of each procedure follows. This is not to imply the program cannot be formally verified; the reader can verify the program from the pre- and post-conditions given for each procedure.

Function swap possible returns true if and only if a single swap is possible. All four of r, the lead scout, w and b are considered in this determination. Procedure advance advances a pointer over pebbles of a given color and returns the color of the first pebble encountered which is not of that color.

Procedure single_swap performs a single swap, given that one is possible. Since it could involve either r or the lead scout, a test is made at the beginning of the procedure to determine which, if either, is involved. The lead scout is used unless it has already been swapped and hence scolor = red. (Note that if $s \neq 0$ and scolor = rcolor, either r or the lead scout could be used. However, the algorithm is simpler if

we choose to use the lead scout in this case. If $s=0$ they are identical, so it makes no difference. If scolor $\neq$ red and scolor $\neq$ rcolor, the last two terms in the invariant state that a swap is possible with the lead scout.) The procedure then determines which pair of pebbles is to be swapped, swaps them, and advances the appropriate pointers. Procedure double-swap is similar.

Advance_r advances the r pointer and the scouts immediately after a red pebble has been swapped. Note that it does not advance the lead scout; this is done only in advance_scout. Four cases, based on whether or not scolor = red and whether or not $s=0$, are conceivable. However, one case (scolor = red and $s=0$) is impossible, as it would imply buck (r) = red. This fact is stated in the third term of the precondition. If scolor $\neq$ red and $s \neq 0$, the lead scout was swapped and r was unaffected. All that is needed is to record this fact by setting scolor := red. In the other two cases, we know r was swapped: Either we have scolor = red and $s \neq 0$, and since scolor = red single-swap chose to use r, or we have scolor $\neq$ red and $s=0$ and single-swap chose to use the lead scout which, since $s=0$, was the same as r. Since buck (r) = red, r provides no useful information. If $s \neq 0$, each scout (including r, which is scout [0]) can be "advanced" by setting scout $[i]$:= scout $[i+1]$ for $i=0, \cdots, s-1$. Scout $[s]$ is no longer needed and is discarded by setting $s := s-1$. If s is now zero, r must be advanced to the next nonred pebble by calling advance.

Procedure advance scout advances a scout from the lead scout. If scolor = red, the lead scout itself can be advanced, since buck (scout $[s]$) = red, and hence scout $[s]$ contains no useful information. Otherwise a new scout is advanced.

5. Analysis of the algorithm. The algorithm is analyzed in the following theorem.

THEOREM 5.1. *The average number of swaps required by the algorithm of* § 4 *is* $[(6\,smax+10)/(18\,smax+27)]n+o(n)$ *for any* $smax \geqq 0$.

Proof. We consider only smax > 0. For smax = 0 the Markov chain we will construct degenerates to two states. With the techniques used below, it is easily shown the algorithm will require $\frac{10}{27}n+o(n)$ swaps, and hence the theorem is true in this case.

DEFINITION. The predicate hascolors (x, y, z) is defined to be true if and only if (rcolor $= x$) and (wcolor $= y$) and (bcolor $= z$).

To model the algorithm by a Markov chain, we examine the program variables each time control reaches the top of the while loop in the main program. The values of the variables determine the current state (see Fig. 5.1). We briefly discuss the motivation for these definitions. To predict the behavior of the algorithm, we must know the values of rcolor, wcolor and bcolor. (Note that for each triple of values with rcolor = white, there is a corresponding "equivalent" triple with rcolor = blue. Thus the eight possible triples result in only four types of states.) In addition, if hascolors (white, blue, red) or hascolors (blue, red, white), we must know whether the lead scout can be swapped (i.e., scolor $\neq$ red and scolor $\neq$ rcolor) since this determines whether a swap is possible. Finally, we need l defined by

$$l=\begin{cases} s & \text{if scolor} \neq \text{red}, \\ s-1 & \text{if scolor} = \text{red}. \end{cases}$$

(Note that for analysis purposes s scouts when the lead scout has been swapped (i.e., scolor = red) is equivalent to $s-1$ scouts when the lead scout has not been swapped. In both cases, there will be s scouts after a call to Advance r.) As a simplification, we note that all states satisfying $l=0$ and swap-possible are equivalent and these are merged to form only one state.

state	*conditions*
A	$l=0$ and swap_possible
B_i $0 \leqq i \leqq \text{smax}$	(hascolors(white,blue,red) or hascolors(blue,red,white)) and $l=i$ and (scolor = rcolor or scolor = red)
C_i $1 \leqq i \leqq \text{smax}$	(hascolors(white,blue,red) or hascolors(blue,red,white)) and $l=i$ and (scolor $\neq$ rcolor and scolor $\neq$ red)
D_i $1 \leqq i \leqq \text{smax}-1$	(hascolors(white,blue,white) or hascolors(blue,blue,white)) and $l=i$
E_i $1 \leqq i \leqq \text{smax}-1$	(hascolors(white,red,red) or hascolors(blue,red,red)) and $l=i$
F_i $1 \leqq i \leqq \text{smax}-1$	(hascolors(white,red,white) or hascolors(blue,blue,red)) and $l=i$

FIG. 5.1. *The states of the Markov chain.*

The state transition probabilities (shown in Fig. 5.2) can be calculated from several simple observations. In states D_i, E_i, and F_i, swap_possible is true, and a single swap is done. If w is swapped, wcolor becomes *arbitrary* (by this we mean wcolor = red with probability $\frac{1}{2}$ and wcolor = blue with probability $\frac{1}{2}$). Similarly, if b is swapped, bcolor becomes arbitrary (i.e., equal to red or white, either with probability $\frac{1}{2}$). If r or the lead scout is swapped (it makes no difference which for purposes of analysis), rcolor is unchanged and l is decremented. A single swap is also performed in state A, but here the colors of both the pointers swapped (including r) become arbitrary.

In state B_i, $0 \leqq i \leqq \text{smax}-1$, swap-possible is false, and advance_scout is called. With probability $\frac{1}{2}$, it is successful (i.e., afterwards scolor $\neq$ rcolor and scolor $\neq$ red). A single swap is now possible, and a transition to state C_{i+1} results. Otherwise, a transition to B_{i+1} results. In state B_{smax}, double_swap is called, resulting in wcolor and bcolor becoming arbitrary (though rcolor remains the same), and l being decremented by one. Finally, in state C_i, $1 \leqq i \leqq \text{smax}$, we have scolor $\neq$ rcolor and scolor $\neq$ red. This results in the lead scout being swapped. Scolor becomes red, and hence l is decremented, and the color of the pointer with which the lead scout was swapped becomes arbitrary.

A system of equations can be constructed to determine the steady state probabilities (see Fig. 5.3). We use lower case letters to denote the probability of the state named by the same upper case letter. Solving it, we get:

$$a = \frac{5}{5\,\text{smax}+4},$$

$$b_0 = \cdots = b_{\text{smax}-1} = \frac{2}{5\,\text{smax}+4},$$

$$c_1 = \cdots = c_{\text{smax}} = d_1 = \cdots = d_{\text{smax}-1} = b_{\text{smax}} = \frac{1}{5\,\text{smax}+4},$$

$$e_1 = \cdots = e_{\text{smax}-1} = f_1 = \cdots = f_{\text{smax}-1} = \frac{1}{2(5\,\text{smax}+4)}.$$

From these probabilities, we determine the expected number of swaps. Define the following random variables: Let S be the total number of swaps performed by the algorithm, and let S_1 and S_2 be, respectively, the number of single and double swaps performed by Phase 1, and S_3 be the number of swaps required in Phase 2.

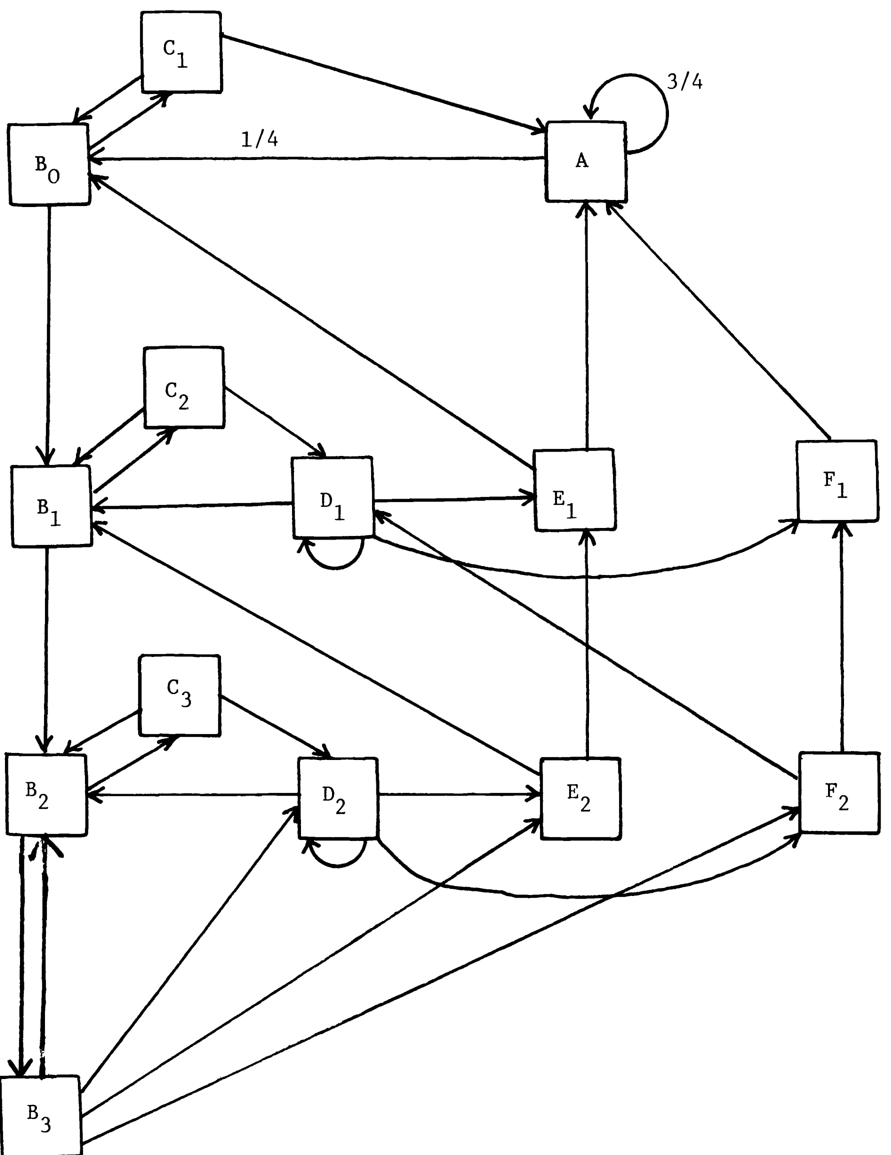


FIG. 5.2. *The Markov chain used to model the algorithm for* smax = 3. *The general form should be clear. Except where otherwise indicated, all transitions from the same state are equally likely.*

Let R be the sum of the lengths of the red, white and blue regions at the end of Phase 1, that is, $R = r + (w - n/3) + (b - 2n/3) - s$. Finally, let I be the number of pebbles that Phase 1 examined but did not swap. We clearly have:

$$S = S_1 + 2S_2 + S_3, \qquad R = 2S_1 + 3S_2 + I.$$

Using the notation $x \approx y$ to mean $x = y + o(n)$ and the relations $E(S_3) \approx 0$, $E(R) \approx n$, and $E(I) \approx n/3$ (derivable in a manner used in the proof of Theorem 2.3) gives

(1) and (2) $\qquad E(S) \approx E(S_1) + 2E(S_2), \qquad \tfrac{2}{3}n \approx 2E(S_1) + 3E(S_2).$

$$
\begin{aligned}
a &= \tfrac{3}{4}a + \tfrac{1}{2}c_1 + \tfrac{1}{2}e_1 + f_1 \\
b_0 &= \tfrac{1}{4}a + \tfrac{1}{2}c_1 + \tfrac{1}{2}e_1 \\
b_i &= \tfrac{1}{2}b_{i-1} + \tfrac{1}{2}c_{i+1} + \tfrac{1}{4}d_i + \tfrac{1}{2}e_{i+1}; && 1 \leqq i \leqq \text{smax}-2 \\
b_{\text{smax}-1} &= \tfrac{1}{2}b_{\text{smax}-2} + \tfrac{1}{4}b_{\text{smax}} + \tfrac{1}{2}c_{\text{smax}} + \tfrac{1}{4}d_{\text{smax}-1}; \\
b_{\text{smax}} &= \tfrac{1}{2}b_{\text{smax}-1}; \\
c_i &= \tfrac{1}{2}b_{i-1} && 1 \leqq i \leqq \text{smax} \\
d_i &= \tfrac{1}{2}c_{i+1} + \tfrac{1}{4}d_i + \tfrac{1}{2}f_{i+1} && 1 \leqq i \leqq \text{smax}-2 \\
d_{\text{smax}-1} &= \tfrac{1}{4}b_{\text{smax}} + \tfrac{1}{2}c_{\text{smax}} + \tfrac{1}{4}d_{\text{smax}-1} \\
e_i &= \tfrac{1}{2}d_i + \tfrac{1}{2}e_{i+1}; && 1 \leqq i \leqq \text{smax}-2 \\
e_{\text{smax}-1} &= \tfrac{1}{4}b_{\text{smax}} + \tfrac{1}{4}d_{\text{smax}-1} \\
f_i &= \tfrac{1}{4}d_i + \tfrac{1}{2}f_{i+1}; && 1 \leqq i \leqq \text{smax}-2 \\
f_{\text{smax}-1} &= \tfrac{1}{4}b_{\text{smax}} + \tfrac{1}{4}d_{\text{smax}-1}
\end{aligned}
$$

FIG. 5.3. *The steady-state equations.*

As $n \to \infty$, the fraction of time spent in each state approaches the steady-state probability. The probability the chain is in a state which does a single swap is $a + \sum_{i=1}^{\text{smax}} c_i + \sum_{i=1}^{\text{smax}-1} (d_i + e_i + f_i) = (3\ \text{smax}+3)/(5\ \text{smax}+4)$. A double swap is done with probability $b_{\text{smax}} = 1/(5\ \text{smax}+4)$. Hence

$$E(S_1) \approx (3\ \text{smax}+3)E(S_2). \tag{3}$$

Substituting (3) into (2) gives

$$E(S_2) \approx \frac{2n}{18\ \text{smax}+27}$$

and

$$E(S_1) \approx \frac{6\ \text{smax}+6}{18\ \text{smax}+27}\, n.$$

Therefore

$$E(S) \approx E(S_1) + 2E(S_2) \approx \frac{6\ \text{smax}+10}{18\ \text{smax}+27}\, n. \qquad \square$$

6. A lower bound for algorithms using Dijkstra's invariant. In this section we derive a lower bound of $\frac{5}{9}n$ swaps for any algorithm which uses Dijkstra's invariant. The significance of this result is discussed in the introduction. By the phrase "using Dijkstra's invariant", we mean that at any time, the sequence of pebbles can be divided into four regions:

$1 \leqq i < r \to$ pebble i is red,
$r \leqq i \leqq w \to$ pebble i has *not* been examined,
$w < i \leqq b \to$ pebble i is white,
$b < i \leqq n \to$ pebble i is blue.

Let us assume the algorithm has been running for a while and that $3x$ pebbles have been examined. Obviously, by the law of large numbers, each of the red, white and blue regions will be of size $x + o(x)$. As a good approximation, we assume each is of size x. Now suppose the algorithm is allowed to execute and examine $3\Delta x$ pebbles from the unexamined region. Of course, we see approximately Δx of each color. How many pebbles are out of place and will have to be moved? (see Fig. 6.1). The final red region initially contained x red pebbles and Δx unexamined pebbles. The final white region initially contained $x - \Delta x$ white pebbles and $2\Delta x$ unexamined pebbles. The final blue region initially contained x blue pebbles and Δx white pebbles. If we

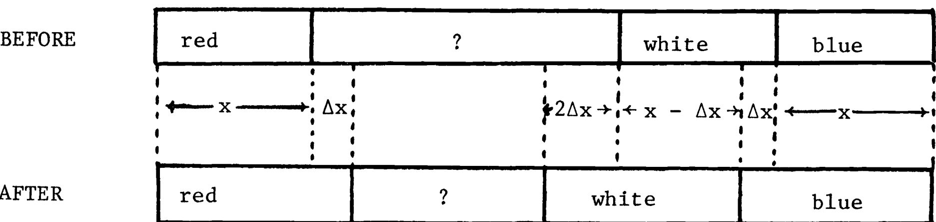


FIG. 6.1. *The arrangement of pebbles before and after the partial execution of any algorithm using Dijkstra's invariant.*

assume $\frac{1}{3}$ of the unexamined pebbles are of each color, we can construct an eulerian digraph which describes the changes required to get from the first to the second arrangement (see Fig. 6.2). By Theorem 3.1, the maximal decomposition is into $3x$ 1-cycles, Δx 2-cycles ($\frac{1}{3}\Delta x$ from vertex "R" to "W" and $\frac{2}{3}\Delta x$ from "W" to "B") and $\frac{1}{3}\Delta x$ 3-cycles. By Theorem 2.2, the number of swaps required is $e(G)-m(G)=\frac{5}{3}\Delta x$. Thus $\frac{5}{3}\Delta x$ swaps must be used to put $3\Delta x$ pebbles into place. Clearly by choosing Δx sufficiently small and repeatedly using this argument, we can prove that to put n pebbles into place must require $\frac{5}{9}n$ swaps.

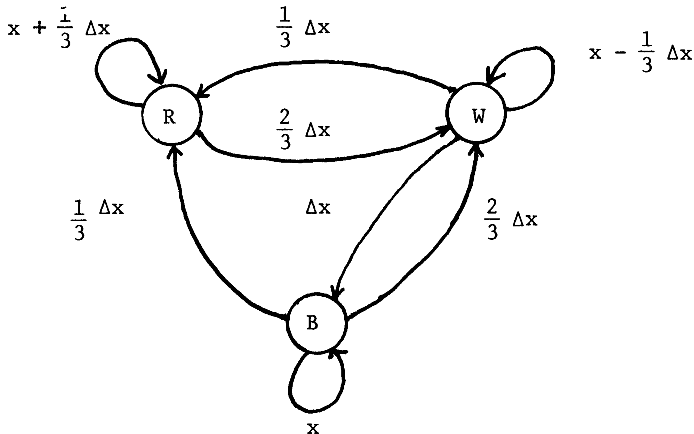


FIG. 6.2. *The eulerian digraph corresponding to the execution shown in Fig. 6.1. The number beside each edge gives the number of edges between the two vertices.*

Appendix.

```
const SMAX = <maximum number of scouts; smax >= 0.>;
      N = <size of the sequence to be sorted>;
      NPLUS1 = N + 1;
type  colortype = (NONE,RED,WHITE,BLUE);
      indextype = 0 · · NPLUS1;

{ We use the following predicates:
  "I" is defined to be
      ((0 <= i < scout[s] and (0 <=j <= s -> i <> scout[j])) ->
          buck(i) = RED) and
      ((N div 3 <= i < w) -> buck(i) = WHITE) and
      ((2*N div 3 <= i < b) -> buck (i) = BLUE) and
      rcolor <> RED and wcolor <> WHITE and bcolor <> BLUE and
      ((0 < i < s) -> buck(scout[i]) = rcolor) and
      0 <= s <= SMAX and
```

```
        ((rcolor = WHITE and scolor = BLUE) ->
             (wcolor = BLUE and bcolor = RED)) and
        ((rcolor = BLUE and scolor = WHITE) ->
             (wcolor = RED and bcolor = WHITE)).

"J" is defined to be
        rcolor = buck(scout[0]) and wcolor = buck(w) and
        bcolor = buck(b) and scolor = buck(scout[s])
        "I and J" is the loop invariant of the main program. }

procedure dnf;
var     w, b : indextype;
        scout : array[0 · · SMAX] of indextype;
        scolor,rcolor,wcolor,bcolor : colortype;
        s : 0 · · SMAX;
        limitr,limitw,limitb : indextype;

{ ptr = p0}
procedure advance (var ptr : indextype; skipcolor : colortype;
                          var finalcolor : colortype; limit : indextype);
var x : colortype;
begin { advance }
repeat
         ptr := ptr + 1;
         if ptr < limit then x := buck(ptr)
                        else  x := NONE;
until x <> skipcolor;
finalcolor := x;
end;  { advance }
{ ((p0 + 1 <= i <= ptr - 1) -> buck(i) = skipcolor) and
  (buck(ptr) = finalcolor <> skipcolor) and
  (ptr >= limit <---> finalcolor = NONE) }

procedure initialize;
begin { initialize }
limitr := N div 3;
limitw := 2 * N div 3;
limitb := N+1;
s := 0;
scout[0] := 0;
advance(scout[0],RED,rcolor,limitr);
scolor := rcolor;
if limitr = 0 then w := 0
              else  w := limitr - 1;
advance(w,WHITE,wcolor,limitw);
if limitw = 0 then b := 0
               else  b := limitw - 1;
advance(b,BLUE,bcolor,limitb);
end;  { initialize }

function pointers_valid : boolean;
begin { pointers_valid }
pointers_valid := (scout[s] < limitr) and (w < limitw) and (b < limitb);
end;  { pointers_valid }

function swap_possible : boolean;
         function has_pair(r,w,b : colortype) : boolean;
         begin { has_pair }
         has_pair := ((r = WHITE) and (w = RED)) or
                     ((r = BLUE) and (b = RED)) or
                     ((w = BLUE) and (b = WHITE))
         end;   { has_pair }
```

```
begin { swap_possible }
swap possible := has_pair(rcolor,wcolor,bcolor) or
                 has_pair(scolor,wcolor,bcolor);
end;  { swap_possible }
{ I and buck(scout[s]) = RED and
  (scolor = red -> s <> 0) and
  (    ( scolor <> RED and s <> 0) -> buck(scout[0]) = rcolor) and
  (not ( scolor <> RED and s <> 0) -> buck(scout[0]) = RED) }
procedure advance_r;
var i : 0 . . SMAX;
begin { advance_r }
if (scolor <> RED) and (s <> 0) then scolor := RED
  else begin
      if s <> 0 then begin
            for i := 0 to s - 1 do scout[i] := scout[i + 1];
            s := s - 1;
      end;
      if s = 0 then begin
           advance(scout[0],RED,rcolor,limitr);
           scolor := rcolor;
      end;
  end;
end;  { advance_r }
{ I and rcolor = buck(scout[0]) and scolor = buck(scout[s]) }

{ not swap_possible and I and J and (s < SMAX or scolor = RED) }
procedure advance_scout;
begin { advance_scout }
if scolor <> RED then begin
      s := s + 1;
      scout[s] := scout[s - 1];
end;
advance(scout[s],RED,scolor,limitr);
end; { advance_scout }
{ I and J }


{ I and J and swap_possible }
procedure single_swap;
var t : 0 . . SMAX;
    tcolor : colortype;
begin { single_swap }
if scolor = RED then begin
        t := 0;
        tcolor := rcolor;
end
else begin
        t := s;
        tcolor := scolor;
end;
if (tcolor = WHITE) and (wcolor = RED) then begin
        swap(scout[t], w);
        advance_r;
        advance(w, WHITE,wcolor,limitw);
end
else if (tcolor = BLUE) and (bcolor = RED) then begin
        swap(scout[t], b);
        advance_r;
        advance(b,BLUE,bcolor,limitb);
end
```

```
else { (wcolor = BLUE) and (bcolor = WHITE) } begin
        swap(w, b)
        advance(w,WHITE,wcolor,limitw);
        advance(b,BLUE,bcolor,limitb);
end;
end;
{ I and J }

{ I and J and not swap_possible and s = SMAX and scolor <> RED }
procedure double_swap;
begin { double_swap }
if scolor = WHITE then swap(scout[SMAX], b)
                  else swap(scout[SMAX], w);
swap(w, b);
advance_r;
advance(w,WHITE,wcolor,limitw);
advance(b,BLUE,bcolor,limitb);
end;  { double_swap }
{ I and J }

begin { dnf }
initialize;
while pointers_valid do begin
        if swap_possible then single_swap
    else if (s < SMAX) or (scolor = RED) then advance_scout
    else double_swap;
end;
end;  { dnf }
```

Acknowledgment. I am grateful to Jay Missa for his helpful discussions on this problem, and to both referees for exceptionally careful and helpful reviews.

REFERENCES

[1] E. W. DIJKSTRA, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[2] S. J. MEYER, *A failure of structured programming*, Zilog Corp., Software Dept. Technical Report No. 5, Cupertino, CA, 1978.

[3] C. L. MCMASTER, *An analysis of algorithms for the Dutch National Flag problem*, Comm. ACM, 21 (1978), pp. 842–846.

[4] C. L. LIU, *Introduction to Combinatorial Mathematics*, McGraw-Hill, New York, 1968, pp. 176.

SIAM J. COMPUT.
Vol. 11, No. 2, May 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1102-0005 $01.00/0

SOME PROPERTIES OF DISJOINT SUMS OF TENSORS RELATED TO MATRIX MULTIPLICATION*

FRANCESCO ROMANI†

Abstract. Let t be a disjoint sum of tensors associated to matrix multiplication. The rank of the tensorial powers of t is bounded by an expression involving the elements of t and an exponent for matrix multiplication. This relation leads to a trascendental equation defining a new exponent for matrix multiplication.

Key words. computational complexity, matrix multiplication, tensor rank, exponent

1. Introduction. In 1969 V. Strassen showed that the complexity of matrix multiplication is lower than $O(n^3)$ operations [7]. Then the question arose of determining the intrinsic complexity of the problem. A lower bound to the number of operations is n^2, but for ten years the best known upper bound was $O(n^{2.81})$. Recently the use of new techniques has considerably reduced this upper bound. The methods of trilinear uniting, aggregating and canceling [4], [5], the introduction of approximate algorithms (also called the field extension method) [1], [2], [3], and the powerful theory of partial matrix multiplication [6], have led to several improvements in the upper bound, down to the exponent 2.5218.

In this paper we start from the final argument of A. Schönhage in [9] to derive some theorems on the rank of the sth tensorial power of disjoint sums of tensors. The application of these theorems results in an exponent of 2.5166 for matrix multiplication.

2. Notation and preliminaries. The reader is assumed to be familiar with the theory of matrix multiplication algorithms. For a survey see [6], the notation of which is followed here with some minor changes.

Let mam (n) be the total number of arithmetic operations needed to compute the product of two square real matrices of order n. The quantity $\inf\{x: \text{mam}(n) = O(n^x)\}$ is denoted by w.

To any matrix multiplication problem is associated a 3-dimensional array called a *tensor*. The length of the minimal decomposition of a tensor t into triads is called the *rank of t* and denoted rk (t). The number of nonscalar multiplications needed to compute any matrix product in the bilinear noncommutative model is exactly the rank of the associated tensor.

Given two tensors t' and t'', $t' \otimes t''$ denotes their tensor product and $t' \oplus t''$ their disjoint sum. For our purposes the two operations are assumed to commute.

We use the notation t^s for $\otimes_{i=1}^{s} t$ and $k * t$ for $\oplus_{i=1}^{k} t$; the tensor associated to the product of a matrix $m \times n$ with a matrix $n \times p$ is denoted $\langle m, n, p\rangle$.

The following properties hold:

$\langle m, n, p\rangle \otimes \langle m', n', p'\rangle = \langle mm', nn', pp'\rangle$.

$\langle m, n, p\rangle^s = \langle m^s, n^s, p^s\rangle$.

rk $(\langle m, n, p\rangle)$ is symmetrical in m, n, p.

rk $(\langle m, n, p\rangle) \leqq r$ implies $w \leqq 3 \log r/\log mnp$.

$\langle m, n, p\rangle \oplus \langle m', n', p'\rangle$ is associated to two distinct multiplications of matrices, $m \times n$ by $n \times p$ and $m' \times n'$ by $n' \times p'$.

rk$(\langle m, n, p\rangle \oplus \langle m', n', p'\rangle) \leqq$ rk $(\langle m, n, p\rangle)$ + rk $(\langle m', n', p'\rangle)$.

---

* Received by the editors March 21, 1980, and in revised form February 1, 1981.

† Istituto di Elaborazione dell'Informazione, via S. Maria 46, 56100 Pisa, Italy.

By an *approximate decomposition of order h and length r for a tensor t*, we mean a representation

$$T(l)=\sum_{j=1}^{r}\mathbf{x}_j(l)\otimes\mathbf{y}_j(l)\otimes\mathbf{z}_j(l)=l^h t+O(l^{h+1}),$$

where $\mathbf{x}_j$, $\mathbf{y}_j$, $\mathbf{z}_j$, are vectors of polynomials in l. $r_h(t)$ is the minimal length of a decomposition of order h and $r_0(t)=\operatorname{rk}(t)$. $\min_h r_h(t)=\underline{\operatorname{rk}}(t)$ is called the *border rank of t.*

The following properties hold:

$\underline{rk}(t)\leqq \operatorname{rk}(t)$.

$r_{h'+h''}(t'\otimes t'')\leqq r_{h'}(t')+r_{h''}(t'')$.

$r_{sh}(t^s)\leqq(r_h(t))^s$.

$\operatorname{rk}(t)\leqq(1+2h)r_h(t)$ (see [2]).

$\underline{\operatorname{rk}}(\langle m,n,p\rangle)\leqq r$ implies $w\leqq 3\log r/\log mnp$.

3. The rank of powers of disjoint sums of tensors. We begin with two simple lemmas.

LEMMA 1. *Let $b>w$. Then there exists a constant c' such that for any n*

$$\operatorname{rk}(\langle n,n,n\rangle)\leqq c'n^b.$$

Proof. There exist c_0 and n_0 such that for $n>n_0$ $\operatorname{rk}(\langle n,n,n\rangle)\leqq c_0n^b$. On the other hand for $n\leqq n_0$ $\operatorname{rk}(\langle n,n,n\rangle)\leqq n_0^3$ and $\operatorname{rk}(\langle n,n,n\rangle)\leqq n_0^3c_0n^b\leqq c'n^b$ for any n. □

LEMMA 2. *Let $b>w$, $m\leqq n\leqq p$. Then there exists a constant c such that*

$$\operatorname{rk}(\langle m,n,p\rangle)\leqq cm^{b-2}np.$$

Proof. Since

$$\operatorname{rk}(\langle m,n,p\rangle)\leqq\operatorname{rk}(\langle m,m,m\rangle\otimes\langle 1,\lceil n/m\rceil,\lceil p/m\rceil\rangle)\leqq c'm^b\lceil n/m\rceil\lceil p/m\rceil$$

and $\lceil n/m\rceil\geqq 1$, $\lceil p/m\rceil\geqq 1$, then $\operatorname{rk}(\langle m,n,p\rangle)\leqq 4c'm^b(n/m)(p/m)=cm^{b-2}np$. □

Now we can prove the central theorem of the paper.

THEOREM 1. *Let $t=\oplus_{i=1}^k\langle m_i,n_i,p_i\rangle$, $b>w$. Then there exists a constant c such that*

$$\operatorname{rk}(t^s)\leqq c(S+1)^k\left[\max\left(\sum_{i=1}^k m_i^{b-2}n_ip_i\right)\right]^S,$$

where the maximum is taken over the permutations of m, n, p.

Proof.

$$\operatorname{rk}(t^S)\leqq\operatorname{rk}\left[\bigoplus_{s_1+s_2+\cdots+s_k=S}\frac{S!}{s_1!s_2!\cdots s_k!}\langle\prod m_i^{s_i},\prod n_i^{s_i},\prod p_i^{s_i}\rangle\right]$$

$$\leqq\sum_{s_1+s_2+\cdots+s_k=S}\frac{S!}{s_1!s_2!\cdots s_k!}\operatorname{rk}(\langle\prod m_i^{s_i}\prod n_i^{s_i},\prod p_i^{s_i}\rangle).$$

Let $s_1,s_2,\cdots,s_k$ be the k-tuple for which the corresponding term in the above expansion is maximal, and assume that

$$\prod m_i^{s_i}\leqq\prod n_i^{s_i}\leqq\prod p_i^{S_i}. \tag{1}$$

Then

$$\operatorname{rk}(t^S)\leqq(S+1)^k\frac{S!}{s_1!s_2!\cdots s_k!}\operatorname{rk}(\langle\prod m_i^{s_i},\prod n_i^{s_i},\prod p_i^{s_i}\rangle)$$

$$\leqq c(S+1)^k\frac{S!}{s_1!s_2!\cdots s_k!}\prod_{i=1}^k m_i^{(b-2)s_i}n_i^{s_i}p_i^{s_i}.$$

Obviously, since one term of a sum is less than the whole expression, then

$$\operatorname{rk}(t^S) \leqq c(S+1)^k\left(\sum_{i=1}^k m_i^{b-2} n_i p_i\right)^S.$$

This formula holds under assumption (1), but it is not known for which permutation of (m, n, p) (1) is true, hence

$$\operatorname{rk}(t^S) \leqq c(S+1)^k\left[\max\left(\sum_{i=1}^k m_i^{b-2}\, n_i p_i, \sum_{i=1}^k m_i n_i^{b-2} p_i, \sum_{i=1}^k m_i n_i p_i^{b-2}\right)\right]^S. \qquad \square$$

COROLLARY 1. *If the set of disjoint tensors in t is symmetrical in m, n, p, the three expressions are equal and*

$$\operatorname{rk}(t^S) \leqq c(S+1)^k\left(\sum_{i=1}^k m_i^{b-2} n_i p_i\right)^S.$$

Schönhage [6] proved the following theorem.

THEOREM 2. *Let* $t = \bigoplus_{i=1}^k t_i$, $t_i = \langle m_i, n_i, p_i\rangle$, $f_i = m_i n_i p_i$, $\underline{\operatorname{rk}}(t) \leqq r$. *Then* $w \leqq 3x$, *where x is the solution of the equation* $\sum_{i=1}^k f_i^x = r$.

Theorem 1 can be considered to be a weak converse of Theorem 2. For example, let $t = \bigoplus_{i=1}^k \langle m_i, m_i, m_i\rangle$. Then from Theorem 1 it follows that

$$\underline{\operatorname{rk}}(t) \leqq r = \sum_{i=1}^k m_i^{3x} \quad \text{implies} \quad w \leqq 3x,$$

and from Theorem 2

$$w \leqq 3x \quad \text{implies} \quad \operatorname{rk}(t^s) \leqq c(s+1)^k\left(\sum_{i=1}^k m_i^{3x}\right)^s.$$

COROLLARY 2. *Let*

$$t_1 = d * \langle 1,1,1\rangle, \quad t_2 = e * \langle 1,1,q\rangle, \quad t_2' = e * \langle 1,q,1\rangle, \quad t_2'' = e * \langle q,1,1\rangle,$$
$$t = (t_1 \oplus t_2) \otimes (t_1 \oplus t_2') \otimes (t_1 \oplus t_2''), \qquad b > w.$$

Then

$$\operatorname{rk}(t^s) \leqq c(s+1)^8[(d+eq)^2(d+eq^{b-2})]^s.$$

Proof. The set of elements of t is symmetrical; in fact,

$$t = d^3 * \langle 1,1,1\rangle \oplus d^2 e * (\langle 1,1,q\rangle \oplus \langle 1,q,1\rangle \oplus \langle q,1,1\rangle)$$
$$\oplus\, de^2 * (\langle 1,q,q\rangle \oplus \langle q,1,q\rangle \oplus \langle q,q,1\rangle) \oplus e^3 * \langle q,q,q\rangle.$$

Applying Corollary 1 we get

$$\operatorname{rk}(t^s) \leqq c(s+1)^8[d^3 + d^2 e(2q + q^{b-2}) + de^2(2qq^{b-2} + q^2) + e^3(q^2 q^{b-2})]^s$$
$$= c(s+1)^8[(d+eq)^2(d+eq^{b-2})]^s. \qquad \square$$

4. Application to matrix multiplication. A sum of r triads T can be viewed as the homomorphic image of the tensor $t = r * \langle 1,1,1\rangle$. In such a case we write $T \to t$. Analogously we have

$$\sum_{j=1}^q \mathbf{x} \otimes \mathbf{y}_j \otimes \mathbf{z}_j \to \langle 1,1,q\rangle.$$

Obviously $T \to t$ implies $\mathrm{rk}\,(T) \leqq \mathrm{rk}\,(t)$; moreover, $T' \to t'$, $T'' \to t''$ implies $T' \otimes T'' \to t' \otimes t''$.

The same considerations can be made for sums of triads depending on a variable l. Thus the following theorems can be stated.

THEOREM 3.

$$T'(l) = l^{h'} t_1' + O(l^{h'+1}) \to t_2', \qquad T''(l) = l^{h''} t_1'' + O(l^{h''+1}) \to t_2''$$

imply

$$\underline{\mathrm{rk}}\,(t_1' \otimes t_1'') \leqq \mathrm{rk}\,(t_2' \otimes t_2'').$$

Proof. From the definitions of $\to$ and of border rank it follows that

$$\underline{\mathrm{rk}}\,(t_1' \otimes t_1'') \leqq r_{h'+h''}(t_1' \otimes t_1'') \leqq \mathrm{rk}\,(T'(l) \otimes T''(l)) \leqq \mathrm{rk}\,(t_2' \otimes t_2''). \qquad \square$$

THEOREM 4. *Let* $t = \oplus_{i=1}^{z} \bar{t}_i$, $\bar{t}_i = \langle m_i, n_i, p_i \rangle$, $m_i n_i p_i = f_i$, *and let*

$$T(l) = l^h t + O(1^{h+1}) \to d * \langle 1, 1, 1 \rangle \oplus e * \langle 1, 1, q \rangle.$$

Then the solution of the equation

$$\left(\sum_{i=1}^{z} f_i^x \right)^3 = (d+eq)^2 (d+eq^{3x-2})$$

satisfies $w \leqq 3x$.

Proof. Let $t_1 = d * \langle 1, 1, 1 \rangle$, $t_2 = e * \langle 1, 1, q \rangle$, $t_2' = e * \langle 1, q, 1 \rangle$, $t_2'' = e * \langle q, 1, 1 \rangle$, and let t', t'' be the tensors obtained from t with the corresponding permutations. Then

$$T'(l) = l^{3h}(t \otimes t' \otimes t'') + O(l^{3h+1}) \to (t_1 \oplus t_2) \otimes (t_1 \oplus t_2') \otimes (t_1 \oplus t_2'') = t_3$$

and

$$T_s''(l) = l^{3sh}(t \otimes t' \otimes t'')^s + O(l^{3sh+1}) \to t_3^s.$$

Applying Corollary 2 we obtain

$$\mathrm{rk}\,[(t \otimes t' \otimes t'')^s] \leqq \mathrm{rk}\,(t_3^s) \leqq c(s+1)^8 [(d+eq)^2 (d+eq^{b(0)-2})]^s$$

for any s and $b(0) > w$.

Now $(t \otimes t' \otimes t'')^s$ has z^{3s} independent components. Their volumes are given by the terms of the expansion of $(\sum_{i=1}^{z} f_i)^{3s}$.

Then by Theorem 2 the solution of

$$\left(\sum_{i=1}^{z} f_i^x \right)^{3s} = c(s+1)^8 [(d+eq)^2 (d+eq^{b(0)-2})]^s \quad \text{satisfies} \quad w \leqq 3x.$$

The solution of this equation depends on s. Note that $x' = \inf \{x(s), s \in N\}$ is the solution of the equation

$$\left(\sum_{i=1}^{z} f_i^{x'} \right)^3 = (d+eq)^2 (d+eq^{b(0)-2}) \quad \text{and} \quad w \leqq x'.$$

It is easy to see that substituting the new value $b^{(1)} = 3x'$ for $b^{(0)}$ and iterating the process results in values which converge to the unique solution of

$$\left(\sum_{i=1}^{z} f_i^{b/3} \right)^3 = (d+eq)^2 (d+eq^{b-2}),$$

and any value of the sequence $\{b^{(0)}, b^{(1)}, \cdots\}$ is an upper bound for w. $\square$

COROLLARY 3. $w \leqq 2.516648 \cdots$.

Proof. Pan [4] presented a decomposition $T(l)$ for the tensor $t = \langle 1, k, 2n\rangle \oplus \langle n, 2, k\rangle \oplus \langle 2k, n, 1\rangle$ [5] and Schönhage noted that $T(l) \to [2(n+1)(k+1) * \langle 1, 1, 1\rangle \oplus (k+1) * \langle 1, 1, 2\rangle]$ [6]. Then, from Theorem 4, $w \leqq b$, where b is the solution of $27(2kn)^b = [2(n+2)(k+1)]^2[2(n+1)(k+1)+(k+1)2^{b-2}]$. In fact, the symmetrization of t yields 27 independent components of the same volume $(2kn)^3$.

The minimal value of b is attained for $n = 10$, $k = 5$; i.e.,

$$100^b = 144^2 \frac{(132+62^{b-2})}{27} \quad \text{gives } b = 2.516648 \cdots. \qquad \square$$

Acknowledgment. The same exponent has been independently derived by Dr. Don Coppersmith of IBM T. J. Watson Research Center, Yorktown, NY.

REFERENCES

[1] D. BINI, M. CAPOVANI, G. LOTTI AND F. ROMANI, *$o(n^{2.7799})$ complexity for $n \times n$ approximate matrix multiplication*, Inform. Processing Lett., 8 (1979), pp. 234–235.

[2] D. BINI, *Relations between exact and approximate bilinear algorithms applications*, Calcolo, 17 (1980), pp. 87–96.

[3] D. BINI, G. LOTTI AND F. ROMANI, *Approximate solutions for the bilinear form computational problem*, this Journal, 9 (1980), pp. 692–697.

[4] V. YA. PAN, *New fast algorithms for matrix operations*, this Journal, 9 (1980), pp. 321–341.

[5] ———, *New combinations of methods for the acceleration of matrix multplication*, Comp. and Maths. with Appl., 7 (1981), pp. 73–125.

[6] A. SCHÖNHAGE, *Partial and total matrix multiplication*, this Journal, 10 (1981), pp. 434–455.

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

SIAM J. COMPUT.
Vol. 11, No. 2, May 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1102-0006 $01.00/0

ON TRANSFORMING CONTROL STRUCTURES*

JOHN KEOHANE,† JOHN C. CHERNIAVSKY‡ AND PETER B. HENDERSON§

Abstract. Transformations of reducible flowcharts to REPEAT-EXIT or RE_n-charts and transformations of RE_n-charts to more structured forms are investigated. In particular, transformations within the hierarchy of flowcharts (D-charts, RE_1-charts, RE_2-charts, $\cdots$, RE_n-charts, $\cdots$) studied by Kosaraju [J. Comput. System Sci., 9 (1974), pp. 232–255] are presented. It is shown that the transformation of RE_n-charts to D-charts requires at most $\lceil \log_2 (n+1) \rceil$ auxiliary Boolean variables (flags), and that for each $n \geqq 1$ this bound is tight for at least one RE_n-chart. The existence of a hierarchy of flowcharts with respect to the number of flags needed for conversion into equivalent D-charts is also shown.

Key words. control structures, structured programming, reducible flowgraphs, data flow analysis, graph grammars

1. Introduction. The development of structured programming as a discipline has stimulated a great deal of research in the area of abstract control structures. Most research on abstract control structures has been conducted by comparing various classes of flowcharts or flowgraphs. The results have usually been a measurement of the relative powers of control structures with respect to various notions of equivalence [11].

In this paper, we examine the relationships among classes of flowcharts within a hierarchy studied by Kosaraju [10]. We first examine the relationship between reducible [8] flowcharts and REPEAT-EXIT flowcharts (RE_n-charts [10]). We then examine transformations for restructuring RE_n-charts by introducing auxiliary Boolean variables (flags) [4], [5]. Techniques for converting RE_n-charts to while-charts or D-charts (after Dijkstra) are presented. Furthermore, we show that one of these techniques results in the introduction of the minimal number of flags required for the conversion. Finally, we demonstrate the existence of a hierarchy of flow charts with respect to the number of flags required for their expression as D-charts.

The transformations presented in this paper are in no way intended to be applied to actual programs, in contrast to those in [3]. We feel that the application of such transformations would contradict the intent of structured programming. Our purpose is to examine the relative powers of various control structures in their limits and, thereby, provide the programming language designer with a set of objective facts to aid in his decision as to which control structures to include in his language.

2. Flowcharts and flowgraphs. We will use concepts from both the area of abstract control structures and that of data flow analysis. Most research on abstract control structures has been conducted using either flowcharts [5], [10] or flowchart schemas [12], and most research on data flow analysis has been conducted using flowgraphs [1], [8]. Flowcharts and flowgraphs are quite similar; both are instances of finite, labeled directed graphs.

* Received by the editors September 30, 1977, and in revised form March 18, 1981. This research was supported in part by the National Science Foundation under grant MCS 7702708.

† Department of Computer Science, State University of New York, Stony Brook, New York 11794. Current address: Department of Computer Science, University of Kentucky, Lexington, Kentucky 40506.

‡ Computer Science Section, National Science Foundation, Washington, DC 20550.

§ Department of Computer Science, State University of New York, Stony Brook, New York 11794.

DEFINITION. A *finite labeled directed graph* is a 5-tuple $G=(N, E, L, \theta_N, \theta_E)$, where:

(1) N is a finite set of objects called *nodes*.
(2) $E \subseteq N \times N$ is a set of objects called *edges*.
(3) L is a set of symbols called *labels*.
(4) θ_N is a partial node labeling function such that $\theta_N : N \to L$.
(5) θ_E is a partial edge labeling function such that $\theta_E : E \to L$.

Following Kosaraju [10], we assume the existence of two disjoint, infinite denumerable sets of symbols $A=\{f, g, \cdots\}$ and $P=\{p, q, \cdots\}$. An element of A is called a *basic action*, and an element of P is called a *basic predicate*.

DEFINITION. A *flowchart* F is a finite, labeled directed graph composed of:

(1) nodes labeled with basic actions and basic predicates;
(2) *auxiliary nodes*, which are
 (a) nodes labeled RPT, END and EXIT(i), where i is a positive integer,
 (b) unlabeled nodes called *merge nodes*;
(3) two distinguished nodes, one of which is labeled START and the other is labeled STOP;
(4) paths, such that for any node $n \in N$, there is a path in F from the START node to the STOP node which contains n.

The START node has in-degree zero, and the STOP node has out-degree zero. Nodes labeled with basic actions, auxiliary nodes and the START node have out-degree one. Nodes labeled with basic predicates have out-degree two. One of the edges from a node labeled with a basic predicate may be labeled T (true) and the other labeled F (false). All other edges are unlabeled.

The use of auxiliary nodes in flowcharts will be restricted and clarified in what follows. Nodes that are not auxiliary nodes are called *nonauxiliary nodes*. Edge labels will be disregarded whenever the labeling is either obvious or irrelevant. We will be concerned with the following classes of syntactically structured flowcharts.

DEFINITION. The class of D-*charts*, denoted **D**, is the class of flowcharts contained in the language of the following graph grammar:

(1) ⟨D-*chart*⟩ ::= (START) →⟨D⟩→ (STOP).

(2) →⟨D⟩→ ::= →⟨D⟩→⟨D⟩→.

(3) →⟨D⟩→ ::= (p) → ⟨D⟩ / → ⟨D⟩ → 0→, where $p \in P$.

(4) →⟨D⟩→ ::= →(p)→ with loop ⟨D⟩, where $p \in P$.

(5) →⟨D⟩→ ::= →(f)→, where $f \in A$.

(6) →⟨D⟩→ ::= → .

DEFINITION. For any integer $n \geqq 0$, the class of RE$_n$-*charts*, denoted **RE**$_n$, is the class of flowcharts in the language of the following graph grammar:

(1) $\langle \mathbf{RE}_n\text{-}chart \rangle ::=$ (START) $\rightarrow \langle \mathbf{RE}_n \rangle \rightarrow$ (STOP).

(2) $\rightarrow \langle \mathbf{RE}_n \rangle \rightarrow\ ::= \rightarrow \langle \mathbf{RE}_n \rangle \rightarrow \langle \mathbf{RE}_n \rangle \rightarrow$.

(3) $\rightarrow \langle \mathbf{RE}_n \rangle \rightarrow\ ::= \rightarrow$ (p) with branches $\rightarrow \langle \mathbf{RE}_n \rangle \rightarrow$ and $\rightarrow \langle \mathbf{RE}_n \rangle \rightarrow$ joining at $0 \rightarrow$, where $p \in P$.

(4) $\rightarrow \langle \mathbf{RE}_n \rangle \rightarrow\ ::= \rightarrow$ (RPT) $\rightarrow \langle \mathbf{RE}_n \rangle \rightarrow$ (END) $\rightarrow$.

(5) $\rightarrow \langle \mathbf{RE}_n \rangle \rightarrow\ ::= \rightarrow$ (f) $\rightarrow$, where $f \in A$.

(6) $\rightarrow \langle \mathbf{RE}_n \rangle \rightarrow\ := \rightarrow$ (EXIT(i)) $\rightarrow$, where $1 \leqq i \leqq n$.

(7) $\rightarrow \langle \mathbf{RE}_n \rangle \rightarrow\ ::= \rightarrow$.

The unlabeled nodes in production (3) of the two definitions above are merge nodes. The nodes labeled RPT, END and EXIT(i) in the productions above are auxiliary nodes whose semantics will soon be clarified. Note, however, that nodes labeled RPT and END form pairs, which we call RPT-END pairs. Each node in an $\langle \mathrm{RE}_n \rangle$ G is said to be *surrounded* by the RPT-END pair shown in $\rightarrow \mathrm{RPT} \rightarrow G \rightarrow \mathrm{END} \rightarrow$. The number of RPT-END pairs surrounding a node in an $\langle \mathrm{RE}_n \rangle$ or RE_n-chart G is its *depth of nesting* in G.

We use **RE** to denote $\bigcup_{n=0}^{\infty} \mathbf{RE}_n$. In what follows, the inclusion of auxiliary nodes other than merge nodes is restricted to flowcharts in **RE**, and the inclusion of merge nodes is restricted to $\mathbf{RE} \cup \mathbf{D}$.

The following definitions are necessary for the presentation of various notions of equivalence between classes of flowcharts.

DEFINITION. A *computation path* in a flowchart F is a finite sequence of nodes $P = n_1 n_2 \cdots n_k$ such that n_1 is the START node, n_k is the STOP node, and for $1 \leqq i \leqq k-1$, the following hold:

(1) If n_i is a nonauxiliary node, a merge node or a RPT node, then n_{i+1} is a successor of n_i in F.
(2) If n_i is an END node, then n_{i+1} is the RPT node with which it forms a RPT-END pair.
(3) If n_i is labeled EXIT(j) and its depth of nesting in F is at least j, then n_{i+1} is the successor of the END node of the jth innermost RPT-END pair surrounding n_i.
(4) If n_i is labeled EXIT(j) and its depth of nesting in F is less than j, then n_{i+1} is the STOP node.

Note the convention adopted in (4) above.

DEFINITION. The *execution sequence* resulting from a computation path $P = n_1 n_2 \cdots n_k$ in a flowchart is the string that is the sequence of basic actions and predicates along the computation path. For $1 \leqq i \leqq k-1$, the instance of a basic predicate that is the label of a node n_i is subscripted with the label of the edge (n_i, n_{i+1}).

Note that the primary difference between computation paths and execution sequences is that auxiliary nodes are included in computation paths and not represented in execution sequences. Consider the RE_2-chart F shown in Fig. 1. Here the node set $N = \{n_1, n_2, n_3, \cdots, n_{13}\}$, the basic action set $A = \{f, g, h\}$, the basic predicate set $P = \{p, q\}$, and edge label set for basic predicates $= \{T, F\}$. The set of computation paths may be represented by the regular expression $n_1(n_2n_3n_4n_5n_6n_7n_8n_9)^* n_2n_3n_4(n_5n_{10}n_{11} + n_{12})n_{13}$, and its set of execution sequences may be represented by the regular expression $(fp_Fq_Fg)^* f(p_Fq_Th + p_T)$.

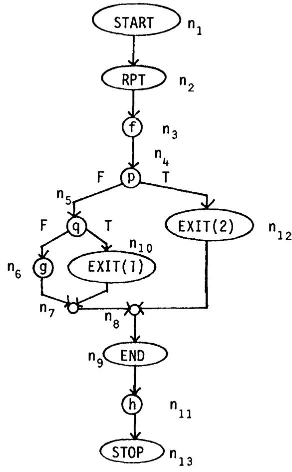


FIG. 1

The following definitions of equivalence between flowcharts and classes of flowcharts are two of those compiled by Ledgard and Marcotty [11].

DEFINITION. Two flowcharts F and G are said to be *pathwise equivalent*, denoted $F \equiv_{\text{pw}} G$, provided that their sets of execution sequences are identical.

DEFINITION. Two flowcharts F and G are said to be *very strongly equivalent*, denoted $F \equiv_{\text{vs}} G$, provided that $F \equiv_{\text{pw}} G$ and given any basic action $b \in A$ (respectively basic predicate $q \in P$), both F and G have the same number of nodes labeled with b (respectively q).

These two notions of equivalence may be extended to relations between classes of flowcharts.

DEFINITION. A class of flowcharts $\mathbf{C}_1$ is said to be *no more powerful under pathwise equivalence* than a class of flowcharts $\mathbf{C}_2$, denoted $\mathbf{C}_1 \leqq_{\text{pw}} \mathbf{C}_2$, provided that, for any $F \in \mathbf{C}_1$, there exists $G \in \mathbf{C}_2$ such that $F \equiv_{\text{pw}} G$.

DEFINITION. A class of flowcharts $\mathbf{C}_1$ is said to be *pathwise equivalent* to a class of flowcharts $\mathbf{C}_2$, denoted $\mathbf{C}_1 \equiv_{\text{pw}} \mathbf{C}_2$, provided that $\mathbf{C}_1 \leqq_{\text{pw}} \mathbf{C}_2$ and $\mathbf{C}_2 \leqq_{\text{pw}} \mathbf{C}_1$.

DEFINITION. A class of flowcharts $\mathbf{C}_1$ is said to be *less powerful under pathwise equivalence* than a class of flowcharts $\mathbf{C}_2$, denoted $\mathbf{C}_1 <_{\text{pw}} \mathbf{C}_2$, provided that $\mathbf{C}_1 \leqq_{\text{pw}} \mathbf{C}_2$ and there exists $G \in \mathbf{C}_2$ such that for no $F \in \mathbf{C}_1$ is it true that $F \equiv_{\text{pw}} G$.

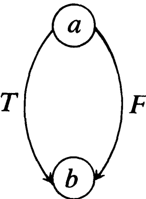
The relations $\leqq_{\text{vs}}$, $\equiv_{\text{vs}}$ and $<_{\text{vs}}$ may be defined analogously to those above and will be assumed in what follows.

Flowgraphs are similar to flowcharts and have been studied extensively because of their utility in performing data flow analysis [8]. For each positive integer k, we assume the existence of a distinct infinite denumerable set $S_k = \{a_k, b_k, \cdots\}$, the elements of which are called *k-ary actions*. We call $S = \bigcup_{k=1}^{\infty} S_k$ the set of *actions*. An action represents a basic block or extended basic block in the sense of [1].

DEFINITION. A *flowgraph* G is a finite, labeled directed graph $G = (N, E, L, \theta_N, \phi)$ composed of

(1) nodes labeled with actions;
(2) auxiliary nodes (as in flowcharts);
(3) two distinguished nodes, one of which is labeled START and the other is labeled STOP;
(4) paths, such that for any node $n \in N$, there is a path in G from the START node to the STOP node which contains n.

The START node has in-degree zero, and the STOP node has out-degree zero. The START node and all auxiliary nodes have out-degree one. For any positive integer k, all nodes labeled with k-ary actions have out-degree k. All edges are unlabeled, and hence $\theta_E = \phi$. Also, we intend this to imply that given $n_1, n_2 \in N$, there is a most one directed edge (n_1, n_2) in E. This restriction does not apply to flowcharts, as the subflowchart below is a valid flowchart.



Classes of structured flowgraphs may be defined analogously to classes of structured flowcharts.

DEFINITION. For any integer $n \geqq 0$, the class of CRE_n-*flowgraphs* (CASE-REPEAT-EXIT), denoted $\mathbf{CRE}_n$, is the class of flowgraphs contained in the language of the following graph grammar:

(1) $\langle \text{CRE}_n\text{-}\mathit{flowgraph} \rangle$ ::= (START) $\rightarrow\langle \mathbf{CRE}_n \rangle\rightarrow$ (STOP).

(2) $\rightarrow\langle \mathbf{CRE}_n \rangle\rightarrow$ ::= $\rightarrow\langle \mathbf{CRE}_n \rangle\rightarrow\langle \mathbf{CRE}_n \rangle\rightarrow$.

(3) $\rightarrow\langle \mathbf{CRE}_n \rangle\rightarrow$ ::= $\rightarrow$ (a_k) [branches $\rightarrow\langle \mathbf{CRE}_n \rangle\rightarrow$ … $\rightarrow\langle \mathbf{CRE}_n \rangle\rightarrow$ joining at] $0\rightarrow$, where $a_k \in S_k$, with $k \geqq 2$.

(4) $\rightarrow\langle \mathbf{CRE}_n \rangle\rightarrow$ ::= $\rightarrow$ (RPT) $\rightarrow\langle \mathbf{CRE}_n \rangle\rightarrow$ (END) $\rightarrow$.

(5) $\rightarrow\langle \mathbf{CRE}_n \rangle\rightarrow$ ::= $\rightarrow$ (a_1) $\rightarrow$, where $a_1 \in S_1$.

(6) $\rightarrow\langle \mathbf{CRE}_n \rangle\rightarrow$ ::= $\rightarrow$ (EXIT(i)) $\rightarrow$, where $1 \leqq i \leqq n$.

(7) $\rightarrow\langle \mathbf{CRE}_n \rangle\rightarrow$ ::= $\rightarrow$.

We use **CRE** to denote $\bigcup_{n=0}^{\infty} \mathbf{CRE}_n$. Everything defined with respect to **RE** may be defined with respect to **CRE** and will be assumed. Furthermore, everything that has been defined with respect to flowcharts may be defined with respect to flowgraphs. Those definitions will be assumed although not explicitly stated. A formalization of the relationship between flowcharts and flowgraphs may be found in [9].

In what follows the term *flowgraph* is used to refer only to those flowgraphs containing no auxiliary nodes. CRE_n-flowgraphs will be referred to explicitly.

3. Reducibility. The class of reducible flowgraphs was introduced by Allen and Cocke [1] and was further characterized by Hecht and Ullman [8]. We will, in fact, use one of Hecht and Ullman's characterizations as our definition. The following is necessary for its presentation.

DEFINITION. A *trivial flowgraph* is a flowgraph consisting of a START node, a STOP node, and a node labeled with a unary action, which is the successor of the START node and the predecessor of the STOP node.

Hecht and Ullman characterized the class of reducible flowgraphs as those flowgraphs which can be "collapsed" into a trivial flowgraph using two graph transformations T_1 and T_2. Informally, T_1 deletes a self-loop.

DEFINITION. For a flowgraph $G=(N, E, L, \theta_N, \phi)$ with $(n, n)\in E$, the flowgraph $T_1(G, (n, n))=(N, E-\{(n, n)\}, L, \theta'_N, \phi)$. Note that $\theta'_N \neq \theta_N$ since, if in G node n was a k-ary action, then in $T_1(G, (n, n))$ n will be a $(k-1)$-ary action.

Informally, the transformation T_2 merges two nodes into a single node.

DEFINITION. For any flowgraph $G=(N, E, L, \theta_N, \phi)$, with $n_1, n_2\in N$ such that neither n_1 nor n_2 is the START or STOP node and n_1 is the unique predecessor of n_2 in G, the flowgraph $T_2(G, (n_1, n_2))=(N', E', L, \theta'_N, \phi)$, where:

(a) $N'=(N-\{n_1, n_2\})\cup\{n_1/n_2\}$. Here n_1/n_2 is a new node which represents the merge of n_1 and n_2.

(b) The new edge set E' is more difficult to define formally, since edges to/from the nodes n_1 and n_2 must be deleted, and new edges to/from n_1 and n_2 must be added.

$$
\begin{aligned}
E' = (E-(\{(n_1, n)\in E\}\cup\{(n_2, n)\in E\}\cup\{(n, n_1)\in E\}))\\
\cup\{(n, n_1/n_2)|\text{ for all } n \text{ such that } (n, n_1)\in E\}\\
\cup\{(n_1/n_2, n)|\text{ for all } n \text{ such that } (n_2, n)\in E\}\\
\cup\{(n_1/n_2, n)|\text{ for all } n \text{ such that } (n_1, n)\in E \text{ and } n\neq n_2\}.
\end{aligned}
$$

Again $\theta'_{N'}\neq\theta_N$, since two nodes have been deleted and one added.

DEFINITION. The class of *reducible flowgraphs*, denoted **RFG**, is the class of all flowgraphs G such that there exists a finite sequence of flowgraphs $G_0, G_1, \cdots, G_k$, where:

(1) G_0 is G.

(2) For $0\leqq i\leqq k-1$, either $G_{i+1}=T_1(G_i, (n, n))$ for some (n, n) or $G_{i+1}=T_2(G_i, (n_1, n_2))$ for some (n_1, n_2).

(3) G_k is a trivial flowgraph.

Hecht and Ullman [8] showed that, for any $G\in\mathbf{RFG}$, if $T_1(G, (n, n))$ is defined for any (n, n), then $T_1(G, (n, n))\in\mathbf{RFG}$, and if $T_2(G, (n_1, n_2))$ is defined for any (n_1, n_2), then $T_2(G, (n_1, n_2))\in\mathbf{RFG}$. We define an object of the form $(T_1, (n, n))$ an *application of* T_1 and an object of the form $(T_2, (n_1, n_2))$ an *application of* T_2.

DEFINITION. A *parse* π of $G\in\mathbf{RFG}$ is a sequence of applications of T_1 and T_2 defined recursively by:

(1) If G is a trivial flowgraph, then the empty sequence is a parse of G.

(2) If $G'=T_1(G, (n, n))$, then $(T_1, (n, n))$ followed by a parse of G' is a parse of G.

(3) If $G'=T_2(G, (n_1, n_2))$, then $(T_2, (n_1, n_2))$ followed by a parse of G' is a parse of G.

In general, a reducible flowgraph may have more than one parse. The *length of a parse* is the number of applications of T_1 and T_2 in it. A *minimal parse* of $G\in\mathbf{RFG}$ is a parse no longer than any other parse of G. The following will help in determining the length of any minimal parse of a reducible flowgraph.

DEFINITION. The *reduction sequence* resulting from a parse π of $G\in\mathbf{RFG}$, denoted $R(G, \pi)$, is the sequence of flowgraphs defined recursively by:

(1) If π is the empty sequence, then $R(G, \pi)=G$.

(2) If $\pi=(T_1, (n, n))\pi'$ then $R(G, \pi)=G, R(T_1(G, (n, n)), \pi')$.

(3) If $\pi=(T_2, (n_1, n_2))\pi'$ then $R(G, \pi)=G, R(T_2(G, (n_1, n_2)), \pi')$.

A more complete version of the following definitions may be found in [8]. We present only what we need.

DEFINITION. For any parse π of $G \in \mathbf{RFG}$, with $R(G, \pi) = G_0, G_1, \cdots, G_k$, each node in G_i, $0 \leqq i \leqq k$, is said to *represent with respect to* π a subset of the nodes of G as follows, where $\mathrm{REP}_{G,\pi}(n, G_i)$ denotes what n in G_i represents with respect to π:

(1) For any node n in G, $\mathrm{REP}_{G,\pi}(n, G) = \{n\}$.

(2) For $0 \leqq i \leqq k-1$, if $G_{i+1} = T_1(G_i, (n, n))$, then for any n_i in G_{i+1}, $\mathrm{REP}_{G,\pi}(n_i, G_{i+1}) = \mathrm{REP}_{G,\pi}(n_i, G_i)$.

(3) For $0 \leqq i \leqq k-1$, if $G_{i+1} = T_2(G_i, (n_1, n_2))$, then for any n in G_{i+1}, $n \neq n_1/n_2$, $\mathrm{REP}_{G,\pi}(n, G_{i+1}) = \mathrm{REP}_{G,\pi}(n, G_i)$. For $n = n_1/n_2$, $\mathrm{REP}_{G,\pi}(n, G_{i+1}) = \mathrm{REP}_{G,\pi}(n_1, G_i) \cup \mathrm{REP}_{G,\pi}(n_2, G_i)$.

DEFINITION. For any parse π of $G \in \mathbf{RFG}$ with $R(G, \pi) = G_0, G_1, \cdots, G_k$, each edge in G_i, $0 \leqq i \leqq k$, is said to represent with respect to π a subset of the edges of G as follows, where $\mathrm{REP}_{G,\pi}(e, G_i)$ denotes what e in G_i represents with respect to π (by convention if an edge e is not in G_i then $\mathrm{REP}_{G,\pi}(e, G_i)$ is the empty set):

(1) For any e in G, $\mathrm{REP}_{G,\pi}(e, G) = \{e\}$.

(2) For $0 \leqq i \leqq k-1$, if $G_{i+1} = T_1(G_i, (n, n))$, then for any e in G_{i+1}, $\mathrm{REP}_{G,\pi}(e, G_{i+1}) = \mathrm{REP}_{G,\pi}(e, G_i)$.

(3) For $0 \leqq i \leqq k-1$, if $G_{i+1} = T_1(G_i, (n_1, n_2))$, then depending upon the edge e in G_{i+1}, we have the following cases:

$$e = (m, n),\ m, n \neq \frac{n_1}{n_2}: \mathrm{REP}_{G,\pi}(e, G_{i+1}) = \mathrm{REP}_{G,\pi}(e, G_i),$$

$$e = \left(m, \frac{n_1}{n_2}\right),\ m, \neq \frac{n_1}{n_2}: \mathrm{REP}_{G,\pi}(e, G_{i+1}) = \mathrm{REP}_{G,\pi}((m, n_1), G_i),$$

$$e = \left(\frac{n_1}{n_2}, m\right),\ m \neq \frac{n_1}{n_2}: \mathrm{REP}_{G,\pi}(e, G_{i+1}) = \mathrm{REP}_{G,\pi}((n_1, m), G_i) \cup \mathrm{REP}_{G,\pi}((n_2, m), G_i)),$$

$$e = \left(\frac{n_1}{n_2}, \frac{n_1}{n_2}\right): \mathrm{REP}_{G,\pi}(e, G_{i+1}) = \mathrm{REP}_{G,\pi}((n_1, n_1), G_i) \cup \mathrm{REP}_{G,\pi}((n_2, n_1), G_i)).$$

DEFINITION. For any parse π of $G \in \mathbf{RFG}$ with $R(G, \pi) = G_0, G_1, \cdots, G_k$, an edge e of G is called a *backward edge with respect to* π provided that for some i, $0 \leqq i \leqq k-1$, $G_{i+1} = T_1(G_i, (n, n))$ and $e \in \mathrm{REP}_{G,\pi}((n, n), G_i)$.

Hecht and Ullman [8] showed that the set of backward edges of any $G \in \mathbf{RFG}$ is independent of any parse of G. Thus we may speak of the set of backward edges of a reducible flowgraph without any reference to a specific parse of it. Hecht and Ullman also characterized the set of backward edges of any $G \in \mathbf{RFG}$ as those edges from descendants to ancestors in any spanning tree created by a depth-first search (see [14]).

Notation. For $s \geqq 3$ and $b \geqq 0$, let $\mathbf{RFG}_{s,b}$ denote the class of all reducible flowgraphs having s nodes of which $b \leqq s$ nodes have backward edges to them.

We obtain a characterization of all minimal parses of a flowgraph in $\mathbf{RFG}_{s,b}$ with the following lemmas.

LEMMA 1. *For any parse π of $G \in \mathbf{RFG}$ with $R(G, \pi) = G_0, \cdots, G_k$ and for any node n_k in any G_i, $0 \leqq i \leqq k$, the backedges incident upon n_k represent, with respect to π, a subset of those edges in G incident upon a single node in* $\mathrm{REP}_{G,\pi}(n_k, G_i)$.

Proof. For any n in G_0 the conclusion is immediate.

Inductive case. This splits into the two subcases as to whether T_1 or T_2 was the transformation applied to obtain G_{i+1}.

Subcase 1. If G_{i+1} results from G_i by applying T_1, then, since the incident backedges and the nodes in G_{i+1} are the same as those in G_i, the conclusion follows from the inductive hypothesis.

Subcase 2. If G_{i+1} results from G_i by applying T_2 to $(G_i, (n_1, n_2))$, then n_2 has no incident backedges (otherwise T_2 could not be applied). Hence the backedges incident upon n_1/n_2 represent, with respect to π, in G_{i+1}, the same set of edges as the backedges incident upon n_1 in G_i. Further, no other nodes obtain additional backedges. Thus the induction hypothesis can again be applied.

LEMMA 2. *There is a parse of* $G \in \mathbf{RFG}_{s,b}$ *consisting of* $s-3$ *applications of* T_2 *and b applications of* T_1.

Proof. We proceed by induction on the number of edges in $G \in \mathbf{RFG}_{s,b}$.

Basis. Any $G \in \mathbf{RFG}$ having only two edges is a trivial flowgraph.

Induction. For $G \in \mathbf{RFG}_{s,b}$ having e edges, let π be any parse of G, with $R(G, \pi) = G_0, G_1, \cdots, G_k$. Let $i = \min j | G_{j+1} = T_2(G_j, (n_1, n_2))$ for some nodes n_1, n_2 in G_j. Note that the set of nodes of G_i and that of G are identical. Furthermore, the only edges of G that are not edges of G_i are self-loops. Thus, the only possible predecessors of n_2 in G are n_1 and n_2. If (n_2, n_2) is in G, then the induction hypothesis is applied to $T_1(G, (n_2, n_2)) \in \mathrm{RFG}_{s,b-1}$. If (n_2, n_2) is not in G, then the induction hypothesis is applied to $T_2(G, (n_1, n_2)) \in \mathrm{RFG}_{s-1,b}$.

LEMMA 3. *Any minimal parse of* $G \in \mathbf{RFG}_{s,b}$ *consists of* $s-3$ *applications of* T_2 *and b applications of* T_1.

Proof. Let $G \in \mathbf{RFG}_{s,b}$. By Lemma 2, there is a parse of G consisting of $s-3$ applications of T_2 and b applications of T_1. We now must show that both are necessary. Clearly $s-3$ applications of T_2 are necessary, for T_2 applications reduce the number of nodes by 1 for each application. We use Lemma 1 to show that at least b applications of T_1 are necessary. By Lemma 1, each application of T_1 eliminates, at most, an edge representing the entire set of incident backedges upon a single node of G. Since there are b such nodes in G, at least b applications of T_1 are necessary.

We wish to define reducibility with respect to flowcharts. There are, however, flowgraphs having only nodes of out-degree two or less such that the application of T_2 to them results in flowgraphs having nodes of out-degree greater than two. Rather than expand the notion of a flowchart, we take a simpler approach. It is consistent with the notions of a basic block and an extended basic block in [1] to assume that the set of basic actions is a subset of the set of unary actions and that the set of basic predicates is a subset of the set of binary actions. We say that a flowchart F consisting of nonauxiliary nodes results in a flowgraph G provided that G is simply F with all edges unlabeled. A flowchart consisting of nonauxiliary nodes is called a *reducible flowchart* provided that the flowgraph in which it results is reducible. We denote the class of all reducible flowcharts by **RFC**, and for $s \geqq 3$ and $b \geqq 0$, we use $\mathbf{RFC}_{s,b}$ to denote the class of all reducible flowcharts resulting in flowgraphs in $\mathbf{RFG}_{s,b}$.

4. Reducibility and structure. In this section, we investigate the relationship between **RFG** and **CRE** and that between **RFG** and **RE**. The following result, due to Kosaraju [10], is reproduced here for purposes of comparison.

Result 1. [10]. For any flowchart containing k nodes labeled with basic predicates, there is $a \equiv_{\mathrm{pw}} \mathrm{RE}_k$-chart.

By defining inverses to T_1 and T_2, we obtain an upper bound on the minimal k for which there is a CRE_k-flowgraph $\equiv_{\mathrm{pw}}$ a reducible flowgraph. As a corollary we obtain a comparable result with respect to **RFC** and **RE**. It can be seen that our bound is better than Kosaraju's infinitely often, and that even for **RFC**, his is better

than ours infinitely often. The significance of this bound will become evident in a later section.

The following definitions will be helpful in the presentation of the proofs of the theorems that follow.

DEFINITION. A node labeled EXIT(i) in a $\langle \mathrm{CRE}_n \rangle$ or $\langle \mathrm{RE}_n \rangle$ G is called a *jump node* of G provided that its depth of nesting in G is less than n.

DEFINITION. In any $G \in \mathbf{CRE}$, a sequence of nodes $P = m\ n_1 n_2 \cdots n_k\, n$, with $k \geqq 0$, is called an *effective edge* from m to n denoted $[m, n]$, provided that for $1 \leqq i \leqq k$, n_i is an auxiliary node and that there exist sequences of nodes S and T such that SPT is a computation path in G.

THEOREM 1. $\mathbf{RFG}_{s,b} \leqq_{\mathrm{pw}} \mathbf{CRE}_b$.

Proof. It is shown by induction on the length of a minimal parse of $G \in \mathbf{RFG}_{s,b}$ that there exist $H \in \mathbf{CRE}_b$ and a surjection α from the nonauxiliary nodes of H onto the nodes of G such that the following three properties hold:

(1) For any nonauxiliary node m in H, $\alpha(m)$ and m have the same label (and, hence, the same out-degree).

(2) For any two, not necessarily distinct nonauxiliary nodes m_1 and m_2 in H, $[m_1, m_2]$ is in H provided that $(\alpha(m_1), \alpha(m_2))$ is in G.

(3) If a nonauxiliary node m in H is as shown in a $\langle \mathrm{CRE}_b \rangle$ of the form shown in Fig. 2, then for all nonauxiliary nodes m' in any $\langle \mathrm{CRE}_b \rangle$ F_i, as shown, $\alpha(m) \neq \alpha(m')$.

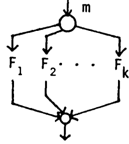


FIG. 2

If the relationship described above exists between G and H, then it can be seen that $G \equiv_{\mathrm{pw}} H$.

Basis. Any trivial flowgraph is in $\mathbf{CRE}_0$.

Induction. Let $G \in \mathbf{RFG}_{s,b}$ be such that any minimal parse of G has a length l, and let π be any minimal parse of G. By Lemma 3, $l = s + b - 3$.

Case 1. $\pi = (T_1, (n, n))\pi'$. Any minimal parse of $G' = T_1(G, (n, n))$ has length $l-1$ and G' contains s nodes. By Lemma 3, $G' \in \mathbf{RFG}_{s,b-1}$; so, by the induction hypothesis, there exist $H' \in \mathbf{CRE}_{b-1}$ and a surjection α from the nonauxiliary nodes of H' onto the nodes of G' such that the three desired properties hold. Recall that G and G' differ only in the presence of (n, n). Let M be the set of all nonauxiliary nodes m of H' such that $\alpha(m) = n$. If the out-degree of n in G' is one, let H be obtained from H' by replacing each $m \in M$ with a $\langle \mathrm{CRE}_b \rangle$ of the form shown in Fig. 3a. If the out-degree of n in G' is greater than one, let H be obtained from H' by replacing each $\langle \mathrm{CRE}_{b-1} \rangle$ of the form shown in Fig. 2 (with m as shown such that $m \in M$) with a $\langle \mathrm{CRE}_b \rangle$ of the form shown in Fig. 3b, where for $1 \leqq i \leqq k$, INC(F_i) is the $\langle \mathrm{CRE}_b \rangle$ obtained by incrementing the index of each jump node in F_i by one. Condition (3) ensures that the index of each jump node in F_i is only incremented by one (i.e., there is no recursive application of this induction step). Clearly, there exists a surjection α which is from the nonauxiliary nodes of H onto the nodes of G and

Case 2. $\pi = (T_2, (n_1, n_2))\pi'$. Any minimal parse of $G' = T_2(G, (n_1, n_2))$ has length $l-1$, and G' contains $s-1$ nodes. By Lemma 3, $G' \in \mathbf{RFG}_{s-1,b}$; so, by the inductive hypothesis, there exist $H' \in \mathbf{CRE}_b$ and a surjection α from the nonauxiliary nodes of

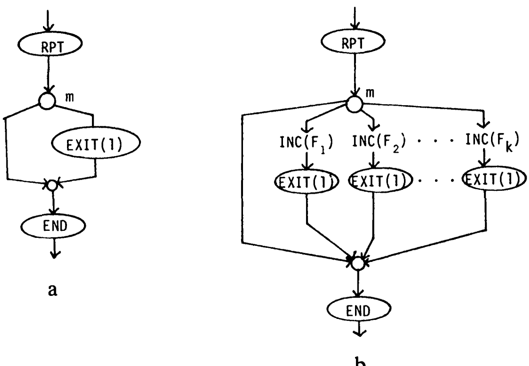


FIG. 3

H' onto the nodes of G' such that the three desired properties hold. Let M be the set of all nonauxiliary nodes m of H' such that $\alpha(m)=n_1/n_2$.

The edges from n_1/n_2 in G' may be partitioned into three sets, C_1, C_2 and C_3. Let e be an edge from n_1/n_2 in G'. From the definition of $T_2(G, (n_1, n_2))$ there must be a unique node n in G such that at least one of the following conditions hold:

(1) (n_1, n) is in G and $(n_1/n_2, n)=e$.

(2) (n_2, n) is in G and $(n_1/n_2, n)=e$.

(3) (n_2, n_1) is in G and $(n_1/n_2, n_1/n_2)=e$.

The edge $e\in C_1$ provided that only (1) holds; $e\in C_2$ provided that both (1) and (2) hold; and $e\in C_3$ provided that only (2) or (3) holds.

The edges from each node $m\in M$ in H' may be partitioned into $\{D_1, D_2, D_3\}$ according to $\{C_1, C_2, C_3\}$. Let m and m' be two nodes of H' such that $m\in M$ and (m, m') is in H'. By the first and second conditions on α, there must be a nonauxiliary node m'' in H' such that $[m, m'']=mm'm_1m_2\cdots m_jm''$ for some auxiliary nodes $m_1, m_2, \cdots, m_j$ with $j\geqq 0$. For $1\leqq i\leqq 3$, $(m, m')\in D_i$ provided that $(n_1/n_2, \alpha(m''))\in C_i$.

If n_1/n_2 has out-degree one in G', the edge e from n_1/n_2 must be in C_2 or C_3; otherwise, n_2 has out-degree zero in G. If $e\in C_2$, let H be obtained from H' by replacing each $m\in M$ with a ⟨CRE_0⟩ of the form shown in Fig. 4a. If $e\in C_3$, let H be obtained from H' by replacing each $m\in M$ with a ⟨CRE_0⟩ of the form shown in Fig. 4b.

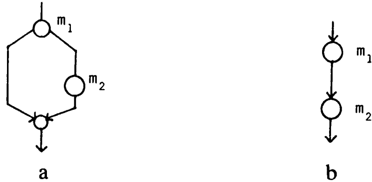


FIG. 4

If n_1/n_2 has out-degree greater than one in G', then each node $m\in M$ must be as shown in a ⟨CRE_b⟩ of the form shown in Fig. 2. Without loss of generality, for each $m\in M$, assume that the edges to $F_1, F_2, \cdots, F_i$ are in D_1, the edges to $F_{i+1}, F_{i+2}, \cdots, F_j$ are in D_2, and the edges to $F_{j+1}, F_{j+2}, \cdots, F_k$ are in D_3. Let H be obtained from H' by replacing, for each $m\in M$, the ⟨CRE_b⟩ of the form shown in Fig. 2 with the ⟨CRE_b⟩ of the form shown in Fig. 5.

In all cases, it can be seen that a surjection from the nonauxiliary nodes of H onto the nodes of G exists and has the desired properties. Furthermore, $H\in\mathbf{CRE}_b$.

COROLLARY. $\mathbf{RFC}_{s,b}\leqq_{\mathrm{pw}}\mathbf{RE}_b$.

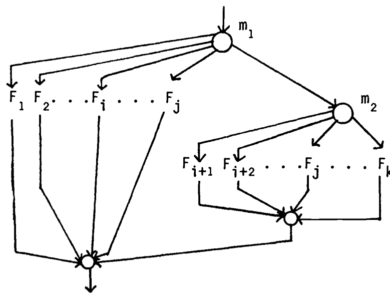


FIG. 5

Note that in the proof of Theorem 1, the inversion of T_1 does not produce duplicate "copies" of nodes of G in H, while the inversion of T_2 may. By modifying the construction, this situation may be avoided. The following concept will be helpful.

DEFINITION. An edge in $H \in \mathbf{CRE}$ that is incident upon an EXIT node, a merge node or the STOP node is called *safe*.

Note that in the inversion of T_2, if all edges in D_2 are safe, there is no duplication of nodes.

DEFINITION. If there is at most one edge that is not safe from each node in $H \in \mathbf{CRE}$, then H is said to have the *safe edge property*.

THEOREM 2. *For* $s \geqq 4$, $\mathbf{RFG}_{s,b} \leqq_{vs} \mathbf{CRE}_{s+b-4}$ *and* $\mathbf{RFG}_{3,b} \leqq_{vs} \mathbf{CRE}_b$.

Proof. By a modification of the proof of Theorem 1, it is shown by induction on the length of a minimal parse of $G \in \mathbf{RFG}$ that there exists $H \in \mathbf{CRE}$ such that H has the safe edge property and that there exists a *bijection* α from the nonauxiliary nodes of H onto the nodes of G such that the three properties listed in the proof of Theorem 1 hold. Furthermore, if $G \in \mathbf{RFG}_{3,b}$, then $H \in \mathbf{CRE}_b$, and if $G \in \mathbf{RFG}_{s,b}$ for some $s \geqq 4$, then $H \in \mathbf{CRE}_{s+b-4}$. Since α is a bijection, it can be seen that $G \equiv_{vs} H$. Note that we are, in fact, demonstrating a form of isomorphism between G and H rather than merely $\equiv_{vs}$.

The construction is similar to that in the proof of Theorem 1. For the base case, our additional criteria are satisfied. For the inductive case, the inversion of T_1 at most increments the index of any EXIT node by one. Furthermore, no new nonauxiliary nodes and only safe edges are introduced. Thus, if our additional criteria are satisfied prior to the inversion of T_1, they are satisfied afterwards. The inversion of T_2, however, may produce duplicate "copies" of nonauxiliary nodes. We will consider this case in detail.

Assume that $G \in \mathbf{RFG}_{s,b}$ having a minimal parse of length l, and let π be any minimal parse of G. By Lemma 3, $l = s + b - 3$. Assume further that $\pi = (T_2, (n_1, n_2))$, and let $G' = T_2(G, (n_1, n_2))$. Since G' contains $s-1$ nodes, $s \geqq 4$. Furthermore, since any minimal parse of G' has length $l-1$, $G' \in \mathbf{RFG}_{s-1,b}$ by Lemma 3. Thus, by the inductive hypothesis, there exists $H' \in \mathbf{CRE}$ and a bijection α from the nonauxiliary nodes of H' onto the nodes of G' such that the three desired properties hold. Furthermore H' has the safe edge property. If $s = 4$, then $H' \in \mathbf{CRE}_b$, and if $s \geqq 5$, then $H' \in \mathbf{CRE}_{s+b-5}$. Note that there is only one nonauxiliary node m in H' such that $\alpha(m) = n_1/n_2$.

If the out-degree of n_1/n_2 in G' is one, let H be constructed from H' as in Case 2 of Theorem 1. It can be seen that H has the safe edge property and that there exists a bijection α from the nonauxiliary nodes of H onto the nodes of G such that the three desired properties hold. Finally, if $s = 4$, then $H \in \mathbf{CRE}_b = \mathbf{CRE}_{s+b-4}$, and if $s \geqq 5$, then $H \in \mathbf{CRE}_{s+b-5} \subseteq \mathbf{CRE}_{s+b-4}$.

If the out-degree of n_1/n_2 in G' is greater than one, recall the partition $\{D_1, D_2, D_3\}$ of the edges from the unique nonauxiliary node m in H' such that $\alpha(m) = n_1/n_2$. If all edges from m are safe or if the edge from m that is not safe is in D_3, let H be constructed as in Case 2 of the proof of Theorem 1. It can be seen that H has the safe edge property and that a bijection exists and has the desired properties. Finally, if $s = 4$, then $H \in \mathbf{CRE}_b = \mathbf{CRE}_{s+b-4}$, and if $s \geqq 5$, then $H \in \mathbf{CRE}_{s+b-5} \subseteq \mathbf{CRE}_{s+b-4}$.

If the edge from m that is not safe is in D_1 or D_2, then it can be seen that $s \geqq 5$. Without loss of generality, assume that the edge from m that is not safe is incident upon F_1 as shown in Fig. 2, where m is as shown. Let H'' be constructed from H' by replacing the $\langle \mathrm{CRE}_{s+b-5} \rangle$ of the form shown in Fig. 2 with the $\langle \mathrm{CRE}_{s+b-5} \rangle$ of the form shown in Fig. 6, where INC(F_1) is as in Case 1 of the proof of Theorem 1 and for $2 \leqq i \leqq k$, UP(F_i) is EXIT($j_i + 1$) if F_i is EXIT(j_i) and EXIT(1) if F_i is $\rightarrow$. Clearly a bijection α' from the nonauxiliary nodes of H'' onto the nodes of G' exists and has the desired properties. Not only does H'' have the safe edge property, but all edges from the unique node m in H'', such that $\alpha'(m) = n_1/n_2$, are safe. Finally, $H'' \in \mathbf{CRE}_{s+b-4}$. Let H be constructed from H'' as in Case 2 of the proof of Theorem 1.

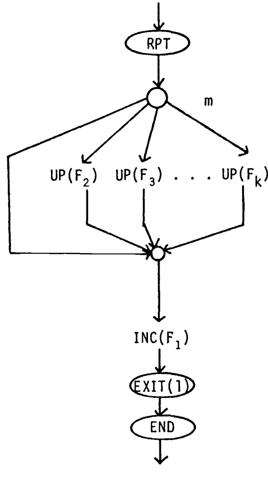


FIG. 6

It can be seen that a bijection from the nonauxiliary nodes of H onto the nodes of G exists and has the three desired properties. Finally, H has the safe edge property, and $H \in \mathbf{CRE}_{s+b-4}$.

COROLLARY 2. *For* $s \geqq 4$, $\mathbf{RFC}_{s,b} \leqq_{\mathrm{vs}} \mathbf{RE}_{s+b-4}$, *and* $\mathbf{RFC}_{3,b} \leqq_{\mathrm{vs}} \mathbf{RE}_b$.

A result similar to Corollary 2 appears in Peterson et al. [13]. Their construction takes an arbitrary flowchart, constructs a reducible flowchart $\equiv_{\mathrm{pw}}$ to it if it is not reducible, and then constructs an RE $\equiv_{\mathrm{vs}}$ to the reducible flowchart. Our construction has the advantages of consisting of local transformations and of producing an upper bound on the minimal k for which there is an RE$_k$-chart $\equiv_{\mathrm{pw}}$ to the reducible flowchart.

Hecht and Ullman [8] also characterized the class of nonreducible flowgraphs as those flowgraphs that contain a subgraph of the form shown in Fig. 7, where wavy lines represent node-disjoint paths. From this characterization, the following corollaries may be obtained.

COROLLARY 3. $\mathbf{RFG} \equiv_{\mathrm{vs}} \mathbf{CRE}$.

COROLLARY 4. $\mathbf{RFC} \equiv_{\mathrm{vs}} \mathbf{RE}$.

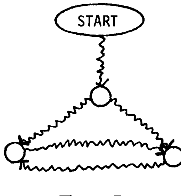


FIG. 7

Thus, by applying a notion from the area of abstract control structures to flowgraphs, we obtain a characterization of **RFG**, and by applying a concept from data flow analysis to flowcharts, we obtain a characterization of **RE**.

5. Interpretations and Kosaraju's hierarchy. We have so far regarded flowcharts as completely uninterpreted entities. We now provide a mechanism whereby semantics may be associated with basic actions and predicates.

DEFINITION. An *interpretation* is a triple $I = \langle D, \phi_A, \phi_P \rangle$, where:

(1) D is a set called the *domain* of I;

(2) ϕ_A is a partial function such that $\phi_A : A \to \{\alpha | \alpha : D \to D$ and α is recursive$\}$;

(3) ϕ_P is a partial function such that $\phi_P : P \to \{\beta | \beta : D \to \{T, F\}$ and β is recursive$\}$.

Since flowcharts are actually abbreviations for Ianov schemas [12], we assume the concept of a *Ianov schema under an interpretation* or a *Ianov program* as defined in [12]. We also assume the definition of the *partial function computed by a Ianov schema under an interpreation* or *the partial function computed by a Ianov program* as in [12], as well as all concepts related to those definitions. For reasons that will become evident, we consider the following notions of equivalence between flowcharts.

DEFINITION. Two flowcharts F and G are said to be *functionally equivalent*, denoted $F \equiv_f G$, provided that F and G compute the same partial function under any interpretation of both F and G.

DEFINITION. Two flowcharts F and G are said to be *semantically equivalent*, denoted $F \equiv_{sem} G$, provided that $F \equiv_f G$ and the set of basic actions and predicates of F and that of G are the same.

Relations between classes of flowcharts based on $\equiv_{sem}$ and $\equiv_f$ may be defined analogously to those based on $\equiv_{pw}$, and those definitions will be assumed although not explicitly stated.

We wish to examine the relationship between $\mathbf{RE}_n$ and more structured classes such as **D** and $\mathbf{RE}_m$ with $m < n$. The following results due to Kosaraju [10] indicate that the techniques we have used so far are insufficient for describing transformations between such classes.

Result 2 [10]. $\mathbf{D} <_{sem} \mathbf{RE}_1$.

Result 3 [10]. For all $n \geqq 0$, $\mathbf{RE}_n <_{sem} \mathbf{RE}_{n+1}$.

Thus, if we are to develop transformations from $\mathbf{RE}_n$ to **D** or to $\mathbf{RE}_m$ with $m < n$, the transformations must introduce something extraneous to the flowchart. We consider such transformations in the next section.

6. Auxiliary Boolean variables. The use of auxiliary Boolean variables, or flags, to structure flowcharts is presented in [2], [4] and [5]. The arguments for and against the use of flags are primarily subjective and, hence, will not be considered in this paper.

A *flag* is a Boolean variable used exclusively to structure a flowchart or program. A formal treatment of flags may be found in [5] and [9]. We regard assignments to flags as basic actions and tests on flags as basic predicates; however, we require that an interpretation of a flowchart containing operations on flags be entirely independent

of the flags. Furthermore, the value of a flag is not a component of the function computed by a flowchart under an interpretation. For any integer $k \geqq 0$, we use $\mathbf{D}_k$ to denote the class of D-charts containing operations on at most k flags, and we use $\mathbf{RE}_{n,k}$ to denote the class of RE_n-charts containing operations on at most k flags.

THEOREM 3. $\mathbf{RE}_{n,k} \leqq_{\text{f}} \mathbf{RE}_{\lceil n/2 \rceil, k+1}$.

Proof. Let l be a flag such that there are no operations on l in $F \in \mathbf{RE}_{n,k}$. Consider the following inductively defined transformation $T: \mathbf{RE}_{n,k} \to \mathbf{RE}_{\lceil n/2 \rceil, k+1}$. Informally, if at any point on a computation path the value of l is *true*, then $\lceil n/2 \rceil$ more loops must be exited before a basic action or predicate other than an operation on l is reached. It can be seen that, for all $F \in \mathbf{RE}_{n,k}$, $T(F) \equiv_{\text{f}} F$,

(1) $T(\text{(START)} \to G \to \text{(STOP)}) = \text{(START)} \to (l \leftarrow \text{false}) \to T(G) \to \text{(STOP)}$.

(2) $T(\to G \to H \to) = \to T(G) \to T(H) \to$.

(3) $T\left(\to (p) \begin{matrix} \nearrow G \searrow \\ \searrow H \nearrow \end{matrix} 0 \to \right) = \to (p) \begin{matrix} \nearrow T(G) \searrow \\ \searrow T(H) \nearrow \end{matrix} 0 \to$.

(4) $T(\to \text{(RPT)} \to G \to \text{(END)} \to) = \to \text{(RPT)} \to T(G) \to \text{(END)} \to (l) \begin{matrix} \xrightarrow{T} (l \leftarrow \text{false}) \to (\text{EXIT}(\lceil n/2 \rceil)) \searrow \\ \xrightarrow{F} \nearrow \end{matrix} 0 \to$.

(5) $T(\to (f) \to) = \to (f) \to$.

(6) $T(\to (\text{EXIT}(i)) \to) = \begin{cases} \to (\text{EXIT}(i)) \to, & \text{if } i \leqq \lceil n/2 \rceil \\ \to (l \leftarrow \text{true}) \to (\text{EXIT}(i - \lceil n/2 \rceil)) \to, & \text{if } \lceil n/2 \rceil < i \leqq n. \end{cases}$

(7) $T(\to) = \to$.

COROLLARY 5. $\mathbf{RE}_{n,k} \leqq_{\text{f}} \mathbf{RE}_{1, k + \lceil \log_2 n \rceil}$.

THEOREM 4. $\mathbf{RE}_{1,k} \leqq_{\text{f}} \mathbf{D}_{k+1}$.

Proof. Let l be a flag such that there are no operations on l in $F \in \mathbf{RE}_{1,k}$. Consider the following inductively defined transformation $U: \mathbf{RE}_{1,k} \to \mathbf{D}_{k+1}$. It can be seen that for all $F \in \mathbf{RE}_{1,k}$, $F \equiv_{\text{f}} U(F)$.

(1) $U(\text{(START)} \to G \to \text{(STOP)}) = \text{(START)} \to (l \leftarrow \text{false}) \to U(G) \to \text{(STOP)}$.

(2) $U(\to G \to H \to) = \to U(G) \to (l) \begin{matrix} \xrightarrow{F} U(H) \searrow \\ \xrightarrow{T} \nearrow \end{matrix} 0 \to$.

(3) $U\left(\to (p) \begin{matrix} \nearrow G \searrow \\ \searrow H \nearrow \end{matrix} 0 \to \right) = \to (p) \begin{matrix} \nearrow U(G) \searrow \\ \searrow U(H) \nearrow \end{matrix} 0 \to$.

(4) $U(\to \text{(RPT)} \to G \to \text{(END)} \to) = \to (l) \begin{matrix} \xrightarrow{F} U(G) \circlearrowleft \\ \xrightarrow{T} (l \leftarrow \text{false}) \to \end{matrix}$.

(5) $U(\to (f) \to) = \to (f) \to$.

(6) $U(\to (\text{EXIT}(l)) \to) = \to (l \leftarrow \text{true}) \to$.

(7) $U(\to) = \to$.

From Corollary 5 and Theorem 4, we may infer that $\mathbf{RE}_{n,k} \leqq_f \mathbf{D}_{k+\lceil \log_2 n \rceil+1}$. If we allow basic predicates representing Boolean functions on flags, a slightly more efficient result may be obtained.

THEOREM 5. $\mathbf{RE}_{n,k} \leqq_f \mathbf{D}_{k+\lceil \log_2 (n+1) \rceil}$.

Proof. Rather than present the transformation using operations on $\lceil \log_2 (n+1) \rceil$ additional flags directly, we use a variable LEVEL, which assumes values from $\{0, 1, 2, \cdots, n\}$. The tests on LEVEL may be replaced by basic predicates representing a Boolean function on $\lceil \log_2 (n+1) \rceil$ flags, and the assignments to LEVEL may be replaced by ⟨D⟩'s constructed from operations on these $\lceil \log_2 (n+1) \rceil$ flags.

Consider the following inductively defined transformation $V: \mathbf{RE}_{n,k} \rightarrow \mathbf{D}_{k+\lceil \log_2 (n+1) \rceil}$. Essentially the variable LEVEL holds the number of loops which must be exited on a computation path before a basic action or predicate that is not an operation on LEVEL is reached. If $F \in \mathbf{RE}_{n,k}$, it can be seen that $F \equiv_f V(F)$.

(1) $V($(START) $\rightarrow G \rightarrow$ (STOP)$)$ = (START) → (LEVEL ← 0) → $V(G)$ → (STOP) .

(2) $V(\rightarrow G \rightarrow H \rightarrow) = \rightarrow V(G) \rightarrow$ (LEVEL = 0) —T→ $V(H)$ → 0 →; —F→ 0 → .

(3) V(→ (P) —→ G → 0 →; —→ H → 0 →) = → (P) —→ $V(G)$ → 0 →; —→ $V(H)$ → 0 → .

(4) $V(\rightarrow$ (RPT) $\rightarrow G \rightarrow$ (END) $\rightarrow) = \rightarrow$ (LEVEL = 0) —T→ $V(G)$ ↺ (back to LEVEL = 0); —F→ (LEVEL ← LEVEL − l) → .

(5) $V(\rightarrow (f) \rightarrow) = \rightarrow (f) \rightarrow$.

(6) $V(\rightarrow$ (EXIT(i)) $\rightarrow) = \rightarrow$ (LEVEL ← i) → .

(7) $V(\rightarrow) = \rightarrow$.

Note that the transformations presented in the proofs of Theorems 4, 5 and 6 do not introduce anything but operations on the flags introduced. There is no duplication of existing basic actions and predicates. The equivalence between a flowchart and its image under one of those transformations may be viewed as "isomorphism up to flag operations".

We now show that there exist RE_n-charts which require $\lceil \log_2 (n+1) \rceil$ additional flags to be transformed into functionally equivalent D-charts. Hence, the upper bound given in Theorem 5 is tight.

THEOREM 6. *For any* $n \geqq 1$, *there exists an* RE_n *flowchart that cannot be* $\equiv_f$ *to any* D*-chart constructed from the same basic actions and predicates and from operations on only* $\lceil \log_2 (n+1) \rceil + 1$ *additional flags.*

Proof. The proof is a modification of that used by Kosaraju [10] to prove Result 3. Let F be the flowchart in Fig. 8, which is drawn to resemble a grid.

Kosaraju [10] gives a construction for a $G \in \mathbf{RE}_n$ such that $G =_{pw} F$. Note that any cycle in F either contains at least one node from each column or contains at least one node from each row. Thus, each cycle C intersects $n+1$ node-disjoint paths $P_{C,1}, P_{C,2}, \cdots, P_{C,n+1}$ such that for $1 \leqq i \leqq n$, $P_{c,i}$ is a cycle and $P_{C,n+1}$ ends in the STOP

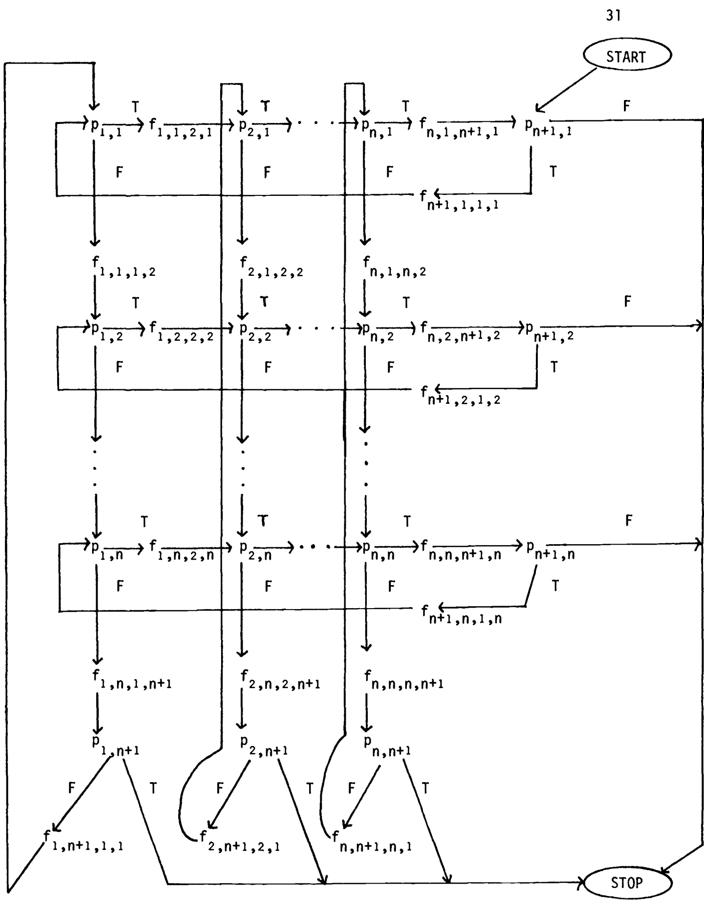


FIG. 8

node. We will provide an interpretation for F whereby the basic actions record the computation path resulting for each input. We assume that there exists $G \in \mathbf{D}$ such that $G \equiv_f F$ and G is constructed from the basic actions on predicates of F and from operations on only $\lceil \log_2 (n+1) \rceil - 1 = \lfloor \log_2 n \rfloor$ flags. We show that we may also assume that for some input, the equivalent of a cycle C of F exists within an innermost loop L of G. We then provide $n+1$ inputs such that for each input, the loop L in G is entered. For each of n of the inputs, the equivalent of a distinct cycle $P_{C,i}$, $1 \leqq i \leqq n$, is then followed until the loop L must have been left. In each of the n cases, a distinct basic action $f_{i,j,i',j'}$ must be the first such basic action reached when the loop is left. For the $n+1$st input, the equivalent of the path $P_{C,n+1}$ to the STOP node is followed from within L. We call a STOP node or a node labeled with a basic action $f_{i,j,i',j'}$ a *relevant node*. Note that for each of the $n+1$ inputs the first relevant node reached when the loop is left must be distinct. The first relevant node reached when the loop is left is entirely determined by the values of the basic predicates $p_{i,j}$ and by the vector of values of the $\lfloor \log_2 n \rfloor$ flags when the test of the loop is reached. However, our interpretation will be such that for each of the $n+1$ inputs, the values of all basic predicates $p_{i,j}$ will be the same when the loop is left. The $\lfloor \log_2 n \rfloor$ flags may assume only $2^{\lfloor \log_2 n \rfloor} \leqq n$ vectors of values. Thus for at least two of the inputs, the first relevant node reached when the loop L is left must be the same.

Following Kosaraju [10], we use the following interpretation of F:

program variable: the variable x;
input alphabet: $\{c, d\}$;
output alphabet: $\{c, d\} \cup Q$, where

$Q = \{[i, j, i+1, j], [i, j, i, j+1], [n+1, j, 1, j], [i, n+1, i, 1] | i, j \in [n]\}$;

basic predicates: $p_{i,j} = p$, where $p =$ "prefix of $x = c$?";
basic actions: $f_{i,j,i',j'} =$ "delete the prefix symbol of x and add $[i, j, i', j']$ as its suffix symbol".

Note that for some inputs infinite looping may occur; however, this situation will be of no concern. We, in fact, restrict our inputs to strings from $\{c, d\}^+$ for which the STOP node is reached with x having a prefix from $\{c, d\}^+$. As mentioned before, the basic actions of F trace the computation path followed. At any instance of a node on a computation path, x must be of the form yz, where:

(1) $y \in \{c, d\}^+$, and the input must have been of the form wy for some $w \in \{c, d\}^*$.
(2) $z = [i_1, j_1, i_2, j_2][i_2, j_2, i_3, j_3] \cdots [i_{m-1}, j_{m-1}, i_m, j_m]$ for some $m \geqq 0$, where for $1 \leqq k \leqq m$, $(i_{k+1}, j_{k+1}) = (i_k + 1, j_k)$ or $(i_k, j_k + 1)$, with sums "end-around at $n+1$" (i.e., $n+2=1$).

Assume there exists $G \in \mathbf{D}$ such that $G \equiv_f F$ and G is constructed from the same basic actions and predicates as F and from operations on $\lceil \log_2 (n+1) \rceil - 1 = \lfloor \log_2 n \rfloor$ flags. Let $k = \lfloor \log_2 n \rfloor$. The following will be helpful in presenting the proof. A node of G is said to be *relevant* provided that it is labeled with a basic action $f_{i,j,i',j'}$ or it is the STOP node. A relevant node n_j is called the *relevant successor* of the indicated instance of n_i on a computation path $P = n_1 n_2 \cdots n_i \cdots n_j \cdots n_k$ provided that $n_{i+1}, n_{i+2}, \cdots, n_{j-1}$ are not relevant. A node n_j is said to be a *potential relevant successor* of n_i provided that it is the relevant successor of an instance of n_i on some computation path. For simplicity, we call a node labeled with a basic predicate a *test node*. Note that the relevant successor of an instance of a test node n on a computation path P is entirely determined by the prefix symbol of x and by the vector of values of the k flags at that instance of n on P. Since the prefix symbol of x either is c or is not c and since k flags may assume 2^k vectors of values, any test node may have at most 2^{k+1} potential relevant successors. We call a potential relevant successor of a test node n a *potential relevant c-successor* provided that it is the relevant successor of some instance of n on some computation path P and the prefix of x is c at that instance of n on P. A *potential relevant d-successor* is defined analogously. Note that any test node may have at most 2^k potential relevant c-successors and at most 2^k potential relevant d-successors.

Since $G \equiv_f F$, G must compute the same partial function as F under the interpretation described above.

CLAIM 1. *G contains an innermost loop.*

CLAIM 2. *For some input x_0 and some innermost loop L of G, x_0 results in the computation path $P = P_0 m P_1 m P_2 \cdots m P_{2^{k+1}} m P_{2^{k+1}+1}$, where m is the test of L and for $1 \leqq i \leqq 2^{k+1}$, the nodes of P_i are within the loop L.*

Proof. If the claim is not true, any innermost loop L may be replaced as indicated in Fig. 9 until either a D-chart satisfying the claim is obtained or until a D-chart containing no loops is obtained. If the latter occurs, Claim 1 is violated.

CLAIM 3. *For the computation path $P = P_0 m P_1 m P_2 \cdots m P_{2^{k+1}} m P_{2^{k+1}+1}$ as described in Claim 2, the relevant successor of at least two of the indicated instances of m are the same.*

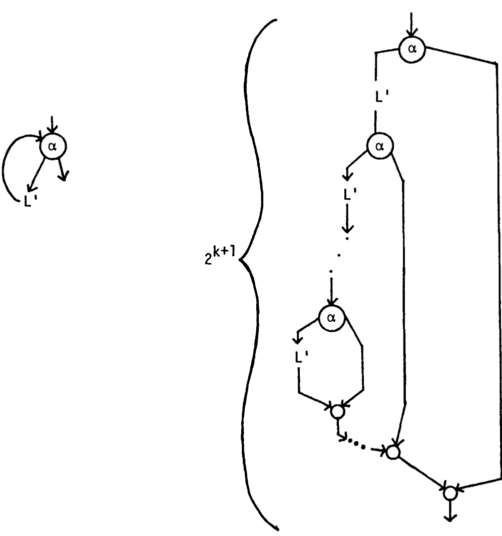


FIG. 9

Proof. There are $2^{k+1}+1$ indicated instances of m on P, and m may have at most 2^{k+1} potential relevant successors.

Thus, there is an innermost loop L in G which is entered on the computation path followed for some input x_0. Furthermore, there is an equivalent of a cycle in F entirely within L. Let $x_0 = y_1 y_2 z$, where $y_1 \in \{c, d\}^*$, $y_2 \in \{c, d\}^+$, and the two instances of the test of L having the same relevant successor are reached with the intermediate values $x = y_2 z y_1'$ and $x = z y_1' y_2'$, respectively. Note that $y_2 \in \{c, d\}^+$, otherwise either infinite looping would occur or the STOP node would be reached with x not having a prefix from $\{c, d\}^+$. By the structure of F, $y_2 = [i_1, j_1, i_2, j_2][i_2, j_2, i_3, j_3] \cdots [i_m, j_m, i_{m+1}, j_{m+1}]$, where for $1 \leqq s \leqq m+1$, $(i_{s+1}, j_{s+1}) = (i_s+1, j_s)$ or (i_s, j_s+1) (with sums "end-around at $n+1$") and $(i_{m+1}, j_{m+1}) = (i_1, j_1)$. Thus, there exist $x_1, x_2, \cdots, x_{n+1}$ and $z_1, z_2, \cdots, z_{n+1}$ such that $x_0 = x_1 z_1 = x_2 z_2 = \cdots = x_{n+1} z_{n+1}$ and the intermediate values $z_i w_i[\cdot, \cdot, u_i, v_i]$ are formed within L in G with either $u_i = i$ for $1 \leqq i \leqq n+1$ or $v_i = i$ for $1 \leqq i \leqq n+1$. Furthermore, those intermediate values are formed within L before the prefix symbol of each of the z_i's is tested.

We consider the case in which $u_i = i$ for $1 \leqq i \leqq n+1$ (the other case is argued analogously). Let s be any integer such that $s(n+1)$ is greater than the number of relevant nodes in L. For $1 \leqq i \leqq n+1$, consider the $n+1$ inputs $x_i (d^{n+1})^s z_i'$, where $z_i' = c^{n+1-i} d$. First consider those inputs with respect to F. Since $F \equiv_f G$, for $1 \leqq i \leqq n+1$, the intermediate value $(d^{n+1})^s z_i' w_i[\cdot, \cdot, i, v_i]$ will be formed. Because of the structure of F, for $1 \leqq i \leqq n$, $z_i' w_i[\cdot, \cdot, i, v_i]([i, v_i, i, v_{i+1}] \cdots [i, v_{i+n}, i, v_{i+n+1}])^s$ will be formed, where $v_{i+1} = v_i + 1$ and $v_{i+n+1} = v_i$. Finally, for $1 \leqq i \leqq n$, the STOP node is reached with the value of $x = d w_i[\cdot, \cdot, i, v_i]([i, v_i, i, v_{i+1}] \cdots [i, v_{i+n}, i, v_{i+n+1}])^s z_i''$, where $z_i'' = [i, v_{i+n=1}, i+1, v_{i+n+1}][i+1, v_{i+n+1}, i+2, v_{i+n+1}] \cdots [n, v_{i+n+1}, n+1, v_{i+n+1}]$. For the input $x_{n+1}(d^{n+1})^s z_{n+1}'$, the STOP node is reached with the value of $x = (d^{n+1})^w z_i' w_i[\cdot, \cdot, n+1, v_i]$. Since $F \equiv_f G$, for each of the $n+1$ inputs, the STOP in G node must be reached with the appropriate value of x.

With respect to G, for each of the $n+1$ inputs $x_i(d^{n+1})^s z_i'$, the intermediate value $(d^{n+1})^s z_i' w_i[\cdot, \cdot, i, v_i]$ must be formed within the loop L before its prefix is tested. Hence, for $1 \leqq i \leqq n$, the intermediate value $z_i' w_i[\cdot, \cdot, k, v_i] \cdot ([i, v_i, i, v_{i+1}] \cdots [i, v_{i+n}, i v_{i+n+1}])^s$ must be formed if the STOP node is to be reached

with the appropriate value. Since in each of the n cases, $s(n+1)$ basic actions must be reached before the formation of the appropriate values, the test of the loop L must be reached because of our choice of s. In each of those n cases a distinct relevant successor of the test of L must be reached. Furthermore, for the input $x_{n+1}(d^{n+1})^s z'_{n+1}$, the STOP node must be reached with the value of $x=(d^{n+1})^s z'_{n+1} w_{n+1}[\cdot, \cdot, n+1, v_i]$. Thus, in all $n+1$ cases, the test of L is reached with d as the prefix of x and a distinct relevant successor of the test of L reached thereafter. Thus, the test of L must have $n+1$ potential relevant d-successors. However, any test node in G may have at most $2^k = 2^{\lfloor \log_2 n \rfloor} \leqq n$ potential relevant d-successors.

The case in which $v_j = j$, for $1 \leqq j \leqq n+1$ is argued analogously.

COROLLARY 6. *For $n \geqq 0$, there exists a* D-*chart containing operations on $n+1$ flags that cannot be* $\equiv_f$ to any D-*chart constructed from the same basic actions and predicates and operations on only n flags.*

Proof. The result follows from Theorems 5 and 6.

7. Conclusion. We have examined the relationship between program structure and reducibility. We have shown that for any reducible flowchart (flowgraph) having n nodes of which b have backward edges incident upon them, there exist a $\equiv_{pw}$ RE$_b$-chart (CRE$_b$-flowchart) and a $\equiv_{vs}$ RE$_{n+b-4}$-chart (CRE$_{n+b-4}$-flowgraph). We have also shown that for any RE$_n$-chart there is a $\equiv_f$ D-chart containing operations on $\lceil \log_2 (n+1) \rceil$ additional flags. We have also provided a partial answer to an open question of Ashcroft and Manna [2] concerning the minimal number of flags that must be introduced to convert an arbitrary flowchart into a D-chart. Furthermore, Theorem 6 and its corollaries coupled with the results found in [6] provide a case against the exclusive use of the control structures reflected in D-charts.

REFERENCES

[1] F. E. ALLEN AND J. COCKE, *A catalogue of optimizing transformations*, in Design and Optimization of Computers, R. Rustin, ed., Prentice-Hall, Englewood Cliffs, NJ, 1973.

[2] E. A. ASHCROFT AND Z. MANNA, *Translating program schemas to while-schemas*, this Journal, 4 (1975), pp. 125–146.

[3] B. S. BAKER, *An algorithm for structuring flowgraphs*, J. Assoc. Comput. Mach., 24 (1977), pp. 98–120.

[4] C. BOEHM AND G. JACOPINI, *Flow diagrams, Turing machines and languages with only two formation rules*, Comm. ACM, 9 (1966), pp. 366–371.

[5] J. BRUNO AND K. STEIGLITZ, *The expression of algorithms by charts*, J. Assoc. Comput. Mach., 19 (1972), pp. 517–525.

[6] R. A. DEMILLO, S. C. EISENSTAT AND R. LIPTON, *The complexity of control structures and data structures*, Proc. Seventh ACM Symposium on Theory of Computing, Assoc. Comput. Mach., New York, 1975, pp. 186–193.

[7] C. C. ELGOT, *Algorithmic properties of structures*, Math. Systems Theory, 3 (1976), pp. 183–195.

[8] M. S. HECHT, *Flow Analysis of Computer Programs*, Elsevier-North Holland, Amsterdam, 1977.

[9] J. KEOHANE, *On reducibility, structure and the need and use of flags*, Ph.D. Dissertation, Department of Computer Science, SUNY, Stony Brook, NY, 1978.

[10] S. R. KOSARAJU, *Analysis of structured programs*, J. Comput. System. Sci., 9 (1974), pp. 232–255.

[11] H. F. LEDGARD AND M. MARCOTTY, *A geneology of control structures*, Comm. ACM, 18 (1975), pp. 629–639.

[12] Z. MANNA, *Mathematical Theory of Computation*, McGraw-Hill, New York, 1974.

[13] W. W. PETERSON, P. KASAMI AND N. TOKURA, *On the capabilities of while, repeat and exit statements*, Comm. ACM, 16 (1973), pp. 503–512.

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

SIAM J. COMPUT.
Vol. 11, No. 2, May 1982
© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1102-0007 $01.00/0

A TIME-SPACE TRADEOFF FOR SORTING ON A GENERAL SEQUENTIAL MODEL OF COMPUTATION

A. BORODIN† AND S. COOK†

Abstract. In a general sequential model of computation, no restrictions are placed on the way in which the computation may proceed, except that parallel operations are not allowed. We show that in such an unrestricted environment TIME·SPACE $=\Omega(N^2/\log N)$ in order to sort N integers, each in the range $[1, N^2]$.

Key words. time-space tradeoffs, conputational complexity, sorting, time lower bounds, space lower bounds

1. Introduction. Within the field of computational complexity, our inability to establish lower bounds on the complexity of "natural problems" stands in marked contrast to the progress that has been made in algorithmic design and analysis, and the progress in characterizing the central issues. To be fair, there are the following important exceptions:

1. Relative to an appropriate reducibility, a problem can be shown "hard" for an entire complexity class. Then diagonalization can be used to infer a corresponding complexity lower bound. For example, see the discussion in Aho, Hopcroft and Ullman [1, Chapt. 11].

2. For certain natural but "structured" models of computation, we have a number of interesting lower bounds. We use "structured" in the sense of Pippenger and Valiant's [2] use of "conservative" to mean that the computation can only proceed within a fixed mathematical structure (e.g., a partial order for comparison based models, a ring or field for algebraic complexity) and only uses the relations and functions within that structure for the computation (see also Borodin [3]). For example, using comparison trees it is well known that sorting n elements requires at least $n \log n + O(n)$ comparisons.

3. On certain nonstructured but restricted models of computation we have a few results. For example, to recognize the set $\{w \# w^R\}$ on a *one-tape* Turing machine requires $\Omega(n^2)$ steps.

A *general* sequential model of computation can be viewed as a string processing machine. While the input string may arise as the encoding of a set of mathematical objects, there is no obligation to process these objects in ways prescribed by the mathematical structure. In this context complexity is measured as a function of the input (plus output) length. If we ignore "diagonalization based results", the following barriers are well recognized:

a. To establish a nonlinear lower bound on time.

b. To establish a nonlogarithmic lower bound on space.[1]

c. To establish a nonlogarithmic lower bound on depth (= parallel time).

Having recognized these barriers, it might seem wise to see if we can at least show that for some problem we cannot simultaneously achieve (say) linear time and logarithmic space. Such a result already appears in Cobham [4], where he shows that for recognizing the set of perfect squares (or for recognizing $\{w \$ w^R\}$) we must have $T \cdot S = \Omega(n^2)$ for any computational device (including a multitape T·M.) having a

* Received by the editors July 28, 1980, and in final form May 4, 1981.
† Department of Computer Science, University of Toronto, Toronto, Ontario, Canada M5S 1A1.
[1] That is, prove space is not $O(\log n)$.

separate *one head-read only input tape.* Here T = number of steps, S = "capacity" = $\log_2$ (number of configurations the machine enters when processing all strings of length n). The concept of "capacity" introduced by Cobham seems to capture just that property of space which lends itself to lower bound analysis. But whereas we accept a capacity lower bound on one of Cobham's general machines to be an "intrinsic" lower bound (i.e., independent of the choice of any reasonable computational models) on space requirements, we cannot say that a $T \cdot S = \Omega(n^2)$ lower bound has the same intrinsic quality, because of the restriction of having only one input head. More specifically, by easily adapting Cobham's argument (based on Hennie's [5] crossing sequence technique), Tompa [6] shows that sorting m numbers, each of length $\log m$ bits (hence $n = m \log m$), requires $T \cdot S = \Omega(n^2)$. But the proof literally states and shows that merging two lists of m sorted numbers would require the same lower bound. But for merging, the use of (say) two input heads would trivially (via a linear merge) permit a simultaneous linear time and logarithmic space merge. We are then led to the following question: Given k "random access" input heads, can we sort (say on a multitape T · M. or unit cost RAM) in simultaneous linear time and logarithmic space? The main result of this paper shows that indeed this is not possible. In fact we will establish a lower bound analogous to (and based upon) the lower bound of $T \cdot S = \Omega(n^2)$ established for sorting in the structured context of "branching programs" by Borodin, et al. [7]. Specifically we show $T \cdot S = \Omega(N^2/\log N)$, N the number of inputs and $N = \Omega(n/\log n)$ where n is the input length. To the best of our knowledge this is a unique result in that it establishes a lower bound (without diagonalization) on a completely unrestricted general model of computation. Unfortunately, we have not yet been able to establish a similar bound for a set recognition problem and we should also note that our methods do not appear applicable to Knuth's [8] problem of in situ sorting.

2. The formal model and an outline of the proof. In a general model of computation, we might be able to solve a given problem by processing the input string in a manner which is completely outside the mathematical domain within which the problem has been defined. For example, consider solving for the existence of a path on a graph by using Strassen's matrix multiplication algorithm and modular arithmetic (see Fischer and Meyer [9]). It seems almost impossible to make sense out of the individual bit operations in terms of the original problems.

The "fortunate" fact for sorting is that such a problem, with its explicit requirement for "ongoing progress" (in the sense of having to output ranks) allows us to enjoy a structured view of the computation even though we are working within a general computational model. Indeed we shall try to mimic the proof for the structured case [7]. That proof was based on the following intuitive idea: if we don't compare many elements, then we can't know the ranks of many elements for many input permutations. We will need a somewhat more involved argument to show an analogous statement for the general model.

Before discussing the model, we should define the problem formally. We consider an input of the form $x_1 \# x_2 \# \cdots \# x_N$ where each x_i is an integer in $[1, N^2]$ and is coded in binary. Hence the total length of the input is $O(N \log N)$. The sorting problem is to output a sequence of distinct pairs $i_1, r_1; i_2, r_2; \cdots; i_N, r_N$ such that x_{i_j} has rank r_j. (Without loss of generality we can assume that x_i's are distinct.) As in Borodin et al. [7] we can define the k-ranking subproblem; namely, output a sequence $i_1, r_1; \cdots; i_l, r_l, l \geqq k$, which correctly represents the ranks of l of the x_{i_j}'s. (The l indices i_j which are assigned ranks may be different for different input values.)

This definition of sorting is not standard. Usually, one requires outputting the x_i values in sorted order. However, for the model we are considering, any algorithm which sorts in this usual manner can be adapted to one which outputs pairs $\langle i_j, r_j\rangle$ by assuming that the index i has been concatenated onto x_i as the low order bits. It will follow that a TIME · SPACE lower bound in our framework for inputs in the range $[1, N^2]$ will imply the same lower bound in the usual setting for inputs in the range $[1, N^3]$.

Our formal models are as follows:

DEFINITION. An *R-way integer tree program* is an R-ary tree (hereafter called an R tree), where each branch is labelled by elements of $[1, R]$, and each internal node is labelled by some index i (referring to x_i). The interpretation is that if the computation has proceeded to an internal node labelled by "x_i" then it will continue to proceed along edge u iff $x_i = u$. Output takes place at the leaves. In particular, it should now be clear to say how a computation tree solves the k-ranking problem, or more generally how a computation tree solves the k-ranking problem for some subset I of the possible inputs. The time complexity of a computation tree is its depth; that is, the maximum number of times inputs are accessed in a computation. Since we assume all x_i are distinct, any branch which has two edges with the same label u for distinct x_i will be inaccessible. We assume these inaccessible paths have been pruned.

An R-way integer branching program (hereafter called an R branching program) is the nonstructured analogue of a comparison branching program [6]. Namely, it is a directed acyclic rooted graph with each nonsink node having out-degree R, with the R out edges labelled $1, 2, \cdots, R$. Without loss of generality, we can assume that the graph is in levels and that an edge out of a node at level l is directed to a node at level $l+1$. (See Tompa [6] for a discussion of the analogous assumption for comparison branching programs.) Outputs can now occur on any edge. The time complexity is again the depth and space = capacity = $\log_2$ (number of nodes in the graph). We can now state:

MAIN THEOREM. *Let τ be an R branching program for sorting N integers and let $R = R(N) = N^2$. Then $T \cdot S = \Omega(N^2/\log N)$ where T and S denote, respectively, the time and space complexity of τ.*

Before proceeding to the proof, we should comment briefly on the generality of the model. Suppose we have a general computational machine with k read-only, "random access" heads. It should be clear that by assuming $k = 1$ we will only slow down the machine by at most a constant factor (i.e., k). Our tree and branching programs assume that we will know an entire input x_i if we access any bit of that input. Hence, we are willing to ignore the $\log N$ factor it might cost to look at a given input. Each node of the computation graph represents a distinct state of the computation. Like Cobham [4], it is profitable for us to ignore completely how (and if) the storage can be represented and manipulated. Again, we are willing to ignore the time spent manipulating the storage between accesses of the input. We thus argue that our model and the time and space measures are sufficiently general that any lower bounds do reflect an intrinsic property of the function (sorting) being computed.

Having presented and justified the model, we can now informally sketch the proof. To do so, it is helpful to review the proof for the structured case [7]. The basic lemma in that proof states that a $\{<, >\}$ comparison tree program on n inputs of depth (time) t can solve the k ranking problem for at most $(t+1)^k(n-k)!$ input permutations. This lemma is applied with $k = S$ = space. Thus for any $c > 1$, by making $t = \alpha n$ for α sufficiently small, we can say that the S ranking problem has been solved for at most a fraction $(1/c)^S$ of the $n!$ possible input permutations. Now if τ is a

$\{<, >\}$ branching program for sorting, we consider the computation at the ith "stage" $= (i \cdot t)$th step, $i \leqq n/S$. In going from stage i to stage $i+1$, we can only correctly calculate S more ranks for at most $n!2^S \cdot (1/c)^S$ input permutations, since there are at most 2^S nodes at the ith stage, each of which can be considered the root of a tree program. Thus by an appropriate choice of $t = \Omega(n)$ we have $c > 2$ so that in going from stage i to stage $i+1$ we have computed more than S new ranks for at most a fraction $(1/d)^S$ of all possible input permutations. It follows that we will need at least $i = n/S$ stages to complete the computation, and hence $T = \Omega(n^2/S)$.

We want to establish the analog of the basic lemma, after which the rest of the proof follows exactly as before. We will show that for any $c > 1$, we can find suitable α such that any R-tree program ($R = N^2$) of depth $t = \alpha N$ can solve the $S \log N$ ranking problem for at most a fraction $(1/c)^S$ of the the possible inputs. In our case, that are $N!\binom{R}{N}$ possible input sequences $\langle x_1, \cdots, x_N \rangle$ since we are assuming distinct $\{x_i\}$.

In viewing the proof of the structured case, we can observe that *every* path in a computation tree can successfully solve the k ranking problem for at most a fraction $(t+1)^k/[n \cdot (n-1) \cdots (n-k+1)]$ of the permutations following that path. In our case, we can see that some short paths can be very successful; indeed if we discover that some $x_i = 1$ (or $x_i = N^2$) on a given path, then we know the smallest (respectively, largest) element for every input sequence on that path. Moreover, if we find some $x_i = 2$ (and no x_j seen so far is equal to 1) we still have a pretty good chance if we guess that x_i is the smallest element. But, we can also see intuitively that our chance of guessing correctly as to which is the smallest element starts to decrease if we have only seen a few not so small numbers.

So this will be our approach for establishing the analogous main lemma: We assert that, with sufficiently high probability, at a leaf of an R-tree program the elements that we have seen on this path will be "spread out" in such a way that there is only a small probability (i.e., for only a small fraction of all possible input sequences) that we will correctly output S ranks.

3. The proof of the main lemma. Throughout this section we will be considering R tree programs τ such that each leaf θ of τ is labelled with a ranking sequence $i_1, r_1; \cdots; i_l, r_l$, where $l = l_\theta$ may depend on the leaf θ. We say τ *solves the m ranking problem for an input sequence* $\langle x_1, \cdots, x_N \rangle$ provided this input leads to a leaf θ for which $l_\theta \geqq m$, and all l_θ ranks are correctly specified (i.e., x_{i_j} is the r_jth smallest input, $1 \leqq j \leqq l_\theta$). The following notation will be maintained: $t \leqq \frac{1}{2}N$ is the depth of the R tree program τ (we may asssume all paths in τ have length t by extending shorter ones if necessary), $N \geqq 2$ is the number of input elements, k is a positive integer satisfying $2f(k) \leqq N$, and $f(k)$ stands for $k\lceil \log N \rceil$. We will see that $R = R(N) = N^2$ is sufficiently large for our purposes, and since all results hold a fortiori for larger R, we will assume $R = N^2$. Our proofs will be formulated in the language of probability theory and we will speak of a random input in the sense that any of the $N!\binom{R}{N}$ possible input sequences are considered to be equally likely.

We are now ready to state the main lemma, which says that any sufficiently shallow R tree program (regardless of its capacity) cannot output many ranks correctly.

LEMMA 1. *For all $c > 0$ there is an $\alpha > 0$ such that for all τ with $t \leqq \alpha N$ and N sufficiently large, and for all k with $f(k) \leqq t$, the set I of inputs for which τ solves the $2f(k)$ ranking problem satisfies $\#I/(N!\binom{R}{N}) \leqq (1/c)^k$. Restated: with probability at most $(1/c)^k$, τ correctly outputs $2f(k)$ or more ranks for a random input.*

DEFINITION. A set $S=\{x_{i_1}, \cdots, x_{i_t}\}$ of inputs is $\langle \rho, k\rangle$ *spread out* if for every subset $S' \subseteq S$ with $\#S' = f(k)$ there is a subset $\{y_1, \cdots, y_k\}$ (listed in increasing order) of S' such that $y_{j+1} - y_j - 1 \geqq \rho$, for $0 \leqq j \leqq k$. (Here $y_0 = 0$ and $y_{k+1} = R+1$.)

LEMMA 2. *For all integers $\beta > 0$ there is an $\alpha > 0$ such that, for all τ with $t \leqq \alpha N$ and all k with $f(k) \leqq t$, $P[\tau, k, \beta] \leqq (1/N)^k$, where $P[\tau, k, \beta]$ is the probability that a random input $\langle x_1, \cdots, x_N\rangle$ to τ will follow a path along which the accessed input elements are not $\langle \beta R/N, k\rangle$ spread out.*

LEMMA 3. *For all $d > 0$ there is an integer $\beta > 0$ such that for all τ with N sufficiently large and for all k with $2f(k) \leqq N$ if the accessed input elements $\langle x_{i_1}, \cdots, x_{i_t}\rangle$ at a leaf θ of τ are $\langle \beta R/N, k\rangle$ spread out and the ranking sequence labelling θ contains at least $2f(k)$ ranks, then the fraction of those inputs leading to θ that are correctly ranked is at most $(1/d)^k$.*

Lemma 1 follows from Lemmas 2 and 3 as follows. Choose $d \geqq 2c$ in Lemma 3 to get β and apply Lemma 2 to get α. By Lemma 2 it does no harm to assume all leaves whose accessed inputs are not spread out always correctly solve the $2f(k)$ ranking problem, and the remaining leaves either output fewer than $2f(k)$ ranks or (by Lemma 3) are correct for too few inputs.

Proof of Lemma 2. Every leaf θ of τ uniquely determines a t-tuple $(x_{i_1}, \cdots, x_{i_t})$ of accessed elements, written in the order in which they are accessed on the path to θ. Conversely, every t-tuple of distinct integers in the interval $[1, R]$ uniquely determines a leaf. Thus there is a one to one correspondence between leaves and t-tuples, and exactly a $t!$ to one correspondence between leaves and sets of t distinct integers from $[1, R]$. Further, any two leaves have the same number of input sequences $(x_1, \cdots, x_N)$ leading to them. Therefore $P[\tau, k, \beta]$ is just that fraction of sets of t distinct integers from $[1, R]$ which are not $\langle \beta R/N, k\rangle$ spread out.

Divide the interval $[1, R]$ into N equal subintervals called *bins* of length N each (recall $R = N^2$). Let $\tilde{P}[t, N, k, \delta]$ be the probability that, when t balls are drawn (without replacement) from an urn of R balls numbered $1, 2, \cdots, R$, there exists some set of $f(k)$ of the drawn balls which lie in at most δ bins[2]. We claim that $\tilde{P}[t, N, k, \delta]$ is an upper bound on $P[\tau, k, \beta]$ where $\delta = k(\beta+1)+\beta$. For, if S is the set of t drawn balls and if every subset $S' \subseteq S$ of $f(k)$ balls lies in $\delta+1$ or more bins $B_1, \cdots, B_{\delta+1}$ (listed in the order in which these intervals occur in $[1, R]$), then we can choose one ball from each of the k bins $B_{j(\beta+1)}$, $1 \leqq j \leqq k$, to form the required subset $\{y_1, \cdots, y_k\}$ in the definition of $(\beta R/N, k) = (\beta N, k)$ spread out. This is because at least β bins lie entirely to the left of y_1, at least β bins lie entirely to the right of y_k, and any two adjacent y_j's are separated by at least β bins (β bins equals βN elements).

To estimate $\tilde{P}[t, N, k, \delta]$, let $p(b_i)$ be the probability that a particular bin B_i has at least b_i balls (after t are drawn). We claim that $\prod_{i=1}^{\delta} p(b_i)$ is an overestimate of the probability that a particular set of δ bins $B_1, \cdots, B_\delta$ get packed (respectively) with at least $b_1, \cdots, b_\delta$ balls. This is because the condition that a set of bins has some minimum number of elements can only decrease the probability that a particular bin has at least b_i elements. Hence

$$\tilde{P}[t, N, k, \delta] \leqq \sum_{\substack{(b_1, \cdots, b_\delta) \\ \sum b_i = f(k) \\ b_i \geqq 0}} \binom{N}{\delta} \prod_{i=1}^{\delta} p(b_i).$$

[2] Here is the essential place that the $\log N$ factor in our main result $T \cdot S = \Omega(N^2/\log N)$ enters the proof. Specifically, we cannot assert that $\tilde{P}$ would be sufficiently small if $f(k)$ were $O(k)$ rather than $k \log N$.

Here $\binom{N}{\delta}$ gives the number of ways to choose a set of crowded bins, and the summation represents the number of ways to pack a set of crowded bins.

We claim that for all $c \geqq 1$ there is $\alpha > 0$ such that $p(b) \leqq (1/c)^b$ (where $t \leqq \alpha N$).

Proof. The probability that a particular bin has exactly l balls is given by

$$\frac{\binom{N}{l}\binom{N^2-N}{t-l}}{\binom{N^2}{t}} \leqq \frac{N^l(N^2-N)^{t-l}}{(N^2-t)^t} \cdot \frac{t!}{l!(t-l)!} \leqq \left(\frac{2}{N}\right)^l \cdot t^l = \left(\frac{2t}{N}\right)^l \leqq (2\alpha)^l.$$

Thus

$$p(b) \leqq \sum_{l=b}^{t} (2\alpha)^l < \frac{(2\alpha)^b}{1-2\alpha} \leqq \left(\frac{1}{c}\right)^b \quad \text{for } \alpha = \frac{1}{4c},$$

assuming $b \geqq 1$. The claim is obvious if $b = 0$.

We thus have

$$\tilde{P}[t, N, k\delta] \leqq \sum_{b_1+\cdots+b_\delta = f(k)} \binom{N}{\delta} \prod_{i=1}^{\delta} p(b_i)$$

$$\leqq (f(k)+1)^\delta N^\delta \left(\frac{1}{c}\right)^{f(k)}$$

$$(\text{since } f(k) < N, f(k) = k\lceil \log N \rceil) \quad \leqq N^{2\delta}\left(\frac{1}{c}\right)^{k\lceil \log N \rceil}$$

$$(\text{for } \delta = k(\beta+1)+\beta) \quad \leqq N^{2k(\beta+1)+2\beta}\left(\frac{1}{N^{\log c}}\right)^k$$

$$\leqq \left(\frac{1}{N}\right)^k \quad \text{for sufficiently large } c. \quad \square$$

Proof of Lemma 3. Let $\{x_{i_1}, \cdots, x_{i_t}\}$ be the input elements accessed on the path to θ. Suppose at θ the labels assert that x_{j_ν} has rank r_ν for $1 \leqq \nu \leqq 2f(k)$. Note that we are not necessarily implying that any $x_{j_\nu} \in \{x_{i_1}, \cdots, x_{i_t}\}$ but, intuitively, one would expect a better chance at "guessing" the rank of an element which has been seen. Suppose that fewer than half of the indices for which θ assigns ranks are among the set $\{i_1, \cdots, i_t\}$. Then there is a set S of $u \geqq k\lceil \log N \rceil$ indices i for which θ assigns a rank and whose corresponding value x_i can be anything in the set $\{1, 2, \cdots, R\} - \{x_{i_1}, \cdots, x_{i_t}\}$. In particular, all $u!$ possible orderings of the set $\{x_i | i \in S\}$ are possible and equally likely, and a necessary condition that θ rank them properly is that they be in the right order. Hence at most a fraction $1/u!$ of the inputs leading to θ are correctly ranked, and $1/u! \leqq (1/d)^k$ for sufficiently large N, since $u \geqq k \log N$.

The remaining case is that half or more (that is at least $k\lceil \log N \rceil = f(k)$) of the indices for which θ assigns ranks are among the set $\{i_1, \cdots, i_t\}$. Let S' be the set of inputs at these indices (so $\#S' \geqq f(k)$). Since $\{x_{i_1}, \cdots, x_{i_t}\}$ is $\langle \beta R/N, k\rangle$ spread out, there is a subset $\{y_1, \cdots, y_k\}$ of S' such that $y_{j+1} - y_j - 1 \geqq \beta R/N$, $0 \leqq j \leqq k$, where $y_0 = 0$, $y_{k+1} = R+1$. Let θ output the assertions that y_j has rank r_j, $1 \leqq j \leqq k$. These assertions are equivalent to saying that exactly $r_j - r_{j-1} - 1$ of the inputs lie in the open interval (y_{j-1}, y_j), $1 \leqq j \leqq k+1$, where we understand that $r_0 = 0$ and $r_{k+1} = N+1$. This in turn is equivalent to saying that exactly k_j of the inputs which θ *does not access* lie in the set $C_j = (y_{j-1}, y_j) - \{x_{i_1}, \cdots, x_{i_t}\}$, where $k_j = r_j - r_{j-1} - 1 - u_j$, and $u_j = \#(y_{j-1}, y_j) \cap \{x_{i_1}, \cdots, x_{i_t}\}$ (i.e., u_j is the number of inputs which θ knows to lie between y_{j-1} and y_j).

We have thus reduced our problem to a more traditional probability setting, namely that of the hypergeometric distribution (see Feller [10, p. 43]). We have a population of size $n = R - t$, made up of n_i elements of "color i" (i.e., member of the set C_i), $1 \leqq i \leqq l = k+1$. We seek an upper bound on the probability

$$p_{k_1 \cdots k_l} = \frac{\binom{n_1}{k_1}\binom{n_2}{k_2}\cdots\binom{n_l}{k_l}}{\binom{n}{r}} \tag{1}$$

that a sample (without replacement) of size $r = N - t = \sum_{i=1}^{l} k_i$ will contain *exactly* k_i elements of color, i, $1 \leqq i \leqq l$. The required bound is given by Lemma 4 below. For our application we have $rn_i/n = (N-t)(\#C_i)/(R-t) \geqq (N-t)(\beta R/N - t)/(R-t)$. But $t \leqq \frac{1}{2}N$ and furthermore[3] $R = N^2$ so that $\beta R/N - t \geqq \frac{1}{2}\beta R/N$ for $\beta \geqq 1$. Thus $rn_i/n \geqq \beta/4$ since $N - t \geqq \frac{1}{2}N$. Further $l - 1 = k \leqq N/\log N$. Hence the constraints on r, n, n_i and l for Lemma 4 will be satisfied for sufficiently large N. Lemma 3 now follows from the following:

LEMMA 4. *For all $d > 0$ there exists a $\beta > 0$ such that if $rn_i/n \geqq \beta$ for $1 \leqq i \leqq l$, $r \geqq 2l\beta$, and $n \geqq 2r$, then for all $k_1, \cdots, k_l$ the hypergeometric distribution satisfies*

$$p_{k_1 \cdots k_l} \leqq \left(\frac{1}{d}\right)^l.$$

We need the following two lemmas to prove Lemma 4. Note that Lemma 5 states that the value of k_i for which the hypergeometric distribution is maximal is close to the expected value rp_i of the number of elements of color i obtained in r draws. If this optimal value were exactly rp_i, the proof of Lemma 4 would be substantially simpler.

LEMMA 5[4]. *The values of k_i in the maximal term of the hypergeometric distribution $p_{k_1 \cdots k_l}$ satisfy*

$$\frac{rp_i}{1 + l/n} - 1 < k_i < rp_i + (l-1)p_i + 1, \tag{2}$$

where $p_i = n_i/n$, $1 \leqq i \leqq l$.

Proof. For any pair (i, j) of distinct indices we calculate the ratio

$$\frac{p_{\cdots k_i+1, \cdots, k_j-1 \cdots}}{p_{k_1, \cdots, k_l}} = \frac{(n_i - k_i)k_j}{(k_i+1)(n_j - k_j + 1)}.$$

A necessary condition for $p_{k_1,\cdots,k_l}$ to be maximal is that the numerator does not exceed the denominator, or $(n_i - k_i)k_j \leqq (k_i+1)(n_j - k_j + 1)$. If we divide by n and rearrange this becomes

$$p_i k_j \leqq p_j k_i + p_j + \frac{k_i - k_j + 1}{n}. \tag{3}$$

If we sum (3) over all $j \neq i$ and use the identities $\sum p_i = 1$ and $\sum k_i = r$, then we obtain the left half of (2). Similarly, if we sum (3) over all $i \neq j$ we obtain the right-hand side of (2). □

[3] This is the only place we need assume that R is as large as N^2.

[4] Feller [10, p. 171, Exercise 28] states a similar result for the multinomial distribution. Our proof is suggested by Feller's hints.

LEMMA 6. *For all $\varepsilon > 0$ there is a z_ε such that for all θ and for all $z \geqq z_\varepsilon|\theta|$*

$$\left(1+\frac{\theta}{z}\right)^z \geqq (1+\varepsilon)^{-|\theta|} e^\theta.$$

Note that z_ε does not depend on θ.

Proof. From elementary calculus we have $\lim_{z\to\infty}(1+\theta/z)^z = e^\theta$. Setting $\theta = 1$ and -1 we conclude $(1+1/z)^z \geqq (1+\varepsilon)^{-1} e$ and $(1-1/z)^z \geqq (1+\varepsilon)^{-1} e^{-1}$ for $z \geqq z_\varepsilon$. Setting $z = z'|\theta|$ we have $(1+\theta/z)^z = (1+\theta/(z'|\theta|))^{z'|\theta|} \geqq (1+\varepsilon)^{-|\theta|} e^\theta$ for $z' \geqq z_\varepsilon$; i.e., for $z \geqq z_\varepsilon|\theta|$. □

Proof of Lemma 4. We have

$$p_{k_1\cdots k_l} = (\prod n_i!) r! (n-r)!/[n! \prod (k_i!(n_i-k_i)!)].$$

Stirling's approximation implies that $1/C_0 \leqq \sqrt{2\pi m}(m/e)^m/m! \leqq C_0$ for some constant $C_0 \geqq 1$ and all $m \geqq 1$. Using this approximation for each factorial, and substituting $rp_i + \theta_i$ for k_i, $1 \leqq i \leqq l$, where $p_i = n_i/n$ and θ_i has been chosen to maximize (1), we obtain

$$p_{k_1\cdots k_l} \leqq ABC_0^{3l+3}, \tag{4}$$

where

$$A = \sqrt{\frac{(\prod n_i) r(n-r)}{(2\pi)^{l-1} n \prod (rp_i+\theta_i)((n-r)p_i-\theta_i)}} \tag{5}$$

and

$$B = \frac{(\prod n_i^{n_i}) r^r (n-r)^{n-r}}{n^n \prod [(rp_i+\theta_i)^{rp_i+\theta_i}((n-r)p_i-\theta_i)^{(n-r)p_i-\theta_i}]}. \tag{6}$$

For (5) and (6) we have used the identity $n_i - k_i = (n-r)p_i - \theta_i$. Notice that all occurrences of e cancel, since $\sum n_i = n$.

Since $\sum p_i = 1$ and $\sum k_i = r$, it follows that $\sum \theta_i = 0$. This fact can be used to verify that if B' is the number obtained by substituting 0 for the two occurrences of θ_i in the denominator (but not in the exponents) in the expression for B, then $B' = 1$. Thus if we multiply and divide the denominator of (6) by $\prod [(rp_i)^{rp_i+\theta_i}((n-r)p_i)^{(n-r)p_i-\theta_i}]$ we can simplify and obtain

$$B = \left[\prod_{i=1}^{l}\left(\left(1+\frac{\theta_i}{rp_i}\right)^{rp_i+\theta_i}\left(1-\frac{\theta_i}{(n-r)p_i}\right)^{(n-r)p_i-\theta_i}\right)\right]^{-1}.$$

Now we apply Lemma 6 and use the fact that $(1+\theta/z)^\theta \geqq 1$ for all $z > 0$ and all θ to obtain $B \leqq (1+\varepsilon)^{2\sum|\theta_i|}$, provided

$$rp_i \geqq z_\varepsilon|\theta_i| \quad \text{and} \quad (n-r)p_i \geqq z_\varepsilon|\theta_i|, \qquad 1 \leqq i \leqq l. \tag{7}$$

By Lemma 5, we have $|\theta_i| \leqq lrp_i/(n+l) + (l-1)p_i + 2$, so $|\theta_i| \leqq 2lp_i + 2$. By assumption, we have $rp_i \geqq \beta$, $(n-r) \geqq r$ and $r \geqq 2l\beta$. Hence

$$(n-r)p_i \geqq rp_i \geqq \frac{\beta|\theta_i|}{4}, \tag{8}$$

so the provisos (7) are satisfied for $\beta \geqq 4z_\varepsilon$. Now summing the bound $|\theta_i| \leqq 2lp_i + 2$, we obtain $\sum|\theta_i| \leqq 4l$, so

$$B \leqq (1+\varepsilon)^{8l}. \tag{9}$$

It remains to estimate A from (5). We rewrite the product $\prod$ in the denominator as the product of five factors:

$$\prod (rp_i) \cdot \prod \left(1+\frac{\theta_i}{rp_i}\right) \cdot \prod n_i \cdot \left(1-\frac{r}{n}\right)^l \cdot \prod \left(1-\frac{\theta_i}{(n-r)p_i}\right).$$

To estimate the first factor $\prod rp_i$, notice that $\sum rp_i = r$, and each $rp_i \geqq \beta$ by assumption. With these constraints, this product obtains its minimum when all but one of the factors are as small as possible (namely β). Thus

$$\prod (rp_i) \geqq \beta^{l-1}(r-(l-1)\beta) > \tfrac{1}{2} r \beta^{l-1}.$$

From (8), we have $1+\theta_i/(rp_i) \geqq \frac{1}{2}$ for $\beta \geqq 8$, so $\prod (1+\theta_i/(rp_i)) \geqq 2^{-l}$. The same bound applies to the fifth factor and (since $n \geqq 2r$) to the fourth. The third factor cancels with the numerator. Thus

$$A \leqq [(2\pi)^{l-1}\beta^{l-1}2^{-3l+1}]^{-1/2} \leqq \left(\frac{1}{c}\right)^l$$

for any c and $\beta \geqq \beta_c$. Lemma 4 follows from this, (4) and (9). □

4. Proof of the main theorem. As indicated earlier in the paper, we will follow the general argument used in the structured case. As in § 3, we again assume $R = R(N) = N^2$. We let T denote the time (that is, the depth) of a branching program, and let S denote the space (that is, the capacity $= \log_2 \#$ nodes). Since we must clearly (by the simplest adversary argument) have $T \geqq N$ and $S \geqq \log_2 T$, we have $S \geqq \log_2 N$. Let us restate the main theorem.

THEOREM. *Let τ be an R branching program for sorting N integers. Then* $T \cdot S = \Omega(N^2/\log N)$.

Proof. Letting $c = 4$, use Lemma 1 to obtain α for N sufficiently large. We will now consider τ in stages, where every stage represents $t = \lfloor \alpha N \rfloor$ steps.

For $1 \leqq i \leqq N/(2f(S))$, let P_i be the fraction of input sequences for which τ has output at least $2if(S)$ ranks by the end of the ith stage. We shall now prove

$$(*) \qquad P_i \leqq i(\tfrac{1}{2})^S.$$

For each node θ on the $(i \cdot t)$th level ($=$ end of stage i) let $P_{i,\theta}$ be the fraction of input sequences which lead to θ and for which τ outputs at least $2f(S)$ ranks during the $i+1$st stage. If we expand the part of the $(i+1)$st stage that is rooted at θ into an R tree, we see by Lemma 1 that (regardless of what has happened in earlier stages) $P_{i,\theta} \leqq (\frac{1}{4})^S$. Since there are at most 2^S nodes θ at level $i \cdot t$, we have $P_{i+1} \leqq P_i + \sum_\theta P_{i,\theta} \leqq P_i + 2^S \cdot (\frac{1}{4})^S$, or $P_{i+1} \leqq P_i + (\frac{1}{2})^S$. This inequality holds for $0 \leqq i \leqq N/(2(f(S))$, if we define $P_0 = 0$. The inequality $(*)$ now follows by induction on i.

Recall $f(S) = S\lceil \log N \rceil$. If $2f(S) > N$ then $S > N/2\lceil \log N \rceil$, so $ST = \Omega(N^2/\log N)$ in this case. If $2f(S) \leqq N$, then we can set $i = i_0 = \lfloor N/(2f(S)) \rfloor$ in $(*)$ to obtain (since $S \geqq \log N$) $P_{i_0} \leqq (N/(2f(S))) \cdot 1/N = 1/(2f(S)) < 1$. Hence at stage i_0 for some input, τ has output fewer than $2i_0 f(S) \leqq N$ ranks, so $T \geqq i_0 t = \lfloor N/(2f(S)) \rfloor \cdot \lfloor \alpha N \rfloor = \Omega(N^2/(S \log N))$ steps. □

5. Conclusion. In order to better appreciate the application of the main theorem, we offer the following example.

Let M be any machine (say, a unit cost RAM or vector machine with operations $+, -, \times, \div, \uparrow$) whose inputs are accessed from a random access *read-only* input device. We only insist that there is a bound on the number of inputs accessible on a given computation step. Choose any "fair" definition of space, e.g., space =

$\max_j \sum_{i=1}^{t} \lceil \log (r_i^j + 1) \rceil$ where r_i^j is the contents of register i at time j and t is the largest register used. For such a machine the theorem yields $T \cdot S = \Omega(N^2/\log N)$. And, of course, the same result holds for multidimensional Turing machines, etc.

Although the lower bound $T \cdot S = \Omega(N^2/\log N)$ established in this paper for a general model of computation differs by a log factor from the lower bound for the structured case [7], the upper bounds for the structured case apply unchanged. This is because a "structured algorithm" is a $\{<\ ,\ >\}$ branching program, and a comparison $x_i < x_j$ over the domain $[1, R]$ can be carried out on an R branching program in two time steps and $R+2$ nodes. However, in order to be sure that the time and space of the simulating program are of the same order as the time and space of the original program, it is necessary to assume $R = O(N^k)$ for some k. Under this assumption, the upper bound $T \cdot S = O(N^2 \log N)$, for $\Omega(\log N) \leqq S \leqq O(N)$ recently established by Frederickson [16] for a unit cost "structured" random access machine with suitable instructions applies to an R branching program. (Frederickson's bound generalizes the one in [7]). It is worth noting that for "unstructured" (i.e., general) random access machines, the upper bound can be extended, using radix sort, to the case $T = O(N)$ and $S = O(N \log N)$.

We thus have a $\log^2 N$ discrepancy in the upper and lower bounds. We note that we can improve on the upper bounds when $R = N + O(N)$, say by finding the missing elements.

It seems to us, however, that the discrepancy in the bounds is far less important than the need to establish analogous results for a set-recognition problem; for example, determining if $X \cap Y = \emptyset$. At the present time such a time-space result has not yet been established for the structured comparison model. We believe that our results suggest that proofs for the structured model may provide a framework for the general model. However, it must be noted that the less constructive variant of branching programs for "silent sorting" mentioned in Borodin et al. [7, Conclusion] becomes trivial in the general setting.

In retrospect, we can see that our methods are quite "brute-force". In particular, we do not make an essential use of an adversary. Rather what we have is basically a counting argument. Moreover, we do not make full use of the fact that space is limited throughout the computation; we only use the fact that it is restricted at certain points of the computation. We suspect that the set recognition problems will entail a more sophisticated argument.

A more general view of time-space complexity is captured in Cook's class SC [11], [12] (formerly PLOPS); that is, those problems for which there exist algorithms which run *simultaneously* in polynomial (sequential) time and $\log^k$ (for some k) space. Obviously, any problem (e.g., sorting, $X \cap Y = \emptyset$?, etc.) which is in log space, is also in SC. A central issue for computational complexity is to establish the conjecture (assuming it is true) that $\mathrm{P} \cap (\bigcup_k \mathrm{DSPACE}(\log^k)) \not\supseteq \mathrm{SC}$. Cook and Tompa (see Tompa [6]) show that the structured branching program model (with either $\{=, \neq\}$ or $\{<, =, >\}$ as the allowable comparisons) may provide a sufficiently general setting for this conjecture.

Another important direction for future work lies in the related (but apparently different) question of size vs. depth. The recent work of Pippenger [13], Ruzzo [14], and Dymond and Cook [15], has focused attention on the stability and importance of the class NC; that is, those problems for which there are algorithms which run *simultaneously* in polynomial size (= sequential operations) and $\log^k$ depth (= parallel time). Again, it is a central issue in complexity to establish the conjecture $P \cap (\bigcup_k$ parallel time $(\log^k)) \not\supseteq \mathrm{NC}$.

Motivated by the results of this paper, we would like to find a problem for which (say) size · depth $= \Omega(N^2)$. Sorting will not suffice since we can sort simultaneously in $\log^2$ depth and $N \log^2 N$ size using a Batcher sorting network. However, one is tempted to conjecture that any Boolean circuit for sorting which uses only $k \log N$ depth requires $cN^{1+\varepsilon}$ size where c and ε will depend upon k. The class of problems which are computable by a log depth, $N \log^k N$ size circuit is a class of practical importance. We suspect that it will be difficult to prove that a given problem does not belong to this class.

Acknowledgment. We sincerely thank Romas Aleliunas and Patrick Dymond for their many helpful suggestions.

REFERENCES

[1] A. AHO, J. HOPCROFT AND J. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] N. PIPPENGER AND L. G. VALIANT, *Shifting graphs and their applications*, J. Assoc. Comput. Mach. 23 (1976), pp. 423–432.

[3] A. BORODIN, *Structured vs general models in computational complexity*, presented at Internationales Symposium über Logik und Algorithmik zu Ehren von Professor Ernst Specker, Feb., 1980, Zurich, L'Enseignement Mathematique, to appear.

[4] A. COBHAM, *The recognition problem for the set of perfect squares*, Conference Record, IEEE 7th Annual Symposium on Switching and Automata Theory, 1966, pp. 78–87.

[5] F. HENNIE, *Crossing sequences and off-line Turing machine computations*, Conference Record IEEE Symposium on Switching Circuit Theory and Logical Design, 1965, pp. 179–190.

[6] M. TOMPA, *Time-space tradeoffs for straight-line and branching programs*, Tech. Rep. 122/78, Dept. Computer Science, Univ. of Toronto, July 1978.

[7] A. BORODIN, M. J. FISCHER, D. KIRKPATRICK, N. LYNCH, M. TOMPA, *A time-space tradeoff for sorting on non-oblivious machines*, Proc. IEEE 20th Annual Symposium on Foundations of Computer Science, Puerto Rico, Oct. 1979.

[8] D. E. KNUTH, *Mathematical analysis of algorithms*, Proc. of IFIP Congress 71, C. V. Freeman, ed., vol. 1, North-Holland, Amsterdam, 1972, pp. 19–27.

[9] M. FISCHER AND A. MEYER, *Boolean matrix multiplication and transitive closure*, Conference Record, IEEE 12th Annual Symposium on Switching and Automata Theory, 1971, pp. 129–131.

[10] W. FELLER, *An Introduction to Probability Theory and its Applications*, I, John Wiley, New York, 1968.

[11] S. COOK, *Deterministic CFL's are accepted simultaneously in polynomial time and log squared space*, Proc. ACM Symposium on Theory of Computing, 1979, pp. 338–345.

[12] ———, *Towards a complexity theory of synchronous parallel computation*, presented at Internationales Symposium über Logik und Algorithmik zu Ehren von Professor Ernst Specker, Feb. 1980, Zurich, L'Enseignement Mathematique, to appear.

[13] N. PIPPENGER, *On simultaneous resource bounds*, Proc. IEEE 20th Annual Symposium on Foundations of Computer Science, Puerto Rico, Oct. 1979, pp. 307–311.

[14] L. RUZZO, *On uniform circuit complexity*, Proc. IEEE 20th Annual Symposium on Foundations of Computer Science, Puerto Rico, Oct. 1979, pp. 312–318.

[15] P. DYMOND AND S. COOK, *Hardware complexity and parallel computation*, Proc IEEE 21st Annual Symposium on Foundations of Computer Science, Oct. 1980, pp. 360–372.

[16] G. N. FREDERICKSON, *Upper bounds for time-space trade-offs in sorting and selection*, Tech. Rep. CS-80-3, Dept. Computer Science, Pennsylvania State University, January 1980.

SIAM J. COMPUT.
Vol. 11, No. 2, May 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1102-0008 $01.00/0

THE RECOGNITION OF SERIES PARALLEL DIGRAPHS*

JACOBO VALDES,† ROBERT E. TARJAN‡ AND EUGENE L. LAWLER§

Abstract. We present a linear-time algorithm to recognize the class of vertex series-parallel (VSP) digraphs. Our method is based on the relationship between VSP digraphs and the class of edge series-parallel multidigraphs. As a byproduct of our analysis, we obtain efficient methods to compute the transitive closure and transitive reduction of VSP digraphs, and to test isomorphism of minimal VSP digraphs.

Key words. algorithms, complexity, graph decomposition, graph isomorphism, series-parallel graphs, scheduling, transitive closure

1. Introduction. In this paper we study a class of directed acyclic graphs arising in certain scheduling problems. In these problems, the tasks to be scheduled are subject to a partial order. The scheduling problems are NP-complete for an arbitrary partial order but have efficient algorithms if the partial order defines a vertex series-parallel (VSP) digraph ([LAW1], [LAW2], [MON], [SID]). These algorithms apply a "divide-and-conquer" approach to the recursive structure of VSP digraphs.

Our main result is a linear-time algorithm that determines whether an arbitrary digraph is VSP. The algorithm represents the structure of a VSP digraph in a concise form suitable for use by the scheduling algorithms mentioned above. Our method exploits the relationship between VSP digraphs and the class of edge series-parallel (ESP) multidigraphs ([ADA], [DUF], [RIO], [WAL], [WEI]), which arise in the analysis of electrical networks.

Our analysis allows us to prove a simple forbidden subgraph characterization of VSP digraphs. We are also able to give efficient algorithms to compute the transitive closure and transitive reduction of VSP digraphs, and to test two minimal VSP digraphs for isomorphism.

The remainder of this paper is divided into four sections. Section 2 provides the concepts and elementary facts used in the recognition procedure. Section 3 outlines the procedure, proves it correct, and describes in detail a linear-time implementation. Section 4 presents the forbidden subgraph characterization, and § 5 discusses some additional consequences of our work.

2. Basic concepts.

2.1. Graph-theoretic definitions. This section reviews the standard graph-theoretic concepts we shall employ. A *multigraph* $G=\langle V, E\rangle$ consists of a finite set of *vertices* V and a finite multiset of *edges* E. Each edge is a pair (v, w) of distinct vertices. If the edges of G are unordered pairs, then G is an *undirected multigraph*; if the edges are ordered pairs, G is a *directed multigraph* (*multidigraph*). If E is a set,

* Received by the editors May 22, 1980, and in revised form February 4, 1981. A preliminary version of this work was presented at the Eleventh Annual ACM Symposium on Theory of Computing, Atlanta, Georgia, 1979.

† Department of Electrical Engineering and Computer Science, Princeton University, Princeton, New Jersey 08544. The work of this author was partially supported by the National Science Foundation under grant MCS-75-22870 and the Office of Naval Research under contract N00014-76-C-0688.

‡ Computer Science Department, Stanford University, Stanford, California 94305. The work of this author was partially supported by the National Science Foundation under grant MCS-75-22870 and the Office of Naval Research under contract N00014-76-C-0688. Present address: Bell Laboratories, Murray Hill, New Jersey 07974.

§ Computer Science Division, University of California at Berkeley, Berkeley, California 94720. The work of this author was partially supported by the National Science Foundation under grant MCS-76-17605.

then G is a *graph*. The terms we define in the remainder of this section for graphs apply equally well to multigraphs.

If (v, w) is an edge of a graph G, then v and w are *adjacent* and (v, w) is *incident* to v and w. If G is a directed graph (*digraph*), then each edge (v, w) is an ordered pair which *leaves* v and *enters* w; v is a *predecessor* of w and w is a *successor* of v. The *degree* of a vertex v in a graph is the number of vertices adjacent to v; a vertex of degree zero is *isolated.* A vertex v in a digraph is a *source* if no edges enter v and a *sink* if no edges leave v.

A *path* of *length k* in a graph is a sequence of vertices $v_0, v_1, \cdots, v_k$ such that (v_i, v_{i+1}) is an edge for $0 \leqq i < k$. If $v_0 = v_k$ and $k \geqq 2$, the path is a *cycle.* A graph which contains no cycles is *acyclic.* The path *contains* vertices $v_0, v_1, \cdots, v_k$ and edges $(v_0, v_1), (v_1, v_2), \cdots, (v_{k-1}, v_k)$, and *avoids* all other vertices and edges.

A directed acyclic graph (*dag*) is *transitive* if for any two vertices v and w such that there is a path from v to w, either $v = w$ or (v, w) is an edge. The *transitive closure* $G_T = \langle V, E_T \rangle$ of a dag $G = \langle V, E \rangle$ is the dag such that $(v, w) \in E_T$ if and only if $v \neq w$ and there is a path from v to w in G.

An edge (v, w) in a dag is *redundant under transitive closure* or simply *redundant* if there is a path from v to w which avoids (v, w). A dag with no redundant edges is *minimal.* The *transitive reduction* of a dag G is the unique minimal dag having the same transitive closure as G (see [AGU]).

The *line digraph* of a digraph G is the digraph $L(G)$ having a vertex $f(e)$ for each edge e of G and an edge $(f(e_1), f(e_2))$ for each pair of edges e_1, e_2 in G of the form $e_1 = (u, v)$, $e_2 = (v, w)$.

A graph $G = \langle V_1, E_1 \rangle$ is a *subgraph* of another graph $G = \langle V_2, E_2 \rangle$ if $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$. For any subset S of vertices in a graph G, the subgraph *induced* by S is the maximal subgraph of G with vertex set S. A graph G contains a subgraph *homeomorphic* to a graph H if H can be obtained from G by a sequence of the following operations: (i) remove an edge; (ii) remove an isolated vertex; (iii) if a vertex v has degree two, delete v and replace the two edges (u, v), (v, w) incident to v by an edge (u, w).

2.2. Vertex series-parallel digraphs. We define the class of VSP dags in terms of the subclass of its minimal members. The dags in this subclass are called *minimal vertex series-parallel* (MVSP) and are defined recursively as follows:

DEFINITION 1 (*minimal vertex series-parallel dags*).

(i) The dag having a single vertex and no edges is MVSP.

(ii) If $G_1 = \langle V_1, E_1 \rangle$ and $G_2 = \langle V_2, E_2 \rangle$ are two MVSP dags, so are the dags constructed by each of the following operations:

(a) *Parallel composition*: $G_P = \langle V_1 \cup V_2, E_1 \cup E_2 \rangle$.

(b) *Series composition*: $G_S = \langle V_1 \cup V_2, E_1 \cup E_2 \cup (T_1 \times S_2) \rangle$, where T_1 is the set of sinks of G_1 and S_2 is the set of sources of G_2.

We define the class of VSP dags as follows:

DEFINITION 2 (*vertex series-parallel dags*). A dag is VSP if and only if its transitive reduction is MVSP.

Figure 1 shows the construction of an MVSP dag by a sequence of series and parallel compositions. Figure 2 shows a VSP dag whose transitive reduction is the MVSP dag of Fig. 1.

An MVSP dag can be represented in a natural way by a binary tree as shown in Fig. 3. This tree is constructed by (i) associating a tree of one node with the MVSP dag having one vertex and no edges, and (ii) using the rules of Fig. 4 to build larger

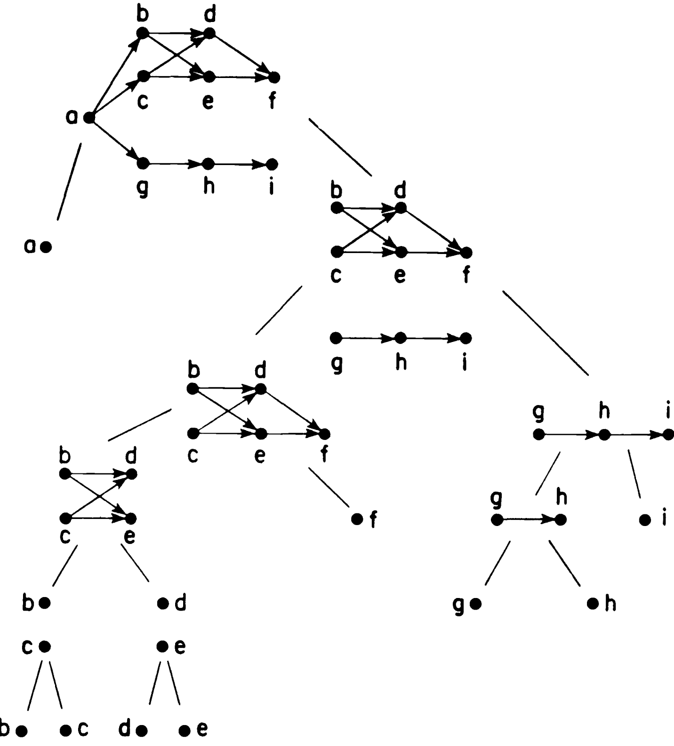


FIG. 1. *Construction of an* MVSP *dag by series and parallel compositions.*

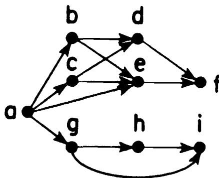


FIG. 2. *A* VSP *dag.*

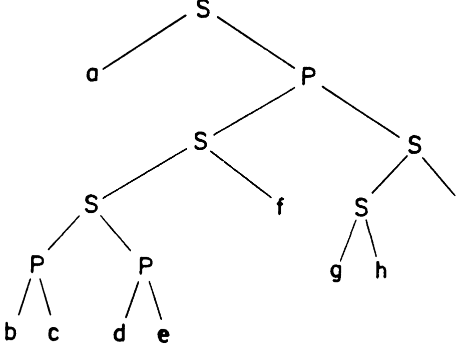


FIG. 3. *Binary decomposition tree representing the* MVSP *dag of Figure* 1.

FIG. 4. *Rules to construct T_S and T_P (the binary decomposition trees of G_S and G_P in Definition* 1) *from T_1 and T_2 (the binary decomposition trees of G_1 and G_2).*

trees from smaller ones as the process of building the MVSP dag by series and parallel compositions progresses. We call such a tree a *binary decomposition tree*. Each external node of the tree represents a vertex in the MVSP dag; each internal node is labeled S or P and represents the series or parallel composition of the MVSP dags represented by the subtrees rooted at the children of the node. A binary decomposition tree thus provides a concise description of the structure of an MVSP dag.

Note that several nonisomorphic binary trees may represent the same dag. Since parallel composition is commutative, the children of any P node may be reordered without changing the MVSP dag represented. Furthermore, both series and parallel composition are associative, which allows the ambiguity typical of unparenthesized infix expressions.

Any dag G induces a partial order on its vertices, defined by $v < w$, if and only if there is a path from v to w in G. Since this definition depends only on the paths in G, the partial order induced by the transitive closure of G or by the transitive reduction of G is the same as that induced by G. The partial orders induced by MVSP and VSP dags are of a special type that plays a role in our algorithm.

Any partial order on a set is the intersection of several total orders on the same set; the minimum number of total orders needed to define the partial order in this fashion is the *dimension* of the partial order. For instance, the MVSP dag in Fig. 1 is two-dimensional, since there is a path from v to w in the dag if and only if v appears before w in both of the following total orders; $a\,b\,c\,d\,e\,f\,g\,h\,i$; $a\,g\,h\,i\,c\,b\,e\,d\,f$. Indeed, *any* partial order induced by an MVSP dag is at most two-dimensional. We shall prove this fact in § 3 after describing how the recognition procedure uses it.

2.3. Edge series-parallel multidigraphs. The relationship between MVSP dags and the class of *edge series-parallel* (ESP) *multidigraphs* plays a central role in our recognition algorithm. In this section we review the relevant properties of ESP multidigraphs (sometimes called two-terminal series-parallel multidigraphs). We define the class of ESP multidigraphs recursively as follows:.

DEFINITION 3 (*edge series-parallel multidigraphs*).

(i) A digraph consisting of two vertices joined by a single edge is ESP.

(ii) If G_1 and G_2 are ESP multidigraphs, so are the multidigraphs constructed by each of the following operations:

(a) *Two-terminal parallel composition*: Identify the source of G_1 with the source of G_2 and the sink of G_1 with the sink of G_2.

(b) *Two-terminal series composition*: Identify the sink of G_1 with the source of G_1.

Figure 5 illustrates the construction of an ESP multidigraph using the operations in Definition 3. Note that every ESP multidigraph is acyclic, since the one-edge ESP multidigraph is acyclic, and the operations of Definition 3 do not create any cycles.

An ESP *multidigraph* is a multigraph formed from an ESP multidigraph by ignoring edge directions. The ESP multigraphs have been extensively studied ([ADA], [DUF], [RIO], [WAL], [WEI]) because of their use in modeling electrical networks. The properties of ESP multidigraphs that we need are simple extensions of known

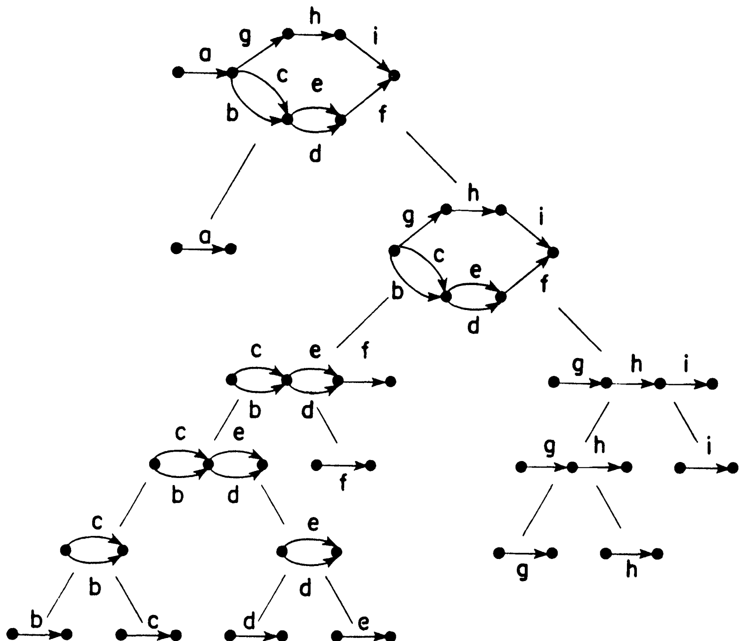


FIG. 5. *Construction of an* ESP *multidigraph by two-terminal series and parallel compositions.*

properties of ESP multigraphs, and we shall provide only summary proofs. A complete description of the relationship between ESP multidigraphs and ESP multigraphs appears in [VAL].

The similarity of Definitions 1 and 3 suggests a vertex-edge duality between MVSP dags and ESP multidigraphs. The following lemma expresses this duality.

LEMMA 1. *An acyclic multidigraph with a single source and a single sink is* ESP *if and only if its line digraph is an* MVSP *dag.*

Proof. By induction on the number of edges in the multidigraph using two facts:

(i) The line digraph of the one-edge ESP multidigraph is the one-vertex MVSP dag.

(ii) The line digraph of the two-terminal series (parallel) composition of G_1 and G_2 is the series (parallel) composition of the line digraph of G_1 and the line digraph of G_2. □

Everything we have said about binary decomposition trees for MVSP dags applies almost verbatim to ESP multidigraphs. In general, if T is a binary decomposition tree of an ESP multidigraph G, then T also represents the corresponding MVSP dag $L(G)$. For instance, the binary decomposition tree in Fig. 3 represents both the ESP multidigraph in Fig. 5 and the MVSP dag in Fig. 1. The external nodes of the tree represent the edges of the ESP multidigraph and the vertices of its line digraph.

The following lemma gives an alternative characterization of ESP multidigraphs based on the reductions in Fig. 6.

LEMMA 2. *A multidigraph is* ESP *if and only if it can be reduced to the one-edge* ESP *multidigraph by a sequence of series and parallel reductions.*

Proof. This lemma corresponds to a result of Duffin [DUF] for ESP multigraphs and follows by an easy induction (on the number of reductions applied for the "if" part, and on the number of edges for the "only if" part). For details see [DUF] or [VAL]. □

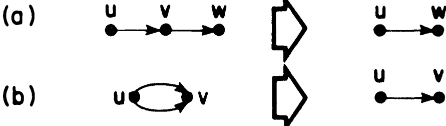


FIG. 6. (a) *Series reduction* (*requires that v have in-degree one and out-degree one*). (b) *Parallel reduction.*

From Lemma 2 we obtain an efficient procedure to recognize ESP multidigraphs. Given a multidigraph G, we repeatedly apply series and parallel reductions until no reduction is possible. If the result is a graph consisting of a single edge, then the original graph G is ESP. If not, G is not ESP. The validity of this procedure depends upon the fact that series and parallel reductions have the *Church–Rosser property* ([ROS], [SET]): if reductions are applied in any order until no reduction is possible, the result is a unique graph independent of the specific reductions applied. Harary, Krarup, and Schwenk [HKS] and Walsh [WAL] prove that the corresponding reduction system for undirected graphs is Church–Rosser; the proof extends easily to the directed case (see [VAL]).

If the reduction process succeeds, we can obtain as a byproduct a decomposition tree of the original ESP multidigraph. We associate a label consisting of a binary tree with each edge of the multidigraph being reduced. Initially the label of each edge is a single-node binary tree. As the reduction process proceeds we use the rules of Fig. 7 to update the edge labels. The label of the last edge remaining after all reductions is the binary decomposition tree of the original multidigraph, as can be proved by an easy induction (see [VAL]).

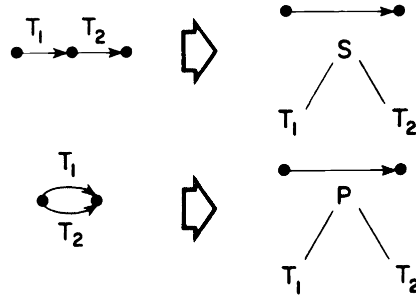


FIG. 7. *Computing the label of an edge introduced by a series or parallel reduction.*

3. The VSP recognition algorithm. We are now able to outline our procedure for recognizing VSP dags and to prove it correct. The input to the algorithm is a dag G. If G is VSP, the algorithm answers YES and produces a binary decomposition tree for G. If G is not VSP, the algorithm answers NO.

Recognition procedure for the class of VSP *dags.*

Step 1. (*Compute the pseudo-transitive reduction of* G.) Given $G=\langle V, E\rangle$, partition E into E_T and E_M such that, if G is VSP, then $G_M=\langle V, E_M\rangle$ is the transitive reduction of G (G_M is thus MVSP). If G is not VSP, G_M may still be MVSP. (We have to pay this price in order to be able to implement this step in linear time, since it is unlikely that a linear-time algorithm exists for transitive reduction of arbitrary dags [AGU].)

Step 2. (*Compute the line digraph inverse of* G_M.) Test whether G_M satisfies a condition (satisfied by all MVSP dags) that guarantees the existence of a line digraph inverse $L^{-1}(G_M)$. If G_M does not satisfy the condition, answer NO and stop. Otherwise, compute a multidigraph $L^{-1}(G_M)$ such that $L(L^{-1}(G_M)) = G_M$. By Lemma 1, G_M is MVSP if and only if $L^{-1}(G_M)$ is ESP.

Step 3. (*Test whether* $L^{-1}(G_M)$ *is* ESP.) Apply Lemma 2 to determine whether $L^{-1}(G_M)$ is ESP. If not, answer NO and stop. Otherwise compute a binary decomposition tree T for $L^{-1}(G_M)$ as an ESP multidigraph. T is also a decomposition tree for G_M as an MVSP dag.

Step 4. (*Test whether* G_M *is the transitive reduction of* G.) Use T to compute two total orders whose intersection defines the partial order $<$ on G_M. Use these orders to test each edge in E_T for redundancy. If every edge in E_T is redundant, answer YES, output T, and stop. Otherwise answer NO and stop.

The following argument shows that this procedure is correct. If G is VSP, then G_M will be MVSP and will pass the test of Step 2. If G_M is MVSP, then by Lemma 1 $L^{-1}(G_M)$ will be ESP and will pass the test of Step 3. Step 4 will certify that Step 1 correctly performed the transitive reduction of G and the algorithm will answer YES.

If, on the other hand, G is not VSP, then either G_M as constructed by the algorithm will not be MVSP, or G_M will not be the transitive reduction of G. In the former case the algorithm will answer NO in either Step 2 or Step 3, since by Lemma 1 $L^{-1}(G_M)$ cannot be ESP if G_M is not MVSP. In the latter case the algorithm will answer NO in Step 4.

In order to verify the linearity of the recognition procedure, we must provide more details of the implementation. The remaining parts of this section describe how to implement each step of the algorithm so that it runs in linear time.

3.1. The transitive reduction of VSP dags. Step 1 requires a method to compute the transitive reduction of any VSP dag; the method may do anything to a non-VSP dag. Our method uses the following functions defined on a dag $G = \langle V, E \rangle$.

DEFINITION 4 (*level, jump, and minimum jump functions*). The *level* function L_G is the function from vertices to non-negative integers such that $L_G(v)$ is the length of the longest path from a source of G to v. Note that $L_G(v) = 0$ if v is a source.

The *jump* function J_G is the function from edges to positive integers defined by $J_G((u, v)) = L_G(v) - L_G(u)$.

The *minimum jump* function M_G is the function from vertices to integers defined by $M_G(v) = 0$ if v is a sink of G, $M_G(v) = \min \{J_G((v, w)) \mid (v, w) \in E\}$ if v is not a sink of G.

The following lemmas justify our interest in these functions.

LEMMA 3. *Let G be a dag. If (v, w) is an edge of G that is redundant under transitive closure, then $M_G(v) < J_G((v, w))$.*

Proof. The vertices along any path in G have strictly increasing levels. If (v, w) is redundant, there must be a path of length at least two from v to w. Thus if (v, x) is the first edge on this path then $J_G(c, x) < J_G(v, w)$, so $M_G(v) < J_G(v, w)$. □

LEMMA 4. *If G is* MVSP *then $M_G(v) = J((v, w))$ for every edge (v, w) in G.*

Proof. We prove the lemma by induction on the number of vertices in G. The lemma is immediate if G has one vertex. Suppose that G has $n \geqq 2$ vertices and the lemma is true for MVSP dags with fewer than n vertices.

If $G = \langle V_1 \cup V_2, E_1 \cup E_2 \rangle$ is the parallel composition of $G_1 = \langle V_1, E_1 \rangle$ and $G_2 = \langle V_2, E_2 \rangle$, then the level of each vertex in V_1 is exactly the same in G as in G_1, and the level of each vertex in V_2 is the same in G as in G_2. The lemma follows by the induction hypothesis applied to G_1 and G_2.

Suppose on the other hand that $G = \langle V_1 \cup V_2, E_1 \cup E_2 \cup (T_1 \times S_2) \rangle$ is the series composition of $G_1 = \langle V_1, E_1 \rangle$ and $\langle V_2, E_2 \rangle$. Then $L_G(v) = L_{G_1}(v)$ for all vertices $v \in V_1$, and $L_G(v) = L_{G_2}(v) + k + 1$ for all vertices $v \in V_2$, where k is the length of the longest path in G_1 ($k = \max \{L_{G_1}(v) \mid v \in V_1\}$). Thus if $(v, w) \in E_1$, $J_G((v, w)) = J_{G_1}((v, w))$; if $(v, w) \in E_2$, $J_G((v, w)) = J_{G_2}((v, w))$; and if $(v, w) \in T_1 \times S_2$, then $J_G((v, w)) = k + 1 - L_{G_1}(v)$. Since each vertex of G has exiting edges which come entirely from one of the sets E_1, E_2, $T_1 \times S_2$, the lemma for G follows from the induction hypothesis applied to G_1 and G_2. □

The jump and the minimum jump functions are defined in terms of the level function, which in turn is defined in terms of longest paths. Because a longest path cannot contain any redundant edge, the values of these three functions are insensitive to the addition and removal of redundant edges. Combining this fact with Lemmas 3 and 4, we obtain the following corollary.

COROLLARY 1. *Let G be a* VSP *dag and let (v, w) be one of its edges. Then (v, w) is redundant under transitive closure if and only if $M_G(v) < J_g(v, w)$.*

We carry out Step 1 of the recognition procedure by computing L_G, J_G, and M_G, and letting $E_M = \{(v, w) \mid M_G(v) = J_G(v, w)\}$. By Corollary 1 this method correctly performs Step 1. We can compute L_G in linear time by processing the vertices of G in topologically sorted order ([KNU]); the rest of the computation clearly requires only linear time.

3.2. The inverse line digraph of a dag. Implementing Step 2 of the recognition procedure requires an understanding of line digraph inverses. Several authors have characterized dags having line digraph inverses by using a nonalgorithmic approach ([HN], [KLE]), and Lehot [LEH] has developed a fast algorithm for computing the inverse line graph of an arbitrary undirected graph. However, Lehot's method does not seem to apply to dags, because several nonisomorphic dags may have the same line digraph. See Fig. 8.

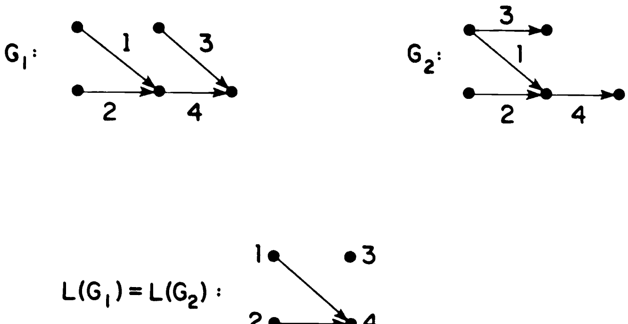


FIG. 8. *Two nonisomorphic multidigraphs that have the same line digraph.*

We compute the line digraph inverse of G_M in two steps. First, we apply a characterization of Harary and Norman [HN] to determine whether G_M has any line digraph inverse. If so, we select a specific inverse as $L^{-1}(G_M)$.

DEFINITION 5 (*complete bipartite composite dags*). A dag G is *complete bipartite composite* (CBC) if there is a set $\{B_1, B_2, \cdots, B_k\}$ of complete bipartite subgraphs of G, called the *bipartite components* of G, such that

(i) every edge of G belongs to exactly one bipartite component;

(ii) for each non-sink vertex v, all edges leaving v belong to the same bipartite component, denoted by $B_H(v)$; and

(iii) for each non-source vertex v, all edges entering v belong to the same bipartite component, denoted by $B_T(v)$.

It is easy to prove that the bipartite components of a CBC dag are unique (see [VAL]). The following lemma gives Harary and Norman's characterization.

LEMMA 5. *A dag has a line digraph inverse if and only if it is* CBC.

Proof. See [HN]. □

In order to select a unique inverse, we use another result of Harary and Norman.

LEMMA 6. *Let G_1 and G_2 be two multidigraphs such that $L(G_1)=L(G_2)$. Let G'_i for $i=1, 2$ be the multidigraph obtained by merging all sources of G_i into a single source and all sinks of G_i into a single sink. Then G'_1 and G'_2 are isomorphic.*

DEFINITION 6 (*line digraph inverse*). If G is a CBC dag, then the *line digraph inverse* of G, denoted by $L^{-1}(G)$, is the multidigraph with a single source and a single sink such that $L(L^{-1}(G))=G$. By Lemma 5, $L^{-1}(G)$ exists. By Lemma 6, $L^{-1}(G)$ is unique.

To use these results in our recognition procedure, we require one more fact.

LEMMA 7. *Every* MVSP *dag is* CBC.

Proof. In the construction of an MVSP dag, new edges are introduced exclusively by series compositions, and each series composition introduces a set of edges forming a complete bipartite subgraph. It is easy to check that these subgraphs satisfy the conditions of Definition 5. □

Lemmas 5–7 give us a way to carry out Step 2. First, we test whether G_M is CBC, as it must be if G_M is MVSP. We perform this test as follows. We select an edge (v, w) that has not yet been assigned to a bipartite component, and we assign it to a new component B_i. We mark all successors of v as belonging to the tail T_i of the component, and we mark all predecessors of w as belonging to the head H_i of the component. We check to see whether G_M contains $\langle T_i \cup H_i, T_i \times H_i\rangle$ as a (complete bipartite) subgraph. If not, G_M is not CBC. It so, we assign all edges in this subgraph to B_i. Then we select a new unassigned edge and repeat the process. We continue until either all edges of G_M are assigned (G_M is CBC), or we attempt to assign an edge to more than one bipartite component (G_M is not CBC), or we mark a vertex as belonging to more than one head or more than one tail (G_M is not CBC). It is straightforward to prove the correctness of this method. It is also not difficult to implement it to run in time linear in the size of the input dag.

Once we have verified that G_M is CBC, we apply the following transformation to compute $L^{-1}(G_M)$. As the vertex set of $L^{-1}(G_M)$, we use $\{B_\alpha, B_1, \cdots, B_k, B_\omega\}$, where $B_1, \cdots, B_k$ represent the complete bipartite components found by the CBC testing method, and B_α and B_ω are two additional vertices. For each vertex v of G_M, we add one edge to $L^{-1}(G_M)$ as follows:

(a) if v is an isolated vertex, we add an edge (B_α, B_ω) to $L^{-1}(G_M)$;

(b) if v is a source but not a sink, we add an edge $(B_\alpha, B_H(v))$ to $L^{-1}(G_M)$;

(c) if v is a sink but not a source, we add an edge $(B_T(v), B_\omega)$ to $L^{-1}(G_M)$;

(d) if v neither a source nor a sink, we add an edge $(B_T(v), B_H(v))$ to $L^{-1}(G_M)$.

This transformation requires linear time given the complete bipartite components of G_M as computed by the CBC testing method. It is routine to verify that $L^{-1}(G_M)$

as constructed by this transformation satisfies $L(L^{-1}(G_M)) = G_M$. We have thus provided a way to carry out Step 2 in time linear in the number of vertices and edges in G_M.

3.3. The recognition of ESP multidigraphs. In section 1.2 we described the algorithm to be used in Step 3 of the recognition procedure: we apply series and parallel reductions until no more are applicable, and we test whether the graph remaining has a single edge. It remains for us to describe how to implement this algorithm so that it runs in linear time. Aho, Hopcroft, and Ullman ([AHU], exercise 5.8), pose a similar problem for undirected graphs but provide no solution to it. Two solutions and their extension to multidigraphs appear in [VAL], and a solution to a special case of the problem appears in [PS]. We shall sketch one of these solutions (suggested by Hopcroft) as applied to the directed case.

We assume that the input multidigraph has a single source and a single sink. We maintain a list of vertices called the *unsatisfied list.* Initially this list contains all vertices except the source and the sink. In general the list contains all vertices other than the source and the sink on which reductions must still be tried. The algorithm repeats the following step until no vertices remain on the unsatisfied list.

General step

(a) Remove some vertex v from the unsatisfied list.

(b) Examine edges entering v. If two edges of the form (u, v) are found, apply a parallel reduction to them. Continue examining edges entering v and applying parallel reductions until either (i) only one edge enters v, or (ii) v is found to have two distinct predecessors.

(c) Examine edges leaving v. If two edges of the form (v, w) are found, apply a parallel reduction to them. Continue examining edges leaving v and applying parallel reductions until either (i) only one edge leaves v, or (ii) v is found to have two distinct successors.

(d) If only one edge (u, v) now enters v and only one edge (v, w) now leaves v, carry out the following steps.

 (i) Apply a series reduction to delete v and replace (u, v) and (v, w) by a new edge (u, w).

 (ii) If u is not the source and not on the unsatisfied list, add it to the unsatisfied list.

 (iii) If w is not the sink and not on the unsatisfied list, add it to the unsatisfied list.

When the unsatisfied list is empty, we test whether any vertices other than the source and sink remain. It so, the multidigraph is not reducible to a single edge. If not, we complete the reduction to a single edge by applying parallel reductions to the edges joining the source and sink.

The following observation guarantees the correctness of this algorithm. Let v be a vertex which is neither the source nor the sink. When v is removed from the unsatisfied list, either v is deleted from the multidigraph or v is verified to have either at least two predecessors or at least two successors. The number of predecessors of v cannot be decreased without adding v to the unsatisfied list. Similarly the number of successors of v cannot decrease without adding v to the list. Now suppose the unsatisfied list is empty and the multidigraph still contains a vertex other than the source and the sink. Then every vertex other than the source and the sink has either at least two predecessors or at least two successors. No number of parallel reductions can change this fact. Thus there is no way to

delete additional vertices (by series reduction), and the multidigraph cannot be reduced to a single edge.

In order to implement this algorithm to run in linear time, we maintain for each vertex a (doubly linked) list of the edges entering it and a list of the edges leaving it. Then each edge examination, parallel reduction, and series reduction requires constant time. Suppose the input multidigraph has n vertices and m edges. Each parallel reduction reduces the number of edges by one. Each series reduction deletes a vertex and reduces the number of edges by one. Thus there are at most $m-1$ parallel reductions and at most $n-2$ series reductions. At most $3(n-2)$ additions to the unsatisfied list occur, $n-2$ initially and two for each series reduction. Thus there are at most $3(n-2)$ executions of the general step.

Consider a given execution of the general step. Each edge examination in (b) (except at most two) causes a parallel reduction. Each edge examination in (c) (except at most two) causes a parallel reduction. Thus the total number of edge examinations in (b) and (c) during all executions of the general step is at most $12(n-2)+m-1$. It follows that the entire algorithm requires $O(m+n)$ time. (Each execution of step (a) or (d) takes constant time.)

It is a simple matter to modify the algorithm so that it computes a binary decomposition tree by applying the rules of Fig. 7 as it carries out reductions. Combining two trees which are edge labels into a new tree labeling the edge produced by a reduction takes only constant time; thus the linear time bound is not affected by this modification.

3.4. The two-dimensionality of MVSP dags. In order to complete our implementation of the recognition procedure, we must show that every MVSP dag is two-dimensional, and we must provide a way to compute two total orders whose intersection is the relation $<$ such that $v<w$ if and only if there is a path from v to w in the dag.

We shall regard a total order on a set of n elements as a bijection between the set and $\{1, 2, \cdots, n\}$. Two total orders on a set thus map each element e into a pair of integers (x_e, y_e), which we can interpret as a point in the Cartesian plane. Our problem is thus to map the vertices of an arbitrary MVSP dag into the plane so that there is a path from v to w if and only if $x_v \leqq x_w$ and $y_v \leqq y_w$; i.e., (x_v, y_v) is below and to the left of (x_w, y_w). Figure 9 shows an embedding of the MVSP dag of Fig. 1 which satisfies this criterion.

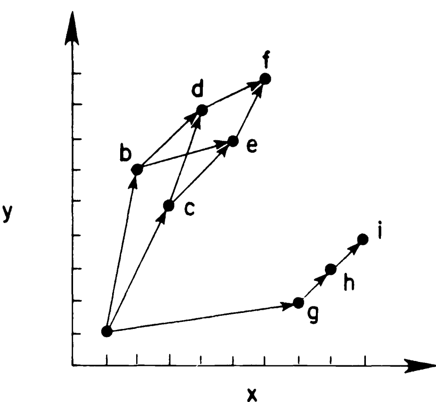


FIG. 9. *Embedding of the* MVSP *dag of Figure* 1 *in the Cartesian plane using the two total orders given in Section* 2.2 *as coordinates.*

We can build up the embedding step-by-step using the constructions of Fig. 10 to deal with series and parallel compositions. If G is formed from G_1 and G_2 by a series composition, there is a path from every vertex of G_1 to every vertex of G_2; in the embedding of Fig. 10, every vertex of G_1 is below and to the left of every vertex in G_2. If G is formed from G_1 and G_2 by a parallel composition, there is no path between G_1 and G_2; in the embedding of Fig. 10, no vertex of G_1 is below and to the right of any vertex in G_2.

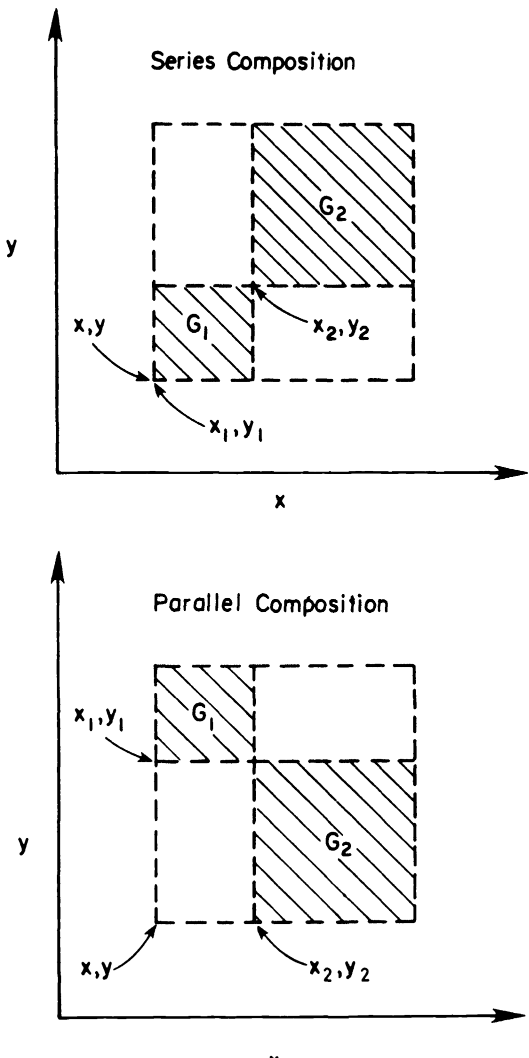


FIG. 10. *Method used to embed an* MVSP *dag in the plane.*

To formally specify this embedding, we shall represent the position of a subgraph by the coordinates of the lower left-hand corner of the smallest square which contains all its vertices. If we let n_1 and n_2 denote the number of vertices in G_1 and G_2, the following formulas provide the positions of G_1 and G_2 given the position of G.

Series composition: $x_1 = x$, $y_1 = y$; $x_2 = x + n_1$, $y_2 = y + n_1$.

Parallel composition: $x_1 = x$, $y_1 = y + n_2$; $x_2 = x + n_1$, $y_2 = y$.

We can compute an embedding in two traversals of the binary decomposition tree T of G. First, we traverse the tree in postorder and assign a *size* to each node; each external node has size one and each internal node has size equal to the sum of the sizes of its children. Next we assign coordinates $(1, 1)$ to the root of the tree. Finally, we traverse the tree in postorder, assigning coordinates to the children of each vertex using the coordinates of the node, the label of the node (S or P), the sizes of the children, and the formulas above. Computing the embedding requires $O(n)$ time, where n is the number of vertices in the MVSP dag. Once we have the embedding, we can test any edge for redundancy in constant time. This provides a linear-time implementation of Step 4, and completes our description of the recognition procedure.

4. Forbidden subgraph characterization. A common goal in classical graph theory is the characterization of a class of graphs by exhibiting a set of *forbidden subgraphs* such that a graph is in the class if and only if it does not contain any forbidden subgraph. Perhaps the most famous of such results is Kuratowski's characterization of planar graphs ([HAR]). We can provide such a result for VSP dags.

THEOREM 1. *A dag G is* VSP *if and only if its transitive closure does not contain the N graph of Fig.* 11 *as an induced subgraph.*

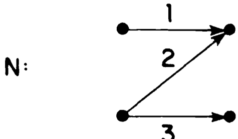


FIG. 11. *The forbidden subgraph for* VSP *dags.*

Proof. (along lines suggested by Peter Avery). The N graph has neither a series nor a parallel decomposition. Furthermore if a transitive dag G contains the N graph as an induced subgraph, and G is composed in either a series or a parallel fashion, then the N graph appears intact in one of the components. For parallel decomposition, this is because no edges join the component, but any partition of the vertices of the N has an edge joining the two blocks of the partition. For series composition this is because every pair of vertices in different components are joined by an edge, but any partition of the vertices of the N produces two vertices in different blocks that are not joined by an edge. Thus, if the transitive closure of a dag G contains an N, G cannot be VSP.

To prove the converse, suppose G is a transitive dag containing at least two vertices and having neither a series nor a parallel decomposition. We want to show that G contains an N. Let s be any source of G and let T be the set of sinks that are successors of s. T is nonempty, or G would have a parallel decomposition into s and the remaining vertices. Let X be the set of vertices with at least one successor in T. We distinguish two cases.

Case 1. There is some vertex $v \notin X \cup T$. Since G has no parallel decomposition, there must be such a vertex v with either a successor or a predecessor in $X \cup T$. Since T contains only sinks, v cannot have a predecessor in T; since $v \notin X$ and G is transitive, v cannot have a successor in $X \cup T$. Thus v has a predecessor, say u, in X. Let t be a successor of u in T. We claim u, v, s, t are the vertices of an N in G.

Certainly, (u, v), (u, t), and (s, t) are edges. Since t is a sink (t, v) is not an edge; since $v \notin X$, (v, t) is not an edge. Since s is a source, neither (u, s) nor (v, s) is an edge.

If v is a sink, (s, v) is not an edge because $v \notin T$. If v is not a sink, there is some sink $w \notin T$ that succeeds v, and (s, v) is not an edge because $w \notin T$ and G is transitive. Finally, since G is transitive and (s, v) is not an edge, (s, u) is also not an edge. Thus u, v, s, t form an N.

Case 2. Every vertex in G is in $X \cup T$. Then T contains all the sinks in G. Let Y be the subset of vertices in X that precede all the sinks. ($Y = \{x \in X \mid \text{for all } t \in T, (x, t) \text{ is an edge}\}$.) Y cannot equal X, or G would have a series decomposition into X and T. Let Z be the subset of vertices in $X - Y$ that succeed all the vertices in Y. ($Z = \{x \in X - Y \mid \text{for all } y \in Y, (y, x) \text{ is an edge}\}$.) Z cannot equal $X - Y$, or G would have a series decomposition into Y and $(X - Y) \cup T$. Let u be a vertex in $X - Y - Z$. Since $u \in (X - Y - Z)$, there is some vertex $y \in Y$ such that (y, u) is not an edge. Since $u \in X - Y$, there is some vertex $v \in T$ such that (u, v) is not an edge. Finally, since $u \in X$, there is some vertex $t \in T$ such that (u, t) is an edge. We claim y, v, u, t are the vertices of an N in G.

We know (u, t) is an edge. Since $y \in Y$ both (y, t) and (y, v) are edges. Since both t and v are sinks, neither (t, v), (v, t), nor (v, u) is an edge. We know (u, v) is not an edge; by transitivity (u, y) is not an edge. Finally, we know (y, u) is not an edge. Thus y, v, u, t form an N. □

Our VSP recognition procedure can be modified, while preserving its linear time bound, so that it finds an induced N subgraph in any non-GSP dag. The details of this modification appear in [VAL].

Another way to prove Theorem 1 is to apply a forbidden subgraph characterization of Duffin [DUF] for undirected ESP multigraphs. Duffin showed that a multigraph is undirected ESP if and only if it does not contain a subgraph homeomorphic to K_4 (the complete graph on four vertices). A trivial modification of his argument shows that a multigraph with a single source and a single sink is ESP if and only if it does not contain a subgraph homeomorphic to the W dag of Fig. 12. Using this characterization of ESP multidigraphs and Lemma 1, it is not hard to show that the transitive closure of a CBC dag G contains N as an induced subgraph if and only if $L^{-1}(G)$ contains a subgraph homeomorphic to W. (Note that the line digraph of W is the dag in Fig. 13, whose transitive closure contains an induced N.) From this the theorem is immediate.

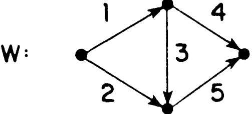


FIG. 12. *The forbidden subgraph for* ESP *multidigraphs.*

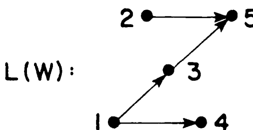


FIG. 13. *The line digraph of the dag of Figure* 12.

5. Remarks. In this section we mention some additional consequences of our results. Step 1 of the recognition procedure provides a linear-time algorithm to compute the transitive reduction of a VSP dag; Step 4 gives a linear-time method to

compute the transitive closure of a VSP dag (in implicit form). These methods are much faster than the best algorithms for arbitrary dags (see [AGU]).

Although several nonisomorphic binary decomposition trees may represent the same MVSP dag, there is a way of modifying these trees to represent MVSP dags in a quasi-unique way: we contract into a single node each connected group of S nodes and each connected group of P nodes. See Fig. 14. The result is a decomposition tree, no longer binary, which is unique up to reordering the children of each P node.

Using these *canonical decomposition trees*, we can test two MVSP dags for isomorphism in linear time by adapting a linear-time tree isomorphism algorithm ([AHU]). The isomorphism problem for VSP dags is as hard as isomorphism of arbitrary graphs; the decomposition tree gives no information about the presence or absence of redundant edges, and we can encode an arbitrary graph into the redundant edges of a VSP dag ([VAL]).

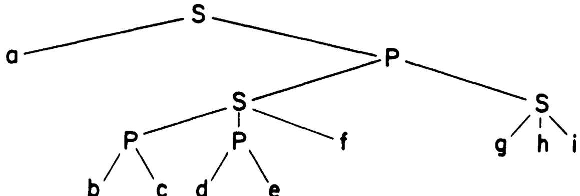


FIG. 14. *Canonical decomposition tree for the* MVSP *dag of Figure* 1.

The subgraph isomorphism problem for MVSP dags is NP-complete, because it contains as a special case the following known NP-complete problem [GJ]: given a (rooted, directed) tree T and a forest (collection of rooted, directed trees) F, determine whether T contains a subgraph isomorphic to F.

Acknowledgment. We thank the referee for finding an error in our original proof of Theorem 1.

REFERENCES

[ADA] A. ADAM, *On graphs in which two vertices are distinguished*, Acta Math. Acad. Sci. Hungary, 12 (1961), pp. 377–397.

[AGU] A. V. AHO, M. R. GAREY AND J. D. ULLMAN, *The transitive reduction of a directed graph*, this Journal, 1 (1972), pp. 131–137.

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

[DUF] R. J. DUFFIN, *Topology of series-parallel networks*, J. Math. Anal. Appl., 10 (1965), pp. 303–318.

[GJ] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, 1979.

[HAR] F. HARARY, *Graph Theory*, Addison-Wesley, Reading, MA, 1971.

[HKS] F. HARARY, J. KRARUP AND A. SCHWENK, *Graphs suppressible to an edge*, Canadian Math. Bull., 15 (1971), pp. 201–204.

[HN] F. HARARY AND R. NORMAN, *Some properties of line digraphs*, Rendiconti del Circolo Mathematico Palermo, 9 (1960), pp. 149–163.

[KLE] J. B. KLERLEIN, *Characterizing line dipseudographs*, Proc. Sixth Conference on Combinatorics, Graph theory, and Computing (1975), pp. 429–442.

[KNU] D. E. KNUTH, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1968.

[LAW1] E. L. LAWLER, *Sequencing jobs to minimize total weighted completion time subject to precedence constraints*, Annals of Discrete Math., 2 (1978), pp. 75–90.

[LAW2] E. L. LAWLER, *Sequencing problems with series parallel precedence constraints*, Proc. Conf. on Combinatorial Optimization, Urbino, Italy, 1978, to appear.

[LEH] P. G. H. LEHOT, *An optimal algorithm to detect a line graph and output its root graph*, J. Assoc. Comput. Mach., 21 (1974), pp. 569–575.

[MON] C. L. MONMA AND J. B. SIDNEY, *A general algorithm for optimal job sequencing with series-parallel constraints*, Technical Report No. 347, School of Operations Research and Industrial Engineering, Cornell University, Ithaca, N.Y., 1977.

[PS] B. PRABHALA AND R. SETHI, *Efficient composition of expressions with common subexpresssions*, J. Assoc. Comput. Mach., 27 (1980), pp. 146–163.

[RIO] J. RIORDAN AND C. E. SHANNON, *The number of two terminal series parallel networks*, J. Math. Physics, 21 (1942), pp. 83–93.

[ROS] B. K. ROSEN, *Tree manipulating systems and Church–Rosser theorems*, J. Assoc. Comput. Mach., 20 (1972), pp. 160–187.

[SET] R. SETHI, *Testing for the Church–Rosser property*, J. Assoc. Comput. Mach., 21 (1974), pp. 671–679.

[SID] J. B. SIDNEY, *The two machine flow line problem with series-parallel precedence relations*, Working paper 76-19, Faculty of Management Sciences, University of Ottawa, Ottawa, Ontario, Canada, 1976.

[VAL] J. VALDES, *Parsing flowcharts and series-parallel graphs*, Technical Report STAN-CS-78–682, Computer Science Department, Stanford University, Stanford, California, 1978.

[WAL] T. R. S. WALSH, *Counting labeled three-connected and homeomorphically irreducible two-connected graphs*, unpublished manuscript, 1978.

[WEI] L. WEINBERG, *Linear graphs: theorems, algorithms, and applications*, Aspects of Network and System Theory, R. E. Kalman and N. DeClaris, eds., Holt, Rinehart, and Winston, N.Y., 1971.

SIAM J. COMPUT.
Vol. 11, No. 2, May 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1102-0009 $01.00/0

PARALLEL ALGORITHMS IN GRAPH THEORY: PLANARITY TESTING*

JOSEPH JA' JA'† AND JANOS SIMON‡

Abstract. We present efficient ($O(\log^2 n)$) parallel algorithms for two classical graph problems: planarity testing and finding triconnected components. The algorithms use only a polynomial number of processors. Previous algorithms used $\Omega(n)$ operations, regardless of the number of available processors.

Key words. parallel algorithms, planarity testing, three-connected components, computational complexity

1. Introduction. One of the major problems in both theoretical and applied computer science is our lack of knowledge about the nature of parallel computation. It is now feasible to build huge conglomerates of processors, and they represent the only hope for order of magnitude improvements in computing power in the near future. At the same time, beyond the problems of interconnection, synchronization and reliability of such conglomerates, there is a more basic problem that is probably the main obstacle to the construction and widespread use of such machines: we have no idea what to do with them. Our algorithms, our formal reasoning, our intuition about computing were all developed for serial models. Independent tasks are obviously well suited to parallel execution, and a few good parallel algorithms have been developed (sorting, portions of numerical linear algebra, evaluation of arithmetic expression, linear recurrences, etc.; see, e.g., [Br], [C], [Ch], [He], [Hi], [P]). However, except for these very special cases, the use of, say, a polynomial number of processors seems to yield only marginal improvements in the running time of algorithms. Our main objective is to gain a better understanding of parallelism. We would like to have a collection of techniques for obtaining efficient parallel algorithms and an intuition of how to recognize features that make a problem amenable to efficient parallel solution.

In this paper we consider a well-known problem in graph theory, namely, testing whether a given graph is planar. This problem has linear time serial algorithms [HT], [BL], [Ev], but has no obvious parallel algorithms with $o(n)$ running time (using a polynomially bounded number of processors). We develop two fast parallel algorithms, each of which runs in $O(\log^2 n)$ time. The first algorithm uses $O(n^4)$ processors (which don't add or multiply), the second (which, in addition, yields a barycentric representation of a planar graph) uses $O(n^{3.29}/\log^2 n)$ arithmetic processors. These results are surprising because all the known serial algorithms for this problem are somehow inherently sequential. We assume here the unbounded parallel model; we have an unlimited number of processors, each identified by a unique label. These processors have access to a common main memory which contains the instructions of a program. We make the assumption that different processors can obtain the content of one memory location at the same time; they may store information simultaneously, but no two processors should attempt to change the content of the same memory location

* Received by the editors July 10, 1980, and in final revised form July 28, 1981.

† Computer Science Department, Pennsylvania State University, University Park, Pennsylvania 16802. The work of this author was supported in part by the National Science Foundation under grant MCS 78 27600.

‡ Computer Science Department, Pennsylvania State University, University Park, Pennsylvania 16802. The work of this author was supported in part by the National Science Foundation under grant MCS 79 05006.

at the same time. We further assume that all the processors are synchronous in the sense that if a set of instructions is executed in parallel, then each must be allowed to finish before the next set of instructions is started. The instruction set of each processor does not include multiplication unless otherwise specified.

Parallel algorithms for several graph problems exist. The list in Table 1 includes some of the known results in this area; n is the number of nodes and m is the number of edges in the graph.

TABLE 1.

Problem	Time bound	# of processors
Transitive closure [H]	$O(\log^2 n)$	$O(n^3)$
Connectivity testing [SJ]	$O(\log^2 n)$	$O(n \log n + m)$
Finding the biconnected components [SJ]	$O(\log^2 n)$	$O(n^3/\log n)$
Finding minimal spanning trees [SJ]	$O(\log^2 n)$	$O(n^2)$
Finding the bridge-connected components [SJ]	$O(\log^2 n)$	$O(n^2 \log n)$
Testing whether G is bipartite	$O(\log^2 n)$	$O(n^2)$
Testing k-connectedness [G2]	$O(\log^2 n \log k)$	$O(n^{k+1})$

Because of well-known results [HS], [PS], [FW], our algorithms may also be viewed as space-efficient. Thus, our investigations may shed some light on the $P \stackrel{?}{=} \bigcup_{k \in N} \text{DSPACE}[(\log n)^k]$ question. A companion paper [JS] presents space-efficient versions of these algorithms, with space bounds that exactly match our time bounds (and, therefore, are better than the algorithms obtained by simply translating parallel time to space).

2. Preliminaries. Our graph terminology is standard and follows ([BM], [H]) unless indicated otherwise. A *graph* is a pair $G=(V,E)$ such that E is a set of unordered pairs of distinct vertices. V is called the set of *vertices* and E the set of *edges*. We sometimes use the notation $E(G)$ to denote the set of edges of G. A graph is complete if E is the set of all edges. A *subgraph* $G'=(V',E')$ of graph G is a graph such that $V' \subseteq V$ and $E' \subseteq E$. A *path* from v_1 to v_n is a sequence of edges $(v_1, v_2), (v_2, v_3), \cdots, (v_{n-1}, v_n)$ where all the vertices are distinct except possibly $v_1 = v_n$, in which case, the path is a *cycle*. A graph is *connected* if for every pair of vertices u and v there exists a path from u to v. The *degree* of a vertex is the number of edges incident on that vertex. A *cutvertex* is a vertex whose removal from V, together with the edges incident on it, leaves G not connected. A graph is *biconnected* if it contains no cutvertices. The *biconnected components* of a graph are its maximal biconnected subgraphs. A *tree* is a connected acyclic graph.

Let $G=(V,E)$ be a connected graph. A *spanning tree* is a tree that connects all vertices in V. Let $H=(V_H, E_H)$ and $K=(V_K, E_K)$ be subgraphs of G. The *symmetric difference* of H and K, written $H \oplus K$, is the subgraph $G'=(V',E')$ of G, where

$$E' = \{e \in E_H \cup E_K \mid e \notin E_H \cap E_K\}$$

and

$$V' = \{v \in V \mid v \text{ is incident with some edge of } E'\}.$$

A *fundamental set of cycles* or *cycle basis* of G is a collection Ω of cycles of G with the property that any cycle C of G can be written as $C = C_1 \oplus C_2 \oplus \cdots \oplus C_k$ for some subcollection of cycles $C_1, C_2, \cdots, C_k \in \Omega$. Let $T=(V, E_T)$ be a spanning tree of G.

Every edge $e \in E - E_T$ will create a cycle C_e if it is added to T. It can be shown that the collection $\{C_e | e \in E - E_T\}$ is a fundamental set of cycles.

The rest of the paper is organized as follows. In the next section we develop a fast algorithm to find the triply connected components of a graph. Our first planarity testing algorithm is described in § 4, while the second algorithm is sketched in § 5.

3. Finding the triply connected components of a graph. Since our planarity algorithms assume that the graph is triconnected, we present in this section a fast parallel algorithm to find the triconnected components of a graph. We define the triconnected components of a graph so that the graph is planar if and only if its triconnected components are. We follow more or less McLane's definitions [McL1]. Our algorithm finds the triconnected components in $O(\log^2 n)$ time with $O(n^4)$ processors and can be viewed as a generalization of a technique used in [SJ] to find the biconnected components of a graph. We now proceed to define precisely what we mean by triconnected components of a graph.

Let $G=(V,E)$ be an undirected biconnected graph, and let $H_1=(V_1,E_1)$ and $H_2=(V_2,E_2)$ be two subgraphs of G. Then H_1+H_2 is defined to be the subgraph $(V_1 \cup V_2, E_1 \cup E_2)$. Let $\{u, v\} \subseteq V$. A *split* of G at $\{u, v\}$ is a pair of nonempty subgraphs $H_1=(V_1,E_1)$ and $H_2=(V_2,E_2)$ such that

$$G=H_1+H_2, \quad E_1 \cap E_2 = \emptyset \quad \text{and} \quad V_1 \cap V_2 = \{u, v\}.$$

If such a split exists, $\{u, v\}$ is called a *separation pair*. Corresponding to the split $\langle H_1, H_2\rangle$, we define the *blocks* B_1 and B_2 associated with H_1 and H_2 as follows:

$$B_1=\begin{cases}(V_1, E_1 \cup \{(u, v)\}) & \text{if } H_1 \text{ is not a path,}\\ \emptyset & \text{otherwise,}\end{cases}$$

$$B_2=\begin{cases}(V_2, E_2 \cup \{(u, v)\}) & \text{if } H_2 \text{ is not a path,}\\ \emptyset & \text{otherwise,}\end{cases}$$

(u, v) will be called a *virtual edge*. A graph $G=(V,E)$ is *triconnected* if it is biconnected and has no split. If a biconnected graph $G=(V,E)$ splits into two blocks, then G may be further decomposed by splitting one of these blocks which may happen not to be triply connected. We continue this process (each time ignoring the empty blocks) until no further splitting is possible. These triply connected blocks are called the *triply connected components* (t.c.c.) of G. We now have some basic facts about triconnected graphs.

LEMMA 3.1. *If* $G=(V,E)$ *is triconnected, then* $\deg v \geqq 3$ *for all* $v \in V$.

Proof. Suppose not. Then there exists $v \in V$ such that $\deg v = 2$, with edges (v, w_1) and (v, w_2). A split at $\{w_1, w_2\}$ is possible. □

LEMMA 3.2. *A graph* $G=(V,E)$ *is triconnected if and only if for all pairs* $\{v, w\} \subseteq V$, *there exist three vertex disjoint paths between* v *and* w.

Proof. The *If* part is easy.

Suppose G has a split $\langle H_1, H_2\rangle$ at $\{u, w\}$, with $H_1=(V_1,E_1)$ and $H_2=(V_2,E_2)$. Take any $v_1 \in V_1$ and $v_2 \in V_2$. We cannot have three vertex disjoint paths joining v_1 and v_2 [H]. □

THEOREM 3.3 [McL1]. *Every t.c.c. of a graph* $G=(V,E)$ *is homeomorphic to a maximal triply connected subgraph. Moreover, if* $\{A_1, A_2, \cdots, A_m\}$ *is a complete set of t.c.c.'s of* G, *then every maximal triconnected subgraph of* G *is homeomorphic to one and only one of the* A_i*'s. The t.c.c.'s of* G *are unique to a homeomorphism.*

THEOREM 3.4 [McL1]. *A biconnected graph* G *is planar if and only if all of its t.c.c.'s are planar.*

One way of attempting to find the t.c.c.'s of a graph is to start by removing all separation pairs, find the connected components and try to reconstruct the t.c.c.'s. This approach won't work because a t.c.c. may only have four vertices $\{v_1, v_2, v_3, v_4\}$ such that each pair $\{v_i, v_j\}$, $i \neq j$, is a separation pair. Consider the graph of Fig. 1.

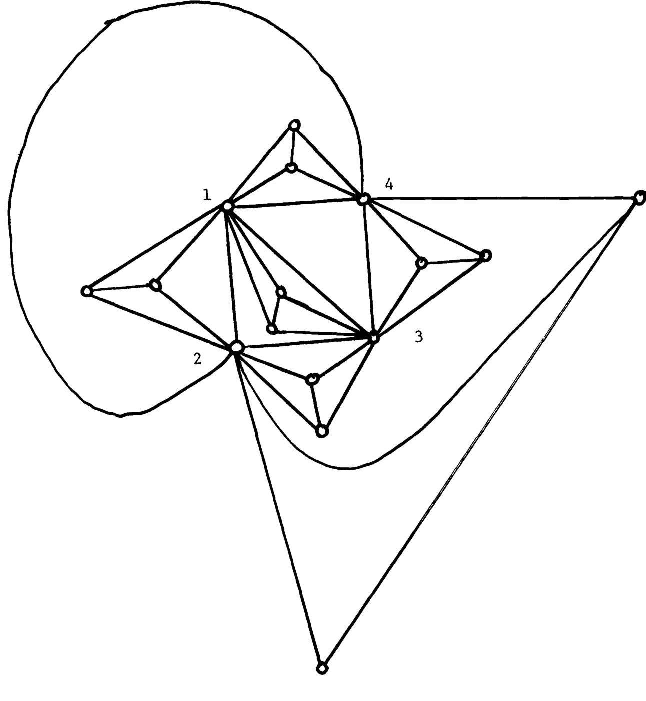


FIG. 1.

$\{1, 2, 3, 4\}$ is the vertex set of a t.c.c. and yet $\{1, 2\}$, $\{1, 3\}$, $\{1, 4\}$, $\{2, 3\}$, $\{2, 4\}$, $\{3, 4\}$ are all separation pairs. Let $G = (V, E)$ be a biconnected graph: Define a relation R on $V \times V$ by iRj if and only if i and j are in a common t.c.c. of G.

LEMMA 3.5. *iRj if and only if for all $\{\alpha, \beta\}$ such that $\alpha, \beta \notin \{i, j\}$, $(i, j) \in G^*_{\alpha\beta} = (V_{\alpha\beta}, E^*_{\alpha\beta})$, where $G_{\alpha\beta}$ is the subgraph of G obtained by removing α and β and all edges incident on α and β, and $G^*_{\alpha\beta}$ is the transitive closure of $G_{\alpha\beta}$.*

Proof. Let i and j be such that iRj, i.e., i and j are in the same t.c.c. If $(i, j) \in E$, then $(i, j) \in E^*_{\alpha\beta}$, and we are done. Suppose now $(i, j) \notin E$. Then, by Menger's theorem [H], no two vertices can separate i and j.

The converse follows from Menger's theorem and Lemma 3.2. □

It is easy to see that we can construct R in $O(\log^2 n)$ time with $O(n^4)$ processors. We now address the problem of actually determining each t.c.c. of G. Notice that the following holds.

LEMMA 3.6. *Two t.c.c.'s can intersect in at most two vertices.*

It follows that every set of three vertices of a t.c.c. determines this t.c.c. uniquely. Define the following relation T on $V \times V \times V$:

$$(i, j, k) \in T \Leftrightarrow iRj \wedge jRk \wedge iRk.$$

Note that since R is *not* transitive, we need all of the three conditions. We now have the following characterization.

LEMMA 3.7. *If $(i, j, k) \in T$, then i, j and k belong to the same t.c.c. of G. Moreover, for each $(i, j, k) \in T$, the set*

$$V_{ijk} = \{l \in V | lRi \wedge lRj \wedge lRk\}$$

determines a t.c.c. and, conversely, each t.c.c. of G can be obtained this way.

Proof. The first statement of the lemma is obvious. Now let $l \in V_{ijk}$. We prove that l is in the same t.c.c. as i, j and k. Suppose not. Let A_1 be the t.c.c. which contains i, j and k. Then there exists a pair of vertices $\{u, v\}$ which separates l from i, j and k. Hence one of lRi, lRj and lRk is false.

On the other hand, if a vertex l belongs to the t.c.c. A_1 containing i, j and k, then it is obvious that $l \in V_{ijk}$. □

The above lemmas suggest the following algorithm.

ALGORITHM 3.1.

1. Find the set of all separation pairs S_G.
2. For each $\{u, v\} \in S_G$, construct the transitive closure G^*_{uv}.
3. Compute R.
4. Compute T.
5. Find the triply connected components of G using T and R.

THEOREM 3.5. *The above algorithm correctly finds the triply connected components of a graph $G = (V, E)$. This algorithm can be implemented to run in $O(\log^2 n)$ time with $O(n^4)$ processors.*

Proof. The correctness follows directly from Lemma 3.7. Table 2 gives the running time and the number of processors for each of the steps of the above algorithm. Detailed algorithms for some of these steps and for similar problems, together with an analysis of time and processor bounds, can be found in [SJ]. □

TABLE 2.

Step	Time	# of processors
1	$O(\log^2 n)$	$O(n^4)$
2	$O(\log^2 n)$	$O(n^4)$
3	$O(log^2 n)$	$O(n^4)$
4	$O(1)$	$O(n^3)$
5	$O(\log^2 n)$	$O(n^3)$

4. The first planarity algorithm. We assume that the reader is familiar with the notions of embeddings of graphs into surfaces, plane embeddings and Jordan curves. For completeness we sketch some of the definitions.

An *embedding* of a graph G into a surface S is an injective map $f: V(G) \to S$ (the embedding of vertices) and a collection of Jordan curves $\Gamma_{(u,v)}$ such that:

(1) for every edge (u, v) of G, there is exactly one curve $\Gamma_{(u,v)}$ with endpoints $f(u), f(v)$;

(2) curves $\Gamma_{(u,v)}$ and $\Gamma_{(x,y)}$ are disjoint.

$\Gamma_{(u,v)}$ is the embedding of edge (u, v), and the collection of Jordan curves and their endpoints is the embedding of the graph. G is planar if it has an embedding into the plane.

In the sequel, unless otherwise noted, we shall always assume graphs to be triconnected. Let G be such a graph, let T be a spanning tree of G, and let $X \subseteq E(G) - E(T)$. Every edge $a \subset X$ defines a unique cycle C_a consisting of a and the unique path in T that joins T's endpoints. Let $F = \{C_a | a \in X\}$. F is a cycle basis.

Let C be a cycle of G, e, f edges of C. Define the equivalence relation $=_c$ by $e =_c f$ if and only if there is a path in G that includes e and f and has no internal vertices in common with C. The *bridges* of G [BM] relative to C are the subgraphs induced by the edges of the equivalence classes of $E(G)-E(C)$ under $=_c$ (these are the "connected pieces of G, ignoring connections due the vertices of C"). C is *peripheral* if G has a single bridge relative to C. Let F' be a cycle basis of peripheral cycles, and $M = F' \cup \{\sum_{C \in F'} C\}$ (cycles are considered, as usual, as vectors in $Z_2^{|E(G)|}$). M is a *plane mesh* of G if every edge of G appears in eactly two cycles of M. The following is a classical result of McLane's [McL1], [BM], [H].

THEOREM 4.1. *G is planar if and only if it has a plane mesh. The cycles of the plane mesh are the faces in the (unique, up to inversion) embedding of G into the sphere.*

Our first algorithm builds a plane mesh or reports that it is impossible to do so. The general strategy is the following: we find a tree cycle basis, i.e., a cycle basis generated from a spanning tree. Note that two cycles of a tree basis are either disjoint or have a common vertex or a common path. If G is planar, the cycles will be mapped into Jordan curves in a plane embedding. In an embedding, bridges will have to be entirely contained in one of the two regions of the plane that the Jordan curve induces. These facts can be stated in a purely combinatorial manner: for every cycle C in the cycle basis and for every edge $e \notin E(C)$ we define a boolean variable $\mathrm{IN}(C, e)$. The intended meaning is that $\mathrm{IN}(C, e) = 1$ if edge e is mapped to the inside region of the Jordan curve defined by C. We call an assignment of truth values to the variables $\{\mathrm{IN}(C, e) \mid e \notin E(C), E \in F\}$ a *pseudoembedding* if the following conditions are met (all cycles mentioned below belong to the cycle basis F):

a) If e, f belong to the same bridge B, relative to cycle C, then $\mathrm{IN}(C, e) \Leftrightarrow \mathrm{IN}(C, f)$.

b) If $\mathrm{IN}(C, e) = 1$, then for every cycle C' such that $e \in E(C')$ and for every edge $f \in [E(C)-E(C')]$ $\mathrm{IN}(C', f) = 0$.

c) If e, f are edges that belong to bridges B, B' that *conflict* (see below) relative to cycle C, then $\mathrm{IN}(C, e) \Leftrightarrow \neg \mathrm{IN}(C, f)$.

The *vertices of attachment* of bridge B to cycle C are the vertices in $V(B) \cap V(C)$. Bridges B and B' *conflict* relative to cycle C if either

(1) there are vertices v_1, v_2, v_3 that are both vertices of attachment of B to C and vertices of attachment of B' to C;

or

(2) there are vertices u, v, w, z of $V(C)$ that, in a cyclic ordering of the vertices of $V(C)$, u, v, w, and z appear in this order with u, w vertices of attachment of B to C and v, z vertices of attachment of B' to C.

Note that vertices of attachment, conflicts and pseudoembeddings are purely combinatorial concepts. We have, however, the following facts: Every embedding is a pseudoembedding. This is precisely stated in Lemma 4.2. Before proceeding we need some terminology. If G is a planar graph and Δ is an embedding of G into the plane, we use the following notations: if C, C_i, C_j are cycles of G and if e is an edge of G we denote by Γ_C, Γ_{C_i}, Γ_{C_j} and γ_e respectively the Jordan curves into which the cycles and the edges are mapped respectively. When C is determined by the context, we drop it as a subscript to avoid double subscripting and denote the curves by Γ, Γ_i, Γ_j, respectively. The *interior* of Γ is the bounded domain of the two domains determined by Γ on the plane, the *exterior* of Γ is the other domain. An edge e (or a cycle D) is *inside* Γ if the curve γ_e (the closed curve Γ_D) has no points mapped to the exterior of Γ. Similarly, we say that Γ_i is mapped *outside* Γ if no points of Γ_i are mapped to the interior of Γ.

LEMMA 4.2. *Let G be a planar graph and Δ an embedding of G into the plane. Let F be a set of fundamental cycles of G (a tree basis). Assign values to the boolean variables* $\mathrm{IN}(C, e)$ *for all* $C \in F$, $e \in E(G) - E(C)$ *according to the rule* $\mathrm{IN}(C, e) = 1$ *if* γ_e *is inside* Γ_C *in* Δ. *Then this assignment is a pseudoembedding.*

Proof. The lemma follows from well-known properties of plane embeddings (see, for example, [BM]: all edges in a bridge must be mapped to the same domain (condition a) for a pseudoembedding; conflicting bridges must be mapped to different domains, one inside and one outside Γ (condition c)); and if Γ_C is inside Γ, every edge f of C not in C' must be mapped to curve γ_f outside Γ (condition b)). Thus, the assignment satisfies conditions a), b) and c) and is a pesudoembedding. □

For a given planar graph G, we say that the pseudoembedding P obtained from an embedding Δ as in the lemma above *represents* the embedding and that the variables $\mathrm{IN}(C, e)$ with the assignments of P *have the intended meaning in* Δ. In general, if $C_1, C_2, \cdots, C_k$ are a subset of F, P is a pseudoembedding and Δ is an embedding, we say that P restricted to $C_1, C_2, \cdots, C_k$ represents Δ (or Δ is represented by P restricted to $C_1, C_2, \cdots, C_k$) if there is an embedding Δ' such that P represents Δ' and the set of edges mapped into the interior of $\Gamma_1, \Gamma_2, \cdots, \Gamma_k$ is the same in Δ and in Δ'.

Unfortunately, the converse of Lemma 4.2 is not true: there are pseudoembeddings that do not represent embeddings. Consider the spanning tree in Fig. 2 for K_{33}. The assignment $\mathrm{IN}(C, e) = 0$ for all $C \in I$ and all $e \notin E(C)$ is a pseudoembedding.

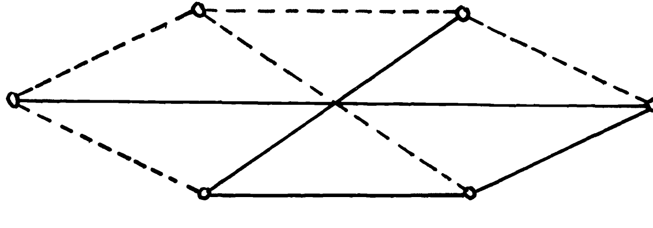


FIG. 2

There is, however, a partial converse that we prove later that enables us to obtain an algorithm: if G is planar every pseudoembedding represents an embedding (Theorem 4.7).

As we shall see, if Δ is an embedding of a planar graph G, it is easy to obtain from Δ a plane mesh for G, and the procedure can be stated in terms of the pseudoembedding P that represents Δ.

The idea behind our first algorithm is as follows: given G, obtain a pseudoembedding P of G (if none exists, G is nonplanar). Now, treating P as if it represented an embedding, carrry out the procedure that will produce a plane mesh for G if P represents an embedding. Call the resulting set of cycles a pseudoplane mesh. If the pseudoplane mesh is indeed a plane mesh, G is planar. On the other hand, if G is planar, then the pseudoplane mesh is in fact a plane mesh. So G is planar if and only if it has a pseudoembedding, and the pseudoplane mesh is a plane mesh, and these conditions can be checked efficiently in parallel.

In the rest of this section, we formalize these notions and prove the facts claimed.

Let $C_1, C_2 \in F$. Define $C_1 \subseteq C_2$ (intended meaning: in the embedding represented by P, Γ_{C_1} lies inside Γ_{C_2}) by

$$C_1 \subseteq C_2 \quad \text{if } \exists e \in E(C_1) \mathrm{IN}(C_2, e) = 1.$$

For fundamental cycles (cycles obtained from a spanning tree), $\subseteq$ is well defined.

LEMMA 4.3. *Given a cycle $C \in F$, the pseudoembeddings of G induce a partition F_1, F_2 of $F-\{C\}$ such that for every pseudoembedding for $i=1, 2$, $C_1 \in F_i$, $C_2 \in F_i$ imply $C_1 \subseteq C$ if and only if $C_2 \subseteq C$.*

Proof. We shall prove that for triply connected graphs conditions a) and b) in the definition of a pseudoembedding ensure that given a cycle C and an edge $e \notin E(C)$ the set of edges of G can be partitioned into three classes:

$E(C)$,

$S_e = \{f | f \in E(G)\ \mathrm{IN}(C, e) = 1 \Leftrightarrow \mathrm{IN}(C, f) = 1\}$ ("same domain as e"),

$\mathrm{Opp}_e = \{f | f \in E(G)\ \mathrm{IN}(C, e) = 1 \Leftrightarrow \mathrm{IN}(C, f) = 0\}$ ("opposite domain").

This implies the truth of the lemma: let

$F_1 = \{C_i \in F : \exists f \in S_e \cap E(C_i)\}$;

$F_2 = \{C_2 \in F : \exists f \in \mathrm{Opp}_e \cap E(C_i)\}$.

By condition a), the F_i's are well defined and $F_i \cap F_2 = \varnothing$. So $\{F_1, F_2\}$ is a partition of F.

Suppose $C_1 \subseteq C$, $C_1 \in F_i$, and let $C_2 \in F_i$. Let $f_1 \in E(C_1) - E(C)$, $f_2 \in E(C_2) - E(C)$. Since $C_1 \subseteq C$, there exists a $g \in E(C_1)$ such that $\mathrm{IN}(C, g) = 1$ (by the definition of $\subseteq$). So $\mathrm{IN}(C, f_1) = 1$ by condition a), and $\mathrm{IN}(C, f_2) = 1$ because of the definitions of S_e, Opp_e and F_i. (For example, if $i = 1$, $\mathrm{IN}(C, f_1) = 1 \Rightarrow \mathrm{IN}(C, e) = 1 \Rightarrow \mathrm{IN}(C, f_e) = 1$.) Thus $C_2 \subseteq C$. Similarly, if $C_2 \subseteq C$ then $C_1 \subseteq C$. To prove the required partition of $E(G)$, we shall reason about the bridges of C in G. If C has a single bridge, the statement is obviously true, with Opp_e (or S_e) empty. So assume C is not peripheral. Let e be an edge not in C, and without loss of generality, suppose $\mathrm{IN}(C, e) = 1$. Let I_1 be the bridge containing e, and let $v_{11}, \cdots, v_{1l_1}$ be the vertices of attachment of I_1 to C. Let I_{u+1} be the set of bridges that conflict with some bridge in I_u and are not in $\bigcup_{j=1}^{u-1} I_j$. By condition b), for every edge g in (a bridge in) I_{u+1}, $\mathrm{IN}(C, g) = 1$ if and only if $\mathrm{IN}(C, f) = 0$ for some edge of I_u (and, therefore, for every edge of I_u). There exists some k such that $I_l = \varnothing$ for $l > k$ since there are only finitely many bridges of C. It remains to show that every bridge is in some I_j—if this is true, then $\bigcup_{j \text{ even}} I_j$ and $\bigcup_{j \text{ odd}} I_j$ form the sets S_e and Opp_e.

Let B be a bridge of C, $B \in F_k$. Let $a_1, a_2, \cdots, a_l$ be vertices of attachment of B to C in a cyclic ordering of the vertices of C. Consider the paths P_i, from a_i to a_{i+1} in C ($a_{l+1} = a_1$). Assume, by contradiction, that there is a bridge D that does not conflict with B nor with any other bridge in $\bigcup I_j$. The vertices of attachment of D to C must lie entirely within a single path P_i (or D would conflict with B). Let a_1 and b_1 be the first and last vertices of attachment of D to C, in the ordering of vertices of P_i. Since the removal of (a_1, b_1) must not disconnect G, there must be a bridge D' that conflicts with D, that has vertices of attachment in P_i and at vertices a_2, b_2 in C, where $a_2 < a_1$ or $b_2 > b_1$ (or both). If one of a_2, b_2 is not in P_i, then D' conflicts with B. Since $I_{k+1} = \varnothing$, $D' \in I_j$ for some $j < k$. But then $D \in I_{j+1}$. If both a_2, b_2 lie in P_i, repeat the procedure with D', eventually yielding a conflict (or a proof that G is not 3-connected). □

From the proof of Lemma 4.4, we can obtain:

COROLLARY 4.4. *Let G be planar, and let C be a cycle of G, $C \in F$. Then the partition S_e, Opp_e of the edges in $E(G) - E(C)$ corresponds to the two possible sets of edges that may be mapped into the bounded domain of the plane in plane embeddings in G.*

Proof. Since the embedding of G into the sphere is unique, it follows that the set of edges in the two domains of the sphere determined by the map of C are uniquely determined. In the plane embeddings of G, all edges in exactly one

of these two classes can be mapped into the interior of the Jordan curve determined by C.

Consider the pseudoembeddings of G obtained from embeddings. By the paragraph above, there are exactly two possible distinct assignments of values to the set of variables $\{\text{IN}(C, e)e \in E(G) - E(C)\}$. On the other hand, Lemma 4.4 shows that there are two possible distinct assignments of values to $\{\text{IN}(C, e) | e \in E(G) - E(C)\}$ that can be extended to pseudoembeddings. Hence, the corollary follows. □

LEMMA 4.5. *Let G be planar and let $C \in F$. Let P be a pseudoembedding, and let $\{D_i\}$ be the set of fundamental cycles such that $D_i \subseteq C$ in P. Then there is a plane embedding Δ of G such that P restricted to $\{C\} \cup \{D_i\}$ represents Δ. (In other words, the pseudoembedding P has the intended meaning in the interior of C.)*

Proof. Corollary 4.4 shows that there are embeddings Δ such that e is inside Γ_c in Δ if and only if $\text{IN}(C, e) = 1$ in P. Let D be a fundamental cycle, $D \subseteq C$. By condition c) of the definition of pseudoembedding, there must be an $f \in E(C) - E(D)$ with $\text{In}(D, f) = 0$ in P. But in Δ, e is inside Γ_c for all edges $e \in E(D) - E(C)$, so we must have f outside Γ_D. By Corollary 4.4, the set of edges f such that $\text{IN}(D, f) = 0$ corresponds to one of the sets of edges mapped into the same domain of the plane, and all such edges are mapped into the same domain as f. Thus, all the edges g with $\text{IN}(D, g) = 0$ are mapped into the unbounded domain, relative to Γ_D. Similarly, all the edges g with $\text{IN}(D, g) = 1$ are mapped into the other domain and, hence, to the interior of Γ_D. Therefore, P restricted to D represents Δ. Since D was arbitrary (subject to $D \subseteq C$) and P restricted to C represents Δ, the lemma follows. □

We note that the proof implies something slightly stronger—not only are there plane embeddings, where P has the intended meaning in the interior of C, but P represents all such embeddings.

The next lemma is a property of plane embeddings. It will be used in the proof of Theorem 4.7. (We use the notation defined after Theorem 4.1.)

LEMMA 4.6. *Let G be planar. Assume G has cycles $C_1, C_2, \cdots, C_k$ and D, and a plane embedding Δ such that:*

(1) in Δ, for all $1 \leqq i < j \leqq k$, Γ_i lies outside Γ_j and Γ_j lies outside Γ_i;

(2) for all $1 \leqq i \leqq k$, Γ_i lies inside Γ_D;

(3) there is an edge $e \in (E(D) - \bigcup_{i=1}^{k} E(C_i))$.

Then there is an embedding Δ' of G, such that:

(1) for all $1 \leqq i \leqq k$, Γ_i' (the image of C_i in Δ') has in its interior (the images of) exactly the same edges as does the interior of Γ_i;

(2) for all i, Γ_i' lies outside Γ_D.

Proof. We only sketch the proof since it is a topological property of embeddings and not relevant to the rest of this paper.

Consider a plane mesh M for G. There are exactly two peripheral cycles in M, say H and H', that contain the edge e of D not in the C_i's. Exactly one of these, say H', is mapped to the interior of Γ_D in Δ. Let Δ' be the plane embedding of G, where H' is the boundary of the external face. Then Δ' has the desired properties. □

We are now ready to prove the partial converse to Lemma 4.2, i.e., that for planar graphs every pseudoembedding represents an embedding.

THEOREM 4.7. *Let G be a planar graph and P a pseudoembedding of G. Then there is a plane embedding Δ of G such that P represents Δ.*

Proof. Let F, the set of fundamental cycles of G used in P, have k cycles. We give an inductive procedure that yields at each step a set E of embeddings and a subset S of cycles, such that for every embedding $\Delta \in E$, P restricted to the cycles in S represents Δ, and, moreover, E is a collection of all such embeddings. At each step,

the cardinality of S increases, so the algorithm terminates since $|F| = k$. We shall prove that E is nonempty at each step. This yields the theorem, when $S = F$. At step i, we will add a collection of cycles to S in such a way that if $D \in F - S$, then $D \subseteq C$ for $C \in S$ (i.e., the cycles not yet in S are not IN any cycle in S in the pseudoembedding P).

We denote the sets S and E, at the end of step i, by S_i and E_i, respectively.

Step 0. $S_0 = \varnothing$ $E = \{$plane embeddings of $G\}$.

Step $i+1$. Choose a cycle C in $F - S_i$ that is maximal with respect to $\subseteq$—i.e., for all $D \in F$ $\neg C \subseteq D$ in P.

$$S_{i+1} = S_i \cup \{C\} \cup \{D \in (F - S_i) \,|\, D \subseteq C\},$$
$$E_{i+1} = \{\Delta \in E_i \,|\, \text{in } \Delta, \Gamma_i \text{ lies outside } \Gamma_c \text{ for all } C_i \in S\}.$$

The construction has the desired properties—we have to show it can be carried out.

(a) There is a maximal cycle C in $F - S_i$. By construction, if $C \subseteq D$ for some $D \in S_i$, then $C \in S_i$. Pick a cycle $C_1 \in F - S_i$. If, for all $D \in F - S_1$, $C_1 \subseteq D$, pick $C = C_1$, otherwise, let C_2 be a cycle such that $C_1 \subseteq C_2$ and continue the procedure with C_2. The procedure either terminates with a cycle C as desired or we have found a sequence of cycles $C_1, C_2, \cdots, C_l$, with $C_2 \subseteq C_2$, $C_2 \subseteq C_3$, $C_{l-1} \subseteq C$, $C_l \subseteq C_1$. But such a sequence contradicts Lemma 4.5; the inside of Γ is represented by P in any embedding that maps an edge e with $\text{IN}(C, e) = 1$ into the interior of Γ (such edge must exist since $C_{l-1} \subseteq C_l$). But in this embedding, by Lemma 4.5, Γ_{l-1}, lies inside Γ_l, Γ_{l-2} inside Γ_{l-1} and hence inside Γ_l, and, finally, Γ_1 lies inside Γ_l. But then, since P restricted to C_2 represents the embedding, $\text{IN}(C_2, f) = 0$ for some $f = E(C) - E(C_2)$ and, hence, $\neg C_l \subseteq C_1$.

(b) E_{i+1} is nonempty. First note that by construction C is such that edges of cycles in S have the same value of $\text{IN}(C, e)$ in all pseudoembeddings. By Corollary 4.4, in any embedding of G, all these edges are mapped into the same domain, relative to Γ_c. If for some $E \in E_i$ these edges are mapped into the unbounded region, then we are done: by induction, the embedding is represented by P restricted S_i, P restricted to C represents E and by Lemma 4.5, P restricted to $S_{i+1} - (S_i \cup \{C\})$ also represents E.

So assume $E \in E_i$ but some edge of S_i is mapped to the interior of Γ_c. Then all cycles of S_i are mapped to the interior of Γ_c. By Lemma 4.6, there is an embedding E' in which the interiors Γ'_j of all $C_j \in S_i$ are the same as in E, but the Γ_j's lie outside Γ'_c. Then P restricted to S_i represents E' (since on edge is mapped inside Γ'_j if and only if it is mapped inside Γ_j) and, hence, $E' \in E_i$. But P restricted to S_{i+1} represents E': P restricted to C has the intended meaning by the construction of E', and by Lemma 4.5, P has the intended meaning for all cycles D in the interior of C.

Finally, note that the definition of S_{i+1} ensures that $D \in F - S_{i+1}$ implies $D \subseteq C$ for some $C \in S$, as claimed.

Thus, the required embedding can be produced. □

Our planarity algorithm, as explained before, does not attempt to mimic the construction above. We need an additional construction that, given a pseudoembedding for a planar graph, finds a plane mesh for it. First we note some properties of plane embeddings.

Let G be a planar graph, F a set of fundamental cycles $F = \{C_i | i = 1, 2, \cdots n\}$ and Δ a plane embedding of G. Let D_{ij} be the set of peripheral cycles that are mapped

to the interior of Γ_i in Δ. Then

$$C_i = \sum_j D_{ij}, \tag{1}$$

where the sum is the usual sum of cycles [H], [BM].

We shall obtain a new cycle basis $\{C'_i \mid i = 1, 2, \cdots, n\}$ for G, where the C'_i are peripheral. We formalize the following idea: some of the C_i are peripheral. Consider some C_j that (in E) contains only peripheral cycles C_i. Then $C'_j = C_j + \sum_{i \in A} C_i$, with $A = \{i \mid C_i \text{ is mapped inside } \Gamma_j \text{ by } \Delta\}$, is a peripheral cycle.

More formally, given a pseudoembedding P, define the relation $C_i < C_j$ (C_j directly surrounds C_i) among cycles of F by $C_i < C_j$ if and only if $C_i \subseteq C_j$ & $\exists D, D \in F, C_i \subseteq D$ & $D \subseteq C_j$. Since P represents an embedding, $<$ is an order relation.

For $C_i \in F$, define

$$\text{level (i)} = \begin{cases} 0 & \text{if } \{C_j \mid C_j < C_i\} = \varnothing, \\ k+1 & \text{if } k = \max \{\text{level } (j) \mid C_j < C_i\}. \end{cases}$$

Define $\{C'_i \mid i = 1, 2, \cdots n\}$ by

$$C'_i = \begin{cases} C_i & \text{if level } (i) = 0, \\ C_i + \sum\limits_{C_j \in \text{Int}(i)} C_j, & \text{where Int } (i) = \{C_j \mid C_j < C_i\}. \end{cases}$$

LEMMA 4.8. *$\{C_i \mid i = 1, \cdots, n\}$ is a peripheral cycle basis for G (and, therefore, $\{C'_i\} \cup \{\sum_{i=1}^n C'_i\}$ is a plane mesh for G).*

Proof. By induction on level (i), assume that for level (i) $\leqq k$, C_j can be expressed as a linear combination of the C'_k such that C_k is mapped inside Γ_j. Let C'_i have level $k+1$. It follows from the definition of C'_i that

$$C'_i = C_i + \sum_{C_j \in \text{Int } (i)} C_j = C_i + \sum_{l \in A} C', \quad \text{where } A \subseteq \{l \mid C_l \text{ is mapped inside } \Gamma_j\}.$$

So $C_i = C'_i + \sum_{l \in A} C'_l$. Since, for level (i) $= 0$, $C'_i = C_i$, by induction, $\{C'_i\}$ is a basis. By (1), and since for every peripheral cycle there is a first time it gets included in an inclusion chain of cycles, C'_i is a peripheral cycle. More precisely, we prove by induction on level (i) that C'_i is peripheral. If level (i) $= 0$, this is clear. Assume true for levels $\leqq k$, and let level (i) $= k+1$. We may write C_i as the sum of all peripheral cycles mapped into Γ_i but not inside Γ_j for $j \in \text{Int}$ (i). Thus, every peripheral cycle appears exactly once in the sums for C'_i, and since there are nC'_i's and the peripheral cycle that is mapped to the external face cannot be mapped inside the C'_i, we conclude that exactly one peripheral cycle lies inside each C'_i, and so, C'_i is this peripheral cycle. □

Note that the definition of the C'_i uses only the combinatorial notions of a pseudoembedding, and the process of computing the C'_i can be carried out for any graph with a pseudoembedding. If the graph is planar, this yields a peripheral basis and a plane mesh.

These results imply the following efficient parallel algorithm for planarity (we assume that the algorithm that finds triconnected components will also check that the component G has at most $3n-6$ edges):

ALGORITHM

Planar-1(G).

1. Find a spanning tree T of G and the corresponding set of fundamental cycles, F.

2. Try to find a pseudoembedding—if this fails, report G is nonplanar. In more detail,

2.1. For each $C_i \in F$, find all the bridges B_i of G relative to C_i.

2.2. For each cycle $C_i \in F$ and for every edge $e \notin E(C_i)$, use a variable $\mathrm{IN}(C_i, e)$. Write down the boolean formulas that ensure that conditions (1), (2) and (3) for a pseudoembedding hold. (Note that this yields a 2-CNF formula.)

2.3. Obtain a satisfying assignment for the formula above. This is a pseudoembedding. If the formula is unsatisfiable, G is nonplanar.

3. Compute the relation $<$ among cycles C_i in F.

4. Obtain the C_i' as in Lemma 4.8. Let $F' = \{C_i'\} \cup \{\sum C_i'\}$.

5. Test whether every edge appears in exactly two cycles of C_i'. If so, G is planar, otherwise it is nonplanar.

THEOREM 4.9. *Planar-1(G) correctly tests planarity of G. It can be implemented to run in $O(\log^2 n)$ time on $O(n^4)$ processors, where $n = |V(G)|$.*

Proof. Correctness follows from results of this section. We sketch the time analysis below.

Step 1 can be done in time $O(\log^2 n)$ with $O(n^2)$ processors ([SJ]).

Step 2.1 is very similar to the connectivity problems in [SJ] or the 3-connected components algorithm in this paper. Every cycle requires $O(n^2)$ processors to find its bridges in $O(\log^2 n)$ time. Since there are at most $O(n)$ cycles (we precede Planar-1 by a procedure that counts the number of vertices and the number of edges and rejects if there are more than $3n-6$ edges) $O(n^3)$ processors suffice for this step.

Step 2.2 can be done in parallel since conflicts among bridges are easy to detect. The number of processors is $O((\text{number of cycles})^2 \times (\text{number of edges})^2) = O(n^4)$, and the time $O(\log n)$. The number of terms in the 2-CNF formula is at most $O(n^4)$.

Step 2.3 is a 2-CNF satisfiability algorithm, described below. There are $O(n^2)$ variables and $O(n^4)$ formulas: $O(\log^2 n)$ time and $O(n^4)$ processors suffice.

Step 3 takes $O(\log n)$ time with $O(n^3)$ processors, as does step 4 (step 4 consists of solving a set of linear recurrences). For step 5, $O(n^2)$ processors and $O(\log n)$ time suffice.

Thus, the whole procedure can be carried out in $O(\log^2 n)$ parallel steps, using $O(n^4)$ processors. □

The 2-CNF satisfiability problem can be solved as follows:

For each variable x, maintain 4 lists of variables:

$$PP_x = \{y \mid \text{the CNF formula forces } y = 1 \text{ if } x = 1\},$$

$$PN_x = \{y \mid \text{the CNF formula forces } y = 0 \text{ if } x = 1\},$$

$$NP_x = \{y \mid \text{the CNF formula forces } y = 1 \text{ if } x = 0\},$$

$$NN_x = \{y \mid \text{the CNF formula forces } y = 0 \text{ if } x = 0\},$$

Initially, $PP_x(0) = \{y \mid (\bar{x} \vee y) \text{ is a clause}\}$, and

$$PP_x(t+1) = PP_x(t) \cup \Big(\bigcup_{z \in PP_x(t)} PP_z(t) \Big) \cup \Big(\bigcup_{z \in PN_x(t)} NP_z(t) \Big).$$

The other lists are similarly updated. For a k-variable CNF at most $\log k$ updates are necessary and each takes at most $O(\log k)$ steps.

COROLLARY 4.10. *Planarity testing can be carried out in space $O(\log^4 n)$ by a Turing machine.*

This follows by standard simulation of parallel computers [HS], [FW] by tape-bounded Turing machines. Actually, we can obtain much sharper results: graph planarity can be tested in $\log^2 n$ space by a Turing machine [JS]. Unfortunately, the space-efficient algorithms do not run in polynomial time.

5. The second planarity algorithm. Tutte describes in [T] a method of embedding a triconnected planar graph in the *Cartesian plane*. We sketch the main ideas behind this method.

Let J be a peripheral cycle of a triconnected graph $G=(V,E)$. Let p be the number of vertices of J. Let Q be a p-sided convex polygon in the Euclidean plane. We view Q as an embedding of J in the Euclidean plane. Let f be a one–one mapping of $V(J)$ onto the set of vertices of Q such that the cyclic order of vertices in J agrees, under f, with the cyclic order of vertices of Q.

Let $V=\{v_1, v_2, \cdots, v_n\}$ be an enumeration of the vertices of V such that $\{v_1, \cdots, v_p\}$ are the vertices of J. We try to extend f to other vertices of G. Let v_i be any vertex, and let $A(i)$ be the set of vertices adjacent to v_i. Suppose $f(v_j)$ is defined for each $v_j \in A(i)$. Put a unit mass m_j at each point $f(v_j)$ for each $v_j \in A(i)$. Then $f(v_i)$ is defined to be the centroid of the masses m_j, $v_j \in A(i)$; i.e., if $f(v_l)=(x_l, y_l)$, $1 \leqq l \leqq n$, we have

$$x_i = \frac{x_{j_1}+x_{j_2}+\cdots+x_{j_k}}{k}, \qquad y_i = \frac{y_{j_1}+y_{j_2}+\cdots+y_{j_k}}{k} \quad \text{if } A(i)=\{v_{j_1}, \cdots, v_{j_k}\}.$$

In order to prove that such an Euclidean embedding is possible if the graph is planar, we define the $m \times n$ matrix $A=(a_{ij})$ such that

$$a_{ij} = \begin{cases} -1 & \text{if } i \neq j \text{ and } G \text{ contains the edge } (v_i, v_j), \\ \deg(i) & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases}$$

Consider the following linear system of equations:

$$\sum_{j=1}^{n} a_{ij}x_j = 0, \qquad \sum_{j=1}^{n} a_{ij}y_j = 0, \qquad 1 \leqq i, \quad j \leqq n.$$

Let (x_i, y_i) be the coordinates of the vertices of Q, $1 \leqq i \leqq p$. Let v_l be any vertex such that $l > p$. Then we have

$$a_{il}x_l = -\sum_{\substack{j=1 \\ j\neq l}}^{n} a_{ij}x_j, \qquad a_{il}y_l = -\sum_{\substack{j=1 \\ j\neq l}}^{n} a_{ij}y_j, \qquad 1 \leqq i \leqq n.$$

If $i=l$, then $a_{ij} = \deg v_l$ and, for $j \neq l$, either $a_{ij}=-1$ or 0, depending on whether v_j is adjacent to v_l or not. Thus,

$$(\deg v_l)x_l = -\sum_{\substack{j \\ (v_j, v_l)\in E}} (-x_j) = \sum_{\substack{j \\ (v_j, v_l)\in E}} x_j$$

and, similarly,

$$(\deg y_l)y_l = \sum_{\substack{j \\ (v_j, v_l)\in E}} y_j.$$

Therefore, if the linear system of equations (*) for $p < i \leqq n$ has a solution, (x_i, y_i) is the centroid of the masses m_j, $v_j \in A(i)$. We now prove that this system has a unique

solution. Decompose A as follows:

$$A = \begin{bmatrix} p & n-p \\ A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{matrix} \\ \}p \\ \}n-p \end{matrix}.$$

Let G_0 be the graph obtained from G by deleting the edges of J and identifying all the vertices of J to form a new single vertex. With suitable enumeration of the vertices of G_0, we can say that A_{22} is obtained from the matrix $A(G_0)$ associated with G_0 by striking out the first row and column, i.e.,

$$\det(A_{22}) = \text{a cofactor } (A(G_0)).$$

But it is a classical result that any cofactor of $A(G_0)$ is equal to the number of minimum spanning trees in G_0 (see [BM § 12.2 and Exercise 12.22], where the matrix $A(G_0)$ is called the *conductance matrix*). It follows that the above system of equations has always a unique solution. The corresponding embedding is called a *barycentric representation* of G. In [T], it is shown that the above embedding has the following property.

THEOREM 5.1. *Let G be a planar triconnected graph. Let J be a peripheral polygon of G with p vertices. Let Q be a p-sided convex polygon in the Euclidean plane. Then there is a unique barycentric representation of G on Q mapping the vertices of J onto the vertices of Q in any arbitrarily specified way preserving the cyclic order.*

We now describe the second planarity algorithm.

ALGORITHM

Planar-2(G).

1. Find a peripheral cycle C of G (which will be the outer face) with p vertices. Find p Cartesian points (x_i, y_i), $1 \leqq i \leqq p$, that form a convex polygon in the Euclidean plane.

2. Compute the entries of the conductance matrix A of G.

3. Solve the linear set of equations

$$\sum_{j=1}^{n} a_{ij}x_j = 0, \qquad \sum_{j=1}^{n} a_{ij}y_j = 0, \qquad p < i \leqq n,$$

obtaining a set of coordinate pairs (x_l, y_l), $l = p+1, \cdots, n$. (Note that, if G is planar, the coordinates (x_i, y_i), $1 \leqq i \leqq n$, constitute a plane embedding of G.)

4. For every pair $\{l, f\}$ of edges of G, write down the equations of the line segments that represent these edges in the embedding obtained in step 3 and check that they do not cross. If so, G is planar, otherwise, G is nonplanar.

THEOREM 5.2. *Planar*-2(G) *correctly checks if a given triconnected graph $G = (V, E)$ is planar. This algorithm can be implemented to run in time $O(\log^2 n)$, using $O(n^{3.29}/\log^2 n)$ processors, where we assume that each processor can multiply and divide.*

Proof. The correctness proof follows directly from Theorem 5.1. Step 1 can be done, as in Planar 1, by finding a spanning tree and the associated cycle basis. One of those cycles is peripheral and computation of the bridges of all the fundamental cycles will find it. The time requirement is $O(\log^2 n)$, with $O(n^3)$ processors. Step 2 can be trivially done in $O(1)$ time, with $O(n^3)$ processors. Step 3 involves solving a linear system of equations which can be done by using the improvement over Csanky's method [C], reported in [PSa]. This takes $O(\log^2 n)$ steps with $O(n^{3.29}/\log^2 n)$ processors. Step 4 can be done in $O(\log n)$ time, with $O(n^3)$ processors. Therefore, Planar-2 runs in $O(\log^2 n)$ time, with $O(n^{3.29}/\log^2 n)$ processors. □

Note that the processor counts in Planar-1 and Planar-2 refer to distinct models of computation: the processors for Planar-2 must have unit cost multiplication, while those in Planar-1 are essentially bit processors.

REFERENCES

[Br] R. P. BRENT, *The parallel evaluation of general arithmetic expressions*, J. Assoc. Comput. Mach., 21 (1974), pp. 201–206.

[BL] K. S. BOOTH AND G. S. LUEKER, *Testing for the consecutive ones property, interval graphs and graph planarity using PQ-tree algorithms*, J. Comp. System. Sci., 13 (1976) pp. 335–379.

[BM] J. A. BONDY AND U. S. R. MURTY, *Graph Theory with Applications*, American Elsevier, New York, 1977.

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

[Ch] A. K. CHANDRA, *Maximal parallelism in matrix multiplication*, IBM Tech. Rept. RC 6193, Sept. 1976.

[Ev] S. EVEN, *Graph Algorithms*, Computer Science Press, Rockville, MD, 1979.

[F] M. J. FLYNN, *Very high-speed computing systems*, Proc. IEEE, 54 (1976), pp. 1901–1909.

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

[G1] C. M. GOLDSCHLAGER, *A unified approach to models of synchronous parallel machines*, Proc. 10th Annual ACM Symposium on Theory of Computing, San Diego, CA, 1978, pp. 89–94.

[G2] ———, Synchronous parallel computation, Tech. Rep. 114, University of Toronto, 1977.

[H] F. HARARY, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.

[He] D. HELLER, *A survey of parallel algorithms in numerical linear algebra*, SIAM Rev., 20, (1978), pp. 740–777.

[Hi] D. C. HIRSCHBERG, *Parallel algorithms for the transitive closure and the connected components problems*, Proc. 8th Annual ACM Symposium on Theory of Computing, 1976, pp. 55–57.

[HS] J. HARTMANIS AND J. SIMON, *On the power of multiplication in random access machines.*, Proc. 15th SWAT Conf., New Orleans, LA, 1974, pp. 13–23.

[HT] J. E. HOPCROFT AND R. E. TARJAN, *Efficient planarity testing*, J. Assoc. Comput. Mach., 21, 4 (1974), pp. 549–568.

[J] J. JA' JA', *Graph connectivity problems on parallel computers*, Tech. Rep. CS-78-05, Dept. Computer Science, Pennsylvania State Univ., University Park, Feb. 1978.

[JS] J. JA' JA' AND J. SIMON, *Some space-efficient algorithms*, Proc. 17th Allerton Conference, 1979, pp. 677–684.

[McL1] S. MCLANE, *A combinatorial condition for planar graphs*, Fundamenta Math., 28 (1937), pp. 22–32.

[McL2] ———, *A structural characterization of planar combinatorial graphs*, Duke Math. J., 3 (1937), pp. 46–472.

[P] F. P. PREPARATA, *Parallelism in sorting*, International Conference on Parallel Processing, Belair, Michigan (August 1977).

[PS] V. R. PRATT AND L. J. STOCKMEYER, *A characterization of the power of vector machines*, J. Comput. Systems Sci., 12, (1976), pp. 198–221.

[PSa] F. P. PREPARATA AND D. V. SARWATE, *An improved parallel processor bound in fast matrix inversion*, IPL, 7, 3 (1978) pp. 148–150.

[SJ] C. SAVAGE AND J. JA' JA', *Fast, efficient parallel algorithms for some graph problems*, this Journal, 10 (1981), pp. 682–690.

[T] W. T. TUTTE, *How to draw a graph*, Proc. London Math. Soc., 3, 13 (1963), pp. 743–768.

SIAM J. COMPUT.
Vol. 11, No. 2, May 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1102-0010 $01.00/0

PLANAR FORMULAE AND THEIR USES*

DAVID LICHTENSTEIN†

Abstract. We define the set of planar boolean formulae, and then show that the set of true quantified planar formulae is polynomial space complete and that the set of satisfiable planar formulae is NP-complete. Using these results, we are able to provide simple and nearly uniform proofs of NP-completeness for planar node cover, planar Hamiltonian circuit and line, geometric connected dominating set, and of polynomial space completeness for planar generalized geography.

The NP-completeness of planar node cover and planar Hamiltonian circuit and line were first proved elsewhere [M. R. Garey and D. S. Johnson, *The rectilinear Steiner tree is NP-complete*, SIAM J. Appl. Math., 32 (1977), pp. 826–834] and [M. R. Garey, D. S. Johnson and R. E. Tarjan, *The planar Hamilton circuit problem is NP-complete*, SIAM J. Comp., 5 (1976), pp. 704–714].

Key words Computational complexity, NP-completeness, P-space-completeness, combinatorial games, planar graphs

1. Motivation. Many properties that are NP-complete for general graphs are also NP-complete for planar graphs. (Others, such as max clique and max cut, are significantly easier to test for on planar graphs, unless $P = NP$.) Proofs of planar NP-completeness often involve two stages, typified by the proof that planar node cover is NP-complete [4]. The first stage is the proof for general graphs, the second is the construction of a complicated crossover box, which is added to the nonplanar reduction everywhere two arcs cross. Unfortunately, such crossover boxes are hard to find and hard to understand.

In this paper, we present a crossover box whose planarity is invariant under many polynomial reductions. In this way, we argue that various planar completeness results are "true for the same reason". Our technique may therefore be a useful tool to use in attempts to strengthen general results to their planar subcases.

2. Preliminaries.

(1) A boolean formula B in conjunctive normal form with at most 3 variables per clause (3CNF) is a set of clauses $B = \{c_1, \cdots, c_m\}$. Each clause is a subset of 3 literals from the sets $V = \{v_1, \cdots, v_n\}$ and $\bar{V} = \{\bar{v}_1, \cdots, \bar{v}_n\}$. For convenience, clauses will be written $(a + b + c)$ instead of $\{a, b, c\}$.

(2) The set of quantified boolean formulae with at most 3 variables per clause $(3\text{QBF}) = \{Q_1v_1Q_2v_2 \cdots Q_nv_nB(v_1, v_2, \cdots, v_n) | Q_i \in \{\forall, \exists\}$, where the v_i are boolean variables and B is in 3CNF$\}$.

(3) TF is the set of true formulae in 3QBF. We will also refer to the problem of recognizing this set as TF.

(4) 3SAT is the subset of TF where all variables are existentially quantified.

(5) The variable v_i occurs m_{n_i} (abbreviated m_i) times, negated or unnegated, in B.

(6) We use as few subscripts as possible, for the sake of readability. Most structures will be described by picture and example, rather than formally.

(7) It will sometimes be convenient to coalesce certain subgraphs into a single macro node. The macro node is then adjacent to all nodes which were originally adjacent to some node in the subgraph replaced by the macronode. This coalescing will be signified pictorially by means of a dotted line around the subgraph.

---

* Received by the editors January 20, 1978, and in final revised form March 25, 1981.

† Department of Computer Science, Yale University, New Haven, Connecticut 06520. This research was partially supported by the National Science Foundation under grant MCS76-17605.

(8) Each problem in the paper is trivially in NP, except for generalized geography, which is trivially in P-space.

3. Planar formulae. In this section, we prove the main results of the paper, the P-space-completeness of the planar quantified boolean formula problem, and the NP-completeness of planar satisfiability. Since 3SAT is just TF with all variables existentially quantified, the same reduction reduces TF to planar TF and 3SAT to planar 3SAT. TF was shown to be P-space-complete in [9]; 3SAT was shown to be NP-complete in [2].

DEFINITION. Let $B \in$ Q3CNF. We call $G(B)=(N, A)$ the graph of B, where $N=\{c_j | 1 \leqq j \leqq m\} \cup \{v_i | 1 \leqq i \leqq n\}$. $A = A_1 \cup A_2$ where

$$A_1 = \{\{c_i, v_j\} | v_j \in c_i \text{ or } \bar{v}_j \in c_i\}, \qquad A_2 = \{\{v_j, v_{j+1}\} | 1 \leqq j < n\} \cup \{\{v_n, v_1\}\}.$$

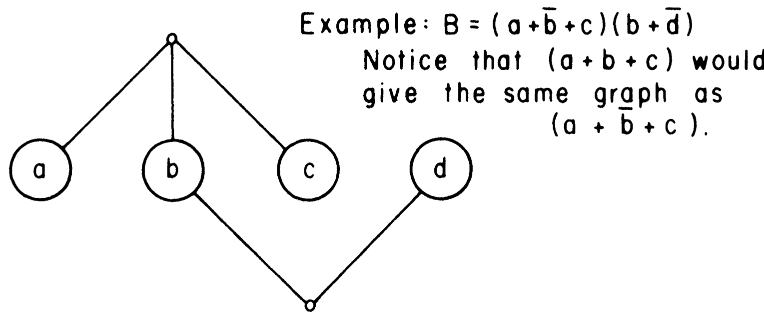


FIG. 1

DEFINITION. *The planar quantified boolean formula problem* (PTF) *is* TF *restricted to formulae B such that $G(B)$ is planar.*

THEOREM 1. PTF *is* P-*space complete.*

Proof. We give a polynomial time algorithm that converts a formula B in 3QBF into a formula PB such that:

(i) *$G(PB)$ is planar*;

(ii) *$PB \Leftrightarrow B$.*

The algorithm proceeds as follows. Draw $G(B)$ on a grid. The grid is $3m \times 3m$, with nodes arranged on the left and bottom borders. The set of clauses $\{c_i\}$ lies along the left border, with each node covering the end points of 3 adjacent horizontal grid lines. The variables $\{v_j\}$ lie along the bottom border, with each node v_i covering the end points of m_i vertical lines of the grid. Grid lines are then darkened in the obvious manner, so that each arc in A_1 consists of a horizontal segment and a vertical segment. A_2 is obtained simply by joining adjacent variables with an arc (see Fig. 2).

We now modify the formula so that nonplanarity is eliminated in A_1, and then further modify the formula so that A_2 can be drawn without introducing nonplanarity.

Pick a point on the graph where two arcs cross, involving, for instance, the variables a and b (see Fig. 3).

Replace that section of the graph by the subgraph shown in Fig. 4, $G(X)$, where the small unlabeled nodes in the picture represent clauses of X. Viewing B as a string, the clauses of X are appended to B, and a new quantifier block existentially quantifying the new variables in X is inserted between the last quantifier of B and the

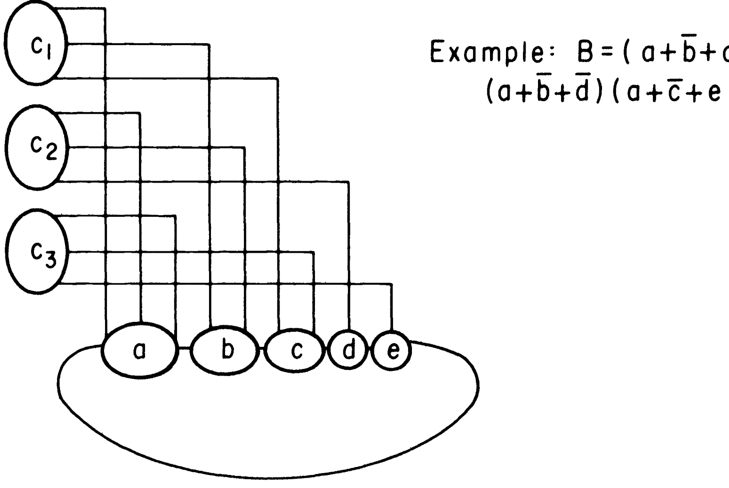


FIG. 2

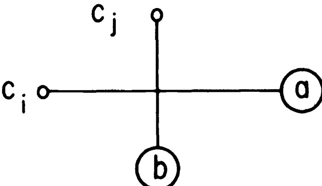


FIG. 3

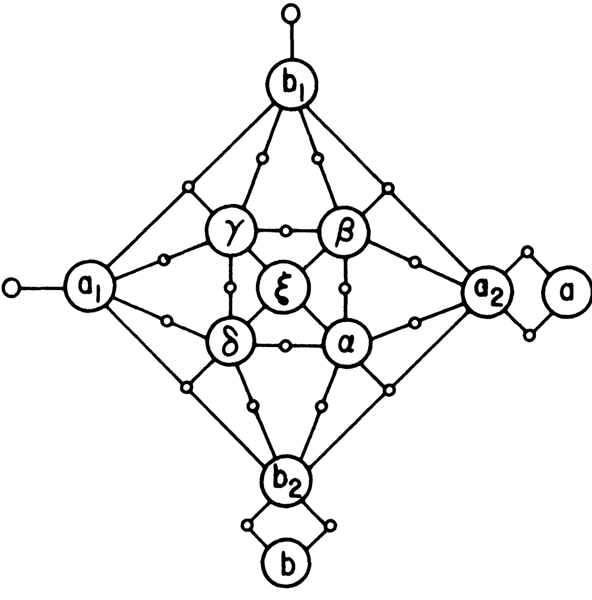


FIG. 4

beginning of the formula. X is comprised of the following!

$$(\bar{a}_2+\bar{b}_2+\alpha)(a_2+\bar{\alpha})(b_2+\bar{\alpha}), \quad \text{i.e., } a_2b_2\Leftrightarrow\alpha;$$

$$(\bar{a}_2+b_1+\beta)(a_2+\bar{\beta})(\bar{b}_1+\bar{\beta}), \quad \text{i.e., } a_2\bar{b}_1\Leftrightarrow\beta;$$

$$(a_1+b_1+\gamma)(\bar{a}_1+\bar{\gamma})(\bar{b}_1+\bar{\gamma}), \quad \text{i.e., } \bar{a}_1\bar{b}_1\Leftrightarrow\gamma;$$

$$(a_1+\bar{b}_2+\delta)(\bar{a}_1+\bar{\delta})(b_2+\bar{\delta}), \quad \text{i.e., } \bar{a}_1b_2\Leftrightarrow\delta;$$

$$(\alpha+\beta+\gamma+\delta);$$

$$(\bar{\alpha}+\bar{\beta})(\bar{\beta}+\bar{\gamma})(\bar{\gamma}+\bar{\delta})(\bar{\delta}+\bar{\alpha});$$

$$(a_2+\bar{a})(a+\bar{a}_2)(b_2+\bar{b})(b+\bar{b}_2), \quad \text{i.e. } a\Leftrightarrow a2, \quad b\Leftrightarrow b2;$$

and a new quantifier block existentially quantifying the new variables is appended to the list of quantifiers.

At the same time a or $\bar{a}$ is replaced in c_i with a_1 or $\bar{a}_1$ and b or $\bar{b}$ is replaced in c_j with b_1 or $\bar{b}_1$.

It is clear from the picture that the new graph has one less crossover point, and one can easily verify that X is satisfiable if and only if $[a_1\Leftrightarrow a]$ and $[b_1\Leftrightarrow b]$.

The algorithm repeats the above replacement at each crossover point, starting at lower right and moving up and left, using new auxiliary variables each time, until the graph is finally planar.

At each stage of the algorithm, only a constant amount of work is done, and there are no more than $9m^2$ stages.

Now we draw in A_2 without disturbing the planarity of the graph. Since all of the new variables are in the same existence block, we are free to order them arbitrarily. Taking another look at our planar crossover box, we notice that there is a simple path linking all of its variables (i.e., the dark lines in Fig. 5). We use this fact to show how to connect all of the new variables together, as in Fig. 6.

Notice that we have used extra boxes to allow arcs in A_2 to cross A_1 arcs as necessary (see Fig. 6). This can add no more than $9m^2$ new boxes, and the algorithm is thus clearly polynomial. Q.E.D.

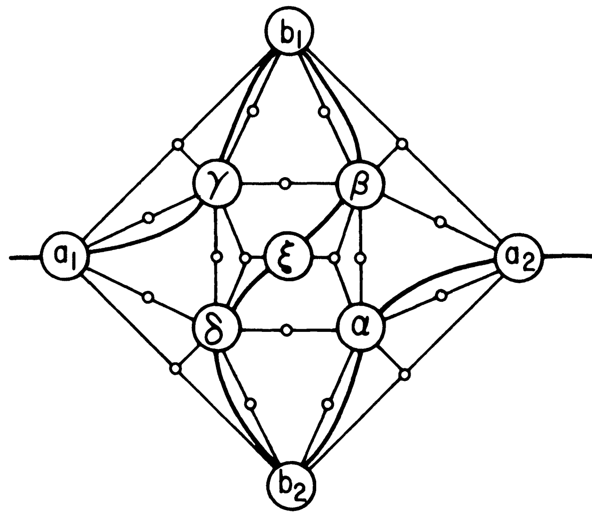


FIG. 5

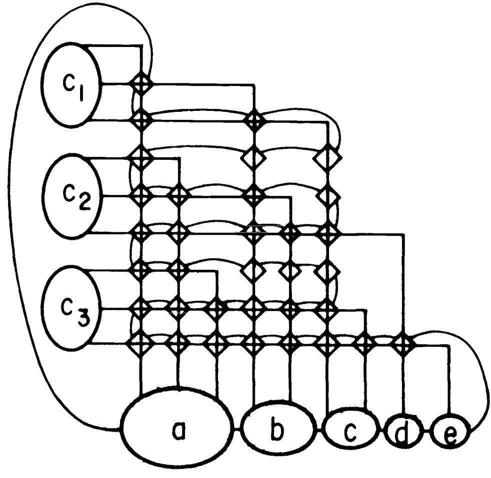


FIG. 6

DEFINITION. Planar 3SAT(P3SAT) is 3SAT restricted to formulae B such that $G(B)$ is planar.

THEOREM 2. P3SAT *is* NP-*complete.*

As remarked above, this is a corollary of Theorem 1.

A word about the arcs in A_2: They are irrelevant for the reduction to node cover, and, for the reductions to Hamiltonian line, geometric connected dominating set and geography, the arc $\{v_n, v_1\}$ will have to be deleted, so as to make the path taken by the A_2 arcs a Hamiltonian line rather than a Hamiltonian circuit.

4. Planar node cover.

DEFINITION. A *node cover* C of a graph G is a subset of the nodes of G with the property that every arc of G is incident to a node in D.

THEOREM 3. *Node cover is* NP-*complete even when restricted to the class of planar graphs.*

Proof. We present Garey, Johnson and Stockmeyer's proof [5] that node cover is NP-complete, and then show how to strengthen it for the planar case.

Given a boolean formula B in 3CNF with m clauses in n variables, form the following node cover problem NC(B), which will have a node cover of size $5m$ if and only if B is satisfiable.

Each clause is represented by a triangle, and each variable is represented by a simple cycle of length $2m_i$. Even numbered nodes in the cycle represent negated instances of the variable, and odd numbered nodes represent unnegated instances.

Arcs go between triangles and cycles whenever the variable represented by the cycle occurs in the clause represented by the triangle. Each node in a triangle is used only once (see Fig. 7).

At least half the nodes from each cycle must be in any node cover, and this local minimum can be achieved only if every other node is chosen. At least two nodes from each triangle must be in any node cover. The rest of the proof involves showing that if these two local minima are achieved, then B is satisfiable.

Note that the choice of which clause node to attach to which node in the cycle is arbitrary, and that this choice determines a cyclic ordering of clauses around each variable and of variables around each clause.

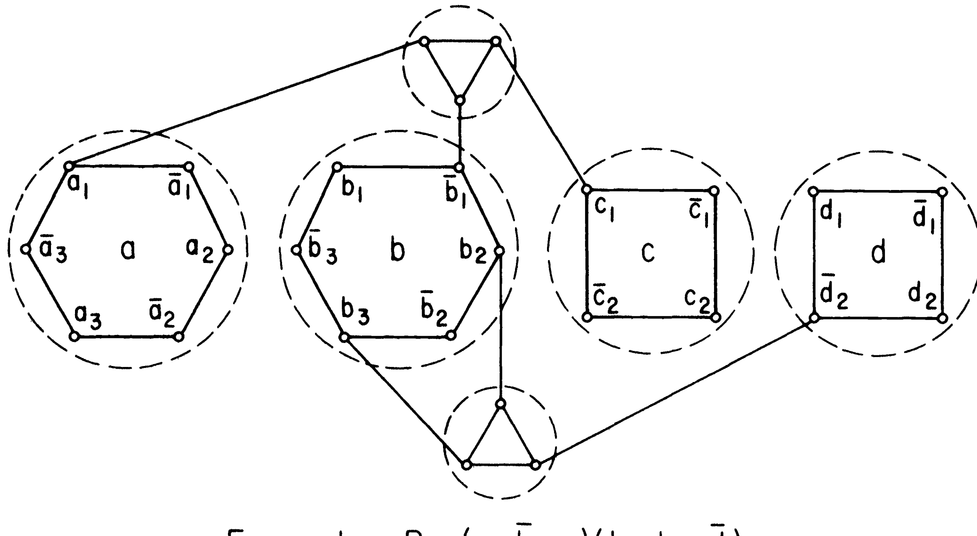


FIG. 7

If each variable structure and each triangle is viewed as a single macro node, as the dotted lines indicate, then the resulting graph is simply $G(B)$. There is a choice of cyclic orderings of clauses around variables and variables around clauses for which NC(B) is planar if and only if $G(B)$ is planar. Since any (polynomial) planarity algorithm can find such an ordering if it exists, Theorem 2 applies and the theorem is proved. Q.E.D.

5. Planar directed Hamiltonian circuits.

THEOREM 4. *Planar directed Hamiltonian circuit is* NP-*complete* [6].

Proof. (This proof is due to Michael Sipser.) We show how to construct H(B), a graph that has a Hamiltonian circuit if and only if B is satisfiable. Variables are represented by ladders, as shown in Fig. 8. Choosing the variable true will mean

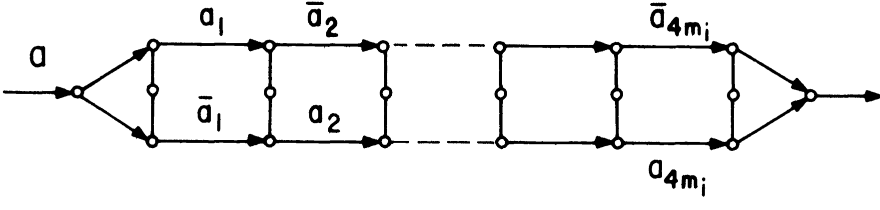


FIG. 8

traversing the nodes in the ladder in a zig-zag starting at the top; choosing the variable false will mean starting at the bottom. The length of the ladder will be the number of (undirected) cross rungs, and the ladder for the variable v_i will be $4m_i$ long. ($4m_i$ is long enough so that we can leave gaps between sections of the ladder linked to two different clauses.)

Clauses are simply single nodes. They are connected to ladders as in the example shown in Fig. 9.

To complete the construction, the ladders are linked together in a global Hamiltonian circuit (drawn in long dashes).

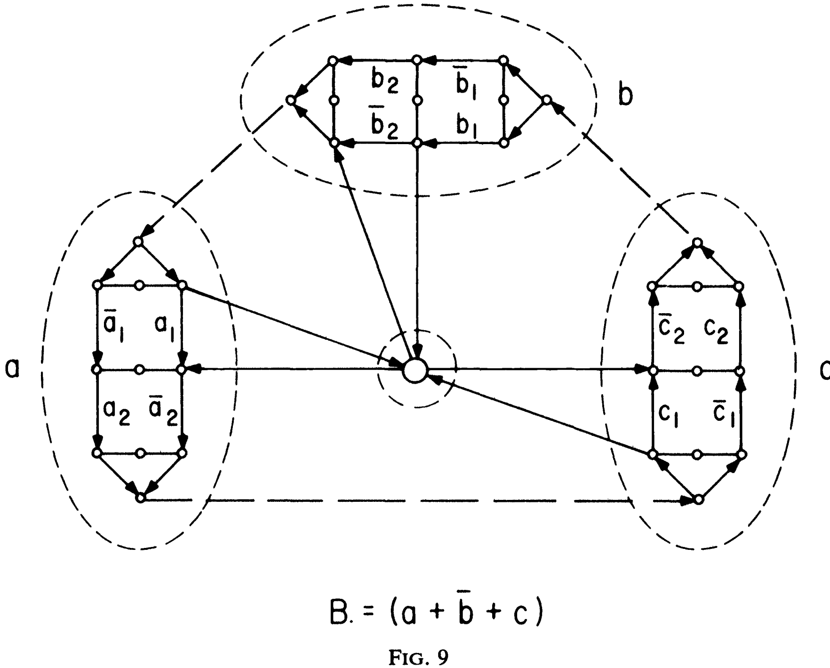


FIG. 9

Claim. If B is satisfiable, the Hamiltonian circuit in H(B) zigs the appropriate way in each ladder, and traverses each clause node by interrupting the path in the ladder to jump up and back down to the ladder, as in Fig. 10. Note that the choice of which variable to use in satisfying the clause is arbitrary for clauses with more than one true literal.

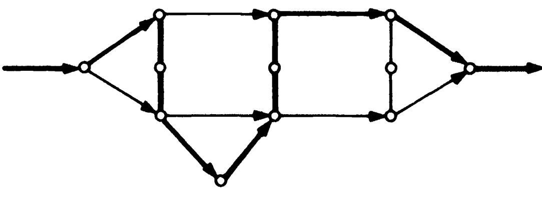


FIG. 10

The converse is nearly as simple. If H(B) has a Hamiltonian circuit, we now show that it cannot leave a ladder in the middle via a clause, but instead must continue down until the end of the ladder.

Suppose then that H(B) has a Hamiltonian circuit which misbehaves, i.e., jumps from one variable to another via a clause, as in Fig. 11.

It should be clear that node u can never be traversed. The converse then follows easily from the fact that each Hamiltonian circuit in H(B) looks right, i.e., that it zigzags correctly through variables and returns immediately to the ladder it came from after traversing a clause node.

We can now invoke Theorem 2 in the same way we did in the previous section, and the theorem is proved. Q.E.D.

COROLLARY. *Planar directed Hamiltonian line is* NP-*complete.*

Proof. Just delete one are from the global circuit, e.g., the one representing $\{v_n, v_1\}$ from A_2. □

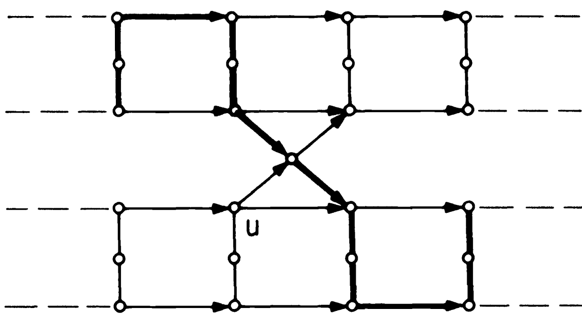


FIG. 11

COROLLARY. *A more involved case analysis shows that the directions on the arcs in the construction are unnecessary, and this gives us the* NP*-completeness of planar undirected Hamiltonian circuit and line. These were first proved in* [6].

The next two proofs are less straightforward than the previous two in that we can not simply demonstrate a reduction from 3SAT and then invoke Theorem 2. This leads us to a choice of where to do the extra tinkering necessary. One can either invent more complicated reductions and use more involved proofs of the correctness of the reduction, or try to massage boolean formulae into forms more easily reducible to the problem at hand. The strategy followed in this paper is to do as much of the work as possible with boolean formulae so as to have to prove as little as possible about unfamiliar, uncooperative combinatorial structures.

6. Geometric connected dominating set.

Problem. Given a set of cities in the plane, each of which has a receiver operational with a radius of d, can k transmitters be apportioned so that a message originating at one transmitter can be relayed to every city?

The above problem is a version of the dominating set problem, and was posed by Phil Spira in connection with packet radio network design.

DEFINITION. A *dominating set of nodes* in a graph is a subset of the nodes in the graph with the property that every node not in the set is adjacent to a node that is in the set.

DEFINITION. The *connected dominating set problem* (CD): Given a graph G and an integer k, is there a connected subset of size k that is a dominating set?

DEFINITION. The geometric connected dominating problem (GCD) is CD when the nodes are a set of points in the Euclidean plane, and an arc is drawn between all pairs of points no greater than distance 1 apart.

THEOREM 5. GCD *is* NP*-complete.*

Proof. B, as usual, is a boolean formula in 3CNF with m clauses and n variables. We wish to construct an equivalent GCD problem, GCD(B).

Our method of presentation will be as follows: First, we present the structures we would like to use in the proof. Then, according to the strategy outlined earlier, we formulate a corollary to Theorem 1 which facilitates the reduction, and last we show the entire construction and prove its correctness.

We want to represent each variable by a set of points n the plane of the form shown in Fig. 12. Choosing a variable true corresponds to putting all the nodes in the top row into the connected dominating set (cds); false puts the bottom row in. The square nodes force at least one of the two nodes adjacent to it into any cds. The structure is long

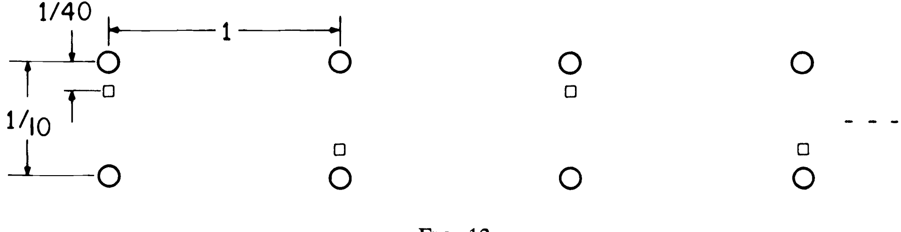


FIG. 12

enough to prevent unwanted interactions between nearby clauses, just as in the Hamiltonian circuit construction.

The variables will all be linked together by a line we call a ground (see Fig. 13).

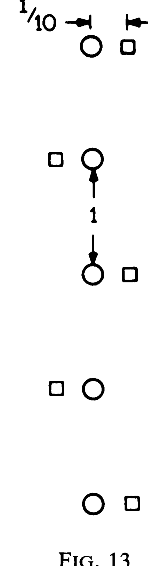


FIG. 13

The ground will follow the path taken by the A_2 arcs from $G(B)$. Figure 14 is a detailed view of the ground passing through a variable. Notice that we have had to move the square forcers outside the variable in the two pairs near the ground, since otherwise the forcers would be near the ground, and would not force at least of the two nearby nodes from the variable into the cds.

Each clause is represented by the kind of structure shown in Fig. 15. If the jth clause is $(a+\bar{b}+c)$, then one circled node will be within 1 of a top node representing a one will be near a bottom node in the structure representing b, and one will be near a top node in the structure representing c.

Notice that the uncircled round nodes are forced into any cds by the square nodes nearby. In general, we will refer to a node which is forced into any cds by a nearby square node as *forced.*

At this point the reader should notice a glaring discrepancy between variable nodes as defined in § 3 and the variable structure we intend to use here to represent them. The latter are bipolar, by which we mean that all clauses containing a positive instance of a variable must be positioned near the top of the variable, and all clauses containing a negative instance of the variable must be positioned near the bottom. We imposed no

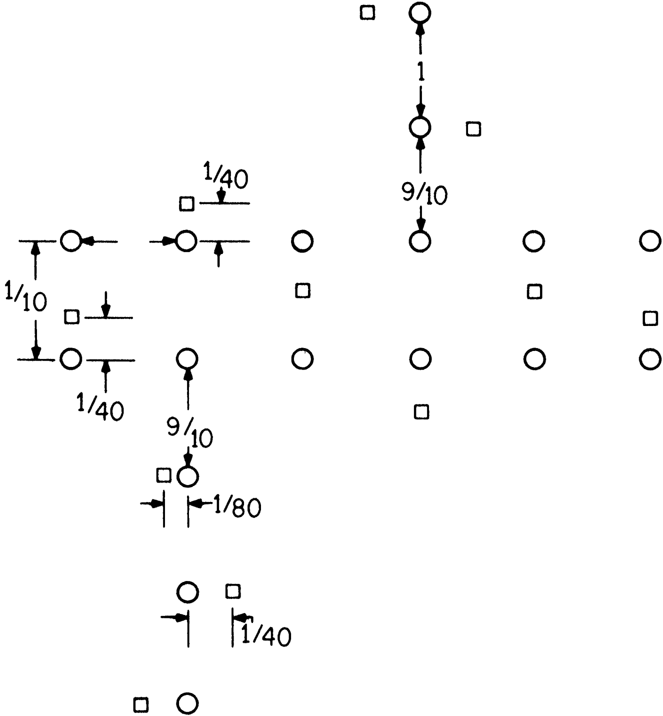


FIG. 14

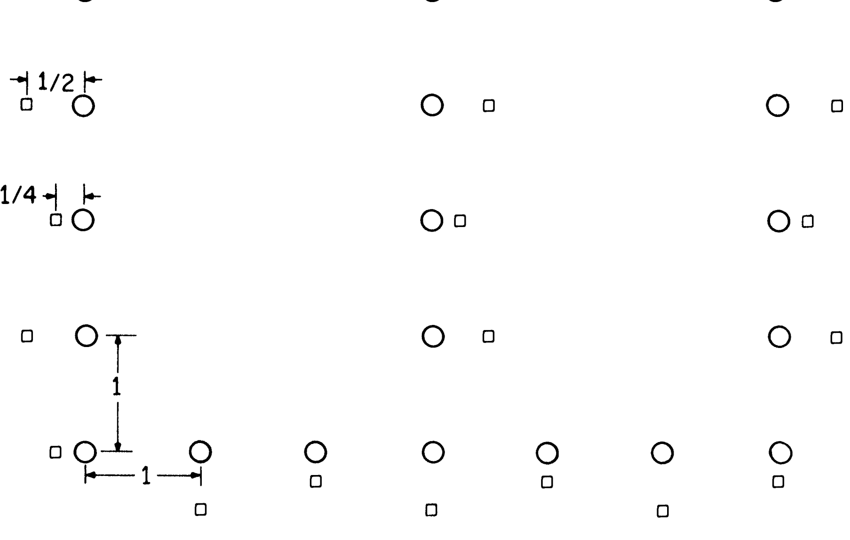


FIG. 15

such restriction in our definition of planar formulae. We do so now, and prove the resulting problem is still NP–complete.

LEMMA 1. *Planar satisfiability is still* NP-*complete even when, at every variable node, all the arcs representing positive instances of the variable are incident to one side of the node and all the arcs representing negative instances are incident to the other side. (Equivalently, we can have separate nodes for positive and negative literals, and add an arc between the (now) two nodes representing a single variable.)*

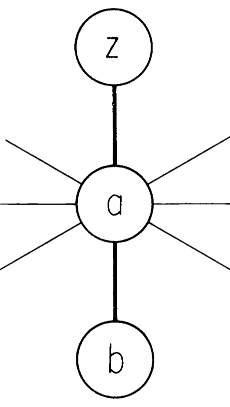


FIG. 16

Proof. Take the planar embedding of the graph of the formula (see Fig. 16), and replace each variable a with a cycle of m_a variables a_i, together with clauses $(\bar{a}_j + a_k)$ for variables a_j and a_k such that ka_k follows a_j in a clockwise traversal of the cycle. (Notice in Fig. 17 that the ordering of variables in the cycle is different from the ordering followed by the A_2 arcs.) These clauses have the effect of forcing $a_j \Leftrightarrow a_k$ for all a_j and a_k in the cycle. A_2 arcs are embedded as in Fig. 17.

Now, back to the problem at hand. Let:

$\mathrm{NV} = \frac{1}{3}$ the number of nodes in all the variable structures;

$\mathrm{NC} =$ the number of forced nodes in all the clause structures;

$\mathrm{NG} =$ the number of forced nodes in the ground.

Let $k = \mathrm{NV} + \mathrm{NC} + \mathrm{NG} + m$.

Claim. GCD(B) has a connected dominating set of size k if and only if B is satisfiable.

$\Leftarrow$ Choose top and bottom rows in variables according to whether the variable is true or false in a given satisfying instance of B. Pick one circled node in each clause that lies within 1 of a variable already chosen. Pick all the forced nodes in each clause and in the ground.

$\Rightarrow$ Let GCD(B) have a connected dominating set of size k. We show that this set must look right. Call a node live if it is in the cds. Suppose some variable switches from true to false at least once. Then suppose we want to find a path from a live node in the left half of the variable to the right half. Since the ground follows the route taken by A_2 arcs, and is therefore a Hamiltonian line through the variables, the path we are looking for must go through at least one clause, c_i. This means some clause has two live circled nodes, since otherwise clauses are culs de sac. Since our threshold, k is a sum of local

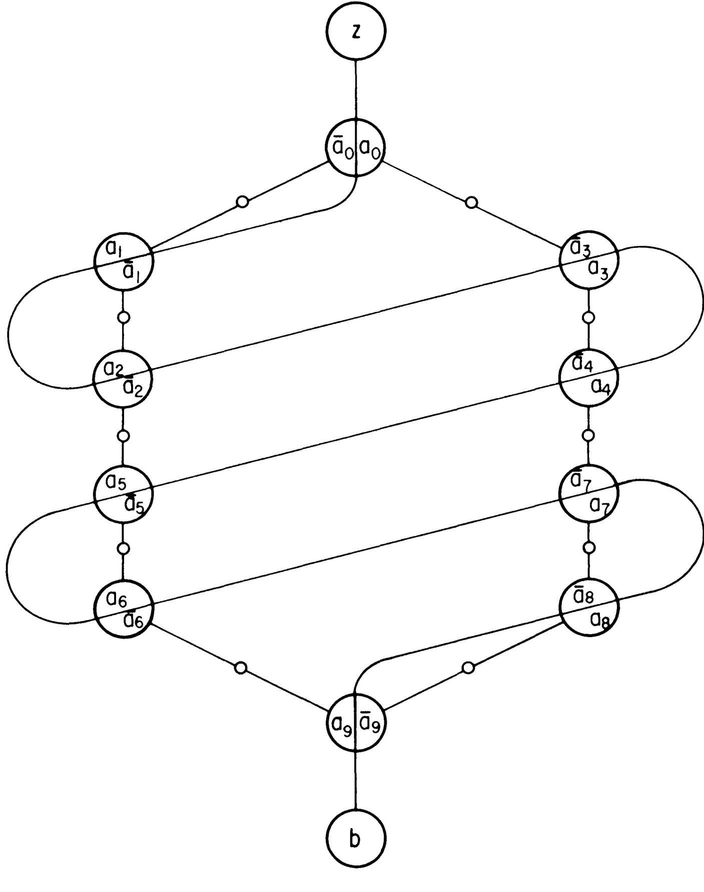


FIG. 17

minima, there is no slack anywhere to make up for the extra live node in c_i. So every variable has either the entire top row live, or the entire bottom row live.

The rest of the proof involves showing that the entire graph can be embedded in the plane in such a way that nodes are at rational points whose precision is bounded by a polynomial in the size of the number of points. This demonstration is straightforward, and we omit it. QED.

7. Generalized geography.

DEFINITION. Generalized geography (GG) is a game played by two players on the nodes of a directed graph. Play begins when the first player puts a marker on a distinguished node. In subsequent turns, players alternately place a marker on any unmarked node q, such that there is a directed arc from the last node played to q. The first player who cannot move loses.

This is a generalization of a commonly played game in which players must name a place not yet mentioned in the game, and whose first letter is the same as the last letter of the last place named. The first player to be stumped loses. This instance of geography would be modelled by a graph with as many nodes as there are places. Directed arcs would go from a node, u, to all those nodes whose first letters are the same as u's last letter.

THEOREM 6. GG *is* P-*space complete* [11].

Proof. We are given a formula $B \in$ Q3CNF, $B = Q_1 v_1, \ Q_2 v_2, \cdots, Q_n v_n F(v_1, v_2, \cdots, v_n)$. Assume without loss of generality, that $Q_1 = \forall$, $Q_n = \exists$, and that $Q_i \neq Q_{i+1}$ for $1 \leqq i \leqq n$. Construct the graph GG(B), which is shown in Fig. 18.

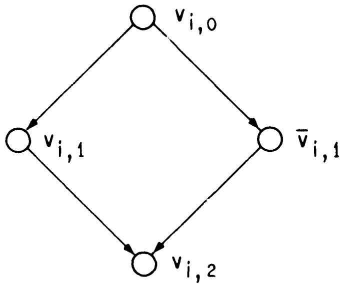


FIG. 18

Each variable, v_i, is represented by a diamond structure, and each clause, c_j, is represented by a single node. In addition, we have arcs $(v_{i,2}, v_{i+1,0})$ for $1 \leqq i < n$, $(v_{n,2}, c_j)$ for $1 \leqq j \leqq m$, and paths of length two going from c_j to $v_{i,1}$ for v_i in c_j, and from c_j to $\bar{v}_{i,1}$ for $\bar{v}_i$ in c_j. $v_{1,0}$ is the distinguished node (see Fig. 19).

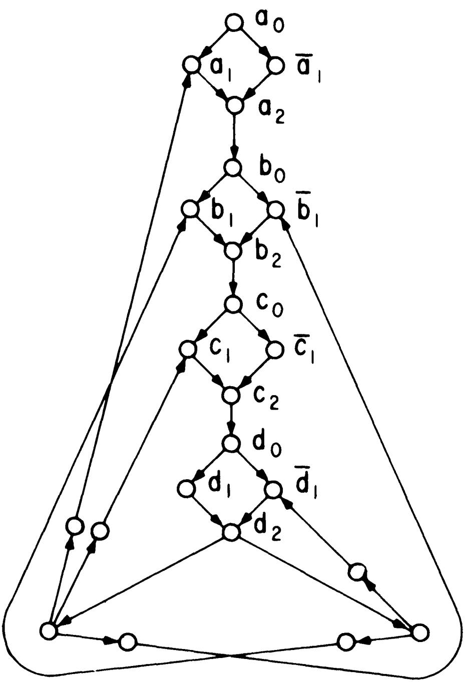


FIG. 19

Play proceeds as follows: One player chooses which path to take through ∀-diamonds (i.e., diamonds representing universally quantified variables), and the other player chooses which path to take through ∃-diamonds. After all diamonds have been traversed, the ∀-player chooses a clause, and the ∃-player then chooses a variable from that clause. ∃ then wins immediately if the chosen variable satisfies the clause; otherwise, ∀ wins on the next move. Assuming both players play optimally, it follows easily that ∃ wins if and only if B is true (we leave the details to the reader).

Planar generalized geography.

THEOREM 6. *Generalized geography is* P-*space complete even when played only on planar graphs.*

Proof. There is a problem which prevents us from merely invoking Lemma 1 to give us the proof, namely, the set of arcs, $\{(v_n, c_j)|1 \leqq j \leqq m\}$.

To solve this problem, we make the following observation: There is no need to wait until all variables have had their truth values chosen before allowing the ∀-player to test the truth of a clause; in fact, each clause can be tested as soon as its last variable has had its value fixed. Moreover, it is only necessary to allow testing of clauses not satisfied by their last chosen variable.

In order to implement this idea, we need variable structures which are large enough so that every clause has a different node of attachment to the structure. Moreover, this node must be one at which ∀ has the choice of direction. Since each variable occurs in at most three clauses (by Lemma 1), the structure in Fig. 20 suffices.

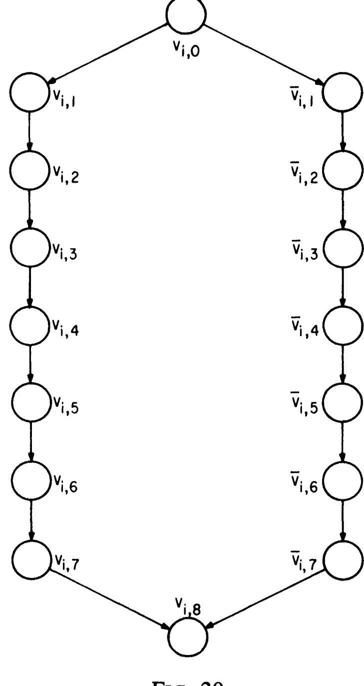


FIG. 20

The clause construction is now as in the following example. Let $c_i = (a + \bar{b} + d)$, where d is the variable with the highest index of the three (i.e., is quantified last). The corresponding arcs in GC(B) are a path of length two going from c_i to an unused node from the set $\{a_1, \cdots, a_7\}$, a path of length two going from c_i to an unused node from the

set $\{b_1, \cdots, b_7\}$, and (d_2, c_i), (d_4, c_i) or (d_6, c_i) if d is a $\forall$-variable, else (d_1, c_i), (d_3, c_i) or $\forall\ (d_5, c_i)$. Notice that if d is chosen true, there is no way for the $\forall$ player to test c_i. In fact, it would not be in $\forall$'s interest to do so.

At this point, we invoke Lemma 1 and the theorem is proved. Q.E.D.

8. Conclusion. We have seen how one planar completeness result easily produces others through the use of transformations under which planarity is invariant. We suspect that it is possible to obtain easy NP-completeness proofs for the planar version of Steiner tree, triangulation existence and minimum weight triangulation. We suggest that it may be profitable to use other artificial sets (e.g., planar exact 3-cover, appropriately defined) to obtain other sets of uniform and easy proofs.

Planar generalized geography has been used to prove P-space completeness of appropriately generalized versions of chess, checkers, go, and hex [13], [3], [8], [10]. A simpler proof of the P-space completeness of generated geography was presented in [7], but the proof in this paper is the original one, and we have included it here more as a justification for planar formulae than for its own sake.

Acknowledgments. I wish to thank Shimon Even, Faith Fich, Michael Garey, Michael Gursky, David Johnson, Richard Karp, Eugene Lawler and Michael Sipser for their help with this paper. Thanks also go to the Weizmann Institute of Science and to the Hebrew University in Jerusalem for allowing me the use of their facilities while writing the paper.

REFERENCES

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

[2] S. A. COOK, *The complexity of theorem-proving procédures*, Proc. Third ACM Symposium on Theory of Computing, 1971, pp. 151–158.

[3] A. S. FRAENKEL, M. R. GAREY, D. S. JOHNSON, T. SCHAEFER AND Y. YESHA, *The complexity of checkers on an $N \times N$ board*, Proc. 19th IEEE Symposium on Foundations of Computer Science, 1978, pp. 55–64.

[4] M. R. GAREY AND D. S. JOHNSON, *The rectilinear Steiner tree is* NP-*complete*, SIAM J. Appl. Math., 32 (1977), pp. 826–834.

[5] M. R. GAREY, D. S. JOHNSON AND L. STOCKMEYER, *Some simplified* NP-*complete problems*, Proc. Sixth ACM Symposium on Theory of Computing, 1974, pp. 47–63.

[6] M. R. GAREY, D. S. JOHNSON AND R. E. TARJAN, *The planar Hamiltonian circuit problem is* NP-*complete*, this Journal 5 (1976), pp. 704–714.

[7] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.

[8] D. LICHTENSTEIN AND M. SIPSER, GO *is polynomial space hard*, J. Assoc. Comput. Mach., 27 (1980), pp. 393–401.

[9] A. R. MEYER AND L. J. STOCKMEYER, *Word problems requiring exponential time*, Proc. 5th ACM Symposium on Theory of Computing, 1973, pp. 1–9.

[10] S. REISCH, *Hex ist* PSPACE-*vollstandig*, private communication.

[11] T. J. SCHAEFER, *On the complexity of some two-person perfect information games*, J. Comput. Systems Sci, 16 (1978), pp. 185–225.

[12] L. STOCKMEYER, *Planar* 3-*colorability is polynomial complete*, SIGACT News 5 (1973), pp. 19–25.

[13] J. A. STORER, *A note on the complexity of chess*, Proc. 1979 Conference on Information Sciences and Systems, Dept. Electrical Engineering, The Johns Hopkins University, Baltimore, Maryland, 1979, pp. 160–166.

SIAM J. COMPUT.
Vol. 11, No. 2, May 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1102-0011 $01.00/0

ON THE ACCEPTING DENSITY HIERARCHY IN NP*

SHLOMO MORAN†

Abstract. Let Al be a polynomial time nondeterministic algorithm accepting a set A, and let $a \in A$. The "accepting density" of Al for a is the ratio between the number of accepting computations and the total number of computations of Al on input a. (If this ratio is $\geqq 1/2$ for all $a \in A$, then Al is a polynomial time probabilistic algorithm accepting A.)

In this paper a characterization of sets in NP according to their "accepting density" is investigated. It is shown that for some relativized form of NP no general, nontrivial lower bound on the accepting density of sets in NP exists. It follows that for this relativized form the accepting density of any NP complete set for inputs of length n cannot be greater than $1/2^{n^c}$ for some fixed $c>0$ and that NP (under that relativization) can be partitioned to infinitely many classes $C_1, C_2, \cdots$, such that the accepting density of sets in C_i is strictly greater, in some precise sense, than that of sets in C_{i+1}. Recent works on the relationship between relativized and unrelativized proof techniques imply that to prove that any of the above results does not hold for the (unrelativized) class NP, possible at all, is probably beyond the ability of today's techniques.

Key words. nondeterministic algorithms, P, NP, probabilistic algorithms, algorithms with oracles

1. Introduction. Let NP be the class of sets which are recognizable by a polynomial time nondeterministic algorithm. It was observed recently that there are some sets in NP which, though they do not have (yet) polynomial time determinstic algorithms which solve them, they can actually be solved in polynomial time by *probabilistic* algorithms [8], [10], [2]. This is due to the fact that there are polynomial time nondeterministic algorithms solving those problems which have the following property: If some input is accepted by such an algorithm, then at least 1/2 of the computations of that algorithm on this input are accepting computations. This fact prompted some researchers to characterize sets in NP according to the ratio between the number of accepting computations and the number of all possible computations for strings in those sets. (See, e.g., [1].) We shall refer to that ratio above as the "accepting density". (This notation was suggested by Adleman in [1].)

We say that a set A in NP has an accepting density $1/u(n)$ for some function $u(n)$ if there is a polynomial time nondeterministic algorithm Al accepting A such that, for each string a of length n, if $a \in A$, then Al accepts a with probability $\geqq 1/u(n)$. Clearly, every set in NP has an accepting density $\geqq 1/2^{n^k}$ for some k.

Let G be a set, and let NP^G denote the class of sets accepted by nondeterministic polynomial time algorithms using oracle G (see [3], [5]). We show that for some G there is no k such that *all* sets in NP^G have an accepting density $\geqq 1/2^{n^k}$. This implies that the accepting density of *any* NP^G complete set is not larger than $1/2^{n^c}$ for some $c>0$ and that NP^G can be partitioned to infinitely many nonempty classes $C_1, C_2, \cdots$, such that the accepting density of sets in C_i is strictly greater (in some precise sense) than that of sets in C_{i+1}. By arguments concerning the relationship between relativized and unrelativized proof techniques [3], [5], [9], it is strongly suggested that unless a new and fundamental technique is found no result which contradicts the above results can be proved for the (unrelativized) class NP.

2. Preliminaries. In the sequel we shall assume that all sets mentioned are subsets of $\Sigma^* = \{0, 1\}^*$. For $x \in \Sigma^*$, $l(x)$ denotes the length of x. Binary predicates are identified

---

* Received by the editors March 21, 1980, and in final revised form December 1, 1980. This research was supported in part by the National Science Foundation under grant MCS 78-01736.

† Computer Science Department, Technion, Haifa 32000, Israel.

with subsets of $\Sigma^* \times \Sigma^*$. For a binary predicate P, "$P(x, y)$ is true," or simply "$P(x, y)$", means $(x, y) \in P$.

DEFINITION 1. A set A is in NP if for some polynomial time computable predicate $P(x, y)$ and some integer k

$$A = \{x \mid \exists y, l(y) = l(x)^k, P(x, y)\}. \tag{1.1}$$

(This is equivalent to: if there exists a nondeterministic polynomial time algorithm accepting A—see, e.g., [6].)

The set above is in R if it also holds that:

$$A = \{x \mid \exists \tfrac{1}{2}(2^{l(x)^k}) y's, l(y) = l(x)^k, P(x, y)\}. \tag{1.2}$$

(I.e.: if there exists a nondeterministic polynomial time algorithm f which recognizes A such that for each $a \in A$ at least $\frac{1}{2}$ of the $2^{l(a)^k}$ possible computations of f on input a are accepting computations, or: for each $a \in A$, f accepts a with probability $\geqq 1/2$. We assume that each of the computation paths of f on input x has the same probability to be executed. For a more elaborate exposition of the set R, and of probabilistic computations in general, see [4], [2].).

It can be shown that if we change the constant $\frac{1}{2}$ in (1.2) to any other positive constant smaller than 1, then still the same class R is defined. (If the constant is 1, then the class defined is P.) Moreover—as was noted already in [2], we can change (1.2) to

$$A = \left\{x \,\middle|\, \exists \frac{2^{l(x)^k}}{l(x)^i} y's, l(y) = l(x)^k, P(x, y)\right\}, \tag{1.2'}$$

where i is any positive constant, and still the same class R is defined.

Equation (1.2') above can be generalized in the following way to define other classes of sets in NP according to the density of the accepting computations for the strings in those sets.

DEFINITION 2. Let A be a set in NP, and let u be a real-valued function defined on the integers. We shall say that A belongs to the density class of u (denoted by "$A \in D(u)$"), if there exists a polynomial time computable predicate $P(x, y)$ and some constant k such that

$$A = \{x \mid \exists y, l(y) = l(x)^k, P(x, y)\} = \left\{x \,\middle|\, \exists \frac{2^{l(x)^k}}{u(l(x))} y's, l(y) = l(x)^k, P(x, y)\right\} \tag{2.1}$$

(or, equivalently, if there exists a nondeterministic polynomial time algorithm f accepting A such that for each $a \in A$ at least $1/u(l(a))$ of the (equiprobable) computations of f on a are accepting computations, or for each $a \in A$, f accepts a with probability $\geqq 1/u(l(a))$).[1]

Examples. $A \in P \leftrightarrow A \in D(1)$, where 1 denotes the unit function. $A \in R \leftrightarrow A \in D(q)$, where q is some polynomial satisfying $q(n) > 1$ for almost all $n \in Z^+$.

Let Al be a polynomial time probabilistic algorithm. By repeating the execution of Al several times, one can increase its accepting density, as described below.

Let A be a set satisfying (2.1), and let $g(n)$ be a positve integer-valued function. We define a nondeterministic algorithm $Al(P, g)$ (P is the predicate in (2.1)), which recognizes A, as follows:

$Al(P, g)$: input: $x \in \Sigma^*$, $l(x) = n$.
Begin
For $i = 1$ to $g(n)$ do

[1] If $u(n) < 1$ for some n's, then, by convention, $D(u(n)) = D[\max(u(n), 1)]$.

begin
(1) generate (in a nondeterminstic way) a string y of length n^k.
(2) if $P(x, y)$ then halt and accept.
end
reject
end.

Assuming that each of the 2^{n^k} strings of length n^k has the same probability to be generated at line (1) above, it is not hard to show the following:

(a) For each $x \in \Sigma^*$, if $x \notin A$, then x will be rejected by $Al(P, g)$.

(b) If $x \in A$, then the probability that x will be rejected is bounded from above by $(1-1/u(n))^{g(n)}$. ($u(n)$ is as defined in (2.1).)

(c) If the time complexity of line (2) (checking whether $P(x, y)$ holds for y such that $l(y)=n^k$) is $O(n^t)$, then the time complexity of the algorithm as a whole is $O(g(n)(n^t+n^k))$. In particular, if $g(n)$ is a polynomial, then $Al(P, g)$ is a polynomial time algorithm.

Note that the technique of repetition implies that if $A \in D(u)$, then there is a probabilistic algorithm accepting A with probability $\geqq 1/2$ in $O(n^k u(n))$ time for some k.

Another corollary of the above technique of repetition is the equivalence of the different definitions of the class R, which is formally stated below.

PROPOSITION 1. *Let A be a set in* NP. *Then if $A \in D(n^i)$ for some $i>0$, then for each $\varepsilon>0$, $A \in D(1/(1-\varepsilon))$.*

The next proposition is an analogue of Proposition 1 for sets in NP $-$ R.

PROPOSITION 2. Let $A \in D(u(n))$, *where $u(n)$ satisfies the following: For each i, $\lim_{n\to\infty}(n^i/u(n))=0$. Then, for each i, $A \in D(u(n)/n^i)$.*

Proof. By definition there exists a polynomial time computable predicate $P(x, y)$ such that for some positive k:

$$A=\{x \mid \exists y, l(y)=l(x)^k, P(x, y)\}=\left\{x \,\middle|\, \exists \frac{2^{l(x)^k}}{u(l(x))} y\text{'s}, l(y)=l(x)^k, P(x, y)\right\}.$$

Let i be given. By the discussion preceding Proposition 1, $Al(P, n^i)$ is a polynomial time algorithm which for each $a \in A$ accepts a with probability $\geqq 1-(1-1/u(l(a)))^{l(a)^i}$. Hence, in order to prove the proposition, it suffices to show that

$$(*) \qquad 1-\left(1-\frac{1}{u(n)}\right)^{n^i}=O\left(\frac{n^i}{u(n)}\right).$$

This will follow from the equality

$$(**) \qquad \lim_{n\to\infty}\frac{1-(1-1/u(n))^{n^i}}{n^i u(n)}=1.$$

Equation (**) follows from the fact that $\lim_{n\to\infty}(n^i/u(n))=0$, by the following equalities:

$$\lim_{n\to\infty}\frac{1-\left(1-\frac{1}{u(n)}\right)^{n^i}}{n^i/u(n)}=\lim_{n\to\infty}\frac{1-\left[\left(1-\frac{1}{u(n)}\right)^{u(n)}\right]^{n^i/u(n)}}{(n^i/u(n))}=\lim_{n\to\infty}\frac{1-\exp(-n^i/u(n))}{(n^i/u(n))}$$

$$=\lim_{n\to\infty}\frac{1-[1-(n^i/u(n))+1/2(n^i/u(n))^2-\cdots]}{(n^i/u(n))}=1. \qquad \square$$

3. The accepting density under relativization. By Proposition 2 above, it can be shown that the power of polynomial time repetition in increasing the accepting density is very limited for sets not in R. For instance, it cannot increase the accepting density from $1/2^{n^k}$ to $1/2^{n^{k-\varepsilon}}$ for arbitrarily small $\varepsilon > 0$. Thus, it is possible a priori that for each k there is a set in NP which is not accepted by any nondeterministic polynomial time algorithm whose accepting density is larger then $1/2^{n^k}$. It is interesting, therefore, to know what can be said about the accepting density of some important sets in NP and, in particular, of NP complete sets. (It can be easily checked that the accepting density of the "naive" nondeterministic algorithms for NP complete sets is at most $1/2^{cn}$ for some constant c). By results of Rackoff [9], which use arguments concerning relativizations (see next), the problem NP $\stackrel{?}{=}$ R probably cannot be solved by techniques known today. We shall show next, by extending the ideas in [9], [7], that no nontrivial lower bound on the accepting density of NP can probably be proved by techniques of today.

Let A be a set, and let F be a class of sets represented by the algorithms which recognize them. We assume that algorithms can be encoded in some definite way (i.e., by encoding of Turing machines which execute them). The class F^A is the class of sets recognized by the same algorithms, which has the additional property of deciding a membership in A in one step. (A is denoted as the "oracle set".) Let F_1 and F_2 be two classes of sets. The problem $F_1^A \stackrel{?}{=} F_2^A$ is called "a relativization of the $F_1 \stackrel{?}{=} F_2$ problem with respect to A". It is widely believed now that if $F_1^A \neq F_2^A$ for some A, then, unless a new and fundamental proof technique is found, it is impossible to prove that $F_1 = F_2$ (and vice versa—if $F_1^A = F_2^A$, then it is impossible to prove that $F_1 \neq F_2$) [3], [5], [9].

Let $B \in$ NP. Then, by an elementary counting argument, $B \in D(2^{n^k})$ for some k. We shall show next that there is a (recursive) set G, such that for each k, $\mathrm{NP}^G \not\subseteq D(2^{n^k})^G$. This implies, by the discussion above, that probably no nontrivial lower bound on the accepting density of the sets in NP can be found by techniques of today.

DEFINITION 3. Let $t(n)$ be a function. Then $T(t(n))^G$ denotes the class of sets accepted in $t(n)$ time by a deterministic algorithm using oracle G.

THEOREM 1. *There exists a set G such that the following conditions hold for each positive integer* q:

(a) NP^G *is not included in* $T(2^{n^q})^G$.

(b) $D(2^{n^q})^G$ *is included in* $T(2^{n^{q+1}})^G$.

In particular, NP^G *is not included in* $D(2^{n^q})^G$.

Proof. The oracle set G is composed of some PSPACE complete set S, to which some additional strings are added. S does not contain strings of even length, while the added strings are all of even length.

Let $(M_1, M_2, \cdots)$ be an enumeration of the deterministic Turing machines, and let $((i_1, q_1), (i_2, q_2), \cdots)$ be an enumeration of $Z^+ \times Z^+$. For each pair (i, q), $M_{i,q}$ denotes a Turing machine which on input of length n simulates M_i up to (at most) 2^{n^q} steps and then halts. The set G is constructed by adding at stage m at most one string of length $2e(m)^{q_m}$ to G. Initially, $m = 1$, $e(1) = 1$. Let $G(m)$ denote the contents of G just before the execution of stage m. ($G(1) = S$.) The procedure is as follows:

Step 1. Run $M_{i_m q_m}^{G(m)}$ on $0^{e(m)}$. If this accepts, then do nothing, else add one string of length $2e(m)^{q_m}$ which has not been queried about (during this procedure) to G.

Step 2. Set $e(m+1) \leftarrow 2^{2^{2e(m)^{q_m}}}$, $m \leftarrow m+1$, and go to step 1.

It is easy to check that step 1 of the procedure can always be carried out (i.e., there always exist a string of length $2e(m)^{q_m}$ which has not yet been queried about) and that if z_m is inserted in G at stage m and $\hat{z}$ is inserted next, then $l(z_m) \leqq \log\log(l(\hat{z}))$.

To prove part (a) of the theorem, let $\mathrm{ROOT}_q(G)$ be defined by

$$\mathrm{ROOT}_q(G)=\{x \mid \exists y,\ l(y)=2l(x)^q,\ y\in G\}.$$

Clearly, $\mathrm{ROOT}_q(G)\in \mathrm{NP}^G$. We show now that $\mathrm{ROOT}_q(G)\notin T(2^{n^q})^G$.

Suppose that $\mathrm{ROOT}_q(G)$ is accepted in 2^{n^q} steps by M_i^G. Then $M_{i,q}^G$ accepts $\mathrm{ROOT}_q(G)$. On the other hand, by the construction of G, there is some m such that $O^{e(m)}$ is accepted by $M_{i,q}^G$ if and only if no string of length $2e(m)^q$ is in G if and only if $O^{e(m)}\notin \mathrm{ROOT}_q(G)$, a contradiction. This completes the proof of part (a) of the theorem.

It remains to prove that for each q, $D(2^{n^q})^G \subset T(2^{n^{q+1}})^G$. Let $A\in D(2^{n^q})^G$. Then, by Definition 2,

$$A=\{x \mid \exists y,\ l(y)=l(x)^k,\ P(x,y)\}=\{x \mid \exists 2^{l(x)^k-l(x)^q} y's,\ l(y)=l(x)^k,\ P(x,y)\},$$

where P is some polynomial time computable (with oracle G) predicate and k is some constant $\geqq q$. Let M^G be algorithm which recognizes the set $\{(x,y) \mid l(y)=l(x)^k, P(x,y)\}$ in $O(l(x)^j)$ time. We describe below an algorithm which recognizes the set A above in $O(n^t 2^{n^q})$ time for some t, and hence in $O(2^{n^{q+1}})$ time. Let x be given; $l(x)=n$, where $n^j<2^n$.

Define $\tilde{G}=G-S$. By querying about all strings of length $2,4,\cdots,2\lfloor \log n/2\rfloor$, compute the set L of strings of length $\leqq \log n$ in $\tilde{G}$. This requires $O(n)$ queries. Let z be the shortest string in $\tilde{G}$ not in L. Then every string in $\tilde{G}$ not in $L\cup\{z\}$ is of length $\geqq 2^n$. Let $G_1=S\cup L$ and $G_2=G_1\cup\{z\}$. It follows that G_1 can be computed in polynomial time and that $x\in A$ if and only if for some y, $l(y)=l(x)^k$, M^{G_2} accepts (x,y).

Define a string w, $\log n<l(w)\leqq n^j$, to be "critical" for x if there are at least $2^{n^k-n^q}$ strings y such that M^{G_1} on input (x,y) queries about w. In the next step of the algorithm, it is checked whether there is a critical string in $\tilde{G}$. (There is at most one).

CLAIM 1. *There are at most $n^j 2^{n^q}$ critical strings for x.*

Proof. Suppose there are c critical strings. Then there are at least $c\cdot 2^{n^k-n^q}$ queries of M^{G_1} on (x,y) as y range over all strings of length n^k. The total number of such queries is at most $n^j\cdot 2^{n^k}$. Hence, $c\cdot 2^{n^k-n^q}\leqq n^j 2^{n^k}$. The claim follows. □

Given G_1, there is a (nondeterministic) polynomial space algorithm which recognizes the set $\{(w',x) \mid w'$ is the prefix of some critical string for $x\}$. (This can be done by checking for each w such that $l(w)\leqq n^j$ and w' is prefix of w, if w is critical for x, by simulating M^{G_1} on (x,y) for all y such that $l(y)=l(x)^k$). Using the PSPACE completeness of A, the above set can be recognized in time which is polynomial in $l(x)$, using G as an oracle. It follows that to check if there is a critical string, and to construct one if there is any, takes only polynomial time, when using G as an oracle. Thus, repeating this procedure for at most $n^j 2^{n^q}$ different strings, it can be checked if there is a critical string in $\tilde{G}$ in $O(n^t 2^{n^q})$ time, for some t.

Let W be the set of critical strings in $\tilde{G}(|W|\leqq 1)$.

CLAIM 2. *If $x\in A$, then $\exists y$, $l(y)=l(x)^k$, $M^{G_1\cup W}$ accepts (x,y).*

Proof. As mentioned above, $x\in A$ if and only if $\exists y$, $l(y)=l(x)^k$, M^{G_2} accepts (x,y). So if $G_2=G_1\cup W$, the claim is obvious. If $G_2\neq G_1\cup W$, then $W=\emptyset$, and G_2 contains a noncritical string z, where $z\notin G_1$. Since z is noncritical, there are less than $2^{n^k-n^q}$ strings y such that M^{G_1} (and hence M^G) on input (x,y) queries about z. But, since $x\in A$, there are at least $2^{n^k-n^q}$ $y's$ such that M^G accepts (x,y). It follows that there is at least one y such that M^G accepts (x,y) without querying about z. □

Since $\mathrm{PSPACE}^S=\mathrm{P}^S$, it can be decided in polynomial time, using oracle G, whether $\exists y$, $l(y)=l(x)^k$ and $M^{G_1\cup W}$ accepts (x,y). If the answer is "no", then by claim 2, $x\notin A$. If the answer is "yes" and $G_2=G_1\cup W$, then $x\in A$. Otherwise, let z be the (noncritical) string in G_2-G_1: If (x,y) is accepted without querying about z,

then $x \in A$. Otherwise, z can be found, and $x \in A$ if and only if $x \in M^{G_1 \cup \{z\}} = M^{G_2}$, which, again, can be decided in polynomial time. This completes the proof of the theorem. □

COROLLARY 1. *Let A be a* NP^G *complete set* (*in the Karp sense* [6]), *where G is as above. Then for some positive c, no nondeterministic polynomial time algorithm which recognizes A* (*using oracle G*) *has accepting density larger than* $1/2^{n^c}$.

Proof. By Theorem 1, there is a set $B \in \mathrm{NP}^G$ such that $B \notin D(2^n)^G$. By the NP^G completeness of A, B is reducible to A in n^i steps for some i. Let $c = 1/i$. We claim that no polynomial time nondeterministic algorithm accepting A has an accepting density $\geqq 1/2^{n^c}$. Suppose, on the contrary, that algorithm Al accepts A and has accepting density $\geqq 1/2^{n^c}$. Then, by reducing B to A and applying Al, there is an algorithm accepting B with accepting density $\geqq 1/2^{(n^i)^c} = 1/2^n$. A contradiction. □

Another related interesting problem concerns the possibility of the existence of "accepting density" hierarchy in NP: can NP be partitioned to nonempty classes $C_1, C_2, \cdots$, such that sets in C_i has an accepting density which is strictly greater, in some precise sense, than the accepting densities of sets in C_{i+1}? For NP^G, the answer is positive.

COROLLARY 2. *There is an infinite sequence of integers,* $0 = i_0 < i_1 < \cdots$, *such that for each* $k \geqq 0$ *there is a set* A_k *in* NP^G *satisfying*:

1) $A_k \in D(2^{n^{i_k}})$,
2) $A_k \notin D(2^{n^{i_{k-1}}})^G$.

Proof. For $k = 0$, $R^G = D(2^{n^0})^G = D(2)^G$. i_k is defined inductively by

$$i_k = \min \{i \mid \exists A \in \mathrm{NP}^G - D(2^{n^{i_{k-1}}})^G, A \in D(2^{n^i})\}.$$

The existence of i_k follows from Theorem 1 and from the observation that for each $A \in \mathrm{NP}^G$, $A \in D(2^{n^i})^G$ for some i. □

Acknowledgment. Part of this paper appeared in the author's Ph.D. thesis, supervised by Professor Azaria Paz, to whom the author wishes to express his thanks. I would also like to thank the referee for helpful remarks.

REFERENCES

[1] L. ADLEMAN, *Two theorems on random polynomial time*, Proc. 19th IEEE Symposium on Foundations of Computer Science, 1978, pp. 75–83.

[2] L. ADLEMAN AND K. MANDERS, *Reducibility, randomness and intractability*, Proc. 9th ACM Symposium on the Theory of Computing, 1977, pp. 151–163.

[3] T. BAKER, J. GILL AND R. SOLOVAY, *Relativization of the* $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$ *problem*, this Journal, 4 (1975), pp. 431–442.

[4] J. GILL, *Computational complexity of probabilistic Turing machines*, Proc. 6th Symposium on the Theory of Computing, 1974, pp. 91–95.

[5] J. E. HOPCROFT AND J. D. ULLMAN, *Introduction to Automata Theory, Languages, and Computations*, Addison-Wesley, Reading, MA, 1979.

[6] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller & J. W. Thatcher eds., Plenum Press, New York, 1972, pp. 85–104.

[7] S. MORAN, *Some results on relativized, deterministic and nondeterministic time hierarchies*, J. Comput. System Sci., 22 (1981), pp. 1–8.

[8] M. O. RABIN, *Probabilistic algorithms*, in Algorithms and Complexity, J. Traub, ed., Academic Press, New York, 1976, pp. 21–40.

[9] C. RACKOFF, *Relativized questions involving probabilistic algorithms*, Proc. 10th Symposium on the Theory of Computing, 1978, pp. 338–342.

[10] R. SOLOVAY AND R. STRASSEN, *A fast Monte-Carlo test for primality*, this Journal, 6 (1977), pp. 84–85.

SIAM J. COMPUT.
Vol. 11, No. 2, May 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1102-0012 $01.00/0

A SCHEME FOR FAST PARALLEL COMMUNICATION*

L. G. VALIANT†

Abstract. Consider $N = 2^n$ nodes connected by wires to make an n-dimensional binary cube. Suppose that initially the nodes contain one packet each addressed to distinct nodes of the cube. We show that there is a distributed randomized algorithm that can route every packet to its destination without two packets passing down the same wire at any one time, and finishes within time $O(\log N)$ with overwhelming probability for all such routing requests. Each packet carries with it $O(\log N)$ bits of bookkeeping information. No other communication among the nodes takes place.

The algorithm offers the only scheme known for realizing arbitrary permutations in a sparse N node network in $O(\log N)$ time and has evident applications in the design of general purpose parallel computers.

Key words. network routing, Monte Carlo algorithm, randomization, parallel computers

1. Introduction. We propose a solution to a fundamental communication problem. Suppose that N devices connected together by a sparse network of wires wish to communicate amongst themselves simultaneously. Suppose also that the communication pattern is unpredictable and rapidly changing as may be required, for example, when the devices are computers cooperating in executing a parallel algorithm. The problem is to specify a network topology and a routing algorithm that can implement arbitrary such communication requests efficiently.

In particular we consider the paradigmatic communication requirement of a permutation. Each device is a node of the network and has a distinct name x from the set $\{0, 1, \cdots, N-1\}$. Initially node x contains a "packet" labelled by an address $a(x) \in \{0, 1, \cdots, N-1\}$ that is the destination node to which the packet is to be sent. If the N addresses are all distinct then the communication requirement is a permutation.

The contraint of having only a few wires from each node, and hence a sparse graph is dictated by physical limitations. It implies that it will take a long time to gather complete information about the permutation request at any one node, as this would have to be done largely sequentially. This strongly suggests that we need to look for a distributed routing strategy that does not require any node ever having more than fragmentary information about the permutation.

There are several quantitative criteria according to which such parallel communication schemes (PCSs) may be judged. The scheme that we propose provably achieves the following parameters:

1. *Speed.* Every permutation can be implemented in $O(\log N)$ steps. (By a step we mean the time taken to transmit a packet along a wire. Computations carried out locally at a node are not counted here).

2. *Sparsity.* Each node has $O(\log N)$ wires from it.

3. *Simplicity.* The bookkeeping information carried with each packet (e.g., its address) is small ($O(\log N)$ bits). Such information is never transmitted except when accompanying a packet. The local computations needed at each node are easy and efficiently parallelizable.

4. *Flexibility.* (i) Besides complete permutations it can also implement partial ones. (ii) No global synchronization is required.

* Received by the editors October 15, 1980 and in revised form March 24, 1981.

† Computer Science Department, Edinburgh University, Edinburgh, Scotland EH8 9YL.

There are essentially only two previously known constructions on which rival schemes could be based: Batcher's sorting networks [2] and the permutation networks of Benes [3], [10]. The former suffers from the disadvantages that it requires $\Omega(\log^2 N)$ steps and cannot directly do partial permutations. The problem with the latter is that the only fast parallel routing algorithms known require global information and take time $\Omega(\log^2 N)$ even on a parallel random access model of computation [6]. An advantage they do share over our scheme is that of greater simplicity in the local computations, but this appears to be of ever diminishing relevance for currently anticipated technologies.

In our scheme the network topology is simply the n-dimensional binary cube. We therefore consider values of N with $N = 2^n$ for some integer n. Between every pair of adjacent nodes of the cube we draw a pair of oppositely directed edges to represent communication in the two directions. This topology has been suggested frequently before. For a survey and bibliography see Siegel [8]. Previous schemes all require at least $\Omega(\log^2 N)$ steps for some inputs.

The claimed speed of our scheme needs one qualification. The routing algorithm makes random choices in the same sense as the famous primality tests of Strassen and Rabin [7], [9]. It is correct for all inputs (i.e., permutations) and, for some constant C will terminate in $C \cdot \log_2 N$ steps with overwhelming probability. A noteworthy feature of the algorithm is that its runtime distribution is provably *identical* for every input. The algorithm is therefore *testable* in the sense that its general behavior for a fixed N can be determined with great confidence by running it often enough on any one input (even the identity permutation!). Since the analytic techniques we use, or can envisage, yield relatively crude complexity bounds, making comparisons among refinements of the algorithm is probably possible only by experimentation. We emphasize that here experimental results can be given a rigorous interpretation, a circumstance that we have not met before in as strong a sense in the context of algorithms.

2. Outline of the algorithm. In describing the algorithm we identify each packet by its starting node. Denoting the set $\{0, \cdots, N-1\}$ by V, the name of each packet is therefore a number $s \in V$.

The algorithm consists of two phases run consecutively. Phase A sends each packet $s \in V$ to a randomly chosen node $u(s) \in V$. For each s every $u \in V$ has the same probability (i.e., $1/N$) of being chosen, and the choices for the different packets are independent of each other. The second phase then routes each packet s from $u(s)$ to its correct destination $t = a(s)$.

At each instant there is just one copy of each packet, and this is either (a) being *transmitted* along an edge, or (b) wating in a *queue* associated with such an edge, or (c) stored as *loose* at a node.

For simplicity of exposition the algorithm is described in synchronized fashion although this is inessential. In this form the algorithm alternates between being in a transmitting mode and a bookkeeping mode. In the former case the packet at the head of each queue is transmitted along the edge associated with it and stored as loose at the recipient node. In the bookkeeping mode each loose packet is assigned to the queue of one of the outgoing edges according to some random choice, unless it has nowhere further to go in the current phase. (For a description of this algorithm in less synchronized form see [11], [12].)

In Phase A each packet makes for itself a random ordering of the n dimensions. It considers each one in turn and according to the toss of a coin makes, or refrains from making, a *move* in that dimension from its current position. (By making a move

we mean here that we add it to the appropriate queue. Actual transmission may be delayed by the presence of other packets in the queue.) It is immediate that with this procedure for any fixed packet every node has the same probability of being its destination. What needs to be proved is that no packet will have to wait in queues for more than $O(n)$ steps.

Phase B is similar except now each packet considers the set of dimensions in which its current location differs from its final destination, and moves along one randomly chosen such dimension at each step. Correctness is again immediate. What needs to be proved is that under the assumption that the packets are initially at randomly chosen nodes (as guaranteed by Phase A) no packet will wait in queues for more than $O(n)$ steps.

Analysis similar to Lemma 1 in § 4 shows that in each phase the probability that Cn different routes visit any one node is bounded above by $\exp\{-Cn/4\}$. This gives a crude upper bound on the maximum number of packets that may reside at any one node at any one time.

3. The algorithm. The n-dimensional cube will be represented by the set $V = \{0, 1, \cdots, N-1\}$ of $N = 2^n$ vertices. For $i \in \{1, \cdots, n\}$ and $x \in V$, x^i denotes the ith most significant bit in the n-bit binary representation of x. Also $x/\!\!/i$ is the number obtained by changing the ith bit of the binary representation of x to its complement. The $n2^n$ edges of V are therefore the pairs

$$\{(x, x/\!\!/i) | x \in V, i \in \{1, \cdots, n\}\}.$$

For each such edge there is a "Queue(x, i)" that feeds it and resides at node x. At each node x there is a set "Loose$_x$" in addition to the n queues.

In the algorithm the subroutine call "Transmit x" means 'for each i transmit the packet at the head of Queue (x, i) to node $x/\!\!/i$ and add it to the set Loose$_{x/\!\!/i}$'.

"Pick $d \in D$" means 'assigning equal probability to each member of set D choose a random element of D and assign it to variable d'.

Each packet $v \in V$ is associated with a set $T \subseteq \{1, \cdots, n\}$. In Phase A it consists of the set of dimensions along which possible transmissions have not yet been considered. In Phase B it is the set of dimensions along which transmission still has to take place. Each of the phases is said to be *finished* when for every $v \in V$ T_v is empty.

The routing algorithm consists of calling Phase A followed by Phase B, with the constants F, G chosen large enough that both algorithms finish with overwhelming probability. Note that when Phase A is finished all the queues are empty, and Loose$_x$ contains the set of packets randomly assigned to node x. Hence when Phase B is entered all the packets are loose and the queues empty.

The algorithms are described in Algol-like notation. Parallel execution of a block with variable d ranging over set D is denoted by

For $d \in D$ **cobegin** $\cdots$ **coend**.

The reader can verify by inspection that both phases are correct: when the first one finishes the packets are at independently chosen random nodes, after the second one they are all at their final destinations.

We remark that neither correctness nor the subsequent analysis depends on the particular disciplines used for maintaining the queues or the loose sets. The only assumption is that they are sets in which elements can be added or taken away. A second remark is that, in practice, the innermost loop could by implemented by a special purpose chip exploiting parallelism rather than by a sequential computation.

Phase A

```
For s ∈ V cobegin Loose_s := {s}.
                  T_s := {1, · · · , n}.
        coend
For f := 1 step 1 until F do
    For s ∈ V cobegin if Loose_s ≠ ∅ then for
                  each v ∈ Loose_s with T_v ≠ ∅ do
                  begin Pick i ∈ T_v.
                        T_v := T_v − {i}.
                        Pick α ∈ {0, 1}.
                        if α = 1 then
                        begin add v to Queue(s, i).
                              Loose_s := Loose_s − {v}.
                        end.
                  end.
                  Transmit s.
              coend
```

Phase B

```
For x ∈ V cobegin if packet with address x is at node u
                  then T_x := {i | x^i ≠ u^i}.
        coend
For g := 1 step 1 until G do
    For u ∈ V cobegin if Loose_u ≠ ∅ then for
                  each v ∈ Loose_u with T_v ≠ ∅ do
                  begin Loose_u := Loose_u − {v}.
                        Pick i ∈ T_v.
                        T_v := T_v − {i}.
                        Add v to Queue(u, i).
                  end
                  Transmit u.
              coend.
```

4. Analysis of the algorithm. The aim of the analysis is to show that for a sufficiently large constant C the routing algorithm will finish within $2Cn$ steps with overwhelming probability.

THEOREM. *For any constant S there is a C such that for $F = G = Cn$ both phases of the routing algorithm finish with probability greater than* $1-2^{-Sn}$.

For the analysis we need some facts from probability theory. Suppose that we have N independent Bernoulli trials each with probability p. Then the probability $B(m, N, p)$ that at least m of the trials are successful is bounded above by the normal distribution in the following way [1]:

Fact 1. If $m = Np(1+\beta)$ where $\beta \in [0, 1]$ then $B(m, N, p) \leqq e^{-\beta^2 Np/2}$.

We shall be interested in independent trials with varying probabilities (i.e., Poisson trials). If we have N such trials with probabilities $p_1, \cdots, p_N$, such that $\Sigma p_i = Np$ then, as is well known, the variance in the number of successes is maximal when $p_1 = p_2 = \cdots = p_N = p$. The following theorem of Hoeffding [5] is a stronger version of this.

Fact 2. If T is the number of successes in N independent Poisson trials with probabilities $p_1, \cdots, p_N$, then if $\Sigma p_i = Np$ and $m \geqq Np + 1$ is an integer, then

$$\Pr(T \geqq m) \leqq B(m, N, p).$$

For combinatorial formulae we shall use the notation n_r to denote $n!/(n-r)!$, and $\binom{n}{r}$ to denote $n_r/r!$. From elementary considerations it is easy to verify the following:

Fact 3. For all n

$$\sum_{r=1}^{n} \frac{1}{\binom{n}{r}} \leqq \frac{5}{3}.$$

Fact 1 is an estimate of the tail of the binomial distribution near the mean. For our main theorem we need estimates further from the mean and for this we use the following bound.

Fact 4. If $n \geqq Np$ is an integer then

$$B(m, N, p) \leqq \left(\frac{Np}{m}\right)^m \cdot e^{m-Np}.$$

Proof. Chernoff's bound [4] is

$$B(m, N, p) \leqq \left(\frac{Np}{m}\right)^m \left(\frac{N-Np}{N-m}\right)^{N-m}.$$

Putting $x = (N-m)/(m-Np)$ and using $(1+x^{-1})^x < e$ gives the required bound. □

The analysis of the two phases A and B are rather similar and will be given in tandem. The basic notion is that of a *route*, denoted typically by $R = \{e_1, e_2, \cdots, e_h\}$ where each e_i is an edge (x_i, y_i). A route is any path in the cube in which no two edges traverse the same dimension, i.e., if for some k

$$x_i /\!/ k = y_i \quad \text{and} \quad x_j /\!/ k = y_j$$

then $i = j$. Thus routes are minimum distance (acyclic) paths between their end points.

For any fixed route R and node $s \in V$ the event that "in running Phase A at least one edge occurring in the route from s also occurs in R" will be denoted by "$|R \cap s\rightarrow| \geqq 1$". For any fixed route R and node $t \in V$ the event that "in running Phase B, with the packet destined for t initially at a randomly chosen node, at least one of the edges occurring in the route of that packet also occurs in R" will be denoted by "$|R \cap \rightarrow t| \geqq 1$". More generally, "$s\rightarrow$" denotes the route from s, "$\rightarrow t$" the route to t, and "$x \rightarrow y \rightarrow z$" a route from x via y to z. The intersection $Q \cap R$ of two routes Q and R is the set of common edges. "R through x" is the event that the route R goes through x.

The first lemma bounds the number of routes that intersect with any one route.

LEMMA 1. *For all* $R = \{e_1, \cdots, e_h\}$ *and* $C \geqq 1$

(A) $\Pr(|R \cap s\rightarrow| \geqq 1$ *for at least* Cn *values of* $s) \leqq \exp\{-Cn/4\}$.

(B) $\Pr(|R \cap \rightarrow t| \geqq 1$ *for at least* Cn *values of* $t) \leqq \exp\{-Cn/r\}$.

Proof. (A) Let

$$p_s = \Pr(|R \cap s\rightarrow| \geqq 1) \leqq \sum_{i=1}^{h} \Pr(e_i \in s\rightarrow) = \sum_{i=1}^{h} p_{si} \quad \text{say}.$$

Since the $n2^n$ edges in the cube have identical roles one can argue that, by symmetry,

$$\sum_{s \in V} p_{si} = \sum_{s \in V} p_{sj}$$

for any $i, j \in \{1, \cdots, h\}$. Hence

$$\sum_{s \in V} p_s \leqq h \sum_{s \in V} p_{s1}.$$

But $\sum_s p_{s1}$ is simply the fraction $1/(n2^n)$ of the expected total number of edges occurring in the 2^n routes. Since the expected number of edges on each route is $n/2$,

$$\sum_{s \in V} p_s \leqq h \sum_{s \in V} p_{s1} \leqq \frac{h}{2}.$$

We therefore have N independent Poisson trials with respective probabilities $p_0, p_1, \cdots, p_{N-1}$ that have sum $h/2$.
By Fact 2

$$\Pr(\text{at least } Cn \text{ successes}) \leqq B\left(Cn, N, \frac{h}{(2N)}\right) \leqq B\left(Cn, N, \frac{Cn}{2N}\right)$$

since $h \leqq n$ and $C \geqq 1$. Hence applying Fact 1 with $\beta = 1$

$$\Pr(\text{at least } Cn \text{ successes}) \leqq \exp\left(-\frac{Cn}{4}\right).$$

(B) Let $p_t = \Pr(|P \cap \to t| \geqq 1)$. Then by the same argument as used in (A) we get

$$\sum_{t \in V} p_t \leqq \frac{h}{2}$$

and hence the required result. □

We note that the packet routes in Phase B when viewed in reverse are identical to the packet routes in Phase A. The source nodes in Phase A, like the targets in Phase B, represent each node of the graph exactly once. The targets in Phase A, like the source nodes in Phase B, represent random mappings of the N packets to the N nodes. Since our proofs concern only the routes themselves, independent of timing considerations, the proofs for Phase B will be always essentially the same as for Phase A.

If we could assume that two paths never intersect more than once then Lemma 1 would suffice to prove the Theorem. Unfortunately this is not the case. The following rough argument shows that in Phase A with large probability at least one pair of routes will be identical for about $n/\log n$ consecutive edges: consider two routes from neighboring starting nodes. With probability $(2n)^{-1}$ the first packet goes to the starting node of the second at the first step, and with probability $((n-1)_r)^{-1}$ it then follows the second path for r steps (provided both make r steps, which is most likely if $r = n/\log n \ll n/2$). But $(2n_{r+1})^{-1}$ exceeds $2^{-(n-1)}$ if $r \lesssim n/\log n$, and there are 2^{n-1} such pairs of routes to consider. It is therefore likely that at least one such pair follow each other for $n/\log n$ steps.

We therefore need the following lemma to bound the probability of two routes having r edges in common.

LEMMA 2. *Let R and Q be routes of length j from node x to node y. Suppose that R is fixed and Q is randomly chosen from all such routes. Then for $K = \frac{5}{3}$*

$$\Pr(|R \cap Q| \geqq r) \leqq \frac{K^r}{j_r}.$$

Proof. The result is established by induction on r. It is clearly true for $r=0$. Suppose it is true for $r=0$. Suppose it is true for $r-1$. Let $R=e_1e_2\cdots e_j$. Then $\Pr(|R\cap Q|\geqq r)$ is less than

$$\sum_{m=1}^{j-r+1}\Pr(|e_{m+1}\cdots e_j\cap Q|\geqq r-1\,|\,e_m\in Q)\cdot\Pr(e_m\in Q)$$

$$\leqq\sum_{m=1}^{j-r+1}\frac{K^{r-1}}{(j-m)_{r-1}}\cdot\frac{1}{\binom{j}{m-1}(j-m+1)}$$

$$\leqq\frac{K^{r-1}}{j_r}\cdot\sum_{m=1}^{j-r+1}\frac{(m-1)!(j-m-r+1)!}{(j-r)!}\leqq\frac{K^r}{j_r}\quad\text{by Fact 3.}\quad\square$$

Lemma 4 will show that the probability of a random route intersecting any fixed route r times vanishes exponentially with r increasing. As a preliminary we need to examine an effect of distance in the cube. For $y,z\in V$ the Hamming distance $H(y,z)$ is the number of bits in which the binary representations of y and z differ, i.e.,

$$H(y,z)=|\{i\,|\,y^i\neq z^i\}|.$$

LEMMA 3. *For any $s,t,x\in V$ with $H(s,x)=H(x,t)=k$,*
(A) *in Phase* A

$$\Pr(s\to\textit{through }x)\leqq 1\Big/\binom{n}{k},$$

(B) *in Phase B*

$$\Pr(\to t\textit{ through }x)\leqq 1\Big/\binom{n}{k}.$$

Proof. (A) The probability that the route from s has length at least k is clearly $B(k,n,\frac{1}{2})$. Now there are $\binom{n}{k}$ nodes at Hamming distance k from s. Assuming that the route from s does have length at least k the probability of it passing through any one of these nodes must be, by symmetry, the same as for any other, namely $1/\binom{n}{k}$. Hence

$$\Pr(s\to\text{through }x)=\Pr(s\to\text{through }x\,|\,|s\to|\geqq k)\cdot\Pr(|s\to|\geqq k)\leqq\frac{1}{\binom{n}{k}}\cdot 1.$$

(B) By a similar argument to the one above:

$$\Pr(\to t\text{ through }x)=\Pr(\to t\text{ through }x\,|\,|t\to|\geqq k)\cdot\Pr(|\to t|\geqq k)\leqq\frac{1}{\binom{n}{k}}\cdot 1.\qquad\square$$

LEMMA 4. *For any fixed route R the expected number of packet routes in Phase* A *(and similarly Phase* B*) which have at least r edges in common with R is*

$$A_r=\sum_{s\in V}\Pr(|s\to\cap R|\geqq r)\leqq\min\left\{\frac{n^4 2^r}{n_r},n\right\}.$$

Proof. We give the proof for Phase A. The analysis for Phase B is identical when the packet movements are played in reverse.

The bound on n is a restatement of the fact observed in Lemma 1 that

$$\sum_{s\in V}p_s<n.$$

Now consider the other bound. Suppose that $R = e_1 e_2 \cdots e_h$ and that it visits $y_0, y_1, \cdots, y_h$ in turn. Let V_{ikg} be the set of nodes such that $H(s, y_i) = k$, and y_{i-g} is the first node of R for which route $s \to y_{i-g} \to y_i$ is possible (i.e., $s \to y_{i-g-1} \to y_i$ is impossible). This is illustrated in Fig. 1.

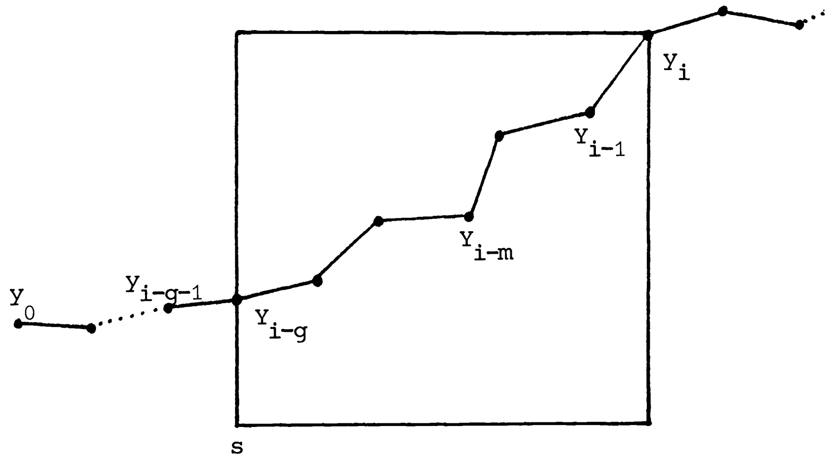


FIG. 1.

Now

$$|V_{ikg}| \leqq \binom{n-g}{k-g} \tag{1}$$

Let $p_{si} = \Pr(|s \to \cap R| \geqq r \,|\, s \to \text{through } y_i)$. Consider now $s \in V_{ikg}$ for some fixed i, k and g, and let y be the first node of R that $s \to$ visits. In this case

$$p_{si} = \sum_{m=r}^{g} \Pr(y = y_{i-m} \,|\, s \to \text{through } y_i)$$

$$\cdot \Pr(|s \to \cap e_{i-m+1} \cdots e_i| \geqq r \,|\, y = y_{i-m} \text{ and } s \to \text{through } y_i)$$

$$\leqq \sum_{m=r}^{g} \frac{1}{\binom{k}{k-m}} \cdot \frac{K^r}{m_r} \tag{2}$$

$$= \frac{K^r}{k!} \cdot \sum_{m=r}^{g} (k-m)!(m-r)! \leqq \frac{K^r}{k!} \cdot (g-r+1) \cdot (k-r)! \quad \text{since } r \leqq m \leqq g \leqq k.$$

By definition

$$A_r = \sum_{s \in V} \Pr(|s \to \cap R| \geqq v) \leqq \sum_{i=1}^{h} \sum_{s \in V} p_{si} \cdot \Pr(s \to \text{through } y_i).$$

Since for each i the sets V_{ikg} partition V

$$A_r \leqq \sum_{i=0}^{h} \sum_{k=r}^{n} \sum_{g=r}^{i} \sum_{s \in V_{ikg}} p_{si} \Pr(s \to \text{through } y_i).$$

Using (2) and Lemma 3, we can bound the summand by

$$\frac{K^r}{k!} \cdot (g-r+1) \cdot (k-r)! \cdot \frac{1}{\binom{n}{k}}.$$

By virtue of (1) the sum of the above over V_{ikg} is

$$\frac{K^r}{k!}\cdot\frac{(g-r+1)(k-r)!\cdot k!(n-k)!}{n!}\cdot\frac{(n-g)!}{(k-g)!(n-k)!}$$
$$=\frac{(g-r+1)}{n_r}\cdot\frac{(k-r)_{g-r}}{(n-r)_{g-r}}\cdot K^r\leqq\frac{(g-r)K^r}{n_r}.$$

Hence

$$A_r\leqq\frac{K^r}{n_r}\cdot\sum_{i=0}^{h}\sum_{k=r}^{n}\sum_{g=r}^{i}(g-r+1)\leqq\frac{nh^3K^r}{n_r}$$

as required. □

Although the exponent of n in Lemma 4 can be improved by more careful analysis its value is immaterial unless we wish to study the exact relationship between S and C in the theorem.

Proof of Theorem. We give a proof for Phase A. The argument for B is identical. Consider any route R and a suitably large constant C. The number of edges a route has in common with R we shall call its *overlap* r. In the analysis we deal with overlaps in three different ways depending on which of the following ranges it falls in:

$$[1,4\alpha],\quad\left[4\alpha,\frac{n}{\log_2 n}\right],\quad\left[\frac{n}{\log_2 n},n\right],$$

for an appropriate constant α to be defined. The second range is itself split into about $\log_2 n$ subranges $[2^i, 2^{i+1}]$ and C is large enough that

$$\frac{Cn}{24(r-\alpha)(\log_2 n-\alpha)}$$

exceeds unity for all values of r in the second range.

In particular we observe that

$$\Pr\left(\sum_s |s\to\cap R|\geqq Cn\right)$$
$$\leqq\Pr\left(|s\to\cap R|\geqq 1\text{ for at least }\frac{Cn}{12\alpha}\text{ values of }s\right)\tag{5}$$
$$+\sum_i\Pr(E_i)+\Pr\left(|s\to\cap R|\geqq\frac{n}{\log_2 n}\text{ for at least }\frac{C}{3}\text{ values of }s\right)$$
$$=\Pr(E^1)+\Pr(E^2)+\Pr(E^3),\quad\text{say}$$

where each component E_i of E^2 is itself the event

$$\text{"}|s\to\cap R|\geqq 2^i\text{ for at least }m_i(n)=\frac{Cn}{24(2^i-\alpha)(\log_2 n-\alpha)}\text{ values of }s\text{"}.$$

In order to verify (5) we note that if the overlap sum exceeds Cn then $E^1,\cdots,E_i$, $E_{i+1},\cdots,E^3$ cannot all be false, for if they were then the contribution from each of the three ranges E^1, E^2 and E^3 would be less than $Cn/3$. In particular, if E^1 is false then the overlap sum in the range $[1,4\alpha]$ is clearly less than $Cn/3$. The same holds for E^3. Finally, if every E_i of E^2 is false also then the overlap sum in the second range

is at most

$$\sum_{i=\lfloor \log_2 4\alpha \rfloor}^{\log_2 n} \frac{Cn}{24(2^i-\alpha)(\log_2 n-\alpha)} \cdot 2^{i+1} \leqq \frac{Cn}{3}$$

for all sufficiently large n.

It remains to show that for a suitable α however large S is chosen there exists C such that the probabilities of E^1, E^2 and E^3 are all bounded by $(\frac{1}{3})2^{-(S+1)n}$. For we can then deduce from (5) that

$$\Pr\left(\sum_s |s \to \cap R| \geqq Cn\right) \leqq 2^{-(S+1)n}.$$

Since there are $N=2^n$ routes R the probability that some of them do have such a large overlap sum is then at most 2^{-Sn} as required.

That for every α and S $\Pr(E^1) \leqq (\frac{1}{3})2^{-(S+1)n}$ for a suitable C is merely a restatement of Lemma 1 and nothing further needs to be proved.

To bound E^2 it is sufficient to prove the claim that for some C for each i $\Pr(E_i) \leqq (3n)^{-1}2^{-(S+1)n}$. By Fact 2 it suffices to consider the event that at least $m=m_i(n)$ successes occur in N trials with equal probabilities. By Lemma 4 this probability is at most

$$p \leqq \frac{n^4 2^r}{n_r N}, \quad \text{where } r=2^i,$$

$$\leqq \frac{n^{4-r}2^{2r}}{N}, \quad \text{since } r=2^i<\frac{n}{2} \text{ in this range.}$$

Using Fact 4

$$B(m, N, p) \leqq (Np)^m e^m \leqq \left(\frac{2^{2r}e}{mn^{r-4}}\right)^m = X, \quad \text{say.}$$

Then

$$\log_2 X \leqq \frac{Cn}{24(r-\alpha)(\log_2 n-\alpha)} \cdot (2r-(r-4)\log_2 n+\log_2 e)$$

$$\leqq \frac{-Cn}{24} \quad \text{if } \alpha=4 \text{ and } r=2^i>4\alpha.$$

We conclude that $\Pr(E_i) \leqq 2^{-Cn/24}$, which is less than $(3n)^{-1}2^{-(S+1)n}$ for C chosen large enough.

Finally to bound E^3 we need an estimate with $r=n/\log_2 n$ and $m=C/3$. Here the analysis for E^2 applies since for $r=n/\log_2 n$

$$\frac{Cn}{24\left(\frac{n}{\log_2 n}-\alpha\right)(\log_2 n-\alpha)} \leqq \frac{C}{3} \quad \text{for all sufficiently large } n.$$

Hence the required bound of $(1/3)2^{-(S+1)n}$ certainly holds for E^3.

We have shown therefore that the overlap sum for the route of each packet is suitably small. Since the overlap sum for a packet bounds the total time it waits in queues the result is established. Although the proof was valid only for sufficiently large values of n, the Theorem can be made to hold for all values of n by always choosing C large enough to cover the remaining small cases. □

5. Remarks.

1. *Testability.* The two phases of the algorithm are testable in the following sense. Phase A is independent of the input altogether. Running it on, say, the identity permutation for different values of F and testing whether it has finished is therefore a method of sampling the distribution of the runtime needed for finishing. In the overall routing algorithm Phase B is used with the packets initially placed randomly in the cube. Hence the distribution of the runtime needed for finishing in Phase B can be sampled by running it with packets placed initially at random. In this way suitable values of F and G can be obtained experimentally. Such experimental results are reported in [11].

2. *Variations.* For practical purposes there is no reason why every packet should wait for Phase A to finish before embarking on Phase B. A modified algorithm in which any packet v immediately enters Phase B as soon as T_v becomes empty in Phase A will also clearly have its runtime bounded by the analysis above, provided the queuing discipline always gives preference to packets still executing Phase A.

Another kind of modification is required if we want the algorithm always to finish. Instead of cutoffs we would have global checks at every n steps to test whether the algorithm has finished. This can be done by collecting one bit of information from each node and collecting their conjunction at one node. For this $O(n)$ steps are sufficient where, now, a step consists of transmitting a fixed piece of bookkeeping along a wire. With this modification each phase is always correct and for all sufficiently large H finishes in time $H \cdot \log_2 N$ with overwhelming probability.

A further variation is possible if we wish to simplify the local computations at the expense of carrying more bookkeeping information. In that case we can precompute the whole of the route for each packet before it leaves its initial node. This also avoids holdups in Phase A that occur whenever $\alpha = 0$ is picked. Note that such holdups occur at most n times for any route and could also be avoided by adding an "until $\alpha = 1$" inner loop to Phase A.

3. *Queuing disciplines.* The proofs given apply to all disciplines, e.g., "first in-first out", "first in-last out". Experimentation appears to be the only method of choosing amongst them. More complicated alternatives include "packets with farthest to go first out".

4. *Obliviousness.* An essential feature of this algorithm is that the route taken by each packet is determined entirely by itself. The other packets can only influence the rate at which the route is traversed. For this reason no global synchronization is required. Indeed, the scheme appears to be well suited to supporting a continuous stream of communication requests from packets generated at the nodes, as long as the traffic flow does not saturate the system either as whole, or by requesting one node or region of it too frequently.

As an alternative "adaptive" algorithms could be considered. For example, one could route the packets from a node so as to minimize the maximal queue length there. Unfortunately such strategies appear to be beyond rigorous analysis or testability.

5. *Necessity for Phase* A. Phase A may appear unnatural at first sight since it may route a packet to distant parts of the network even when its destination is near its source. It is natural to ask therefore whether Phase B on its own works in $O(n)$ steps for *all* inputs, rather than merely for most inputs. A negative answer to this can be derived as follows: Consider any edge $e = (x, y)$. There are $\binom{n-1}{r}$ nodes at distance r from x from which routes can go through e. For each such node z choose as its destination one of the $\binom{n-1}{r}$ nodes at distance r from y to which a route can go from

z via e. The reader can verify that if $r \leqq (n-1)/2$ then this is always possible. Now consider Phase B applied to such a set of $\binom{n-1}{r}$ packets. Since each route will intersect e with probability

$$(r+1)^{-1}\binom{2r+1}{r}^{-1}$$

the expected number of routes intersecting e will be

$$\frac{(n-1)!}{r!(n-r-1)!}\cdot\frac{r!r!}{(2r+1)!}\leqq(r+1)^{-1}\cdot\frac{(n-1)_r}{(2r+1)_r}\leqq(r+1)^{-1}\left(\frac{(n-1)}{(2r+1)}\right)^r$$

If $r = n/J$ this quantity grows as $2^{\gamma n}$ where γ equals $(\log_2(J/2))/J$, which is positive if $J > 2$. Hence if Phase B is started on a suitably bad input it will require N^γ rather than logarithmic time.

We note that the above estimation is asymptotic. For small values of n it is clearly advantageous to omit Phase A.

6. Alternative algorithms. Recently [12] it has been shown that if in each phase the dimensions are traversed in order of dimension number then the proof of the (log N) runtime is much simplified, essentially because the 1ntersection of any two routes must then be a contiguous sequence of edges.

REFERENCES

[1] D. ANGLUIN AND L. G. VALIANT, *Fast probabilistic algorithms for hamiltonian circuits and matchings*, J. Comput. System Sci., 18 (1979), pp. 155–193.
[2] K. E. BATCHER, *Sorting networks and their applications*, AFIPS Spring Joint Comp. Conf., 32 (1968), pp. 307–314.
[3] V. E. BENES, *Mathematical Theory of Connecting Networks and Telephone Traffic*, Academic Press, New York 1965.
[4] H. CHERNOFF, *A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations*, Ann. Math. Statist. 23 (1952), pp. 493–507.
[5] W. HOEFFDING, *On the distribution of the number of successes in independent trials*, Ann. Math. Statist., 27 (1956) pp. 713–721.
[6] G. LEV, N. PIPPENGER AND L. G VALIANT, *A fast parallel algorithm for routing in permutation networks*, IEEE Trans. Comput., C-30 (1981), pp. 93–100.
[7] M. O. RABIN, *Probabilistic algorithms*, Algorithms and Complexity, J. F. Traub, ed., Academic Press, New York, 1976.
[8] H. J. Siegel, *Interconnection networks for* SIMD *machines*, Computer, (June 1979), pp. 57–65.
[9] R. SOLOVAY AND V. STRASSEN, *A fast Monte-Carlo test for primality*, this Journal., 6 (1977), pp. 84–85.
[10] A. WAKSMAN, *A permutation network*, J. Assoc. Comput. Mach., 15 (1968), pp. 159–163.
[11] L. G. VALIANT, *Experiments with a parallel communication scheme*, Proc. 18th Allerton Conf. on Communication, Control and Computing, University of Illinois, Oct. 8–10, 1980, pp. 802–811.
[12] L. G. VALIANT AND G. J. BREBNER, *Universal Schemes for parallel communication*, Proc. 13th ACM Symposium of Theory of Computing, 1981, pp. 263–277.

SIAM J. COMPUT.
Vol. 11, No. 2, May 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1102-0013 $01.00/0

COMPUTATION OF MATRIX CHAIN PRODUCTS. PART I*

T. C. HU† AND M. T. SHING†

Abstract. This paper considers the computation of matrix chain products of the form $M_1 \times M_2 \times \cdots \times M_{n-1}$. If the matrices are of different dimensions, the order in which the product is computed affects the number of operations. An optimum order is an order which minimizes the total number of operations. We present some theorems about an optimum order of computing the matrices. Based on these theorems, an $O(n \log n)$ algorithm for finding an optimum order will be presented in Part II.

Key words. matrix multiplication, polygon partition, dynamic programming

1. Introduction. Consider the evaluation of the product of $n-1$ matrices

$$M = M_1 \times M_2 \times \cdots \times M_{n-1}, \tag{1}$$

where M_i is a $w_i \times w_{i+1}$ matrix. Since matrix multiplication satisfies the associative law, the final result M in (1) is the same for all orders of multiplying the matrices. However, the order of multiplication greatly affects the total number of operations to evaluate M. The problem is to find an optimum order of multiplying the matrices such that the total number of operations is minimized. Here, we assume that the number of operations to multiply a $p \times q$ matrix by a $q \times r$ matrix is pqr.

In [1], [7], a dynamic programming algorithm is used to find an optimum order. The algorithm needs $O(n^3)$ time and $O(n^2)$ space. In [2], Chandra proposed a heuristic algorithm to find an order of computation which requires no more than $2T_o$ operations where T_o is the total number of operations to evaluate (1) in an optimum order. This heuristic algorithm needs only $O(n)$ time. Chin [3] proposed an improved heuristic algorithm to give an order of computation which requires no more than $1.25T_o$. This improved heuristic algorithm also needs only $O(n)$ time.

In this paper we first transform the matrix chain product problem into a problem in graph theory—the problem of partitioning a convex polygon into nonintersecting triangles, see [9], [10], [11], [12]; then we state several theorems about the optimum partitioning problem. Based on these theorems, an $O(n \log n)$ algorithm for finding an optimum partition is developed.

2. Partitioning a convex polygon. Given an n-sided convex polygon, such as the hexagon shown in Fig. 1, the number of ways to partition the polygon into $(n-2)$ triangles by nonintersecting diagonals is the Catalan number (see for example, Gould [8]). Thus, there are 2 ways to partition a convex quadrilateral, 5 ways to partition a convex pentagon, and 14 ways to partition a convex hexagon.

Let every vertex V_i of the polygon have a positive weight w_i. We can define the cost of a given partition as follows: The cost of a triangle is the product of the weights of the three vertices, and the cost of partitioning a polygon is the sum of the costs of all its triangles. For example, the cost of the partition of the hexagon in Fig. 1 is

$$w_1w_2w_3 + w_1w_3w_6 + w_3w_4w_6 + w_4w_5w_6. \tag{2}$$

* Received by the editors May 19, 1980, and in final revised form September 9, 1981. This research was supported in part by the National Science Foundation under grant MCS-80-03362 and by the U.S. Army Research Office under grant DAAG29-80-C-0029.

† Department of Electrical Engineering and Computer Sciences, University of California, San Diego, California 92093.

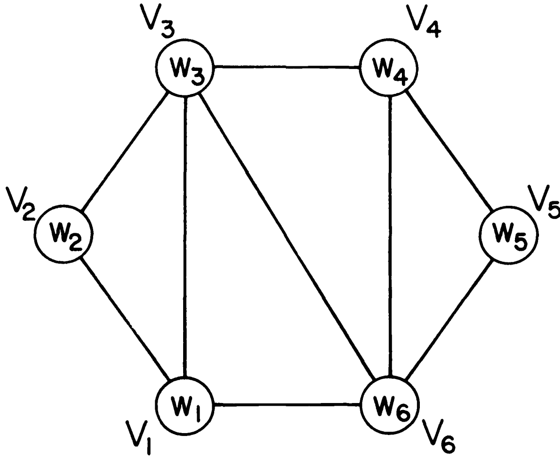


FIG. 1

If we erase the diagonal from V_3 to V_6 and replace it by the diagonal from V_1 to V_4, then the cost of the new partition will be

$$w_1w_2w_3 + w_1w_3w_4 + w_1w_4w_6 + w_4w_5w_6. \tag{3}$$

We will prove that an order of multiplying $n-1$ matrices corresponds to a partition of a convex polygon with n sides. The cost of the partition is the total number of operations needed in multiplying the matrices. For brevity, we shall use n-gon to mean a convex polygon with n sides, and the partition of an n-gon to mean the partitioning of an n-gon into $n-2$ nonintersecting triangles.

For any n-gon, one side of the n-gon will be considered to be its base, and will usually be drawn horizontally at the bottom such as the side $V_1 - V_6$ in Fig. 1. This side will be called the base; all other sides are considered in a clockwise way. Thus, $V_1 - V_2$ is the first side, $V_2 - V_3$ the second side, $\cdots$ and $V_5 - V_6$ the fifth side.

The first side represents the first matrix in the matrix chain and the base represents the final result M in (1). The dimensions of a matrix are the two weights associated with the two end vertices of the side. Since the adjacent matrices are compatible, the dimensions $w_1 \times w_2, w_2 \times w_3, \cdots, w_{n-1} \times w_n$ can be written inside the vertices as $w_1, w_2, \cdots, w_n$. The diagonals are the partial products. A partition of an n-gon corresponds to an alphabetic tree of $n-1$ leaves or the parenthesis problem of $n-1$ symbols (see, for example, Gardner [6]). It is easy to see the one-to-one correspondence between the multiplication of $n-1$ matrices to either the alphabetic binary tree or the parenthesis problem of $n-1$ symbols. Here, we establish the correspondence between the matrix-chain product and the partition of a convex polygon directly.

LEMMA 1. *Any order of multiplying $n-1$ matrices corresponds to a partition of an n-gon.*

Proof. We shall use induction on the number of matrices. For two matrices of dimensions $w_1 \times w_2$, $w_2 \times w_3$, there is only one way of multiplication; this corresponds to a triangle where no further partition is required. The total number of operations in multiplication is $w_1w_2w_3$, the product of the three weights of the vertices. The resulting matrix has dimension $w_1 \times w_3$. For three matrices, the two orders of multiplication $(M_1 \times M_2) \times M_3$ and $M_1 \times (M_2 \times M_3)$ correspond to the two ways of partitioning a 4-gon. Assume that this lemma is true for k matrices where $k \leqq n-2$, and we now consider $n-1$ matrices. The n-gon is shown in Fig. 2.

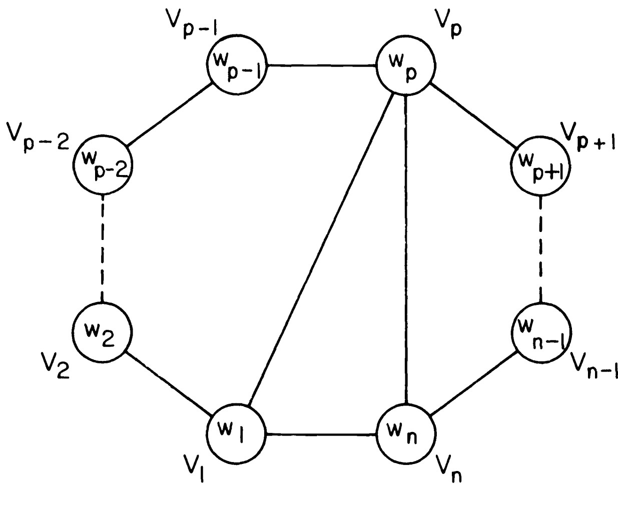


FIG. 2

Let the order of multiplication be represented by

$$M=(M_1\times M_2\times\cdots\times M_{p-1})\times(M_p\times\cdots\times M_{n-1});$$

i.e., the final matrix is obtained by multiplying a matrix of dimension $(w_1\times w_p)$ and a matrix of dimension $(w_p\times w_n)$. Then in the partition of the n-gon, we let the triangle with vertices V_1 and V_n have the third vertex V_p. The polygon $V_1-V_2-\cdots-V_p$ is a convex polygon of p sides with base V_1-V_p and its partition corresponds to an order of multiplying matrices $M_1,\cdots,M_{p-1}$, giving a matrix of dimension $w_1\times w_p$. Similarly, the partition of the polygon $V_p-V_{p+1}-\cdots-V_n$ with base V_p-V_n corresponds to an order of multiplying matrices $M_p,\cdots,M_{n-1}$, giving a matrix of dimension $w_p\times w_n$. Hence the triangle $V_1V_pV_n$ with base V_1-V_n represents the multiplication of the two partial products, giving the final matrix of dimension $w_1\times w_n$. □

LEMMA 2. *The minimum numbers of operations needed to evaluate the following matrix chain products are identical.*

$$M_1\times M_2\times\cdots\times M_{n-2}\times M_{n-1},$$
$$M_n\times M_1\times\cdots\times M_{n-3}\times M_{n-2},$$
$$\vdots$$
$$M_2\times M_3\times\cdots\times M_{n-1}\times M_n,$$

where M_i has dimension $w_i\times w_{i+1}$ and $w_{n+1}\equiv w_1$. Note that in the first matrix chain, the resulting matrix is of dimension $w_1\times w_n$. In the last matrix chain, the resulting matrix is of dimension $w_2\times w_1$. But in all the cases, the total number of operations in the optimum orders of multiplication is the same.

Proof. The cyclic permutations of the $n-1$ matrices all correspond to the same n-gon and thus have the same optimum partitions. □

(This lemma was obtained independently in [4] with a long proof.)

From now on, we shall concentrate only on the partitioning problem.

The diagonals inside the polygon are called arcs. Thus, one easily verifies inductively that every partition consists of $n-2$ triangles formed by $n-3$ arcs and n sides.

In a partition of an n-gon, the degree of a vertex is the number of arcs incident on the vertex plus two (since there are two sides incident on every vertex).

LEMMA 3. *In any partition of an n-gon, $n \geqq 4$, there are at least two triangles, each having a vertex of degree two.* (*For example, in Fig.* 1, *the triangle $V_1V_2V_3$ has vertex V_2 with degree* 2 *and the triangle $V_4V_5V_6$ has vertex V_5 with degree* 2.) (*See also* [5].)

Proof. In any partition of an n-gon, there are $n-2$ nonintersecting triangles formed by $n-3$ arcs and n sides. And for any $n \geqq 4$, no triangle can be formed by 3 sides. Let x be the number of triangles with two sides and one arc, y be the number of triangles with one side and two arcs, and z be the number of triangles with three arcs. Since an arc is used in two triangles, we have

$$x+2y+3z=2(n-3). \tag{4}$$

Since the polygon has n sides, we have

$$2x+y=n. \tag{5}$$

From (4) and (5), we get

$$3x=3z+6.$$

Since $z \geqq 0$, we have $x \geqq 2$. □

LEMMA 4. *Let P and P' both be n-gons where the corresponding weights of the vertices satisfy $w_i \leqq w'_i$. Then the cost of an optimum partition of P is less than or equal to the cost of an optimum partition of P'.*

Proof. Omitted. □

If we use $C(w_1, w_2, w_3, \cdots, w_k)$ to mean the minimum cost of partitioning the k-gon with weights w_i optimally, Lemma 4 can be stated as

$$C(w_1, w_2, \cdots, w_k) \leqq C(w'_1, w'_2, \cdots, w'_k) \quad \text{if } w_i \leqq w'_i.$$

We say that two vertices are *connected* in an optimum partition if the two vertices are connected by an arc or if the two vertices are adjacent to the same side.

In the rest of the paper, we shall use $V_1, V_2, \cdots, V_n$ to denote vertices which are ordered according to their weights, i.e., $w_1 \leqq w_2 \leqq \cdots \leqq w_n$. To facilitate the presentation, we introduce a tie-breaking rule for vertices of equal weights.

If there are two or more vertices with weights equal to the smallest weight w_1, we can arbitrarily choose one of these vertices to be the vertex V_1. Once the vertex V_1 is chosen, further ties in equal weights are resolved by regarding the vertex which is closer to V_1 in the clockwise direction to be of less weight. With this tie-breaking rule, we can unambiguously label the vertices $V_1, V_2, \cdots, V_n$ for each choice of V_1. A vertex V_i is said to be *smaller than* another vertex V_j, denoted by $V_i < V_j$, either if $w_i < w_j$ or if $w_i = w_j$ and $i < j$. We say that V_i is the *smallest* vertex in a subpolygon if it is smaller than any other vertices in the subpolygon.

After the vertices are labeled, we define an arc $V_i - V_j$ to be *less than* another arc $V_p - V_q$ if

$$\min(i, j) < \min(p, q) \quad \text{or} \quad \begin{cases} \min(i, j) = \min(p, q), \\ \max(i, j) < \max(p, q). \end{cases}$$

(For example, the arc $V_3 - V_9$ is less than the arc $V_4 - V_5$.) Every partition of an n-gon has $n-3$ arcs which can be sorted from the smallest to the largest into an ordered sequence of arcs, i.e., each partition is associated with a unique ordered sequence of arcs. We define a partition P to be *lexicographically less than* a partition Q if the ordered sequence of arcs associated with P is lexicographically less than that associated with Q.

When there is more than one optimum partition, we use the *l-optimum partition* (i.e., lexicographically-optimum partition) to mean the lexicographically smallest optimum partition, and use *an optimum partition* to mean some partition of minimum cost.

We shall use V_a, V_b, $\cdots$ to denote vertices which are unordered in weights, and T_{ijk} to denote the product of the weights of any three vertices V_i, V_j and V_k.

THEOREM 1. *For every way of choosing* V_1, V_2, $\cdots$ *(as prescribed), there is always an optimum partition containing* $V_1 - V_2$ *and* $V_1 - V_3$. *(Here,* $V_1 - V_2$ *and* $V_1 - V_3$ *may be either arcs or sides.)*

Proof. The proof is by induction, For the optimum partitions of a triangle and a 4-gon, the theorem is true. Assume that the theorem is true for all k-gons ($3 \leqq k \leqq n-1$) and consider the optimum partitions of an n-gon.

From Lemma 3, in any optimum partition, we can find at least two vertices having degree two. Call these two vertices V_i and V_j. We can divide this into two cases.

(i) One of the two vertices V_i (or V_j) is not V_1, V_2 or V_3 in some optimum partition of the n-gon. In this case, we can remove the vertex V_i with its two sides and obtain an $(n-1)$-gon. In this $(n-1)$-gon, V_1, V_2, V_3 are the three vertices with smallest weights. By the induction assumption, V_1 is connected to both V_2 and V_3 in an optimum partition.

(ii) Consider the complementary case of (i), in all the optimum partitions of the n-gon, all the vertices with degree two are from the set $\{V_1, V_2, V_3\}$. (In this case, there will be at most three vertices with degree two in every optimum partition.) We have the following three subcases:

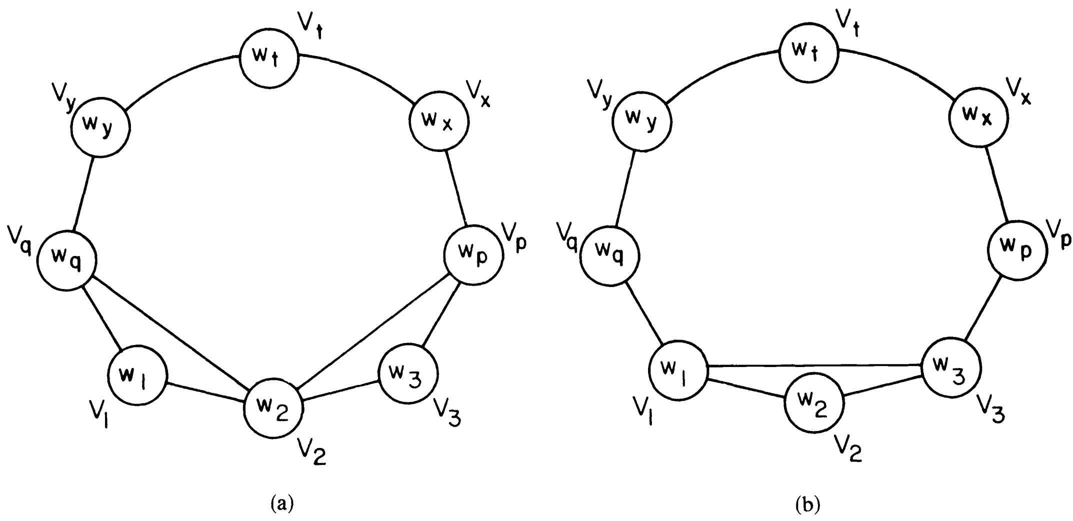
(a) $V_i = V_2$ and $V_j = V_3$ in some optimum partition of the n-gon, i.e., both V_2 and V_3 have degree two simultaneously. In this case, we first remove V_2 with its two sides and form an $(n-1)$-gon. By the induction assumption, V_1, V_3 must be connected in some optimum partition. If $V_1 - V_3$ appears as an arc, it reduces to (i). So $V_1 - V_3$ must appear as a side of the $(n-1)$-gon, and reattaching V_2 to the $(n-1)$-gon shows that either V_1, V_2 and V_3 are mutually adjacent or $V_1 - V_3$ is a side of the n-gon. In the former case, the proof is complete, so we assume that $V_1 - V_3$ is a side of the n-gon. Similarly, we can remove V_3 with its two sides and show that V_1, V_2 are connected by a side of the n-gon.

(b) $V_i = V_1$ and $V_j = V_2$ in some optimum partition of the n-gon, i.e., V_1 and V_2 both have degree two simultaneously. In this case, we can first remove V_1 and form an $(n-1)$-gon where V_2, V_3, V_4 are the three vertices with smallest weights. By the induction assumption, V_2 is connected to both V_3 and V_4 in an optimum partition. If $V_2 - V_3$ or $V_2 - V_4$ appears as an arc, it reduces to (i). Hence, $V_2 - V_3$ and $V_2 - V_4$ must both be sides of the n-gon. Similarly, we can remove V_2 with its two sides and form an $(n-1)$-gon where V_1, V_3, V_4 are the three vertices with smallest weights. Again, V_1 must be connected to V_3 and V_4 by sides of the n-gon. But for any n-gon with $n \geqq 5$, it is impossible to have V_3 and V_4 both adjacent to V_1 and V_2 at the same time, i.e., V_1 and V_2 cannot both have degree two in an optimum partition of any n-gon with $n \geqq 5$.

(c) $V_i = V_1$, $V_j = V_3$ in some optimum partition of the n-gon. By argument similar to (b), we can show that V_2 must be adjacent to V_1 and V_3 in the n-gon. The situation is as shown in Fig. 3(a). Then the partition in Fig. 3(b) is cheaper because

$$T_{123} \leq T_{12q}$$

and $C(w_1, w_q, w_y, w_t, w_x, w_p, w_3) \leqq C(w_2, w_q, w_y, w_t, w_x, w_p, w_3)$ according to Lemma 4. □



(a) (b)

FIG. 3

COROLLARY 1. *For every way of choosing $V_1, V_2, \cdots$ (as prescribed), the l-optimum partition always contains $V_1 - V_2$ and $V_1 - V_3$.*

Proof. It follows from Theorem 1 and the definition of the l-optimum partition. □

Once we know $V_1 - V_2$ and $V_1 - V_3$ always exist in the l-optimum partition, we can use this fact recursively. Hence, in finding the l-optimum partition of a given polygon, we can decompose it into subpolygons by joining the smallest vertex with the second smallest and third smallest vertices repeatedly, until each of these subpolygons has the property that its smallest vertex is adjacent to both its second smallest and third smallest vertices.

A polygon having V_1 adjacent to V_2 and V_3 by sides will be called a *basic* polygon.

THEOREM 2. *A necessary but not sufficient condition for V_2–V_3 to exist in an optimum partition of a basic polygon is*

$$\frac{1}{w_1}+\frac{1}{w_4} \leqq \frac{1}{w_2}+\frac{1}{w_3}. \tag{6}$$

Furthermore, if $V_2 - V_3$ is not present in the l-optimum partition, then V_1, V_4 are always connected in the l-optimum partition.

Proof. If V_2, V_3 are not connected in the l-optimum partition of a basic polygon, the degree of V_1 is greater than or equal to 3. Let V_p be a vertex in the polygon and V_1, V_p be connected in the l-optimum partition. V_4 is either in the subpolygon containing V_1, V_2 and V_p or in the subpolygon containing V_1, V_3 and V_p. In either case, V_4 will be the third smallest vertex in the subpolygon. From Corollary 1, V_1, V_4 are connected in the l-optimum partition of the subpolygon and it also follows that V_1, V_4 are connected in the l-optimum partition of the basic polygon.

If V_2, V_3 are connected in an optimum partition, then we have an $(n-1)$-gon where V_2 is the smallest vertex and V_4 is the third smallest vertex. By Theorem 1, there exists an optimum partition of the $(n-1)$-gon in which V_2, V_4 are connected. Thus by induction on n, we can assume that V_4 is adjacent to V_2 in the basic polygon as shown in Fig. 4(a).

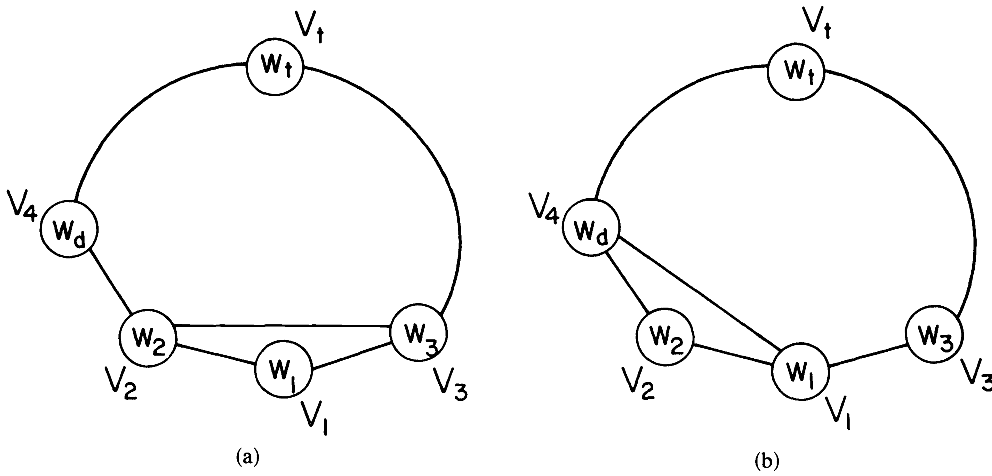


FIG. 4

The cost of the partition in Fig. 4(a) is

$$T_{123}+C(w_2, w_4, \cdots, w_t, \cdots, w_3), \tag{7}$$

and the cost of the partition in Fig. 4(b) is

$$T_{124}+C(w_1, w_4, \cdots, w_t, \cdots, w_3).$$

According to Lemma 4,

$$C(w_1, w_4, \cdots, w_t, \cdots, w_3) \leqq C(w_2, w_4, \cdots, w_t, \cdots, w_3). \tag{9}$$

Since the weights of the vertices between V_4 and V_3 in the clockwise direction are all greater than or equal to w_4, the difference between the right-hand side and the left-hand side of (9) is at least

$$T_{243}-T_{143}.$$

So the necessary condition for (7) to be no greater than (8) is

$$T_{123}+T_{243} \leqq T_{124}+T_{134}$$

or

$$\frac{1}{w_1}+\frac{1}{w_4} \leqq \frac{1}{w_2}+\frac{1}{w_3}. \qquad \square$$

LEMMA 5. *In an optimum partition of an n-gon, let V_x, V_y, V_z and V_w be four vertices of an inscribed quadrilateral (V_x and V_z are not adjacent in the quadrilateral). A necessary condition for V_x–V_z to exist is*

$$\frac{1}{w_x}+\frac{1}{w_z} \geqq \frac{1}{w_y}+\frac{1}{w_w}. \tag{10}$$

Proof. The cost of partitioning the quadrilateral by the arc V_x–V_z is

$$T_{xyz}+T_{xzw}, \tag{11}$$

and the cost partitioning the quadrilateral by the arc V_y-V_w is

$$T_{xyw}+T_{yzw}. \tag{12}$$

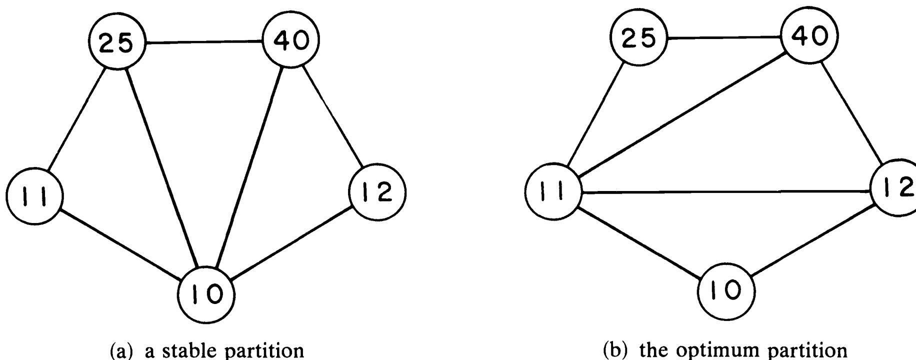
For optimality, we have (11) ≦ (12) which is (10). □

Note that if strict inequality holds in (10), the necessary condition is also sufficient. If equality holds in (10), the condition is sufficient for $V_x - V_z$ to exist in the l-optimum partition provided min $(x, z) <$ min (y, w). This lemma is a generalization of [3, Lemma 1] where V_y is the vertex with the smallest weight and V_x, V_w, V_z are three consecutive vertices with w_w greater than both w_x and w_z.

A partition is called *stable* if every quadrilateral in the partition satisfies (10).

COROLLARY 2. *An optimum partition is stable but a stable partition may not be optimum.*

Proof. The fact that an optimum partition has to be stable follows from Lemma 5. Figure 5 gives an example that a stable partition may not be optimum. □



(a) a stable partition (b) the optimum partition

FIG. 5

In any partition of an n-gon, every arc dissects a unique quadrilateral. Let V_x, V_y, V_z, V_w be the four vectices of an inscribed quadrilateral and $V_x - V_z$ be the arc which dissects the quadrilateral. We define $V_x - V_z$ to be a *vertical* arc if (13) or (14) is satisfied.

$$\min(w_x, w_z) < \min(w_y, w_w), \tag{13}$$

$$\min(w_x, w_z) = \min(w_y, w_w), \qquad \max(w_x, w_z) \leqq \max(w_y, w_w). \tag{14}$$

We define $V_x - V_z$ to be a *horizontal* arc if (15) is satisfied

$$\min(w_x, w_z) > \min(w_y, w_w), \qquad \max(w_x, w_z) < \max(w_y, w_w). \tag{15}$$

For brevity, we shall use *h-arcs* and *v-arcs* to denote horizontal arcs and vertical arcs from now on.

COROLLARY 3. *All arcs in an optimum partition must be either vertical arcs or horizontal arcs.*

Proof. Let $V_x - V_z$ be an arc which is neither vertical nor horizontal. There are two cases:

Case 1. $\min(w_x, w_z) = \min(w_y, w_w)$ and $\max(w_x, w_z) > \max(w_y, w_w)$;

Case 2. $\min(w_x, w_z) > \min(w_y, w_w)$ and $\max(w_x, w_z) \geqq \max(w_y, w_w)$.

In both cases, the inequality (10) in Lemma 5 cannot be satisfied. This implies that the partition is not stable and hence cannot be optimum. □

THEOREM 3. *Let V_x and V_z be two arbitrary vertices which are not adjacent in a polygon, and V_w be the smallest vertex from V_x to V_z in the clockwise manner ($V_w \neq V_x$, $V_w \neq V_z$), and V_y be the smallest vertex from V_z to V_x in the clockwise manner*

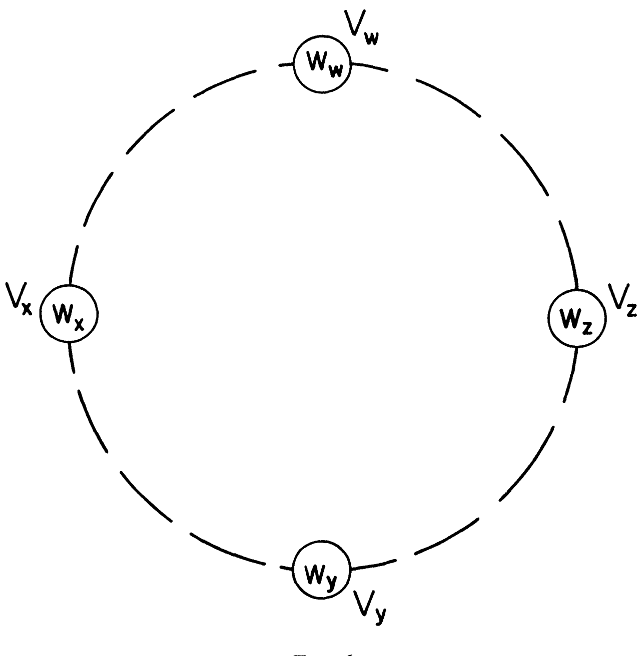


FIG. 6

$(V_y \neq V_x, V_y \neq V_z)$. This is shown in Fig. 6, where without loss of generality we assume that $V_x < V_z$ and $V_y < V_w$. A necessary condition for $V_x - V_z$ to exist as an h-arc in the l-optimum partition is that

$$w_y < w_x \leqq w_z < w_w.$$

(Note that the necessary condition still holds when the positions of V_y and V_w are interchanged.)

Proof. The proof is by contradiction. If $w_x \leqq w_y$, w_x must be equal to the smallest weight w_1, and $V_x - V_z$ can never satisfy (15). Hence, in order that $V_x - V_z$ exist as an h-arc in the l-optimum partition, we must have $w_y < w_x \leqq w_z$. Since V_y is the smallest vertex from V_z to V_x in the clockwise manner and $V_x < V_w$, we must have $V_y = V_1$.

Assume for the moment that $V_3 < V_x < V_z$. From Corollary 1, both $V_1 - V_2$ and $V_1 - V_3$ exist in the l-optimum partition, and the two arcs would divide the polygon into subpolygons. If V_x and V_z are in different subpolygons, then they cannot be connected in the l-optimum partition. Without loss of generality, we can assume that the polygon is a basic polygon. In this basic polygon, either $V_2 - V_3$ or $V_1 - V_4$ exists in the l-optimum partition (Theorem 2).

If V_2, V_3 are connected, then V_x and V_z are both in a smaller polygon in which we can treat V_2 as the smallest vertex and repeat the argument. If V_1, V_4 are connected, the basic polygon is again divided into two subpolygons and V_x and V_z both have to be in one of the subpolygons and the subpolygon has at most $n-1$ sides. (Otherwise $V_x - V_z$ can never exist in the l-optimum partition.) The successive reduction in the size of the polygon will either make the connection $V_x - V_z$ impossible, or force V_x and V_z to become the second smallest and the third smallest vertices in a basic subpolygon. Let V_m be the smallest vertex in this basic subpolygon. In order that $V_x - V_z$ appear as an h-arc, we must have $w_x > w_m$. From Theorem 2, the necessary condition for $V_x - V_z$ (i.e., $V_2 - V_3$) to exist in an optimum partition of the subpolygon

is

$$\frac{1}{w_x}+\frac{1}{w_z}\geqq\frac{1}{w_m}+\frac{1}{w_w}.$$

Since $w_x > w_m$, the inequality is valid only if $w_z < w_w$. □

COROLLARY 4. *A weaker necessary condition for $V_x - V_z$ to exist as an h-arc in the l-optimum partition is that*

$$V_y < V_x < V_z < V_w.$$

Proof. This follows from Theorem 3. □

We call any arc which satisfies this weaker necessary condition a *potential h-arc*. Let P be the set of potential h-arcs in the n-gon and H be the set of h-arcs in the l-optimum partition, we have $P \supseteq H$ where the inclusion could be proper.

COROLLARY 5. *Let V_w be the largest vertex in the polygon and V_x and V_z be its two neighboring vertices. If there exists a vertex V_y such that $V_y < V_x$ and $V_y < V_z$, then $V_x - V_z$ is a potential h-arc.*

Proof. This follows directly from Corollary 4 where there is only one vertex between V_x and V_z. □

Two arcs are called *compatible* if both arcs can exist simultaneously in a partition. Assume that all weights of the vertices are distinct, then there are $(n-1)!$ distinct permutations of the weights around an n-gon. For example, the weights 10, 11, 25, 40, 12 in Fig. 5(a) correspond to the permutation w_1, w_2, w_4, w_5, w_3 (where $w_1 < w_2 < w_3 < w_4 < w_5$). There are infinitely many values of weights which correspond to the same permutation. For example, 1, 16, 34, 77, 29 also corresponds to w_1, w_2, w_4, w_5, w_3 but its optimum partition is different from that of 10, 11, 25, 40, 12. However, all the potential h-arcs in all the n-gons with the same permutation of weights are compatible. We state this remarkable fact as Theorem 4.

THEOREM 4. *All potential h-arcs are compatible.*

Proof. The proof is by contradiction. Let V_x, V_y, V_z and V_w be the four vertices described in Theorem 3. Hence, we have $V_y < V_x < V_z < V_w$ and $V_x - V_z$ is a potential h-arc. Let $V_p - V_q$ be a potential h-arc which is not compatible to $V_x - V_z$, as shown in Fig. 7. Without loss of generality, we can assume $V_p < V_q$. (The proof for the case $V_q < V_p$ is similar to that which follows.)

Since V_w is the smallest vertex between V_x and V_z in the clockwise manner, we have $V_z < V_w < V_q$. Hence, we have either $V_y < V_p < V_z < V_q$ or $V_y < V_z < V_p < V_q$. Both cases violate Corollary 4 and $V_p - V_q$ cannot be a potential h-arc. □

Note that the potential h-arc $V_x - V_z$ always dissects the n-gon into two subpolygons and one of these subpolygons has the property that all its vertices except V_x and V_z have weights no smaller than max (w_x, w_z). We shall call this subpolygon the *upper* subpolygon of $V_x - V_z$. For example, the subpolygon $V_x - \cdots - V_w - \cdots - V_z$ in Fig. 7 is the upper subpolygon of $V_x - V_z$.

Using Corollary 4 and Theorem 4, we can generate all the potential h-arcs of a polygon.

Let $V_x - V_z$ be the arc defined in Corollary 5, i.e., $V_1 < V_x < V_z < V_w$. The arc $V_x - V_z$ is a potential h-arc compatible with all other potential h-arcs in the n-gon. Furthermore, there is no other potential h-arc in its upper subpolygon. Now consider the $(n-1)$-gon obtained by cutting out V_w. In this $(n-1)$-gon, let $V_{w'}$ be the largest vertex and $V_{x'}$ and $V_{z'}$ be the two neighbors of $V_{w'}$ where $V_1 < V_{x'} < V_{z'} < V_{w'}$. Then $V_{x'} - V_{z'}$ is again a potential h-arc compatible with all other potential h-arcs in the

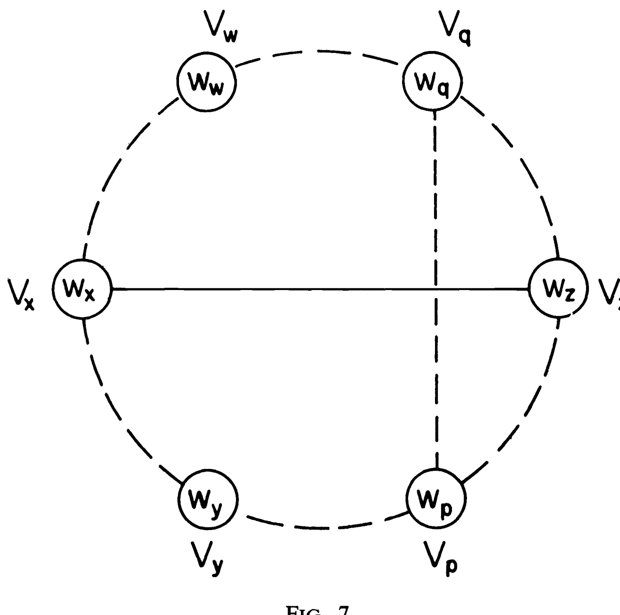


FIG. 7

n-gon and there is no other potential h-arc in its upper subpolygon which has not been generated. This is true even if V_w is in the upper subpolygon of $V_{x'} - V_{z'}$. If we repeat the process of cutting out the largest vertex, we get a set P of arcs, all of which satisfy Corollary 4. The h-arcs of the l-optimum partition must be a subset of these arcs.

The process of cutting out the largest vertex can be made into an algorithm which is $O(n)$. We shall call this algorithm the *one-sweep algorithm*. The output of the one-sweep algorithm is a set S of n-3 arcs. S is empty initially.

The one-sweep algorithm. Starting from the smallest vertex, say V_1, we travel in the clockwise direction around the polygon and push the weights of the vertices successively onto the stack as follows (w_1 will be at the bottom of the stack).

(a) Let V_t be the top element on the stack, V_{t-1} be the element immediately below V_t, and V_c be the element to be pushed onto the stack. If there are two or more vertices on the stack and $w_t > w_c$, add $V_{t-1} - V_c$ to S, pop V_t off the stack; if there is only one vertex on the stack or $w_t \leqq w_c$, *push* w_c onto the stack. Repeat this step until the nth vertex has been pushed onto the stack.

(b) If there are more than three vertices on the stack, add $V_{t-1} - V_1$ to S, pop V_t off the stack and repeat this step, else stop.

Since we do not check for the existence of a smallest vertex whose weight is no larger than those of the two neighbors of the largest vertex, i.e., the existence of the vertex V_y in Corollary 4, not all the $n-3$ arcs generated by the algorithm are potential h-arcs. However, it is not difficult to verify that the one-sweep algorithm always generates a set S of $n-3$ arcs which contains the set P of all potential h-arcs which contains the set H of all h-arcs in the l-optimum partition of the n-gon, i.e.,

$$S \supseteq P \supseteq H,$$

where each inclusion could be proper. For example, if the weights of the vertices around the n-gon in the clockwise direction are $w_1, w_2, \cdots, w_n$ where $w_1 \leqq w_2 \leqq \cdots \leqq w_n$, none of the arcs in the n-gon can satisfy Corollary 4 and hence there are no

potential h-arcs in the n-gon. The one-sweep algorithm would still generate $n-3$ arcs for the n-gon but none of the arcs generated is a potential h-arc.

3. Conclusion. In this paper, we have presented several theorems on the polygon partitioning problem. Some of these theorems are characterizations of the optimum partitions of any n-sided convex polygon, while the others apply to the unique lexicographically smallest optimum partition. Based on these theorems an $O(n)$ algorithm for finding a near-optimum partition can be developed [12]. The cost of the partition produced by the heuristic algorithm never exceeds $1{\cdot}155\ C_{\text{opt}}$, where C_{opt} is the optimum cost of partitioning the polygon. An $O(n \log n)$ algorithm for finding the unique lexicographically smallest optimum partition will be presented in Part II [13].

Acknowledgment. The authors would like to thank the referees for their helpful comments in revising the manuscript.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
[2] A. K. Chandra, *Computing matrix chain product in near optimum time*, IBM Res. Rep. RC5626 (#24393), IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1975.
[3] F. Y. Chin, *An $O(n)$ algorithm for determining a near optimal computation order of matrix chain product*, Comm. ACM, 21, (1978), pp. 544–549.
[4] L. E. Deimel, Jr. and T. A. Lampe, *An invariance theorem concerning optimal computation of matrix chain products*, Rep. TR79-14, North Carolina State Univ., Raleigh.
[5] G. A. Dirac, *On rigid circuit graphs*, Abh. Math. Sem., Univ. Hamburg, 25 (1961), pp. 71–76.
[6] M. Gardner, *Catalan numbers*, Scientific American, June 1976, pp. 120–124.
[7] S. S. Godbole, *An efficient computation of matrix chain products*, IEEE Trans. Comput., C-22, (1973), pp. 864–866.
[8] H. W. Gould, *Bell and Catalan numbers*, Combinatorial Research Institute, Morgantown, WV., June 1977.
[9] T. C. Hu and M. T. Shing, *Computation of Matrix Chain Product*, Abstract, Amer. Math. Soc., Vol. 1, (1980), p. 336.
[10] ———, *Some theorems about matrix multiplications*, Proc. 21st Annual IEEE Symposium on the Foundations of Computer Science, October 1980, pp. 28–35.
[11] ———, *Computation of matrix chain products*, Proc. 1981 Army Numerical Analysis and Computations Conference, August 1981, pp. 615–628.
[12] ———, *An $O(n)$ algorithm to find a near-optimum partition of a convex polygon*, J. Algorithms, 2 (1981), pp. 122–138.
[13] ———, *Computation of matrix chain products. Part II*, submitted.

SIAM J. COMPUT.
Vol. 11, No. 2, May 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1102-0014 $01.00/0

ON PRIMALITY TESTS*

DANIEL J. LEHMANN†

Abstract. Whether an odd number m is prime can be decided on the knowledge of the image of the function $a \mapsto a^{(m-1)/2}(m)$. As a consequence, an algorithm for testing primality is proposed (under the extended Riemann hypothesis) which is more efficient than ones proposed by Miller [Proc. 7th ACM Symp. Theory of Computing, 1975, pp. 234–239] and Vélu [SIGACT News, 10 (1978), pp. 58–59]. A probabilistic version is compared with the algorithm of Solovay and Strassen [SIAM J. Comput., 6 (1977), pp. 84–85; erratum, 7 (1978), p. 118].

Key words. primality, extended Riemann hypothesis, probabilistic algorithms

Let $(Z/mZ)^*$ be the multiplicative group of units of Z/mZ. Let $f_{n,m}:(Z/mZ)^* \to (Z/mZ)^*$ be the group homomorphism defined by $f_{n,m}(a)=a^n$. If $m=p^\alpha$ for a prime p and a positive integer α, we shall say that m is a prime power (n.b: a prime is a prime power).

LEMMA 1. *If m is not a prime power, then the image of $f_{n,m}$ is not the subgroup $\{+1,-1\}$ (it may be in that subgroup, though).*

Proof. If m is not a prime power, it is the product of two relatively prime integers greater than 1. Let $m=m_1m_2$, $(m_1,m_2)=1$, $m_1, m_2>1$. Suppose

$$f_{n,m}(a)=-1 \quad \text{for some } a \in (Z/mZ)^*.$$

Then $a^n \equiv -1(m_1)$. By the Chinese remainder theorem, there is a (unique) $x \in (Z/mZ)^*$ such that $x \equiv a\ (m_1)$ and $x \equiv 1\ (m_2)$. Then $f_{n,m}(x)$ is not in $\{1,-1\}$. □

THEOREM 1. *Let m be an odd integer. The image of $f_{(m-1)/2,m}$ is the subgroup $\{+1,-1\}$ if and only if m is prime.*

Proof. The *if* part follows from well-known theorems of elementary number theory. The *only if* part follows from Lemma 1 and the fact that if $m=p^\alpha$ for p a prime and $\alpha>1$, $(Z/mZ)^*$ is cyclic of order $p^{\alpha-1}(p-1)$, which is divisible by p; if the image of $f_{m-1,m}$ were the trivial subgroup, p would divide $m-1=p^\alpha-1$. □

Note that $f_{(m-1)/2,m}$ can be trivial only if m is not a prime power.

THEOREM 2 (Ankeny–Montgomery). *If the extended Riemann hypothesis is true, there is a number M such that for every integer m every Abelian group G and every nontrivial homomorphism $f:(Z/mZ)^* \to G$ there is an element a less than $M(\log m)^2$ such that $f(a) \neq 1$.*

ALGORITHM 1. For each element a of $(Z/mZ)^*$ less than $M(\log m)^2$, compute $f_{(m-1)/2,m}(a)$.

If at least one of the values found is different from 1 and -1, then m is composite. Otherwise, if at least one of the values found is -1, then m is prime and else m is composite.

Proof of correctness. If at least one of the values found is different from 1 and -1, then by Theorem 1, m is composite. If all the values found are $+1$ or -1, Theorem 2 implies that the homomorphism $g:(Z/mZ)^* \to (Z/mZ)^*/\{+1,-1\}$ defined by $g = j \circ f_{(m-1)/2,m}$, where j is the canonical projection $(Z/mZ)^* \to (Z/mZ)^*/\{+1,-1\}$, is trivial. The image of $f_{(m-1)/2,m}$ is therefore a subgroup of $\{+1,-1\}$. If the value -1 is

* Received by the editors January 24, 1979, and in revised form April 13, 1980.

† Department of Computer Science, Institute of Mathematics, The Hebrew University of Jerusalem, Jerusalem, Israel. Part of this work was done while the author was at the University of Southern California, Los Angeles, California.

found at least once, then the image is $\{+1, -1\}$ and by Theorem 1, m is prime. If all the values found are $+1$, then by Theorem 2, $f_{(m-1)/2,m}$ is trivial and by Theorem 1 m is composite. □

Algorithm 1 is more efficient than the one proposed by Vélu because it avoids computing the Jacobi function $(\frac{a}{m})$. It is closely related to Miller's as will be shown now but seems simpler, and the number of operations involved is slightly smaller in the worst case.

To link Algorithm 1 with Miller's, let us examine in more detail the case where $f_{(m-1)/2,m}$ is trivial. In such a case, m is not a prime power. By Lemma 1, then, for any n the image of $f_{n,m}$ is either the trivial subgroup or is not included in $\{1, -1\}$. Let $m-1=2^l \cdot m'$ with m' odd. Because m' is odd, $f_{m',m}$ cannot be trivial ($f_{m',m}(-1)=-1$); there is therefore a largest $k \leqq l$ such that $f_{2^k \cdot m',m}$ is not trivial. There is an a for which $b=f_{2^k \cdot m',m}(a) \neq \pm 1$ and $b^2=f_{2^{k+1} \cdot m',m}(a)=1$. Such an a is a witness of the fact that m is not prime and even enables the factorization of m, since

$$b^2=1\ (m) \Rightarrow (b+1)(b-1)=0\ (m) \Rightarrow (b+1, m) \neq 1 \text{ or } (b-1, m) \neq 1.$$

A probabilistic version of Algorithm 1 can be given, based on the following.

LEMMA 2. *Let G_1 and G_2 be finite groups, $f: G_1 \rightarrow G_2$ a group homomorphism and G_3 a subgroup of G_2. If the image of f is not contained in G_3, then $f(a) \notin G_3$ for at least half the elements a of G_1.*

Proof. $G_3 \cap \mathrm{Im}(f)$ is a strict subgroup of $\mathrm{Im}(f)$; therefore, at least half the elements of $\mathrm{Im}(f)$ are in $\mathrm{Im}(f)-G_3$. But if the kernel of f has size k, then every element of $\mathrm{Im}(f)$ is the image of exactly k different elements of G_1. □

ALGORITHM 2. Let m be an odd integer. Draw k random elements of $(Z/mZ)^*$: $a_1, a_2, \cdots, a_k$ for each one compute $a^{(m-1)/2}$.

If at least one of the values found is different from $+1$ and -1, then m is certainly composite. If only values of $+1$ and -1 are found and the value -1 is found at least once, then m is prime with probability greater than $1-2^{-k}$. If all the values found are $+1$, then m is composite with probability greater than $1-2^{-k}$.

Proof. Immediate from Lemma 2. □

Algorithm 2 is more efficient than the one proposed by Solovay and Strassen (it does not involve the computation of the Jacobi symbol $(\frac{a}{m})$ but though the probability of error is the same in the worst case, both conclusions "composite" and "prime" may be incorrect, whereas in Solovay and Strassen's method the conclusion "composite" is always totally reliable.

Acknowledgments. Conversations with Dennis Estes and Sidney Graham are gratefully acknowledged.

REFERENCES

[1] GARY L. MILLER, *Riemann's hypothesis and tests for primality*, Proc. 7th Annual ACM Symposium on the Theory of Computing, 1975, pp. 234–239.

[2] HUGH L. MONTGOMERY, *Topics in multiplicative number theory*, Lecture Notes in Mathematics 227 Springer, New York, 1971.

[3] MICHAEL O. RABIN, *Probabilistic algorithms*, in Algorithms and Complexity, New Directions and Recent Results, J. F. Traub, ed., Academic Press, New York, 1976.

[4] R. SOLOVAY AND V. STRASSEN, *A fast Monte-Carlo test for primality*, this Journal, 6 (1977) pp. 84–85; erratum, 7 (1978), p. 118.

[5] J. VÉLU, *Tests for primality under the Riemann hypothesis*, SIGACT News, 10, 2 (1978), pp. 58–59.

SIAM J. COMPUT.
Vol. 11, No. 2, May 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1102-0015 $1.00/0

THE COMPLEXITY OF FINDING CYCLES IN PERIODIC FUNCTIONS*

ROBERT SEDGEWICK†, THOMAS G. SZYMANSKI‡ AND ANDREW C. YAO§

Abstract. Given a function f over a finite domain D and an arbitrary starting point x, the sequence $f^0(x), f^1(x), f^2(x), \cdots$ is ultimately periodic. Such sequences are typically the output of random number generators. The *cycle problem* is to determine the first repeated element $f^n(x)$ in the sequence. Previous algorithms for this problem have required $3n+O(1)$ operations. In this paper we show that $n(1+\Theta(1/\sqrt{M}))$ steps are both necessary and sufficient, if M memory cells are available to store values of the function. We explicitly consider the performance of the algorithm as a function of the amount of memory available and the relative cost of evaluating f and comparing sequence elements for equality.

Key words. computational complexity, time-space tradeoffs, cycle detection

1. Introduction. Suppose that we are given an arbitrary function f which maps some finite domain D into D. If we take an arbitrary element x from D and generate the infinite sequence $f^0(x)$, $f^1(x)$, $f^2(x)$, $\cdots$, then we are guaranteed by the "pigeonhole" principle and the finiteness of D that the sequence becomes cyclic. That is, for some l and c we have $l+c$ distinct values $f^0(x)$, $f^1(x)$, $\cdots$, $f^{l+c-1}(x)$ but $f^{l+c}(x) = f^l(x)$. This implies, in turn, that $f^{i+c}(x) = f^i(x)$ for all $i \geq l$. The problem of finding this unique pair (l, c) will be termed the *cycle problem* for f and x. The integer c is the *cycle length* of the sequence, and l is termed the *leader length*. Similarly, the elements $f^l(x), f^{l+1}(x), \cdots, f^{l+c-1}(x)$ are said to form the *cycle* of f on x and f^0, $f^1(x)$, $\cdots$, $f^{l-1}(x)$ are said to form the *leader* of f on x. For notational convenience, the number $l+c$ of distinct values in the sequence will be denoted by n.

The cycle problem arises when analyzing pseudo-random number generators that produce successive "random" values by applying some function to the previous value in the sequence [1, §3.1]. Solving the cycle problem gives the number of distinct random numbers which can be produced from a given seed. Algorithms for the cycle problem are used in checking the characteristics of random number generators whose internal properties are unknown. Other applications include checking for loops in self-referent lists (see [2]), and studying the performance of certain numerical calculations (see [5]). Beyond these practical motivations, the problem is of some intrinsic combinatorial interest.

A graphic restatement of the problem is provided by imagining a directed graph whose nodes are the elements of D and which contains an arc from y to $f(y)$ for every $y \in D$. For example, Fig. 1a shows the graph corresponding to $f(x) = (2x+1) \bmod 10$, with $D = 0,1, \cdots ,9$. The cycle structure for a function consists of a number of

*Received by the editors November 13, 1980, and in revised form July 24, 1981. This paper was typeset at Bell Laboratories, Murray Hill, New Jersey, using the ***troff*** program running under the UNIX™ operating system. Final copy was produced on December 29, 1981.

†Department of Computer Science, Brown University, Providence, Rhode Island 02912. This work was done in part while this author was visiting the Institute for Defense Analyses, in part under support from the National Science Foundation, grant MCS-75-23738, and in part while this author was visiting the Xerox Palo Alto Research Center.

‡Bell Laboratories, Murray Hill, New Jersey 07974. This work was done in part while this author was visiting the Institute for Defense Analyses.

§Department of Electrical Engineering and Computer Science, University of California, Berkeley, California 94720. This work was done in part under support from the National Science Foundation, grant MCS-77-05313, and in part while this author was visiting the Xerox Palo Alto Research Center.

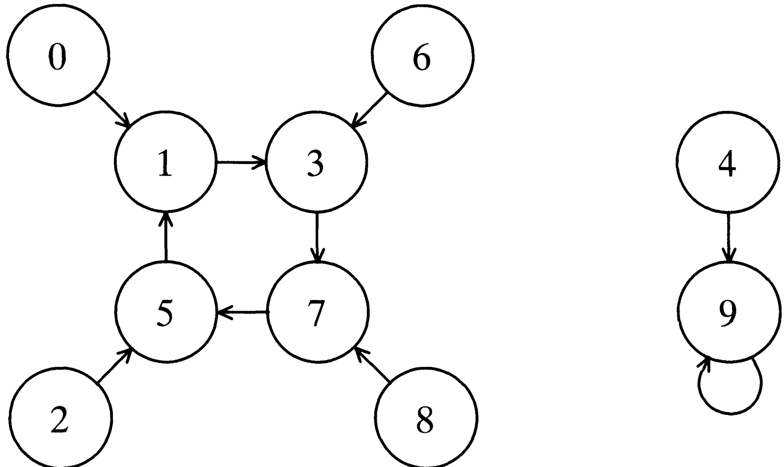


FIG. 1a. *A typical cycle structure.*

disjoint cycles, with disjoint trees feeding points on the cycles. To solve the cycle problem, we need consider only the subgraph consisting of a single cycle and leader, which can be drawn as shown in Fig. 1b.

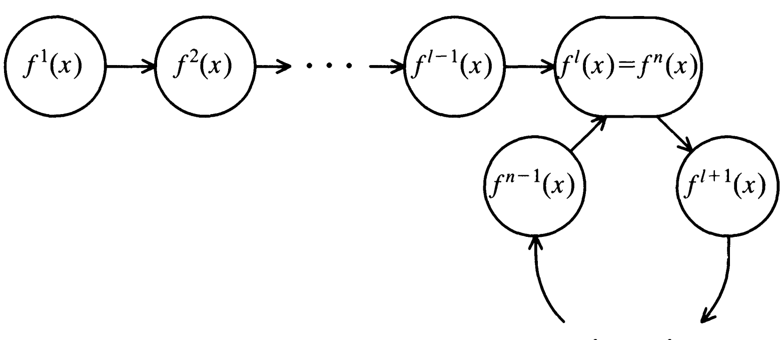


FIG. 1b. *A typical cycle and leader.*

One method for cycle detection, commonly referred to as "the tortoise and hare" algorithm, has been given by Floyd [1, Exercise 3.1-7]. The idea is to have two variables taking on the values in the sequence, one advancing twice as fast as the other. A program implementing this idea is given in Fig. 2.

```
y ← z ← x;
repeat
    y ← f(y);
    z ← f(f(z));
until y=z;
```

FIG. 2. *Floyd's algorithm.*

This algorithm stops with $y=f^i(x)=f^{2i}(x)=z$, where i is the smallest positive multiple of c which is greater than or equal to l. If $l=0$ then $3n$ function evaluations are performed, and if $l=c+1$ or if $c=1$ with $l\neq 0$ then a total of $3(n-1)$ function evaluations are performed. This number may be objectionable when the cost of evaluating f is

high relative to the cost of comparisons.

Another method, due to Gosper [2, page 64], was designed to circumvent the overhead of advancing two independently operating "copies" of the generating function as required in Floyd's method. His method is to save certain values of the sequence in a small table (whose size must be at least $\log_2 n$) and to search for each new value to see if it has previously been generated. The table update rule is to save the ith value generated in the jth cell of the table, where j is the number of trailing zeroes in the binary representation of i. This method can require as many as $l+2c$ function evaluations, or $2n$ if $l=0$, and $3n/2$ if $l=c$. Moreover, it requires at least an equal number of table searches, which would be important if the cost of comparisons were high relative to the cost of evaluating f.

These algorithms are suitable for detecting the existence of a cycle. The value of c can be found by proceeding around the cycle one additional time. Of course, this may be undesirable if c is very large. Moreover, neither algorithm has provision for directly finding l except by starting back at the initial value.

In this paper, we develop an algorithm that solves the cycle problem using $n(1+O(1/\sqrt{M}))$ function evaluations in the worst case, where M is the amount of memory available for storing generated function values. The number of memory operations (i.e., stores and searches) used is $O(n/\sqrt{M} + M\log n/M)$. The algorithm is developed in §2 in two parts: one stage which detects the cycle, and a companion stage which recovers the values of l and c. A worst case analysis of the algorithm is given in §3. In §4, we derive a lower bound which shows that no algorithm using the same fundamental operations can have a substantially better worst case performance. Our algorithm for the cycle problem thus demonstrates a tight, non-trivial tradeoff in which time is a continuous function of memory size. A generalization of the problem and some concluding remarks are offered in §5.

2. The algorithm. Any algorithm for the cycle problem must have a running time of at least nt_f where t_f is the (assumed constant) time to perform one evaluation of f. It should be clear that by using a large amount of memory we can produce an algorithm whose running time is $nt_f+O(n\log n)$ by employing, for example, a balanced tree scheme to save all elements generated in the sequence. Such an algorithm is unsatisfactory for at least two reasons. First, it is unrealistic to assume an unlimited supply of memory. Second, it does not take into consideration the relative complexity of evaluating f and comparing two domain elements for equality. Let us therefore construct a framework in which these considerations can be addressed. We shall be particularly interested in the tradeoff between memory size and execution time.

Let *TABLE* be an associative store capable of storing up to M pairs (y, i) of domain elements and integers. Both elements of the pair are *keys* in the sense that it is possible to search *TABLE* for an entry that contains a specified value for its first (or second) component. Let t_u be the time needed to insert or delete a pair from *TABLE*, i.e., to update *TABLE*, and let t_s be the time needed to search *TABLE* for a given key. Depending on the implementation of *TABLE*, t_u and t_s might be constants, logarithmic functions of M or even linear functions of M (see §5). All other operations of the algorithm are assumed to be free.

Within this model, we are ready to develop an algorithm for the cycle problem. The basic idea is to limit the number of operations performed on *TABLE* by only storing and searching for occasional values in the sequence $f^0(x), f^1(x), \cdots$. Thus, most of the time consumed by the algorithm is spent advancing the function f. In order to implement this idea, let us introduce two parameters, b and g. Fig. 3

contains an algorithm which only stores every bth function value in *TABLE* and which performs searches on blocks of b consecutive values spaced gb apart in the sequence.

```
y ← x;
i ← 0;
repeat
    if (i mod b) = 0 then insert(y, i);
    y ← f(y);
    i ← i+1;
    if (i mod gb) < b then search(y);
until found;
output i, j;
```

FIG. 3. *Preliminary version of the algorithm.*

Here the procedure *insert*(y, i) puts the pair (y, i) into *TABLE* without checking to see if there is another entry already there with the same first component. The procedure *search*(y) sets the variable *found* to false if no pair in *TABLE* has y for its first component, otherwise it sets *found* to true and j to the minimum value of j for which (y, j) is in *TABLE*. The modulus computations in this program are used for clarity; an actual implementation would use counters instead.

The program is guaranteed to halt because once the cycle is reached at least one function value out of every block of b consecutive values searched for must be in *TABLE*. Although it is possible to overshoot the point at which the cycle first returns to itself, it is clear that the algorithm will always detect the cycle before the $(n+gb)$th evaluation of f. Since the algorithm performs g updates and b searches for every gb evaluations of the function, the worst case running time of the algorithm is no greater than $(n+gb)(t_f+t_s/g+t_u/b)+bt_s$. The bt_s term is caused by the fact that the searches are not uniformly distributed within the sequence of function evaluations.

It is interesting to note that a dual algorithm can be developed by interchanging the roles of *search* and *insert* in Fig. 3, that is, every bth function value is searched for, and a block of b function values is stored every gb evaluations. Most of the results of this paper then carry through for the dual algorithm. Further development along these lines is left to the reader.

We could arrange to have the algorithm spend virtually all its time doing the (unavoidable) task of stepping f by choosing b and g suitably, were it not for the fact that *TABLE* will soon fill up. Accordingly, we introduce the following memory management mechanism. Whenever *TABLE* gets filled, we invoke a procedure *purge*(b) which removes all entries (z, j) from *TABLE* for which $j \not\equiv 0 \pmod{2b}$. We then double b, and continue. This has the same effect as restarting the program from the beginning (with the larger value of b) and running it to the current value of i. Notice that this effect is achieved at the cost of a few memory operations and, more importantly, no additional function evaluations. The algorithm thus adapts its behavior to the problem at hand. (A similar memory allocation strategy can be devised for the dual version of the algorithm mentioned above.)

The final version of the algorithm is shown in Fig. 4. The variable m is used to count the number of entries currently in *TABLE*. Notice that b is now a variable of the program, while g is still a parameter. The memory size M must be at least 2 and g can be any integer in the range $1 \le g < M$. If $g=1$ then every generated function value is looked for in *TABLE* and the algorithm will halt very soon after the nth function evaluation. Larger values of g result in fewer searches but delay the point at which a duplicate element is discovered. It will be explained later how to best choose

the value of g.

```
y ← x;
i ← 0;
m ← 0;
b ← 1;
repeat
    if (i mod b) = 0 and m = M then
        begin
            purge(b);
            b ← 2b;
            m ← ⌈m/2⌉;
        end;
    if (i mod b) = 0 then
        begin
            insert(y, i);
            m ← m+1;
        end;
    y ← f(y);
    i ← i+1;
    if (i mod gb) < b then search(y);
until found;
output i, j;
```

FIG. 4. *The cycle detecting algorithm.*

The following lemma provides the key invariant relations necessary for understanding the operation of the algorithm. The corollaries to the lemma provide useful facts needed in the analysis and correctness proof.

LEMMA 1. *The following relationships hold among the variables in the cycle detecting algorithm at the start of each* **if** *statement (and therefore at each call of search, insert, or purge):*

(a) $y = f^i(x)$,

(b) $(f^j(x), j)$ *is in TABLE if and only if* $j \equiv 0 \pmod b$ and $0 \le j < i$,

(c) *the number of entries in TABLE is* $m = \lceil i/b \rceil$.

Proof. Clearly, the relations are all true when the **repeat** loop is first entered. Moreover, it is easy to see that (a) is preserved throughout the program because the variables y and i are only changed in one place in the loop. It remains to show that (b) and (c) are preserved by the loop body.

Consider the first **if** statement in the loop. If its predicate is false, no variables are changed and the relations are preserved. If its predicate is true, then by induction we have $m = \lceil i/b \rceil = M$ with $i \equiv 0 \pmod b$. Thus $i = Mb$ and *TABLE* contains $(f^j(x), j)$ for $j \in \{0, b, 2b, \cdots, (M-1)b\}$. After the call on *purge*, *TABLE* contains only those entries with $j \in \{0, 2b, 4b, \cdots, kb\}$ where k is $M-2$ if M is even, and $M-1$ if M is odd. In either case, the number of entries remaining in *TABLE* is $(k/2)+1$, which turns out to be $\lceil M/2 \rceil$. Thus (c) is preserved by the purge and subsequent assignment to m. It should be equally obvious that the purge and subsequent doubling of b preserve (b), so we have thus established that the first **if** statement preserves the relations in the lemma.

The preservation of (b) and (c) by the rest of the loop body is straightforward. □

COROLLARY 1. *For $k \ge 0$, the $k+1$st call on purge increases b from 2^k to 2^{k+1} and occurs when $i = 2^k M$.*

COROLLARY 2. *The value of b when search is called is i/M, rounded up to the next integral power of 2, that is, $2^{\lceil \log_2 i/M \rceil}$.*

Our main goal in the study of the performance of Algorithm 4 is to prove that it halts fairly soon after the nth evaluation of f. We would expect termination to occur during the execution of the first block of consecutive searches initiated after the time when $i=n$: this turns out to be true, but the proof is complicated substantially by the possibility that purges can occur at inopportune times, thus delaying the start of the search block. The situation is quite complicated because, though g and M are fixed ahead of time, the algorithm must work correctly for *all* values of l and c. It is not obvious that the algorithm is guaranteed to terminate within a reasonable amount of time: for some choices of g and M there might be values of l and c that the search block is delayed for some time by unfortunate purges. The following lemma shows that this cannot be the case, and gives precise bounds on the time at which the algorithm must terminate.

LEMMA 2. *Let b_n be n/M rounded up to the next power of 2 (i.e., b_n is the value of b when i is assigned the value of n). Then i_f, the value of i at the termination of the cycle detecting algorithm, obeys:*

$$n \le i_f < n+(g+1)b_n \qquad \text{if } \lceil M/g \rceil \text{ is even,}$$
$$n \le i_f < n+(2g+1)b_n \qquad \text{if } \lceil M/g \rceil \text{ is odd.}$$

Moreover, these bounds on i_f are as tight as possible.

Proof. Complicated interactions between the occurrence of purge and search operations, based on arithmetic relationships between M, g, l, and c, make this proof an intricate case analysis, which is relegated to the Appendix. To see the flavor of the proof, consider the simplest case, when no purge operations occur between the first evaluation of $f^n(x)$ and the search that terminates the algorithm. In this case, the algorithm performs searches for $f^i(x)$, $i_s \le i < i_s+b_n$, where i_s is the (unique) multiple of gb_n for which $n \le i_s < n+gb_n$. During the search, since there are no purges (the precise conditions for this case are given in the Appendix), the value of of b remains fixed at b_n. Since $i_s \ge n$, the values searched for are equal, respectively, to $f^j(x)$, $i_s-c \le j < i_s+b_n-c$, exactly one of which, by Lemma 1(b), must be in *TABLE*. The algorithm therefore finds a match on one of these searches and terminates with *found* true and $i_f < i_s+b_n < n+gb_n+b_n = n+(g+1)b_n$. Notice that if $c<b_n$, the $f^j(x)$ that is found will have $j=i_s$. □

The algorithm of Fig. 4 halts as soon as it discovers a pair $i>j$ of integers for which $f^i(x)=f^j(x)$. This implies that $j \ge l$ and that $i \equiv j \pmod{c}$, but we need to do some additional processing to find the exact values of l and c. Fig. 5 shows a companion algorithm which recovers the solution (l, c) once the cycle detecting algorithm has terminated.

> **if** $f^j(x)=f^{j+c}(x)$ with $1 \le c \le (g+1)b$ **then**
> $c \leftarrow$ smallest such c;
> **else**
> $c \leftarrow i-j$;
> $i' \leftarrow \max(c, gb\lfloor i/gb \rfloor - gb)$;
> $j' \leftarrow i'-c$;
> $l \leftarrow$ smallest $l>j'$ such that $f^l(x)=f^{l+c}(x)$;
> **output** l, c;

FIG. 5. *The recovery algorithm.*

The second to last statement in this program may require evaluating f starting at a point that is not in *TABLE*. This can be done with little extra overhead. For example, $f^{j'}(x)$ can be found by doing a *search* for a *TABLE* entry whose second component is $b\lfloor j'/b \rfloor$ and then applying f exactly j' (mod b) times to $f^{b\lfloor j'/b \rfloor}(x)$.

LEMMA 3. *The recovery algorithm correctly computes c using at most* $2(g+1)b_n$ *function evaluations.*

Proof. The bound on the number of function evaluations is immediate from the observation that the final value of b is either b_n or $2b_n$.

The correctness of the computation of c has two cases depending on the predicate in the initial **if** statement in the algorithm. If the true branch is taken, then c is correctly computed by definition of cycle size. If the false branch is taken, then we must have $c>(g+1)b$. However, we know from the proof of Lemma 4 that $i<n+(g+1)b$ and hence $i<n+c$. Since $j \geq n-c$, this means that $i-j<2c$. Because $i-j$ must be a multiple of c, this implies that $i-j=c$, which is precisely what the algorithm has computed in this case. □

LEMMA 4. *The recovery algorithm correctly computes l using at most* $4(g+1)b_n$ *function evaluations and two memory searches.*

Proof. The expression $gb\lfloor i/gb \rfloor$ gives the value of i at the start of the final block of searches that the algorithm performed. Subtracting an additional term of gb from this gives the start of some previous block of searches that was completed unsuccessfully. Thus $n-gb \leq gb\lfloor i/gb \rfloor - gb < n$ and we have $\max(c, l+c-gb) \leq i' < l+c=n$. This implies that $\max(0, l-gb) \leq j' < l$. It should be clear from the definition of leader length that the algorithm correctly computes l.

The time bound follows from a more detailed consideration of the implementation of the statement that assigns l. As mentioned above, $f^{j'}(x)$ and $f^{j'+c}(x)$ can each be found by performing a memory search and $b \leq 2b_n$ function evaluations. Assigning these function values to variables and applying f to both of them until they are equal involves (from the range given above on j') at most $gb \leq 2gb_n$ function evaluations apiece. □

It is possible to design faster recovery procedures for many situations. For example, c could be found by applying "divide and conquer" to the prime decomposition of $i-j$, and l could be found by a binary search procedure. However, the recovery time is heavily dominated by the cycle detection time, so such sophisticated implementation tricks might not be worth the effort.

3. Worst case analysis. The algorithms of the previous section can provide an efficient solution to the cycle problem if the parameter g is chosen intelligently. In this section we shall analyze the running time of the algorithms to find the best choice of g. We shall concentrate on minimizing the worst case running time.

THEOREM 1. *In the worst case, the running time of the cycle detecting algorithm is at most*

$$n\left(1+\frac{4g+2}{M}\right)\left(t_f+\frac{t_s}{g}\right)+t_u M \log_2 \frac{8n}{M}.$$

The additional time required to recover the values of l and c is at most

$$n\frac{12(g+1)}{M}t_f+2t_s.$$

Proof. Corollary 2 tells us that $b_n \leq 2n/M$. Lemma 2 then implies that $i_f < n(1+(4g+2)/M)$. The detection algorithm performs i_f evaluations of f for a

contribution of $i_f t_f$ to the running time. Throughout the execution of the algorithm, b searches are performed for every gb function evaluations. Thus the total number of search operations is i_f/g, contributing $i_f t_s/g$ to the running time. This takes care of the first term. (Note that the full result of Lemma 2 implies that coefficient of g in the $O(1/M)$ term could be reduced to 2, by taking $\lceil M/g \rceil$ to be even.)

For the second term, observe that each call on *purge* removes $M/2$ elements from *TABLE*. If we charge each element with t_u time for initially inserting it, and then t_u time for its removal, we get a cost of Mt_u for each purge performed. In addition, *TABLE* can contain up to M elements at the termination of the algorithm, elements which have been inserted but not yet deleted. This contributes at most Mt_u more to the cost. Since the total number of purges performed is no greater than $1+\log_2 b_n$, we get a memory manipulation charge of $Mt_u(2+\log_2 b_n)$ which gives us the second term above.

For the running time of the recovery algorithm, Lemmas 3 and 4 give us an upper bound in terms of b_n, which by Corollary 2 is at most $2n/M$. □

COROLLARY 3. *The total running time of the cycle finding algorithm is*

$$nt_f\left(1+O\left(\left(\frac{t_s}{t_f M}\right)^{1/2}\right)\right)$$

if g is chosen appropriately.

Proof. From Theorem 1, the total running time (of both algorithms) is bounded by

$$n\left(1+\frac{16g+14}{M}\right)\left(t_f+\frac{t_s}{g}\right)+t_u M\log_2\frac{8n}{M}.$$

Choosing g to be the square root of $\dfrac{Mt_s}{16t_f}\left(1+\dfrac{14}{M}\right)$ minimizes this expression, yielding

$$n\left(t_f+8\left(\frac{t_f t_s}{M}\left(1+\frac{14}{M}\right)\right)^{1/2}+\frac{14t_f}{M}+\frac{16t_s}{M}\right)+t_u M\log_2\frac{8n}{M}.$$

Combining terms that are $O(\sqrt{1/M})$ gives the stated result. □

Note, in particular, that a balanced tree implementation will have $t_s = O(\log M)$, and a hashing method could have $t_s = O(1)$. In both cases, the worst case running time will approach nt_f as M gets large. In the next section, we shall see that the algorithms are, in fact, optimal in a much stronger sense than this.

4. Optimality. The cycle detecting algorithm given above can be thought of as a family of algorithms (parameterized by g) that trade off between the two types of basic operations, namely, function evaluations and memory searches. Thus, in the range $1 \le g < M$, if we are willing to use $O(ng/M)$ extra function evaluations in addition to the n function evaluations needed to compute $f^n(x)$, then we need only perform $O(n/g)$ table searches. The corollary to Theorem 1 shows that g can be chosen to allow the cycle problem to be solved in time $nt_f(1+O(\sqrt{t_s/Mt_f}))$. In this section, we shall show these results to be optimal in the sense that no algorithm which does successive function evaluations and has a limited amount of memory to store function values can do substantially better.

We shall consider the problem of *cycle detection*, that is, finding a pair $i \ne j$ such that $f^i(x)=f^j(x)$. The lower bound results will also, of course, apply to the more general cycle finding problem. We need to specify the model of computation to be

considered.

The model. An algorithm A uses an array $T[1], \cdots, T[M]$, each cell of which can store a pair (k, d) with $k \geq 0$ an integer, and d an element of the domain D. At all times, any pair (k, d) stored in a cell will satisfy the relation $d = f^k(x)$; initially all cells contain $(0, x)$. The algorithm can make two types of moves: An *F-move* which picks a pair (i, j) of integers (possibly equal) and sets $T[i] \leftarrow (k, f(d))$ where (k, d) is the contents of $T[j]$ when the move is executed; and an *S-move* which picks an i and tests "is there a $j \neq i$ such that $T[i]=(k, d)$ and $T[j]=(k', d')$ satisfy $k' \neq k$ and $d'=d$". The computation proceeds one move at each time $t = 1, 2, \cdots$, and halts as soon as some S-move results in a "yes" answer. The algorithm is assumed to remember the entire history of the computation (that is, the values of i and j made in all F-moves, and the value of i used in all S-moves) and the choice of the next move can depend on all of this information. Of course, values of D are "remembered" only if they are currently stored in T. The reader will note that algorithms constructed for our model of computation are *oblivious* in that any one algorithm, when run on two different problem instances, will exhibit identical behavior up until the point that one of the computations receives a "yes" response to an S-move and halts.

For any instance (f,x) of the cycle problem, let $F_{(f,x)}(A)$ be the number of F-moves performed by A when run on that instance, and let $S_{(f,x)}(A)$ be the number of S-moves. The running time is thus $F_{(f,x)}(A)t_f + S_{(f,x)}(A)t_s$. We shall use the notation $n_{(f,x)}$ to denote the sum of the leader and cycle lengths for the instance (f,x).

The following theorem gives an explicit lower bound on the tradeoff required between the number of function evaluations and table searches for any algorithm for the cycle problem.

THEOREM 2. *Let k be a positive real and n_0 a positive integer. Suppose that A is an algorithm for the cycle problem, for which $F_{(f,x)}(A) < (1+k)n_{(f,x)}$ whenever $n_{(f,x)} \geq n_0$. Then*

$$S_{(f,x)}(A) > \frac{n_{(f,x)}}{8kM(1+4k)^2}$$

for all (f,x) with $n_{(f,x)}$ sufficiently large.

Proof. Consider the algorithm A working on an input (f,x) with $n_{(f,x)} = \infty$. Let t_m be the time when $f^m(x)$ is first computed by an F-move and stored into the array. Let $s(m, m')$ denote the number of S-moves performed in the time interval $[t_m, t_{m'})$. The method of proof will be to bound $s(m, m')$ for appropriate m, m', and then sum these results over a large range of intervals to prove the theorem.

First we shall show that if $m \geq n_0$ and $m' \geq (1+k)m$ then we must have

$$s(m, m') \geq \frac{m^2}{4(M-1)m'}.$$

To prove this, suppose that A has not yet halted at time $t_{m'}$. Then the S-moves made so far must have given enough information to establish that $f^m(x) \neq f^{m-c}(x)$ for $1 \leq c \leq m$, otherwise there would exist an instance (f,x) of the cycle problem on which A would perform more than $m' \geq (1+k)m = (1+k)n_{(f,x)}$ F-moves.

We shall proceed by determining how many inequalities must be found in order to discount those cycle sizes in the range $\alpha m \leq c \leq m$ for some α to be determined later. For each such c, let $f^{i_c}(x) \neq f^{j_c}(x)$ with $j_c < i_c < m'$ be the "witness" that $f^m(x) \neq f^{m-c}(x)$. Then $i_c - j_c = h_c c$ for some integer $h_c \geq 1$. Since $i_c < m'$ and $c \geq \alpha m$, we must have $h_c \leq m'/\alpha m$. Thus each inequality $f^i(x) \neq f^j(x)$ can eliminate at most $m'/\alpha m$ cycle sizes in the range $\alpha m \leq c \leq m$. Because each S-move can supply up to $M-1$ inequalities, we have

$$s(m, m')(M-1)\frac{m'}{\alpha m} \geq (1-\alpha)m.$$

Taking $\alpha = ½$ yields the bound claimed in the statement of the theorem.

Next we shall bound the total number of S-moves made before time $t_{n_{(f,x)}}$. Since we already know that algorithm A cannot halt before time $t_{n_{(f,x)}}$, this will suffice to prove the theorem. Define $m_i = \lceil n_0(1+2k)^i \rceil$, $0 \leq i < \infty$. We may assume, without loss of generality, that $n_0 \geq 1+1/k$, and so, $1+k \leq m_{i+1}/m_i \leq 1+3k$. Thus we have

$$s(m_i, m_{i+1}) \geq \frac{m_i}{4M(1+3k)}.$$

For any $n_{(f,x)} > n_0$, define t to be the largest integer such that $m_t < n_{(f,x)}$. We thus have

$$S_{(f,x)}(A) \geq \sum_{0 \leq i < t} s(m_i, m_{i+1}) \geq \frac{1}{4M(1+3k)} \sum_{0 \leq i < t} n_0(1+2k)^i = \frac{1}{8Mk} n_0 \frac{(1+2k)^t - 1}{1+3k}.$$

If $n_{(f,x)}$ is sufficiently large, then t is large enough to guarantee that

$$\frac{(1+2k)^t - 1}{1+3k} > \frac{(1+2k)^t}{1+4k} > \frac{(1+2k)^{t+1}}{(1+4k)^2}.$$

Thus

$$S_{(f,x)}(A) > \frac{1}{8Mk} n_0 \frac{(1+2k)^{t+1}}{(1+4k)^2} \geq \frac{1}{8Mk} \frac{n_{(f,x)}}{(1+4k)^2},$$

which completes the proof. □

The reader should note that if our model of computation is altered to allow an unbounded supply of memory, and the basic operations changed to allow F-moves and simple comparisons, that is, in one step, test whether some specified pair of memory locations contain equal elements of D, then our proof techniques imply that any algorithm satisfying the hypotheses of Theorem 2 must perform at least $\frac{n_{(f,x)}}{8k(1+4k)^2}$ comparisons. It should also be noted that Theorem 2 has an alternate proof which is only valid when $k < ½$ but which improves the constant in the bound on $S_{(f,x)}$. The alternate bound is $S_{(f,x)}(A) > \left(\frac{1-k}{k(1+k)M}\right) n_{(f,x)}$.

COROLLARY 4. *If $t_s/t_f = O(M)$, then the running time for any algorithm for the cycle detecting problem is*

$$nt_f\left(1+\Omega\left(\left(\frac{t_s}{t_f M}\right)^{½}\right)\right)$$

Proof. Let k be the square root of $t_s/t_f M$. For any cycle detecting algorithm, there are two cases:

Case 1. The algorithm performs more than $n(1+k)$ F-moves for infinitely many n. Thus the algorithm has a running time of at least

$$n(1+k)t_f = nt_f\left(1+\left(\frac{t_s}{t_f M}\right)^{½}\right)$$

for infinitely many n.

Case 2. The algorithm uses no more than $n(1+k)$ F-moves for all n greater than some n_0. In this case, Theorem 2 applies, and the running time of the algorithm is at

least

$$nt_f + \frac{nt_s}{8kM(1+4k)^2} = nt_f\left(1+\frac{1}{8(1+4k)^2}\left(\frac{t_s}{t_f M}\right)^{1/2}\right)$$

for all instances of the problem with $n_{(f,x)}$ sufficiently large. Since $t_s/t_f = O(M)$, k is bounded from above as M varies, and thus $1/(1+4k)$ is bounded from below. This gives us the claimed result. □

The above results are asymptotic statements about the performance of algorithms for the cycle problem for instances (f,x) with $n_{(f,x)}$ large. Of course, it is implicit in these statements that the size of the domain D must also be large since we have the constraint that $n_{(f,x)} \leq |D|$. If $|D|$ is known to be small, an algorithm might be able to make use of that information.

The lower bounds derived in this section shed some light on the possibility of extending further the algorithms in §2. The cycle finding algorithm has a running time of

$$\left(1+c_1\frac{g}{M}\right)nt_f + c_2\frac{nt_s}{g}$$

for small positive constants c_1, c_2, but only under the constraint that $g<M$. It is interesting to inquire whether algorithms can be found which extend this range.

For example, if one is willing to use many more, say Mn extra function evaluations, can the number of searches be lowered to $O(n/M^2)$? Theorem 2 provides part of the answer, since it says that with this many extra function evaluations, one still has to perform $\Omega(n/M^2)$ table searches. We do not know of any algorithm that achieves this lower bound.

Another direction of research involves finding a non-trivial lower bound on the number of function evaluations independent of the number of searches performed. It is easy to show, for example, that any algorithm which uses memory M must perform at least

$$\left(1+\frac{1}{2M}\right)n_{(f,x)}-1$$

function evaluations in the worst case for sufficiently large $n_{(f,x)}$. This can be proved by considering the contents of memory at time t_{n-1} and choosing the cycle size so that $f^l(x)$ is at least $n/2M$ "away" from any stored element. Details are left to the reader.

5. Concluding remarks. We have dealt exclusively with algorithms with good worst case performance for solving a particular instance of the cycle problem on an unknown function. The problem is also interesting under other variations of the model.

One variation is to take the function to be *random* (in some sense) and to talk about an average case measure of complexity. R. W. Floyd has pointed out that studying the probabilistic structure of random functions over D can lead to savings on the average. For example, l is known to be relatively large in the case of a random function, so it may not be worthwhile to save or search for values at the beginning.

Anotner variation is to let the cost of computing $f^j(x)$ be independent of j. A famous factoring algorithm due to Shanks [4] is based on this problem. Shanks' solution, which uses the additional knowledge that $l=0$ and n is bounded by some constant N, finds n in time proportional to $\sqrt{n}$ using $\sqrt{n}$ memory. The method is similar to the dual algorithm mentioned in §2: one saves the first $\sqrt{n}$ values generated,

then does table searches at subsequent intervals of $\sqrt{n}$. Our cycle detecting algorithm also works in this case, but it runs slightly longer due to its need to discover that $l=0$ and its need to adapt to the value of c. A rough analysis for this case follows. From the program, we see that the function evaluation cost will be about the same as the table update cost, so the expression for the total running time in Theorem 1 tells us that the best thing to do is to pick g as large as possible (about M), for a running time of

$$O\left(\frac{nt_s}{M}+(t_u+t_f)M\log_2\frac{n}{M}\right).$$

This expression is minimized to $O(\sqrt{n\log_2 N})$ by choosing M to be $O(\sqrt{n/\log_2 N})$ if enough memory is available and we know that n is bounded by N.

A generalization of the cycle problem arises when all the points of the domain D are to be studied. In general, D is partitioned by f into disjoint sets with the property that all points in each set lead to the same cycle. Properties of the *cycle structure* (e.g., the number of sets, their sizes, the sizes of their cycles) can be found by solving the cycle problem on all points of D. The algorithms of this paper can be adapted to avoid retraversing long cycles by maintaining versions of *TABLE* for each cycle.

Another generalization of the cycle problem can be formulated in the following way. As before we are given a unary function f, only now we allow the domain of f to be infinite. We suppose that we are also given a binary predicate P on $D\times D$. The problem is to find the smallest n for which there exists an $l<n$ such that $P(f^n(x), f^l(x))$. In the absence of any further information, it is easy to show that this problem requires $\binom{n}{2}$ evaluations of P. However, if P is *preserved* by f, that is, $P(a, b)$ implies $P(f(a), f(b))$, then the algorithms of this paper can be made to run in time $n(t_f+O(t_p))$ where t_p is the time needed to evaluate P. It is interesting to note that the algorithms of [1] and [2] simply do not work for this problem.

An earlier version of this paper [3] left open the question of whether an algorithm exists for the cycle problem which used a bounded amount of memory and an optimal number of function evaluations. We have resolved this question in §4, with the somewhat surprising result that the algorithm of §2 may be viewed as optimal for the range of problem parameters for which it is applicable.

Appendix. In this appendix, we give a detailed proof of the complete result about the termination time of the cycle finding algorithm, Lemma 2 from §2 of the paper. The main purpose of including this proof in detail is that it precisely illustrates why the result given is the most general available for the problem: in fact, each of the cases below was essentially discovered as a counterexample during the search for a simpler or better version of Lemma 2.

A key fact needed to establish the correctness of the algorithm is that at least one complete block of searches on b consecutive values of the function is performed between any two consecutive calls on *purge*. Let the consecutive purges take place at $i=2^{k-1}M$ and 2^kM. During this time interval $b=2^k$. The following fact implies that at least one block of searches will be started early enough in this interval to be completed before the latter purge. More specifically, the search block will start no later than 2^kM-2^k.

FACT 1. *Let M and g be integers with $1\le g<M$ and $M\ge 2$. For any integer $k>0$ there exists an integer i, $2^{k-1}M\le i\le 2^kM-2^k$, such that $i\equiv 0 \pmod{2^kg}$.*

Proof. If $\lceil M/2\rceil\le g<M$, then 2^kg clearly has the required properties. If $1\le g\le\lfloor M/2\rfloor$, simply count the number of multiples of 2^k in the specified interval. If

M is even, there are precisely $M/2$ such multiples, whereas if M is odd, there are $(M-1)/2$ of them. In either case, the interval contains $\lfloor M/2 \rfloor$ consecutive multiples of 2^k so one of them must be congruent to $2^k g$. □

The next fact is a technical result which will be useful in determining the amount by which the algorithm can overshoot n.

FACT 2. *Let M, g, and x be positive integers. Then $\lceil M/g \rceil$ is even if and only if the smallest multiple of gx that is at least Mx is an even multiple of gx.*

Proof. (If) By hypothesis, there exists an even integer z such that $(z-1)gx < Mx \leq zgx$. But then, $z-1 < M/g \leq z$ and so $\lceil M/g \rceil \leq z$. Thus $\lceil M/g \rceil$ is even.

(Only if) By hypothesis, there exists an even integer z such that $z = \lceil M/g \rceil$. Then $z-1 < M/g \leq z$ and so $(z-1)gx < Mx \leq zgx$. Since $(z-1)gx$ and zgx are consecutive multiples of gx, zgx must be the smallest multiple of gx that is at least Mx. Since z is even, zgx is an even multiple of gx. □

At this point we are ready to prove the result of Lemma 2 from the text, which states precisely when the algorithm halts. The key idea is that termination is guaranteed to occur during the execution of the first block of consecutive searches initiated after the time when $i=n$. As mentioned in the text, the proof is complicated substantially by the necessity to account for the effects of purges which could delay the start of the last search block. Fact 1 will be used to show that the delay cannot be indefinite and Fact 2 will be used to establish a bound on the amount of the delay.

LEMMA 2. *Let b_n be n/M rounded up to the next power of 2 (i.e., b_n is the value of b when i is assigned the value of n). Then i_f, the value of i at the termination of the cycle detecting algorithm, obeys:*

$$n \leq i_f < n+(g+1)b_n \qquad \text{if } \lceil M/g \rceil \text{ is even,}$$

$$n \leq i_f < n+(2g+1)b_n \qquad \text{if } \lceil M/g \rceil \text{ is odd.}$$

Moreover, these bounds on i_f are as tight as possible.

Proof. Let $i_p = b_n M$, and let i_s be the (unique) multiple of gb_n for which $n \leq i_s < n+gb_n$. By Corollary 1, $i_p \geq n$. Thus i_p is the first moment after $i=n$ at which a purge operation can occur, and i_s is the first moment after $i=n$ at which a new block of search operations can commence. A number of cases now arise.

Case 1. $i_s < i_p$. Since i_s and i_p are both multiples of b_n, $i_s + b_n \leq i_p$ and the algorithm performs searchs for $f^i(x)$, $i_s \leq i < i_s + b_n$, during which time the value of b remains fixed at b_n. Since $i_s \geq n$, the values searched for are equal, respectively, to $f^j(x)$, $i_s - c \leq j < i_s + b_n - c$, exactly one of which, by Lemma 1(b), must be in *TABLE*. The algorithm therefore immediately terminates after one of these searches with $i_f < i_s + b_n < n + gb_n + b_n = n+(g+1)b_n$.

Case 2. $i_p \leq i_s$ and $\lceil M/g \rceil$ is even. By definition, i_s is the smallest multiple of gb_n that is at least as great as n. Since the condition of this case requires that $n \leq i_p \leq i_s$, i_s is the smallest multiple of gb_n that is at least $i_p = Mb_n$. Fact 2 thus guarantees that i_s is an even multiple of gb_n and hence $i_s \equiv 0 \pmod{2gb_n}$. Fact 1 guarantees that no additional purges take place between the times when $i=i_p$ and $i=i_s$. Thus the algorithm performs search operations for $f^i(x)$, $i_s \leq i < i_s + 2b_n$, during which time $b=2b_n$. Since $i_s \geq n$, at least one of these function values is in *TABLE* and the algorithm halts with $i_f < i_s + 2b_n$. Thus $n \leq i_f < n+(g+2)b_n$.

We shall conclude this case by demonstrating by contradiction that we must actually have $i_f < n+(g+1)b_n$. To do this, suppose that $i_f \geq n+(g+1)b_n$. Consider $i_{ps} = i_s - gb_n$ and $i_{pf} = i_f - gb_n - b_n$. i_{ps} is the point where the previous search block began. We shall show that a previous find would have occurred at i_{pf}, terminating the

algorithm and giving us the desired contradiction. Observe that $i_{ps}<n$ because $i_s<n+gb_n$. Moreover, $n\le i_{pf}$ because $n+(g+1)b_n\le i_f$. Notice that $i_{pf}<i_{ps}+b_n$ because $i_f<i_s+2b_n$. Finally, observe that $i_{ps}+b_n\le i_p$ because both i_{ps} and i_p are multiples of b_n with $i_{ps}<n\le i_p$. Putting these together we have $i_{ps} < n \le i_{pf} < i_{ps}+b_n \le i_p$. Since $i_{ps}\equiv 0 \pmod{gb_n}$, this implies that the algorithm performs searches for $f^i(x)$, $i_{ps}\le i<i_{ps}+b_n$ during which time b remains constant at b_n. By definition of i_f, $i_f-c\equiv 0 \pmod{2b_n}$, and so $i_{pf}-c\equiv 0 \pmod{b_n}$. Thus the search for $f^{i_{pf}}(x)$ should have caused the algorithm to terminate with $i=i_{pf}$.

Case 3. $i_p\le i_s$ and $\lceil M/g\rceil$ is odd. As in case 2 above, i_s is the smallest multiple of gb_n that is at least i_p. This time, however, Fact 2 tells us that i_s is an odd multiple of gb_n and hence $i_s\not\equiv 0 \pmod{2gb_n}$. The first block of searches starting after $i=n$ must therefore begin at i_s+gb_n. Since b at this time is $2b_n$, we see that $i_f < i_s+gb_n+2b_n < n+(2g+2)b_n$.

As in case 2, we can next argue that the last b_n values in this range are not really possible. To do this, suppose that $i_f\ge n+(2g+1)b_n$. Consider $i_{ps}=i_s-gb_n$ and $i_{pf}=i_f-2gb_n-b_n$. Once again, algebra reveals that $i_{ps} < n \le i_{pf} < i_{ps}+b_n \le i_p$ and we can argue that the algorithm would have terminated earlier.

To see that the stated bounds are the tightest possible, let us suppose that M and g are given, with $1\le g<M$. We shall show how to construct an infinite set of instances of the cycle finding problem in which i_f is at the extreme high end of the ranges given in the statement of the lemma.

For any integer $k\ge 0$, let i_p be 2^kM, and let i_{s_1} be the largest multiple of 2^kg that is less than i_p. Let n be $i_{s_1}+2^k$. Since both n and i_p are multiples of 2^k, $n\le i_p$ and $b_n=2^k$.

If $\lceil M/g\rceil$ is even, let i_{s_2} be $i_{s_1}+2^kg = n+gb_n-b_n$ and let l be $1+gb_n-b_n$. If $\lceil M/g\rceil$ is odd, let i_{s_2} be $i_{s_1}+2^{k+1}g = n+2gb_n-b_n$ and let l be $1+b_n$. These choices for l are possible because n is at least gb_n+b_n. It can be shown through the use of Fact 2 that, in either case, i_{s_2} is the smallest multiple of $2^{k+1}g$ that is at least i_p. Now consider the operation of the algorithm on an instance of the cycle finding problem whose solution is given by the n and l defined above. The algorithm will perform searches for $f^i(x)$, $i_{s_1}\le i<i_{s_1}+b_n=n$. These searches all fail. A purge then occurs at i_p increasing b to $2b_n=2^{k+1}$. The next block of searches is performed for $f^i(x)$, $i_{s_2}\le i<i_{s_2}+2b_n$. Since the cycle size $c=n-l$ is sufficiently large, the algorithm will terminate for the first i with $i_{s_2}\le i<i_{s_2}+2b_n$ and for which $i-c\equiv 0 \pmod{2b_n}$. Let us write this i_f as $i_{s_2}+j$ with $0\le j<2b_n$.

If $\lceil M/g\rceil$ is even, then $i_f-c = i_{s_2}+j-c = n+gb_n-b_n+j-c = l+gb_n-b_n+j = 1+2gb_n-2b_n+j$. Thus $i_f-c\equiv 0 \pmod{2b_n}$ when $j=2b_n-1$ and the search halts at $i_f = i_{s_2}+2b_n-1 = n+(g+1)b_n-1$. If $\lceil M/g\rceil$ is odd, then $i_f-c = i_{s_2}+j-c = n+2gb_n-b_n+j-c = l+2gb_n-b_n+j = 1+2gb_n+j$. Thus $i_f-c\equiv 0 \pmod{2b_n}$ when $j=2b_n-1$ and the search halts at $i_f = i_{s_2}+2b_n-1 = n+(2g+1)b_n-1$. In either case, i_f is the largest value permitted in the ranges given in the statement of the lemma. □

Acknowledgments. The authors take pleasure in thanking A. V. Aho, W. Beckman, and M. D. McIlroy for their helpful comments on various drafts of this paper.

Postscript. F. E. Fich has recently shown [5] that the cycle problem is also interesting to study under a complexity measure that counts only the number of function evaluations. Memory references are not counted explicitly in the cost, but

algorithms must respect the limitation of using only a fixed amount of memory. She proves a lower bound of $n(1+1/(M-1))$ function evaluations for algorithms restricted to M memory cells, while the algorithm in this paper uses $n(1+2/(M-1))$ function evaluations. Also, she gives upper and lower bounds for $M=2$ and various other restrictions.

REFERENCES

[1] D. E. KNUTH, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms,* Addison-Wesley, New York, 1969.

[2] R. GOSPER ET AL., *HACKMEM,* M.I.T. Artificial Intelligence Lab Report No. 239, 1971.

[3] R. SEDGEWICK AND T. G. SZYMANSKI, *The complexity of finding periods,* Proc. 11th Annual ACM Symp. on the Theory of Computing, (April 1979), pp. 74-80.

[4] D. SHANKS, *Class number, a theory of factorization, and genera,* Proceedings of Symposia in Pure Mathematics, American Mathematical Society, Providence, Rhode Island, 1970.

[5] F. E. FICH, *Lower bounds for the cycle detection problem,* Proc. 13th Annual ACM Symposium on the Theory of Computing, (May 1981), pp. 96-105.

SIAM J. COMPUT.
Vol. 11, No. 2, May 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1102-0016 $1.00/0

FREEDOM FROM DEADLOCK OF SAFE LOCKING POLICIES*

MIHALIS YANNAKAKIS†

Abstract. The usual method for preserving the consistency of a database when accessed (read and updated) concurrently by several transactions, is by locking the transactions according to some locking policy; a locking policy that guarantees the preservation of consistency of the database is called *safe*. Furthermore, if no deadlocks can arise the policy is called *deadlock-free*. In this paper we are concerned with the freedom from deadlock of safe locking policies. We show that a simple extension of the DAG policy of [Y] is the most general safe and deadlock-free policy for a pair of transactions. We prove however, that it is NP-complete to test whether a set of transactions is not deadlock-free even for the simplest kind of transactions, those that are two-phase locked [E]. We show that for the natural class of safe locking policies, the L-policies, studied in [Y], freedom from deadlock is determined only by the order in which entities are accessed by the transactions and not by the way in which safety is ensured. As a consequence of this fact we develop simple conditions that guarantee the freedom from deadlock of a safe L-policy.

Key words. database, concurrency control, transaction, locking policy, serializability, safety, deadlock

1. Introduction. A database consists of *entities* (e.g., files, relations, records etc.) which satisfy certain *consistency constraints*. Many times, when a user accesses the data base, he may have to violate temporarily these constraints in order to transform it to a new consistent state. For this reason, atomic actions are grouped together into units of consistency, called *transactions*. When several transactions access (read and update) the same database concurrently, there must be some kind of coordination of the various actions to assure that the resulting sequence of actions (or *schedule)* is *correct;* that is, every user must receive a consistent view of the data, and the schedule must leave the database in a consistent state at the end. Such coordination is usually achieved through some locking mechanism [E], [G1], [G2], [LW], [U]. Each transaction locks entities according to some *locking policy* in such a way that when the transaction runs concurrently with any other possible set of transactions that follow the same locking policy, any schedule that might result (if the locking protocol is observed) is guaranteed to be correct. Such a locking policy is called *safe*.

The simplest safe policy is the *two-phase locking policy* (2PL) proposed by Eswaran et al. in [E]. In 2PL a transaction must lock (in any order) every entity it needs before it accesses it. However, after some entity is unlocked, the transaction is not allowed to lock any more items. In [SK] another safe policy, the *tree policy* (TP), and then in [Y] a generalization of it, the *DAG policy* (DP), were proposed. In TP the entities are arranged in a rooted (directed) tree. Any entity can be the first one locked by a transaction T. Subsequently, T may lock an entity A only if its father $f(A)$ is locked, and A has not been locked previously again. In DP the entities are arranged in a single source DAG. A transaction can start by locking any entity; subsequently an entity A can be locked only if all its fathers (immediate predecessors), but not A, have been locked (and possibly unlocked), and the transaction holds a lock on at least one father of A. The basic idea in TP and DP, of locking an entity only if a father of it is currently locked, was first used in papers dealing with the concurrent

*Received by the editors March 25, 1980, and in revised form March 23, 1981. This paper was typeset at Bell Laboratories, Murray Hill, New Jersey, using the ***troff*** program running under the UNIX™ operating system. Final copy was produced on January 6, 1982.

†Bell Laboratories, Murray Hill, NJ 07974.

manipulation of B-trees [S], [BS]. Note that TP and DP are families of policies (one for each underlying tree and DAG). A policy operating on a set Δ of structures (such as trees, DAGs, etc.) is called a *structured policy*. In practice, such structures may model either a physical (e.g., tree or DAG of pointers) or a logical (e.g., flow of consistency constraints) organization of the entities.

A locking policy that does not give rise to any deadlocks is called *deadlock-free*. For example, deadlocks may arise when the transactions are locked according to 2PL, whereas TP is a deadlock-free policy. We should note here that in database systems freedom from deadlock of a locking policy is important only in conjunction with its safety: Transactions are locked according to some policy in order to preserve the consistency of the database; we would like to make the policy deadlock-free, but not at the expense of its safety.

In § 2 we describe our model and define our terminology formally. In [Y], [YPK] a natural class of locking policies, the *L-policies*, was defined; these are the policies that can be stated in terms of a set of conditions that tell us at any given moment in a transaction whether or not a given entity can be locked, depending on the portion of the transaction executed up to this moment. A simple, most general safe L-policy (called the *Hypergraph policy* HP) was found there; simple in the sense that its rules can be efficiently (in the size of the structure) enforced, and most general in the sense that every safe L-policy can be viewed as an instance of it for an appropriate choice of the underlying structure. Our purpose in this paper is to see whether similar results can be derived also for safe *and* deadlock-free L-policies. In § 3 we show that the DAG policy is deadlock-free. We then extend the policy to DAGs with more than one source and show that the extended DAG policy is the most general safe and deadlock-free policy for a pair of transactions. Unfortunately, in § 4 we show that it is NP-complete to test whether a set of transactions is not deadlock-free, even for the simplest kind of transactions, those that are two-phase locked. This implies in particular, that probably there is no simple, most general safe and deadlock-free L-policy. In § 5 we show that L-policies have a characteristic property with respect to deadlocks: the freedom from deadlock of safe L-policies is determined only by the order in which the entities are accessed by the various transactions and not on how safety is enforced (where the entities are unlocked). In § 6 we use this fact to develop simple conditions that guarantee the freedom from deadlock of safe L-policies. We also prove that if a safe (general) policy achieves freedom from deadlock in a way similar to that of TP and DP then this policy can be extended to a safe and deadlock-free L-policy.

2. Definitions and preliminaries. A database is a finite set E of *entities*. A *state* of the database is an assignment of values to the entities. The consistency constraints of the database define a set CS of *consistent states*. A *transaction T* is a finite sequence $<a_i>$ of *actions*. Each action a_i is associated with an entity $x \in E$, and as in [E], [KP] is considered to be the indivisible execution of the instructions: $t_i \leftarrow x$ [read x]; $x \leftarrow f_i(t_1,\ldots,t_i)$ [update x], where the t_i's are local variables of T and f_i is an uninterpreted function symbol.[1] A *transaction system* is a finite set τ of transactions. A

[1]A simple variant of this model is defined in [U, § 10.2]. An action a_i on entity x is considered there to be the execution of the instruction $x \leftarrow f_i(x)$; i.e., the new value of x depends only on the old value of x and not on the other entities previously read by the transaction. The two models are equivalent as far as correctness of schedules is concerned.

schedule S of τ is an ordering of the actions of all the transactions of τ, which preserves the ordering of the actions of each transaction. A *serial schedule* is one in which there is no interleaving; i.e., no action of a transaction T occurs between two actions of some other transaction T'. A sequence of actions is *correct* if execution of it will map any consistent state to a consistent state. We assume that each transaction is correct. Then every serial schedule is also correct.

Most of the literature in the area has dealt with issues of correctness in the presence of purely syntactic information: Each transaction is given as a sequence of entities acted upon in the corresponding actions (i.e., the f_i's are uninterpreted and distinct for every action of each transaction[2]), and our knowledge of CS, the set of consistent states, is limited to the fact that each transaction when run by itself maps any consistent state to a consistent state [E], [P1], [P2], [St]. In this framework, saying that a schedule S of a transaction system $\tau = \{T_1,\ldots,T_m\}$ is correct, means that for every choice of the set CS, and for every interpretation of the function symbols of the T_i's such that execution of each T_i maps CS into itself, execution of S maps also CS into itself. In this paper we will be talking about correctness in this sense. Kung and Papadimitriou show in [KP] that in this model correctness is equivalent to *serializability* [P1], [P2], [St]; i.e., there is a serial schedule S' such that if S and S' start from the same initial state, they will leave the database in the same final state.

Correctness of a schedule S can be easily decided in our model [E] as follows: Construct a labelled directed graph $D(S)$ with a node for each transaction, and an arc (T_i,T_j) labelled x if transaction T_i acts on x in schedule S before T_j does. It is shown in [E] that S is serializable if and only if the digraph $D(S)$ is acyclic.

A *locked transaction* T is a finite sequence of *steps*. Each step is either an action on an entity x (written $a.x$), or an instruction Lock x (written Lx) or Unlock x (Ux), where $x \in E$. We assume that a locked transaction does not lock again an entity that is already locked, nor unlock an entity which is not locked, and that it (eventually) unlocks every entity that it locks. We say that T has locked x through step i, if for some $j<i$ the jth step of T is Lx, and there is no k with $j<k<i$ such that the kth step of T is Ux. A locked transaction T is *well-formed* [E] if whenever the ith step of T is $a.x$ then x is locked through step i; i.e., an action can take place only if the corresponding entity is currently locked. We will be dealing throughout this paper with well-formed locked transactions, without mentioning it again explicitly.

A *legal schedule* of a set τ of locked transactions (or locked transaction system) is a schedule that respects the locks of the transactions of τ; i.e., a transaction cannot lock an entity that is already locked by some other transaction. A locked transaction system τ is *safe* if any legal schedule S of it is correct. A *partial schedule* S of a transaction system $\tau = \{T_1,\ldots,T_m\}$ is a legal schedule of any prefixes of the transactions of τ. The *state* $J(S)$ of a partial schedule S is the vector $<j_1,\ldots,j_m>$ that describes the next step to be executed for each transaction of τ. An entity x is *locked at state* J if x is locked by some T_i through step j_i. The state J is a *deadlock state* if for all i the j_ith step of every unfinished transaction T_i is Lx_i for some entity x_i locked at J. A transaction system τ is *deadlock-free* if the state $J(S)$ of any partial schedule S of τ is not a deadlock state. In other words any partial schedule of τ can be extended to (is a prefix of) a (complete) schedule of τ.

[2]A transaction system can have two transactions with the same syntax (the same sequence of entities acted upon). The two transactions are considered distinct, in that the functions computed by them are unrelated.

A *locking policy* P is a mapping from the set of transactions on E to the power set of (well-formed) locked transactions on E, which satisfies the property: if $\bar{T} \in P(T)$ then T and $\bar{T}$ contain exactly the same actions in the same order. The locking policy P is safe (resp. deadlock-free) if for any transaction system $\tau = \{T_1,\ldots,T_m\}$ with $P(T_i) \neq \varnothing$, for all i, any set $\bar{\tau} = \{\bar{T}_1,\ldots,\bar{T}_m\}$ of locked transactions with $\bar{T}_i \in P(T_i)$ is safe (resp. deadlock-free). If Δ is a class of structures on E (e.g., relations, graphs, etc.) a *structured policy* ΔP operating on Δ is a family of locking policies, one for each structure $D \in \Delta$.

As in [Y], we will assume that if $\bar{T} \in P(T)$, then there is another locked transaction $\bar{T}' \in P(T')$ for some T', such that $\bar{T}'$ has the same sequence of Lock and Unlock steps as $\bar{T}$, and every $Lx - Ux$ pair of steps of $\bar{T}'$ contains an action on x. This is equivalent to saying that the locking policy has the following rule : *a transaction can act on an entity while it holds a lock on it.* All policies that we know of have in fact such a rule. We make this assumption in order to exclude policies that use for locking any set of special variables — not related to the set of entities (see [Y] for more discussion on this subject). Note, however, that our complexity and sufficiency results carry over to the general setting.

A transaction may lock an entity x more than once. If the policy is to be safe, then it better be the case that no transaction can make actual use of the fact that x is unlocked for some period of time between the two intervals. In other words, if we replace the two Locks by a single long Lock, this will not affect the possible legal schedules permitted by the policy; the only thing that can happen is that some deadlock situations may be eliminated. Therefore, there is no point for a transaction of the policy to lock an entity more than once. Thus, from now on we will assume that every transaction locks every entity at most once.

Let P be a locking policy that satisfies the previous assumptions, and let $\tau(P)$ be the set of locked transactions in the image set of P that act on all the entities that they lock; i.e., $\tau(P) = \{\bar{T} | \bar{T} \in P(T)$ for some T, and every $Lx - Ux$ pair of steps of $\bar{T}$ contains an $a.x$ step$\}$. From the previous assumptions and the definitions it follows that P is safe (or deadlock-free) if and only if every transaction system τ all of whose transactions are in $\tau(P)$ is safe (or deadlock-free)[3]. Thus, it suffices to restrict our attention to sets of locked transactions which act on all the entities that they lock. It is not hard to see ([Y]) that if τ is a locked transaction system, the safety or freedom from deadlock of τ is not influenced by the exact position of the actions in its transactions; that is, if we move in some transactions of τ their actions (while keeping of course the transactions well-formed) the resulting transaction system τ' is safe (or deadlock-free) if and only if τ is. For this reason, we can ignore the action steps of the transactions and see a transaction simply as a sequence of Lock and Unlock steps. From now on we will use the term *transaction* in this sense and the term *transaction system* to refer to a set of such transactions. In fact, in most of the literature on locking policies a transaction is modelled in this way (as a sequence of Lock and Unlock steps). We went through the previous discussion in order to justify this modelling and make explicit the underlying assumptions.

The safety (or freedom from deadlock) of a locking policy P is completely determined by its image set $\tau(P)$. We are going usually to identify P with this set of

---

[3]Strictly speaking, τ may not be a subset of $\tau(P)$, since it can contain more than one copy of a transaction in $\tau(P)$.

transactions $\tau(P)$, and use the term *policy* τ to refer to a policy P with $\tau = \tau(P)$. We should note, however, that there is a slight difference between the safety of P and that of the transaction system $\tau(P)$: the safety of P requires the safety also of systems of transactions from $\tau(P)$ that contain possibly more than one transaction with the same syntax (copies of a transaction in $\tau(P)$ with unrelated semantics). If during a transaction T in $\tau(P)$ there is an (intermediate) point at which no entity is locked, then P cannot be safe: Just consider a copy T' of T and a schedule of $\{T,T'\}$ in which T' executes at the *unlocked point* of T. However, if no transaction of $\tau(P)$ has such an unlocked point, then P is safe (or deadlock-free) if and only if $\tau(P)$ is safe (or deadlock-free); a proof of this fact is given in § 3. [4]

If τ is a transaction system and S a partial schedule of it, we can construct from S, as with a complete schedule, a directed graph $D(S)$ with a node for each transaction of τ and an arc (T_i,T_j) labelled x, if T_i locks x in S before T_j does (even if the Lx step of T_j has not been executed yet in S). Then τ is safe and deadlock-free if and only if for every partial schedule S of τ, the digraph $D(S)$ is acyclic.

We will denote by $R(T)$ the set of entities mentioned in a transaction T, and by L_T (step i) (resp. R_T (step i)) the set of entities locked through step i (resp. mentioned up to the ith step) of T.

In [Y], [YPK] we defined a class of natural locking policies, the L-policies, as follows. A locking policy P is an *L-policy* if P can be described by a set of conditions that state whether a given entity can be locked at a certain moment in a transaction, depending on the portion of the transaction executed up to this moment. In other words, with each entity x there is associated a set $W(x)$ of prefixes of transactions; a transaction T is in P if and only if for each entity x referenced by T, the prefix of T to the left of Lx belongs to $W(x)$. For example, the two-phase locking policy has for every $x \in E$, $W(x) = \{\overline{T} | \overline{T}$ does not contain any unlock steps$\}$. A *truncation* of a transaction T at the jth step is a transaction T' that agrees with T in the first j steps, and then unlocks (in any order) the entities locked by T through the $(j+1)$th step. The *closure under truncation* $Ct(\tau)$ of a transaction system τ is $Ct(\tau) = \tau \cup [\cup_{T \in \tau} Tr(T)]$, where $Tr(T)$ is the set of truncations of a transaction T. A system τ is *closed under truncation* if $Ct(\tau) = \tau$. Thus, a policy P is an L-policy if (when viewed as a transaction system) it is closed under truncation; the closure under truncation of a transaction system τ is the smallest L-policy containing τ. A *hypergraph* $H = (N,F)$ has a set of nodes N and a set of hyperedges F. Each hyperedge is a subset of N. With every transaction system τ we associate a hypergraph $H(\tau)$, which has one node for each entity and a hyperedge $R(T)$ for each transaction T of τ. It was shown in [Y] that a policy τ, which is closed under truncation, is safe if and only if for every $T \in \tau$, and x, y in $R(T)$ such that Ux occurs in T before Ly, the set $L_T(Ly)$ (or equivalently $L_T(Ux) - \{x\}$) separates x from y in $H(\tau)$. As a consequence of this result, there is a simple safe structured policy HP which covers all safe L-policies; i.e., every safe L-policy (for example 2PL, the tree policy, the DAG policy) is an instance of HP for an appropriate choice of the underlying structure. This policy HP, called the Hypergraph policy, operates on directed hypergraphs. A *directed hypergraph* $DH = (N,F)$ is a hypergraph, each hyperedge A of which has a node specified as its *head*. The rest of the nodes of A form its *tail*. The underlying hypergraph of DH is simply $H = (N,F)$

[4] An alternative way would be to consider P as the transaction system $\tau_2(P)$ that contains two copies of each transaction $\tau(P)$. Then P is safe (or deadlock-free) if and only if $\tau_2(P)$ is.

without head-tail specification. Terms such as "paths," "cycles," "separators," etc. in a directed hypergraph are used with respect to the underlying hypergraph.

Hypergraph policy (HP): The entities are arranged in a directed hypergraph H. The rules are:

(1) First lock is arbitrary.

Subsequently, an entity x can be locked if and only if

(2) There is a hyperedge A of H with head x, whose tail has been mentioned in the transaction up to this point, and

(3) For each y previously unlocked, the set of entities that are currently locked separate x from y.

3. Freedom from deadlock of the DAG policy. We repeat here the definition of the DAG policy.

DAG policy (DP): The entities are arranged on (correspond to the nodes of) a single source directed acyclic graph (DAG) D. The rules of the policy are as follows:

(1) First lock is arbitrary.

(2) Subsequently, an entity A can be locked only if all its fathers (immediate predecessors) have been mentioned in the transaction up to this point, and at least one father is currently locked.

Let us consider a typical case of deadlock for a set τ of transactions. Deadlock arises in a partial schedule S of τ when every transaction wants in the next step to lock an entity that is already locked by some other transaction. This means that there is a set of transactions $\{T_1,\ldots,T_k\}$ such that the next step of T_i is Lx_i where x_i is currently locked by T_{i+1} ($x_i \in L_{T_{i+1}}(Lx_{i+1})$) — see Fig. 1.

T_1:	. . .	Lx_k	. .	$\mid Lx_1$	. . Ux_k	. .
T_2:	. . .	Lx_1	. .	$\mid Lx_2$	. . Ux_1	. .
.				$\mid$		
.				$\mid$		
.				$\mid$		
T_k:	. . .	Lx_{k-1}	. .	$\mid Lx_k$	. . Ux_{k-1}	. .
				J		

FIG. 1

Thus in the partial schedule S, transaction T_i accesses x_{i-1} before T_{i-1}; if S could possibly finish in any way then the resulting schedule would not be correct. In other words, deadlocks prevent some wrong schedules from finishing. Let us show that the DAG policy is deadlock-free using this fact.

THEOREM 1. *The* DAG *policy is deadlock-free.*

Proof. Suppose S is a partial schedule, of $T_1,\ldots,T_k$ deadlocked at state J where the next step of each T_i is Lx_i as in Fig. 1. We can assume without loss of generality that Lx_i is the last locking step of T_i. Suppose that for some i, the transaction T_i' obtained from T_i by moving the Ux_{i-1} step right before the Lx_i step is also a transaction of the DAG policy. Then the partial schedule S could be extended to a nonserializable schedule of the system $[\cup_{j\neq i}\{T_j\}] \cup \{T_i'\}$; T_i' can execute the Ux_{i-1} step and then $T_{i-1},T_{i-2},\ldots,T_1,T_k,\ldots,T_i'$ can finish in this order. But this fact contradicts the safety of the DAG policy. Therefore, for each i, x_{i-1} must be a father of x_i, which

implies that the DAG has a cycle $\{x_1,\ldots,x_k\}$. □

If we modified the DAG policy by changing the locking rule (2) into: (2′) an entity x can be locked if at least one father of x is currently locked, and all fathers of x are mentioned until the Ux step, then the modified policy is safe — the proof given in [Y] essentially works. However, it is easy to see that it is not any more deadlock-free: postponing the locking of some father of x allows a (partial) schedule to start wrongly, and then be stopped later on by a deadlock. Note that this modified policy is not an L-policy any more. We shall show in § 6 (see Theorem 8) that any such modification of the DAG policy (or the tree policy) has to be an L-policy in order to be deadlock-free.

We will now characterize safety and freedom from deadlock for a pair of transactions. If $F \subseteq E$ is a set of entities, the *restriction of T on F*, is the sequence of steps of T that involve entities from F.

THEOREM 2. *Let $\tau = \{T_1,T_2\}$ be a pair of transactions and $R = R(T_1)\cap R(T_2)$ the set of common entities of T_1 and T_2. τ is safe and deadlock-free if and only if*

(1) *the first entity x of R locked by T_1 is the same as the first entity of R locked by T_2, and*

(2) *for every entity $y \neq x$ of R, the sets $N_1(y) = L_{T_1}(Ly)\cap R_{T_2}(Ly)$ and $N_2(y) = L_{T_2}(Ly)\cap R_{T_1}(Ly)$ are both non-$\varnothing$.*

Proof.

(if) Note at first that entities referenced only by one of the transactions do not affect the safety or freedom from deadlock of τ; that is, if T_1', T_2' are the restrictions of T_1 and T_2 on their common entities R, then τ is safe and deadlock-free if and only if $\tau' = \{T_1',T_2'\}$ is safe and deadlock-free. Construct a digraph D with R as its set of nodes and with $F(z) = N_1(z)\cup N_2(z)$ as the set of fathers of a node $z\in R$. Clearly, if (u,v) is an arc of D then u is locked in both transactions before v, and therefore D is acyclic. Since $N_1(x) = N_2(x) = \varnothing$ (x is the first entity of R locked by T_1 and T_2), node x is a source of D. Since $N_1(y) \neq \varnothing, N_2(y) \neq \varnothing$ for every $y \neq x$, x is the only source of D. Thus, D is a single source acyclic digraph. From the definition of $N_1(y)$ and $N_2(y)$ we have for every $y \neq x$ $N_1(y), N_2(y) \subseteq R_{T_1'}(Ly)\cap R_{T_2'}(Ly) = R_{T_1}(Ly)\cap R_{T_2}(Ly)$ and therefore $F(y)\subseteq R_{T_1'}(Ly), R_{T_2'}(Ly)$. Also $F(y)\cap L_{T_1'}(Ly) = N_1(y) \neq \varnothing$ and $F(y)\cap L_{T_2'}(Ly) = N_2(y) \neq \varnothing$. Consequently, T_1' and T_2' follow the DAG policy on D, and therefore $\tau' = \{T_1',T_2'\}$ is safe and deadlock-free.

(only if) (1) Let x_1 (resp. x_2) be the first entity in R locked by T_1 (resp. T_2). If $x_1 \neq x_2$ then consider the partial schedule S in which T_1 executes its steps up to (including) the Lx_1 step and then T_2 executes its steps up to (and including) the Lx_2 step; the digraph $D(S)$ of S has a cycle and therefore τ is not both safe and deadlock-free.

(2) Suppose that $N_1(y) = \varnothing$ (the case $N_2(y) = \varnothing$ is symmetrical). Consider the following partial schedule S: First run T_1 up to Ly; then run T_2 up to, and including, Ly . This partial schedule is legal since $N_1(y) = \varnothing$. In S, T_1 accesses x before T_2, and T_2 accesses y before T_1. Therefore if S can finish in any way, the resulting schedule will be nonserializable, contradicting the fact that τ is safe and deadlock-free. □

As in § 2, we say that a transaction T has an *unlocked point* if there is a step $i > 1$ of it such that $L_T(\text{step}\,i) = \varnothing$.

COROLLARY 1. *A pair of transactions with the same syntax is safe if and only if it is safe and deadlock-free if and only if the transactions have no unlocked point.*

Proof. Let $\tau = \{T_1,T_2\}$ be a pair of identical transactions. If the transactions have an unlocked point, then τ is not safe: the schedule in which T_1 executes at the

unlocked point of T_2 is obviously nonserializable. On the other hand, if the transactions have no unlocked point, then they satisfy the conditions of Theorem 2: Since the transactions are identical, they start by locking the same entity and for every $y \neq x$ we have $N_1(y) = N_2(y) = L_{T_1}(Ly) \neq \varnothing$, since the transactions have no unlocked point. □

Let P be a locking policy and let $\tau(P)$ be the associated transaction system (the image set of P) without duplicates.

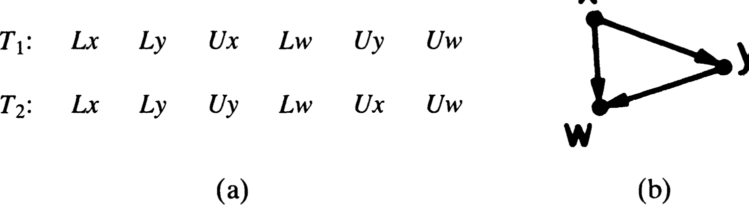
COROLLARY 2. *P is safe (resp. safe and deadlock-free) if and only if* (1) *no transaction in $\tau(P)$ has an unlocked point, and* (2) *the transaction system $\tau(P)$ is safe (resp. safe and deadlock-free).*

Proof. The necessity of the two conditions follows from the definitions and Corollary 1. We will prove the sufficiency part only in the case of safety and freedom from deadlock; the proof for safety only is similar. Suppose that P is not safe and deadlock-free. Then there is a partial schedule S of a set τ of transactions from $\tau(P)$ such that the digraph $D(S)$ contains a cycle. Without loss of generality we can assume that $D(S)$ *is* a chordless cycle $T_1 \to T_2 \to \cdots \to T_k \to T_1$. If τ does not contain two identical transactions, then $\tau(P)$ is not safe and deadlock-free. Assume therefore that τ contains two identical transactions . Since the cycle is chordless, the two identical transactions must be consecutive, say T_1 and T_2. Suppose that $k > 2$. Let x be the label of an arc $T_2 \to T_3$. Then T_2 locks x in S before T_3 does. If T_1 locks x before T_2, then we must have an arc from T_1 to T_3 contradicting the minimality of the cycle. If T_1 locks x after T_2 (or does not reach the Lx step in S), then we must have an arc from T_2 to T_1, again contradicting the minimality of the cycle. Therefore, $k = 2$, and the pair $\{T_1, T_2\}$ is not safe and deadlock-free. It follows then from Corollary 1 that condition (1) is violated. □

From the proof of Theorem 1, we can state the conditions for safety and freedom from deadlock of a pair of transactions in the following form.

COROLLARY 3. *A pair of transactions $\tau = \{T_1, T_2\}$ is safe and deadlock-free if and only if the restrictions of T_1 and T_2 on their common entities $R(T_1) \cap R(T_2)$ follow the* DAG *policy for some* DAG *D on $R(T_1) \cap R(T_2)$.*

Since the entities referenced only by one transaction do not affect the safety or freedom from deadlock of a transaction system, Corollary 3 says essentially that the DAG policy captures these properties for systems of two transactions. Note that the tree policy does not suffice to cover all safe and deadlock-free pairs of transactions. For example consider the pair $\{T_1, T_2\}$ of transactions of Fig. 2(a) which follow the DAG policy for the DAG of Fig. 2(b).



(a) (b)

FIG. 2

Clearly $L_{T_1}(w) \cap L_{T_2}(w) = \varnothing$, and thus no entity can be the father of w in a tree that would cover $\{T_1, T_2\}$.

We can have the transactions themselves (rather than their restrictions to the common entities) follow the DAG policy if we extend the policy to apply also to DAGs with more than one source. The rules of the extended policy are as follows.

Multiple source DAG *policy* (MSDP)

(1) First lock is arbitrary.

(2) Subsequently, an entity A can be locked only if

(a) the transaction has previously referenced all fathers of A that are not separated by A (in the underlying undirected graph) from the first entity of the transaction, and

(b) the transaction holds a lock on at least one father of A.

It is easy to see that MSDP reduces to the DAG policy if the DAG has only one source. The Multiple source DAG policy is also safe and deadlock-free. Its safety can be shown using the Hypergraph criterion in a way similar to that of the DAG policy. The basic idea is that given a transaction T of MSDP that references A but not as the first entity, every other transaction T' referencing A must either reference exactly the same fathers of A as T, or else A separates $R(T')$ from all the entities referenced by T up to the LA step. The freedom from deadlock of MSDP follows from its safety as in Theorem 1.

COROLLARY 4. *A locking policy with two transactions is safe and deadlock-free if and only if it is contained in the multiple source* DAG *policy for some* DAG D.

Proof. Let P be a locking policy with two transactions T_1 and T_2 (in its image set). Let D' be the DAG on $R(T_1)\cap R(T_2)$ constructed in the proof of Theorem 1. Since P is safe, its transactions have no unlocked point. Extend D' to a DAG D on $R(T_1)\cup R(T_2)$ as follows. Let $\bar{R}$ be the set of entities that have no incoming arc in D'; $\bar{R}$ consists of the first common entity locked by the transactions and the entities that occur only in one of the transactions. For every entity x in $\bar{R}$ that appears in T_1 (resp. T_2) but is not the first entity locked by the transaction, choose any entity y locked by T_1 (resp. T_2) when x gets locked and add an arc $y \rightarrow x$. It is easy to see then that T_1 and T_2 follow the multiple source DAG policy on the DAG D. □

4. Testing for safety and freedom from deadlock. The criterion of Theorem 2 for the safety and freedom from deadlock of a pair of transactions gives rise to an efficient algorithm to perform this task. Let $\tau = \{T_1,T_2\}$ be a pair of transactions. Using standard techniques (see, e.g., [AHU]) we can compute in $O(n\log n)$ time $R = R(T_1)\cap R(T_2)$, and check condition (1) of Theorem 2. For $z \in R$ let $i_1(z)$ be the order of the Lz step in T_1 and $i_2(z)$ the order of the Lz step in T_2.(A single pass over T_1 and T_2 suffices to compute these two parameters for all $z \in R$.) Let $M_1(Ly) = \{i_2(z) | z \in L_{T_1}(Ly)\}$ and $M_2(Ly) = \{i_1(z) | z \in L_{T_2}(Ly)\}$. Clearly, condition (2) of Theorem 2 is equivalent to: for every $y \neq x$ in R , $\min M_1(Ly) < i_2(y)$ and $\min M_2(Ly) < i_1(y)$. We can check these conditions by doing a pass over T_1 while keeping at each step in a data structure D the i_2 parameters of the entities of R that are currently locked, and then doing a similar pass over T_2. The data structure D has to be such that the operations INSERT, DELETE, MIN can be performed in $O(n\log n)$ time. (A 2-3 tree for example will do — see [AHU].) Thus, we have:

THEOREM 3. *A pair of transactions can be tested for safety and freedom from deadlock in* $O(n\log n)$ *time.*

Unfortunately, safety and freedom from deadlock cannot be tested efficiently in general (unless P=NP), even for the simplest kind of transaction systems: systems of two-phase locked transactions. Note that two-phase transaction systems are safe and have a very simple kind of deadlocks: If τ is a two-phase transaction system then τ is

not deadlock-free if and only if there exist transactions $T_1,\ldots,T_k$ and entities $x_1,\ldots,x_k$ with $x_i \in R(T_i)$, such that if T_i' is the prefix of T_i up to (not including) the Lx_i step, the partial schedule S which is the serial execution of $T_1',\ldots,T_k'$ (in any order) is legal and its state $J(S)$ is a deadlock state; that is, $R_{T_i}(Lx_i) \cap R_{T_j}(Lx_j) = \varnothing$ for $i \neq j$, and $x_{i-1} \in R_{T_i}(Lx_i)$ (i.e., in Fig. 1 the first two sets of dots in each transaction represent steps that do not involve entities referenced in the other transactions).

THEOREM 4. *It is NP-complete to decide whether a set of two-phase transactions is not deadlock-free.*

Proof. Let $F = C_1 \wedge \cdots \wedge C_p$ be a formula in conjunctive normal form with at most 3 literals per clause, and assume for simplicity that each variable occurs twice and its negation once.

We construct a transaction system τ which contains two transactions T_i and T_i' for each clause C_i, and one transaction for each literal occurrence. Let C_i be $a \vee b \vee c$ (literal c may be missing). T_i is $LB_i\ La_i'\ Lb_i'\ Lc_i'\ UB_i\ Ua_i'\ Ub_i'\ Uc_i'$, and T_i' is $La_i\ Lb_i\ Lc_i\ LB_{i+1}\ Ua_i\ Ub_i\ Uc_i\ UB_{i+1}$, where if $i = p$, B_{i+1} stands for B_1.

If variable x occurs in C_i, C_j and its negation in C_k, then the transactions corresponding to x are:

$$T_{x_i}\colon\ LA_x\ Lx_i'\ Lx_i\ UA_x\ Ux_i'\ Ux_i,$$
$$T_{x_j}\colon\ LA_x'\ Lx_j'\ Lx_j\ UA_x\ Ux_j'\ Ux_j,$$
$$T_{\bar{x}_k}\colon\ LA_x\ LA_x'\ L\bar{x}_k'\ L\bar{x}_k\ UA_x\ UA_x'\ U\bar{x}_k'\ U\bar{x}_k.$$

We claim that τ is not deadlock-free if and only if F is satisfiable.

(if) Suppose that F is satisfiable. Fix a satisfying truth assignment and let d_i be the literal that satisfies C_i. Consider the state J of the partial schedule S which executes the steps of each T_i before Ld_i', the steps of each T_i' before LB_{i+1}, and the steps of each T_{d_i} before Ld_i. (This is a legal schedule since no two of the d_i's are the negation of each other.) State J is a deadlock state: Fig. 3 shows the entity that each transaction has to lock next and the transaction by which it is locked.

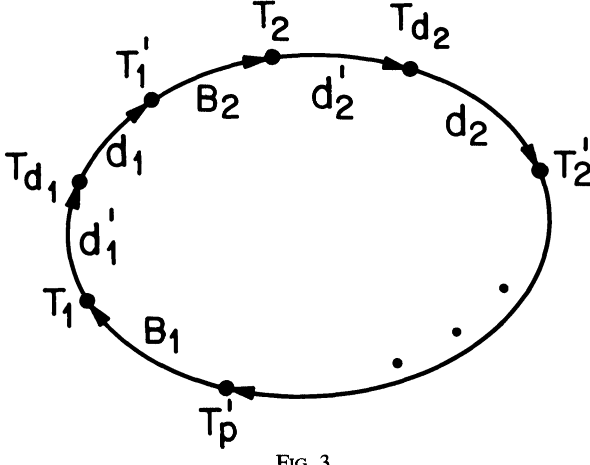


FIG. 3

(only if) Suppose that there is a deadlock, and consider the cycle K of the transactions in the deadlock state J, as in Fig. 1. Suppose that $T_{\bar{x}_k} \in K$ for some variable x, and let T be the transaction that has to lock next entity y, currently locked by $T_{\bar{x}_k}$. Then y cannot be either A_x or A_x' because then T would be T_{x_i} or T_{x_j}, and would not

hold a lock on any entity; since the next step of $T_{\bar{x}_k}$ to be executed is a lock step, y must be $\bar{x}_k'$ and therefore T_{x_i} and T_{x_j} are not in the cycle K (because otherwise A_x or A_x' would be locked by two transactions). Thus, T is T_i, and since the next step of $T_{\bar{x}_k}$ is $L\bar{x}_k$, also $T_i' \in K$. Similarly if $T_{x_i} \in K$ for some variable x, then the next step of T_{x_i} must be Lx_i, and therefore $T_i, T_i' \in K$. Now let $T_j' \in K$. The next step of T_j' cannot be Ld_j for any literal d of C_j; therefore it has to be LB_{j+1}, and T_{j+1} must also belong to K, with next step Ld_{j+1}' for some literal d_{j+1} of C_{j+1}. But then $T_{d_{j+1}} \in K$ and consequently also $T_{j+1}' \in K$.

Thus K contains all T_i, T_i''s and a T_{d_i} for each i, where d_i is a literal of clause C_i. Since K does not contain two transactions corresponding to complemented literals, the d_i's define a satisfying truth assignment for the formula F. □

The *interaction graph* $G(\tau)$ of a transaction system τ is an undirected graph with the transactions of τ as its nodes and an edge between any two transactions that have an entity in common. Note that the digraph $D(S)$ of any (partial or complete) schedule S of τ has a subgraph of $G(\tau)$ as its underlying graph. In [Y] we showed that safety of a transaction system can be tested in time polynomial in the number of minimal cycles of its interaction graph. The direct analogue of this result for freedom from deadlock does not hold, however: In the proof of Theorem 4, we can add a Lz step before the unlocking steps of each transaction, where z is a new entity common to all transactions. It is easy to see that the new transaction system is deadlock-free if and only if the original was.

The reason for this difference is the fact that if a transaction system is not safe then this is due to some chordless cycle, whereas deadlock may be possible due to some cycle with chords, even though all chordless cycles are deadlock-free. Thus the correct analogue of Theorem 3 of [Y] is:

THEOREM 5. *Freedom from deadlock of a safe transaction system can be decided in time polynomial in the number of cycles of its interaction graph.*

Proof. At first we check that all pairs of transactions are deadlock-free as in Theorem 3 above. Consider now a shortest partial schedule S whose digraph $D(S)$ contains a cycle $\{T_1,\ldots,T_k\}$, and let (T_k,T_1) be the last arc labelled, say x_k added to $D(S)$. Then the cycle of $D(S)$ must be chordless and the last step executed in S is the Lx_k step of T_k, since S is a shortest such partial schedule. Let x_i be the first entity of $R(T_i) \cap R(T_{i+1})$ locked by T_i (and T_{i+1} since all pairs are safe and deadlock-free), $J = <l_1,\ldots,l_k>$ the state of S, Y_i the set of entities mentioned in the remaining steps of T_i. Because of our choice of S we have: (1) $R_{T_i}(l_i) \cap Y_{i-1} = \varnothing$, (2) $R_{T_i}(l_i) \cap R(T_j) = \varnothing$, for all j with $|i-j| > 1$ (arithmetic is mod k), (3) the Lx_i step of T_i occurs before the l_ith step, for all i, and the Lx_k step of T_1 occurs after the l_1th step — see Fig. 4.

(Note, however, that after state J the transactions may be referencing other entities in common, and thus they may not form a chordless cycle.)

Let l_1' be the latest step of T_1 before Lx_k, such that $R_{T_1}(l_1') \cap [\cup_{j\neq 1,2} R(T_j)] = \varnothing$, and let Y_1' be the set of entities mentioned by T_1 after l_1'. From (2) and (3) we have $l_1' \geq l_1$, and therefore $Y_1' \subseteq Y_1$. Let l_2' be the latest step of T_2, such that $R_{T_2}(l_2') \cap Y_1' = \varnothing$, and $R_{T_2}(l_2') \cap R(T_j) = \varnothing$, for $|j-2| > 1$. Since $Y_1' \subseteq Y_1$, we have from (1) and (2) that $l_2' \geq l_2$ and therefore $Y_2' \subseteq Y_2$ where Y_2' is the set of entities referenced by T_2 after l_2'. Proceeding in this way, we define l_i' (and the corresponding Y_i') such that $R_{T_i}(l_i') \cap Y_{i-1}' = \varnothing$ (1′), and $R_{T_i}(l_i') \cap R(T_j) = \varnothing$ for $|j-i| > 1$ (2′). Then $l_i' \geq l_i$, and therefore (3′) Lx_i occurs

						|					
T_1:	.	.	Lx_1	.	.	|	.	.	Lx_k	.
T_2:	.	.	Lx_2	.	.	|	.	.	.	.
.					.	|				
.					.	|				
.					.	|				
T_{k-1}:	.	.	Lx_{k-1}	.	.	|	.	.	.	.
T_k:	.	.	.	.	Lx_k	|	.	.	.	.
						J				

FIG. 4

in T_i before l_i'.

Now, to test whether τ is a deadlock-free transaction system we take every cycle $K = \{T_1,...,T_k\}$ of its interaction graph, assign an orientation and choose a first transaction, say T_1. We then define in turn $l_1',...,l_k'$ as above to satisfy (1′) and (2′) (where for $i=1$ we take in (1′) Y_k' to be $R(T_k)$). If the l_i''s satisfy also (3′), then τ is not deadlock-free. For, consider the partial schedule formed by running T_1 up to the l_1'th step, T_2 up to the l_2'th step, ..., T_k up to the l_k'th step. This is a legal schedule since $R_{T_i}(l_i') \cap Y_{i-1}' = \varnothing \Rightarrow R_{T_i}(l_i') \cap L_{T_{i-1}}(l_{i-1}') = \varnothing$ (because $L_{T_{i-1}}(l_{i-1}') \subseteq Y_{i-1}'$) and $R_{T_i}(l_i') \cap L_{T_j}(l_j') = \varnothing$ for $j<i-1$ (because $L_{T_j}(l_j') \subseteq R_{T_j}(l_j') \subseteq R(T_j)$). The digraph $D(S)$ of S contains for each i an edge (T_i,T_{i+1}) labelled x_i (because of (3′)) and therefore is not acyclic. □

5. Deadlock-free *L*-policies. Theorem 4 implies that unless NP=co-NP there is no simple, structured L-policy ΔP that can express in a nice way all safe and deadlock-free L-policies, where by "nice" we mean that the size of the structure in Δ that expresses a policy P be bounded by some polynomial in the size of the set of transactions of P. For, if there were such a policy ΔP we could test in nondeterministic polynomial time whether a set τ of two-phase transactions is deadlock-free (a co-NP-complete problem) by guessing the structure D in Δ which expresses $Ct(\tau)$ and then verifying that all transactions of $Ct(\tau)$ follow ΔP for D. (Note that closing under truncation a set of transactions does not affect its freedom from deadlock.)

Of course, Theorem 4 implies also that testing whether a safe L-policy is not deadlock-free is NP-complete. However, in the case of L-policies, knowing that the policy is safe restricts the kind of possible deadlocks that can occur and as a consequence helps us in proving freedom from deadlock; recall, for example, how we used in Theorem 1 the safety of the DAG policy to prove its freedom from deadlock.

THEOREM 6. *If τ is a safe transaction system that is closed under truncation, then τ is deadlock-free if and only if there do not exist transactions T_1, ..., T_k, and entities x_1, ..., x_k, where $x_i \in R(T_i) \cap R(T_{i+1}) - [\bigcup_{j\neq i,i+1} R(T_j)]$ and $R_{T_i}(Lx_i) \cap [\bigcup_{j\neq i} R(T_j)] = \varnothing$.*

Proof.

(only if) Obvious.

(if) (1) All pairs of transactions are deadlock-free: Let $\{T_1,T_2\}$ be a pair. From the hypothesis with $k = 2$, we have that the first entity of $R(T_1) \cap R(T_2)$ locked by T_1 and T_2 is the same, say x. From Theorem 2 we still have to show that for all $y \in R(T_1) \cap R(T_2)$, $y \neq x$, $N_1(y) = L_{T_1}(Ly) \cap R_{T_2}(Ly) \neq \varnothing$ and $N_2(y) = L_{T_2}(Ly) \cap R_{T_1}(Ly) \neq \varnothing$. Suppose that $N_1(y) = \varnothing$, and let T_2' be the

truncation of T_2 at Ly. Since $x \in R_{T_2}(Ly)$, x must be unlocked by T_1 before Ly. But then $R(T_2')$ forms an $x-y$ path in the hypergraph of $\{T_1,T_2\}$ that avoids $L_{T_1}(Ly)$, and therefore $\{T_1,T_2'\}$ is not safe. (Recall from § 2 the criterion for safety of a system that is closed under truncation.)

(2) Suppose now that all pairs of transactions are deadlock-free but τ is not. Then the configuration of Fig. 4 has to occur, where x_i is the first entity of $R(T_i) \cap R(T_{i+1})$ locked by T_i (and T_{i+1} since all pairs of transactions are safe and deadlock-free).

Suppose that for some i, x_{i-1} is locked in T_i before the l_ith step, with the notation of the proof of Theorem 5, and let i be the smallest index for which this happens ($i \neq 1$). Since $R_{T_i}(l_i) \cap Y_{i-1} = \varnothing$, x_{i-1} must be unlocked by T_{i-1} before l_{i-1}. Let y be the first element of $\cup_{j \neq i,i-1} R(T_j)$ locked by T_{i-1}. (Clearly, there exists such a y since $R(T_{i-1}) \cap R(T_{i-2}) \neq \varnothing$.) Since $R_{T_{i-1}}(l_{i-1}) \cap R(T_j) = \varnothing$ for $j \neq i,i-1,i-2$, and since $R_{T_{i-1}}(l_{i-1}) \cap R(T_{i-2}) = \varnothing$ (because Lx_{i-2} occurs in T_{i-1} after the l_{i-1}th step by our choice of i), it must be the case that Ly occurs in T_{i-1} after the l_{i-1}th step, and therefore after Ux_{i-1}. Let T_i' be the truncation of T_i at the l_ith step. We have then $R(T_i') \cap Y_{i-1} = \varnothing \Rightarrow R(T_i') \cap L_{T_{i-1}}(Ly) = \varnothing$, and therefore $R(T_i')$, $R(T_{i+1})$, $\ldots$, $R(T_{i-2})$ form a path between x_{i-1} and y in the hypergraph of τ that avoids $L_{T_{i-1}}(Ly)$, contradicting the safety of τ.

Thus, for each i, x_{i-1} is locked in T_i after the l_ith step and therefore after Lx_i. Since $R_{T_i}(l_i) \cap R(T_j) = \varnothing$ for $j \neq i$, $i-1$, $i+1$, and because x_i (resp. x_{i-1}) is the first entity common to T_i and T_{i+1} (resp. T_{i-1}) locked by T_i, we have $R_{T_i}(Lx_i) \cap R(T_j) = \varnothing$ for $j \neq i$. □

Note that the conditions of Theorem 6 involve only R sets and not any L sets of the transactions. It follows then that whether a safe L-policy is deadlock-free or not depends only on the order in which entities get locked, and not on how the unlock steps are placed within the transactions. More formally we have:

COROLLARY 5. *Suppose that $\tau = \{T_1,\ldots,T_m\}$ and $\tau' = \{T_1',\ldots,T_n'\}$ are two safe transaction systems that are closed under truncation, and such that for every i, there is a j, where T_j' locks the same entities as T_i in the same order. Then, if τ' is deadlock-free, then so is τ.*

If we take in Corollary 5 τ' to be τ with all the Unlock steps moved to the end of the transactions (i.e., τ' is the two-phase version of τ) it follows that all safe L-policies that are not deadlock-free have the same kind of simple deadlocks that two-phase transaction systems have; thus, for example, if the freedom from deadlock problem for 2PL transaction systems can be solved in $f(n)$ time then the freedom from deadlock problem for any transaction system that is closed under truncation can be solved also in $f(n)$ time.

Note that Corollary 5 is not true for general policies (systems that are not closed under truncation). For example, consider a transaction system τ' that consists of the following transactions:

T_1':	Lx_1	Lx_3	Ux_1	Ux_3		
T_2':	Lx_2	Ly	Lx_1	Uy	Ux_1	Ux_2
T_3':	Lx_2	Lx_3	Ly	Ux_2	Ux_3	Uy

From Theorem 6, τ' is safe and deadlock-free. Suppose that $\tau = \{T_1,T_2,T_3\}$, where $T_1 = T_1'$, $T_3 = T_3'$, and T_2 is $Lx_2\ Ly\ Ux_2\ Lx_1\ Uy\ Ux_1$. Clearly τ is safe (although its closure under truncation is not). However τ is not deadlock-free as the following

partial schedule shows:

T_1:	Lx_1					
T_2:		Lx_2	Ly	Ux_2		
T_3:					Lx_2	Lx_3

In practical terms, Corollary 5 has the following implication. Suppose that we design an L-policy P and that we have already determined the underlying hypergraph H of P (how the various transactions visit the entities). For P to be safe, we must enforce rule 3 of the Hypergraph policy HP. Now, one might choose not to use the full freedom of it in order to get a more efficient policy (at the expense of a loss in concurrency). As a consequence, there may be various ways in which rule 3 can be enforced. Corollary 5 implies that no matter which way is chosen, freedom from deadlock is not going to be affected: if one way leads to a deadlock-free policy then so does every other way. Therefore, if HP on the hypergraph H is not deadlock-free, then in order to get a deadlock-free policy we must restrict the way in which rule 2 of HP is applied. Of course, the NP-completeness result of the previous section implies that there is no most general and simple restriction of rule 2.

Corollary 5 is useful for deriving freedom from deadlock of a safe L-policy from that of another. Let us prove as an example that the DAG policy is deadlock-free using the fact that the tree policy is. Let D be the underlying DAG of DP and let G be the dominator tree of D.[5] Let T be a transaction of DP. If $y \in R(T)$ and y is not the first entity locked by T, then before locking y T has locked (and possibly unlocked) the dominator of y in D [Y]. Thus, there is a transaction T' of TP operating on the dominator tree of D which locks exactly the same entities as T in the same order. (For example, T' can be obtained from T by moving all the Unlock steps to the end.) Since TP is deadlock-free, so is DP.

We will exploit further this property of safe L-policies in the next section to derive sufficient conditions for freedom from deadlock.

6. Sufficient conditions for freedom from deadlock. From a transaction system τ we are going to construct a directed graph $D'(\tau)$ which reflects partially the order in which entities get locked by the transactions of τ. We then use Corollary 5 to derive sufficient conditions on $D'(\tau)$ that guarantee the freedom from deadlock of τ if τ is closed under truncation and safe.

With every transaction system τ we can associate a directed graph $D(\tau)$ as follows: the nodes of $D(\tau)$ are the entities, and there is an arc (x,y) if there is a transaction T of τ that starts by locking x and references y. Suppose that τ is safe and deadlock-free. Then,

(a) *$D(\tau)$ is acyclic.*

Suppose that there is a cycle $x_1 \to x_2 \to \cdots \to x_k \to x_1$. Then there are transactions $T_1, \ldots, T_k$, such that the first step of T_i is Lx_i, and $x_{i+1} \in R(T_i) \cap R(T_{i+1})$. Clearly $\{T_1, \ldots, T_k\}$ cannot be both safe and deadlock-free.

(b) *If x is an ancestor of y, then in all transactions that contain both x and y, x*

---

[5] A node y *dominates* a node x in a single source DAG D, if every path from the source to x passes through y. The *dominator* of x is the (unique) lowest ancestor of x that dominates x. The dominator tree of D is a (rooted) tree G with the source of D as its root and with the dominator of each node x as its father — see, e.g., [AHU].

gets locked before y.

Suppose that there is a transaction T where Ly precedes Lx, and assume without loss of generality that no descendant of x is locked before Ly. Let $x \to z_1 \to \cdots \to z_k \to y$ be a directed $x-y$ path, and $T_0, T_1, \ldots, T_k$ the transactions that give rise to the arcs of the path. The partial schedule that executes the Lx step of T_0, the Lz_i step of T_i, for each i, and the prefix of T up to (and including) the Ly step has a cycle in its corresponding digraph.

Let $F_1(x)$ be the set of fathers y of x, for which there is a transaction T that starts with y, contains x, and there is no ancestor z of x in $R_T(Lx)-y$. Denote by $D'(\tau)$ the subgraph of $D(\tau)$, where the arc (y,x) is in $D'(\tau)$ if there is a transaction T starting with y, containing x, and such that $F_1(x) \cap [R_T(Lx)-y] = \varnothing$. ($D'(\tau)$ has the same transitive closure as $D(\tau)$, but is not necessarily its transitive reduction.) Clearly every transaction T of τ has the property:

(c) *If* $x \in R(T)$ *is not the first entity locked by* T, *then at least some father* y *of* x *in* $D'(\tau)$ *is referenced by* T *before* Lx.

Consequently every transaction T references a connected subgraph of $D'(\tau)$ which can be reached from the first entity locked by T. For example, if τ is the tree policy, then $D'(\tau)$ is the tree itself; in general if τ is the hypergraph policy on a (directed) graph G, then $D'(\tau)$ is the graph G. If τ is the DAG policy, then $D'(\tau)$ is the dominator tree of the DAG.

THEOREM 7. *If* P *is a safe* L*-policy with* $D'(P)$ *a tree, then* P *is deadlock-free.*

Proof. By Corollary 5 and the freedom from deadlock of the tree policy for general (not necessarily rooted) trees [KS]. □

Note that Theorem 7 is not true for general policies. For example, if $\tau = \{T_1, T_2\}$ with T_1 : *LA LC UA LB UB UC* and T_2 : *LA LB UA LC UB UC* then $D'(\tau)$ is the tree of Fig. 5, τ is safe (but its closure under truncation $Ct(\tau)$ is not), and τ is not deadlock-free. This is not a coincidence: if any such τ is deadlock-free, then it can be extended to a safe (and of course still deadlock-free) L-policy.

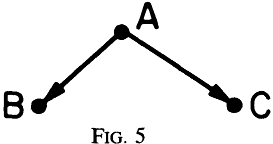


FIG. 5

THEOREM 8. *Suppose that* P *is a safe and deadlock-free policy with* $D'(P)$ *a tree. Then* $Ct(P)$ *is also safe and deadlock-free.*

Proof. The freedom from deadlock of $Ct(P)$ follows from that of P. Let $y_1, \ldots, y_k$ be the fathers of x in $D'(P)$, and for each $i = 1, \ldots, k$ let $N_i(x) = \{L_T(Lx) \mid T \in P, \text{and } y_i \text{ is unlocked in } T \text{ before } x \text{ is locked}\}$. Consider the following L-policy P': (1) First lock arbitrary ; (2) subsequently entity x can be locked if and only if (i) some father y_i of x has been mentioned, (ii) some transversal [6] of $N_i(x)$ has been mentioned, and (iii) if y_i has been unlocked then all the elements of some set in N_i are currently locked.

Every transaction T of P is in P': it satisfies (i) by property (c) of $D'(P)$, it

[6] A *transversal* of a family of sets is a set which contains at least one element from every set in the family.

satisfies (iii) by the definition of $N_i(x)$, and it satisfies (ii) because all pairs of transactions of P are safe and deadlock-free.

Let H be the hypergraph of P'. Let T be a transaction in which x is locked after some other entity z is unlocked, and suppose that y_i is the (necessarily unique) father of x that is mentioned in T. From the definition of P', $L_T(Lx)$ contains either y_i or a set in $N_i(x)$; consequently $L_T(Lx)$ intersects all hyperedges that contain both x and y_i. Since $D'(P)$ is a tree, every $x-z$ path in H has to use a hyperedge that contains both x and y_i. Therefore $L_T(Lx)$ separates x from z in H, and consequently the policy P' is safe by the safety criterion for L-policies (see § 2). □

Actually, it is Corollary 4 that lies behind Theorem 8: It is possible to show that a policy P with $D'(P)$ a tree is safe and deadlock-free if and only if all pairs of transactions of it are so. From Corollary 4 there is an L-policy (the multiple source DAG policy) that describes all safe and deadlock-free pairs of transactions. Thus, taking the closure under truncation of all transactions will not create any pairs that are not safe or deadlock-free and therefore will not destroy safety.

Theorem 8 does not hold if $D'(P)$ is not a tree. Consider, for example, the set $P = \{T_1,T_2,T_3\}$ with $T_1 : LA\ LB\ UA\ LD\ UB\ UD$, $T_2 : LA\ LC\ LB\ UA\ UB\ UC$, $T_3 : LC\ LD\ UC\ UD$. It is easy to see that P is safe and deadlock-free but $Ct(P)$ is not safe. The digraph $D'(P)$ is shown in Fig. 6.

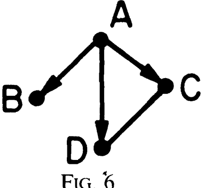


FIG. 6

Suppose now that P is an L-policy . Deadlocks may arise because of undirected cycles in $D'(P)$ (cycles in the underlying undirected graph of $D'(P)$).

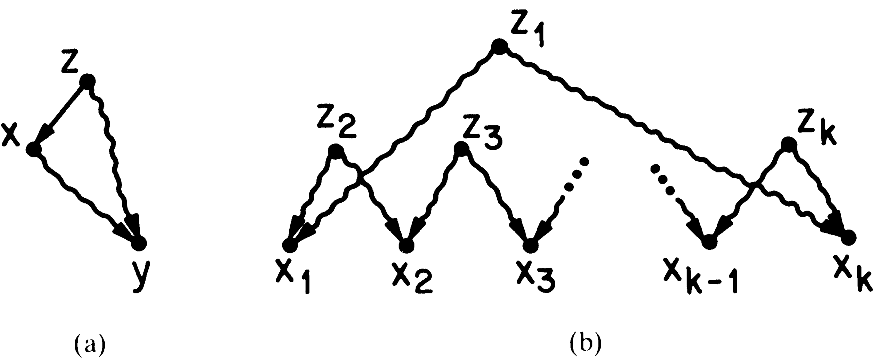


FIG. 7

In a general $D'(P)$ we can distinguish between two kinds of cycles — see Fig. 7(a),(b). A cycle as in Fig. 7(a) may give rise to a pair of transactions $\{T_1,T_2\}$, where T_1 starts from x and follows the path to y, and T_2 starts from z, follows the path to y, and then goes on to lock x; thus, T_2 does not satisfy condition (b). (Note that if P is the hypergraph policy with the graph $D'(P)$ as the underlying hypergraph, then both transactions are allowed.) If condition (b) is checked in the rules of the policy, then cycles as in Fig. 7(a) do not create any problems. That is, we can show that the policy is deadlock-free, unless D' contains a cycle as in Fig. 7(b) with the x_i's pairwise incomparable — i.e. no x_i is an ancestor of another x_j.

THEOREM 9. *Let P be a safe L-policy whose digraph $D'(P)$ is acyclic and contains no (undirected) cycles as in Fig. 7(b). Then P is deadlock-free if and only if it satisfies condition (b).*

Proof. The necessity of condition (b) was shown before. To prove the sufficiency, let τ be the set of transactions of the hypergraph policy HP operating on $D'(P)$, which are two-phase and satisfy condition (b). Clearly, τ is closed under truncation. By Corollary 5, P is deadlock-free if and only if τ is. Suppose that $\{T_1,\ldots,T_k\}$ is a smallest set of transactions of τ deadlocked at state J where the next step of each T_i is Lx_i with $x_i \in L_{T_{i+1}}(Lx_{i+1})$ (see Fig. 1). From property (c) of D' T_i references a common ancestor of x_i and x_{i-1} and a path from it to them. Let z_i be a lowest common such ancestor. Since the transactions are two-phase all these paths are disjoint, and therefore form a cycle as in Fig. 7(b). It remains to show that the x_i's are pairwise incomparable. Suppose that x_1 is an ancestor of x_j, with j as large as possible. Since condition (b) is satisfied $j<k$. Let p be a directed path from x_1 to x_j. Let T_1' be the two-phase transaction that agrees with T_1 up to the Lx_1 step and then locks in order the entities on the path p. Clearly T_1' is also in τ, and the set of transactions $\{T_1',T_{j+1},T_{j+2},\ldots,T_k\}$ is not deadlock-free, contradicting the minimality of k. □

Note that if D' is a tree, then (b) is satisfied automatically. (Thus, Theorem 7 follows also from Theorem 9 .) Also, note that if a cycle as in Fig. 7(b) exists in D', and P is the hypergraph policy operating on D', with condition (b) checked in addition, such a cycle gives rise to a deadlock. Deadlock from such a cycle can be avoided either if we lock nodes according to some specified (acyclic) order or prevent the x_i's from being the first common entities of the corresponding transactions T_i, by forcing the transactions to start locking higher in the DAG D'. For example the following rule guarantees freedom from deadlock (assuming that (b) is enforced): if $x \in R_T(Ly)$ and x is not an ancestor of y, then $R_T(Ly)$ and the descendants of x separate x and y in the underlying graph of D'. (This can be proved similarly to Theorem 9). Thus, for example, the DAG policy enforces this rule by requiring all fathers of x to be locked before x (which results in $D'(DP)$ being a tree rather than the original DAG). Note however that the previous rule (or any other simple rule) is not necessary, because of our NP-completeness result of § 4.

Acknowledgments. I wish to thank J.-D. Ullman for introducing me to the area of concurrency control, and A. V. Aho, M. D. McIllroy and P. J. Weinberger for helpful comments on an earlier version of the manuscript.

REFERENCES

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

[BS] R. BAYER, AND M. SCHKOLNICK, *Concurrency of operations on B-trees,* Acta Informatica, 9 (1977), pp. 1-21.

[E] K. P. ESWARAN, J. N. GRAY, R. A. LORIE AND I. L. TRAIGER, *The notions of consistency and predicate locks in a database system,* CACM 19(11) (1976), pp. 624-633.

[GJ] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* Freeman, 1978.

[G1] J. N. GRAY, R. A. LORIE, G. R. PUTZOLU AND I. L. TRAIGER, *Granularity of locks and degrees of consistency in a shared data base,* IBM Research Report RJ 1654, (1975).

[G2] J. N. GRAY, *Notes on database operating systems,* IBM Research Report RJ 2188, (1978).

[KS] Z. KEDEM AND A. SILBERSCHATZ, *Controlling concurrency using locking protocols,* Proc. of the 20th Annual Symp; pp. 274-285. on Foundations of Computer Science (1979).

[KP] H. T. KUNG AND C. H. PAPADIMITRIOU, *An optimality theory of concurrency control for databases,* ACM/SIGMOD Intern. Symp. on Management of Data (1979), pp. 116-126.

[LW] Y. E. LIEN AND P. J. WEINBERGER, *Consistency, concurrency, and crash recovery*, ACM/SIGMOD Intern. Symp. on Management of Data (1978), pp. 9-15.

[P1] C. H. PAPADIMITRIOU, *The serializability of concurrent database updates,* JACM 26 (1979), pp. 631-653.

[P2] C. H. PAPADIMITRIOU, P. A. BERNSTEIN AND J. B. ROTHNIE, *Computational problems related to database concurrency control,* Conf. on Theoretical Computer Science, University of Waterloo (1977), pp. 275-282.

[S] B. SAMADI, *B-trees in a system with multiple users,* Information Processing Letters, 5 (1976), pp. 107-112.

[SK] A. SILBERSCHATZ AND Z. KEDEM, *Consistency in hierarchical database systems,* JACM 27 (1980), pp. 72-80.

[St] R. E. STEARNS,, P. M. LEWIS AND D. J. ROSENKRANTZ, *Concurrency control for database systems*, Proc. of the 17th Annual Symp. on Foundations of Computer Science (1976), pp. 19-32.

[U] J. D. ULLMAN, *Principles of Database Systems,* Computer Science Press, 1979.

[Y] M. YANNAKAKIS, *A theory of safe locking policies in database systems,* to appear in JACM (1982).

[YPK] M. YANNAKAKIS, C. H. PAPADIMITRIOU AND H. T. KUNG, *Locking policies: safety and freedom from deadlock,* Proc. of the 20th Annual Symp. on Foundations of Computer Science (1979), pp. 286-297.

SIAM J. COMPUT.
Vol. 11, No. 3, August 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1103-0001 $01.00/0

THE COUNTERFEIT COIN PROBLEM REVISITED*

NATHAN LINIAL† AND MICHAEL TARSI‡

Abstract. We find the optimal algorithm in the sense of average run time for the counterfeit coin problem: Given n coins, one of which is heavier or lighter than the rest. Using a balance scale, find the counterfeit coin and whether it is heavy or light. An interesting feature of the solution is that our algorithm is a straight line algorithm. We also find the optimal algorithm if a standard coin is available for the first weighing.

Key words. average-optimal algorithms, search problems, Huffman trees, straight-line algorithms

In this paper we consider the following algorithmic problem which is a variation on the ancient false coin problem. (Find the counterfeit one, out of 12, by 3 weighings.)

The Problem.

Input: $n \geqq 3$ coins, one of which is false, being either heavier or lighter than the other coins.

Output: Find the counterfeit coin and whether it is heavy or light. The output coin j is light (resp. heavy) is denoted jL (resp. jH).

Step: The elementary step is placing an equal number of coins on the two sides of a balance.

Assuming the possible $2n$ outcomes equally likely, we look for algorithms which take the smallest number of steps on the average.

(1) Straight line/nonstraight line, i.e., branching is prohibited or allowed. Branching prohibited means that the location of the coins for every weighing is given and cannot be changed according to the results of previous weighings. We show that this does not change the situation.

(2) A standard coin is available for use in the weighings in addition to the set of n coins. It turns out that the average number of steps can sometimes be reduced using this extra coin.

We define four functions:

$$F(n), \quad F_N(n), \quad S(n), \quad S_N(n).$$

$F(n)$ is defined as $2n$ times the smallest average number of steps that any algorithm takes to solve the problem. (Note that when n coins are given there are $2n$ possible outcomes). The other three functions are defined for the straight line case (no branching allowed), with a standard coin, and straight line with a standard coin, respectively.

Our main results are summarized in the following.

THEOREM. *For* $n \geqq 3$, $F(n) = F_N(n)$, $S(n) = S_N(n)$ (*i.e., branching cannot reduce the average number of steps*). *Let*

$$2n = 3^t + 2k + 1, \qquad (3^t > k \geqq 0).$$

* Received by the editors May 8, 1981 and in revised form June 23, 1981.

† Department of Mathematics, University of California, Los Angeles, California 90024 and Institute of Mathematics and Computer Science, Hebrew University, Jerusalem 91904 Israel. The work of this author was supported by the Chaim Weizman post-doctoral grant.

‡ Department of Mathematics, University of California, Los Angeles, California 90024. The work of this author was supported in part by the National Science Foundation under grant MCS 78-18924.

Then

$$F(n)=2nt+3k+\begin{cases}4, & k \text{ even},\\ 3, & k \text{ odd},\end{cases}$$

$$S(n)=2nt+3k+\begin{cases}2, & k \text{ even},\\ 3, & k \text{ odd}.\end{cases}$$

Before going into the proof, we need some definitions:

We call a tree *ternary* if each node has at most 3 sons, called the L-, N- and R-sons, respectively. For a ternary tree T rooted at r, the subtree of T rooted at the L son of r is called the *L-subtree* of T, and its leaves are the *L-leaves* of T. The N- and R-subtrees and N- and R-leaves are defined similarly. The definition extends in the obvious way to w-subtrees and w-leaves, w being any word over the alphabet $\{L, N, R\}$. Also we define $h(T)=\sum d(x, r)$, the sum taken over all leaves x of T and $d(x, r)$ being the distance in T from x to r.

An algorithm which solves the counterfeit coin problem can be represented by a rooted ternary tree: The problem has $2n$ possible outcomes $1H, 1L, \cdots, nH, nL$. Every possible outcome is discovered by a certain sequence of weighing results. We denote such a sequence by a word over the alphabet $\{L, N, R\}$. The ith letter is $R(L)$ if the right (left) side of the balance is heavier at the ith weighing and N if both sides are equal. Since the process stops when the counterfeit coin is discovered, the obtained set of words is prefix free and naturally defines a ternary tree with $2n$ leaves. This is the tree which we assign to the algorithm. The definitions make it clear that the average number of steps that the algorithm takes equals $(1/2n)h(T)$, where T is the corresponding tree.

It is clear now that the following Huffman problem [*Hu*] is closely related:

Evaluate $H(n)=\min h(T)$ over all ternary trees with $2n$ leaves, and describe all trees which attain this minimum.

We illustrate the situation in the following:

Observation 1.

$$F_N(n)\geqq F(n)\geqq H(n),$$

$$S_N(n)\geqq S(n)\geqq H(n),$$

$$F(n)\geqq S(n),$$

$$F_N(n)\geqq S_N(n).$$

Convention. Given an integer $n\geqq 3$, we represent

$$2n=3^t+2k+1, \qquad 3^t>k\geqq 0.$$

The proof of the main theorem obviously follows from the following four propositions:

PROPOSITION 1.

$$H(n)=2nt+3k+2.$$

The trees which achieve this minimum are the following: Starting from a complete ternary tree of height t, choose any $k+1$ leaves. Attach to k of them 3 *new leaves* ($\cdot\rightarrow$ ⋎) *and attach* 2 *new leaves to the* $(k+1)$st ($\cdot\rightarrow$ ⋁). *For any other tree T with* $2n$ *leaves*, $h(T)>H(n)$.

Proof. See [Hu] for the general Huffman tree problem. □

PROPOSITION 2.

$$S(n) \geqq H(n) + \begin{cases} 0, & k \text{ even}, \\ 1, & k \text{ odd}. \end{cases}$$

PROPOSITION 3.

$$F(n) \geqq H(n) + \begin{cases} 2, & k \text{ even}, \\ 1, & k \text{ odd}. \end{cases}$$

PROPOSITION 4.

$$F_N(n) \leqq 2nt + 3k + \begin{cases} 4, & k \text{ even}, \\ 3, & k \text{ odd}, \end{cases}$$

$$S_N(n) \leqq 2nt + 3k + \begin{cases} 2, & k \text{ even}, \\ 3, & k \text{ odd}. \end{cases}$$

The proof of Propositions 2, 3 is based on the fact that not every ternary tree corresponds to an algorithm for the counterfeit coin problem.

PROPOSITION 5. *In a ternary tree which corresponds to an algorithm for the S-problem (even if branching is allowed and a standard coin given), the following holds:*

(S1) *For all $i \geqq 0$ the number of N^iL-leaves equals the number of N^iR-leaves.*

In an F-problem ternary tree (branching still allowed, no standard coin):

(F1) *The number of L-leaves = number of R-leaves is even.*

Proof. According to the tree representation of an algorithm, the N^iR leaves correspond to the outcomes still possible after i "equal" weighings and one "right side heavier", meaning that one of the coins on the right-hand side in the $(i+1)$st step is heavier or one of those in the left side is lighter. The N^iL leaves correspond to the same coins just switching "heavier" and "lighter", and this proves (S1).

As for (F1), if b coins are placed on each side of the balance in the first step, there are $2b$ L-leaves as well as R-leaves in the tree. (Notice that it is not so if a standard coin participates in the first weighing.) □

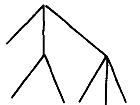
Instead of Proposition 2, we prove the somewhat stronger:

PROPOSITION 2'. *If T is a rooted ternary tree with 2n leaves satisfying* (S1), *then*

$$h(T) \geqq H(n) + \begin{cases} 0, & k \text{ even}, \\ 1, & k \text{ odd}. \end{cases}$$

(We remind the reader that we always represent $2n = 3^t + 2k + 1$, $(3^t > k \geqq 0)$).

Proof. The definition of Huffman trees implies $h(T) \geqq H(n)$, and so we only want to prove that no Huffman tree with $2n$ leaves satisfies condition (S1) if k is odd. We prove it by induction on t. For $t = 1$ the only odd k to be considered is $k = 1$, so $2n = 6$. The only Huffman tree with 6 leaves (up to a permutation) is



This tree and any permutation of it violates condition (S1).

By the description of Huffman trees in Proposition 1, the number of L, N, R-leaves are all between 3^{t-1} and 3^t. Also, two of these numbers are odd and one is even. Together with condition (S1), this implies that

$$\text{number of } L\text{-leaves} = \text{number of } R\text{-leaves} = 3^{t-1} + 2l, \qquad 3^{t-1} \geqq l \geqq 0,$$

and

$$\text{number of } N\text{-leaves} = 3^{t-1} + 2m + 1, \qquad m \text{ is odd}, \quad 3^{t-1} - 2 \geqq m \geqq 1.$$

Using once again our knowledge about Huffman trees, we see that the L, N, R-subtrees of a Huffman tree are all Huffman trees. But the N-subtree satisfies condition (S1) and so cannot be a Huffman tree by the induction hypothesis. □

Proof of Proposition 3. In view of Proposition 2 and the obvious fact that $F(n) \geqq S(n)$, it suffices to discuss the case of even k. For any tree T satisfying (S1) and (F1) with $2n$ leaves and $2n = 3^t + 2k + 1$ (k even, $3^t - 1 \geqq k \geqq 0$), we construct another ternary tree with $2n$ leaves T' so that $h(T) \geqq h(T') + 2$. This will prove our claim. (F1) implies that the number of L-leaves in T which equals the number of R-leaves in T is even. We call it $2l$. The number of N-leaves is even, too, and we call it $2m$. Replace the L, N, R-subtrees of T by Huffman trees with $2l, 2m, 2l$ leaves, respectively. We claim that

$$2l, 2m \geqq 3^{t-1}.$$

By Proposition 1 each of the L, N, R-subtrees of T has a node with exactly two sons which are leaves. Let x, y, z be these nodes with sons x_1, x_2, y_1, y_2 and z_1, z_2 respectively. Assume without loss of generality that $d(x, r) \geqq d(y, r)$. Delete x_1, x_2 and add a new son y_3 to y. If $d(x, r) > d(y, r)$, this already reduces $h(T)$ by at least two. So we may assume that $d(x, r) = d(y, r) = d(z, r)$, which by Proposition 1 implies

$$2l, 2m \geqq 3^{t-1}.$$

Since $2n = 4l + 2m$ and k is even, it follows that

$$2m = 3^{t-1} + 2s + 1, \qquad s \text{ odd}, \quad 3^{t-1} - 2 \geqq s \geqq 1.$$

Now we show how to reduce $h(T)$ by at least two: transfer leaves from the L-subtree to the R-subtree as in the above-described procedure. As mentioned above, this reduces $h(T)$ by one. Now the N-subtree satisfies (S1) and s is odd, so on replacing the N–subtree by a Huffman tree with $2m$ leàves at least another one is gained, as shown in Proposition 2′. □

Proof of Proposition 4. The proof is constructive and inductive on n. Straight line algorithms will be represented by means of a table having n rows: one for each coin and a column for each weighing step. The (i, j) entry of the table being L, R or N indicates that the ith coin is placed on the left, placed on the right, or not placed on the balance at the jth weighing step. Not all the rows must be of equal length. Empty entries at the end of a row indicate that if the corresponding coin is false, then it will be known by the last step in whose column there is an $\{N, L, R\}$ entry in the row.

Associating L with 1, R with -1 and N with 0, it is obvious how arithmetic is done on the "row vectors" of the table. The vectors table constitutes an algorithm to solve the false coin problem if and only if the following 3 conditions hold:

(T1) For $\mathbf{x}$, $\mathbf{y}$ rows of the table, $\mathbf{x}$ is not a prefix of $\mathbf{y}$.

(T2) For $\mathbf{x}$, $\mathbf{y}$ rows of the table, $-\mathbf{x}$ is not a prefix of $\mathbf{y}$.

(T3) The sum of all row vectors is the zero vector.

Necessity. If (T1) is violated we cannot tell between the outcomes $\mathbf{x}H$ and $\mathbf{y}H$. If (T2) fails to hold, then an $\mathbf{x}H$ outcome cannot be told from a $\mathbf{y}L$ outcome. Failure of (T3) implies that one of the weighing steps is worthless since we place a different number of coins on the right side and left side of the balance.

Sufficiency. Let $\mathbf{b}$ be a row vector over $\{N, L, R\}$ indicating the results of the steps given by the balance. The jth entry being L, R or N indicated that on the jth step the left (resp. right) hand was heavy or they were balanced (resp.). Now either a prefix of $\mathbf{b}$ is a row in the table and this row is unique by (T1), or a prefix of $-\mathbf{b}$ is a row in the table and there is only one such row by (T2). In either case we know

the false coin, being heavy in the first case and light in the second case. It is impossible that both situations should occur, by (T2). Since one of the coins is false, **b** or $-\mathbf{b}$ should appear on the table. So we have:

$$F_N(n) = 2 \times \begin{array}{l}\text{smallest number of nonempty entries in an}\\ n\text{-row table satisfying (T1), (T2), (T3).}\end{array}$$

Before we can construct the tables we need some more terminology: If B is table with entries L, N, R and empty, we denote by RB the array resulting from affixing an R at the beginning of each row in B. Similar definitions hold for NB, LB. Also, $-B$ is the array which we obtain on replacing each L entry in B by R and each R by L. For arrays B, C, we let

$$\begin{array}{c}B\\C\end{array}$$

stand for the array whose row set is the union of the row set of B with that of C. If $\mathbf{w}$ is a row in B and $\mathbf{w}_1, \cdots, \mathbf{w}_p$ are words over $\{L, N, R\}$, we denote by

$$\mathbf{w} \to \begin{cases}\mathbf{w}_1\\ \vdots\\ \mathbf{w}_p\end{cases}$$

the operation where the row $\mathbf{w}$ is deleted and the rows $\mathbf{w}_1, \cdots, \mathbf{w}_p$ are introduced into the array. Also N^r is the row of r consecutive N's, etc. Finally, the sum of all row vectors in B is denoted by $\sum B$, and $|B|$ is twice the number of nonempty entries in B.

We construct our tables for the F_N problem first. As usual, we represent $2n = 3^t + 2k + 1$ $(3^t - 1 \geqq k \geqq 0)$, and the corresponding table is called $A_{t,k}$. We start with the case of odd k. We describe a construction for $k = 1$ which is done by induction on t. The other tables for odd k are obtained by a simple procedure.

The case of even k is a little more involved. Using induction on t, we take care of the cases $k = 0, 2$. Most of the remaining cases can be handled easily using the $A_{t,2}$ table. The case $k = 3^t - 1$ is again handled separately by induction on t. Most of the (routine) calculations to ensure that conditions (T1), (T2), (T3) are satisfied and that the definitions make sense are omitted.

The solution for the S_N problem follows from the F_N case in a simple way.

We start the construction by defining a sequence of arrays B_t as follows:

$$B_1 = L \text{ (a one-by-one array with an } L \text{ entry).}$$

For $t \geqq 1$, define

$$B_{t+1} = \begin{cases}RB_t\\ LB_t\\ N(-B_t)\\ LN^t.\end{cases}$$

B_t is a $(3^t - 1)/2$ by t array satisfying (T1), (T2); it does not satisfy (T3) since $\sum B_t = L^t$. Note also that L^t is a row in B_t. Define now the array $A_{t,1}$ which is obtained from B_t by

$$L^t \to \begin{cases}L^tN\\ R^{t+1}\\ N^tL.\end{cases}$$

$A_{t,1}$ supplies an algorithm for $F_N(n)$, where $n=(3^t+3)/2$ and $|A_{t,1}|=2(tn+3)$ as needed.

For $2n=3^t+2k+1$, $k\geqq 3$ odd, say $k=2l+1$, we start from $A_{t,1}$ changing it into $A_{t,k}$ by picking l distinct rows $\mathbf{x}_1, \cdots, \mathbf{x}_l$ of length t and replacing each one of them:

$$\mathbf{x}_i \to \begin{cases} \mathbf{x}_i R \\ \mathbf{x}_i L \\ (-\mathbf{x}_i)N. \end{cases}$$

Note that $A_{t,k}$ satisfies (T1), (T2), (T3) and

$$|A_{t,k}|=F_N(n),$$

so $A_{t,k}$ supplies an optimal algorithm for the n coin problem.

Next we handle the case of even k. For $k=0$,

$$n=(3^t+1)/2, \qquad t\geqq 2,$$

change B_t into $A_{t,0}$ by the operations

$$L^t \to \begin{cases} L^tN \\ R^{t+1} \end{cases} \qquad \mathbf{x}\to \mathbf{x}(L),$$

where $\mathbf{x}$ is any row of length t in B_t and (L) indicates that this coin is placed on the left side of the balance on step $(t+1)$ only to have an equal number of coins on the right and left hands of the balance. If this coin is false, it will be known after t steps since conditions (T1), (T2) hold also if the (L) is omitted. Therefore we do not count the (L) in $|A_{t,0}|$. Hence $|A_{t,0}|=2(tn+2)$ and so it gives the optimal algorithm for $k=0$, $n=(3^t+1)/2$. For $k=2$ and $n=(3^t+5)/2$, change $A_{t,0}$ into $A_{t,2}$ by

$$\mathbf{x}(L)\to \begin{cases} \mathbf{x}L \\ (-\mathbf{x})L \\ \mathbf{x}N \end{cases} \qquad L^tN\to L^tR.$$

Now $|A_{t,2}|=|A_{t,0}|+2(2t+3)=2(nt+5)$, so $A_{t,2}$ solves the problem for $n=(3^t+5)/2$. For even k, $3^t-3\geqq k\geqq 4$, $k=2l+4$, we pick l distinct rows $\mathbf{x}_1, \cdots, \mathbf{x}_l$ of length t in $A_{t,2}$ and replace each of them:

$$\mathbf{x}_i \to \begin{cases} \mathbf{x}_i R \\ \mathbf{x}_i L \\ (-\mathbf{x}_i)N. \end{cases}$$

It is again routine to check that $A_{t,k}$ is an optimal solution table.

For the last remaining case, $n=(3^t-1)/2$, start from B_t, and to have condition (T3) hold replace

$$L^t\to N^tR, \qquad \mathbf{x}\to \mathbf{x}(L),$$

$\mathbf{x}$ being any row of length t and (L) is as explained above.

To complete the proof we only need to show that

$$S_N(n)\leqq 2nt+3k+2 \quad \text{for even } k.$$

(The case of odd k is already settled, since $F_N(n)\geqq S_N(n)$ is obvious.) To construct a table for this problem, if $n\neq(3^{t+1}-1)/2$, start with an $A_{t,k+1}$ table, which has $n+1$ rows and satisfies $|A_{t,k+1}|=2+(n+1)+3(k+1)+3$. Two of the rows in $A_{t,k+1}$ are L^tN

and R^{t+1}. Let $S_{t,k}$ be the table which results on performing

$$L^tN \to (L^tR), \qquad R^{t+1} \to R^t.$$

The row vector (L^tR) is the row for the standard coin. Now $S_{t,k}$ violates only condition (T2) in that $-R^t = L^t$ and L^t is a prefix of L^tR. But if the results of the weighings are R^t or L^t we know that the R^t coin is counterfeit, as the (L^tR) coin is known to be a standard coin. Thus we do not count the (L^tR) row, and we get

$$|S_{t,k}| = |A_{t,k+1}| - 2(t+1) - 2 = 2nt + 3k + 2,$$

as required.

At last, if $n = (3^{t+1} - 1)/2$, the table $\left\{\begin{smallmatrix} B_{t+1} \\ (R^{t+1}) \end{smallmatrix}\right.$, where (R^{t+1}) comes for the standard coin, supplies the required algorithm.

Acknowledgment. The first author wishes to express his thanks for the generous support of the Chaim Weizman postdoctoral grant.

REFERENCE

[Hu] D. A. HUFFMAN *A method for the construction of minimum redundancy codes*, Proc. IRE, 40, (Sept. 1952), p. 1098.

[Me] D. G. MEAD, *The average number of weighings to locate a counterfeit coin*, IEEE Trans. Inform. Theory, IT-25 (1974), pp. 616–617.

SIAM J. COMPUT.
Vol. 11, No. 3, August 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1103-0002 $01.00/0

$\Omega(n \log n)$ LOWER BOUNDS ON LENGTH OF BOOLEAN FORMULAS*

MICHAEL J. FISCHER†, ALBERT R. MEYER‡ AND MICHAEL S. PATERSON§

Abstract. A property of Boolean functions of n variables is described and shown to imply lower bounds as large as $\Omega(n \log n)$ on the number of literals in any Boolean formula for any function with the property. Formulas over the full basis of binary operations ($\wedge$, $\oplus$, etc.) are considered. The lower bounds apply to all but a vanishing fraction of symmetric functions, in particular, to all threshold functions with sufficiently large threshold and to the "congruent to zero modulo k" function for $k > 2$. In the case $k = 4$, the bound is optimal.

Key words. Boolean formula, length, size, symmetric function

1. Introduction. We describe a property of Boolean functions of n variables which implies lower bounds on the size of all Boolean formulas for functions with the property. Let C_k^n be the Boolean function "congruent to zero modulo k" of n arguments, that is, $C_k^n(x_1, \cdots, x_n)$ iff $\sum_{i=1}^n x_i \equiv 0 \pmod k$. We show that C_k^n has the property and conclude that there is a constant $\varepsilon > 0$ such that any Boolean formula for C_k^n over the full basis of binary and unary Boolean operations ($\wedge, \vee, \neg, \oplus$, NAND, etc.) is of length exceeding $\varepsilon n \log(n/k)$ for all $k \geqq 3$ and all n. There are formulas for C_4^n of length asymptotic to $n \log_2 n$ so our bound is achieved to within a constant multiple in this case.

The logarithm of the minimum length of a formula for a Boolean function gives the minimum *time*, i.e., depth, of a combinational circuit computing the function. This remark provides some technological motivation for our results. The depth of formulas is also related to the space and parallel time of computations and so is of basic concern in the theory of computational complexity; see [16], [13], [14], [1] for further discussion.

General counting arguments allow one to conclude that *most* Boolean functions of n variables require formulas of size asymptotic to $2^n/\log_2 n$ [21], [12], [11]. The largest lower bound provable for *explicit* examples, however, is proportional to $n^2/\log n$ by Neciporuk [15].[1] Although Neciporuk's method yields lower bounds for many explicit examples (cf. [16], [17]), no symmetric function possesses the property which implies Neciporuk's lower bounds. Hodes and Specker [4] provide another general property of functions which implies nonlinear lower bounds on the length of formulas, and Hodes [3] demonstrates that it is widely applicable.[2] For example, Hodes' and Specker's results imply that formulas for all but sixteen of the 2^{n+1} symmetric Boolean functions of n variables grow nonlinearly in n [8], [16], [17].

* Received by the editors November 19, 1980, and in revised form June 22, 1981. This work was supported in part by the National Science Foundation under grants MCS 7702474, MCS 7719754, MCS 8010707 and by a grant to the MIT Laboratory for Computer Science by the IBM Corporation. The results reported here appeared in weaker preliminary form in the Proceedings of the 7th Annual ACM Symposium on the Theory of Computing, 1975.

† University of Washington, Seattle, Washington, 98195.

‡ Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139.

§ University of Warwick, Coventry, England CV4 7AL.

[1] A slightly larger lower bound of $\Omega(n^2)$ is due to Khrapchenko [6] for the special basis of operations $\wedge$, $\vee$, $\neg$, but our results are concerned with formulas in which *all* binary operations may appear.

[2] Vilfan [22], [23] extends Hodes' and Specker's results to multivalued logic with arbitrary (not necessarily binary) operations and concludes, for example, that formulas for C_k^n grow nonlinearly in n using d-valued logic for $k > d!$.

Our main theorem resembles that of Hodes and Specker. We essentially show that any function which can be defined by a "small" formula can be restricted to a "large" subset of its variables so that the resulting restricted formula is equivalent to the sum modulo two of a subset of its variables. Since C_k^n, and indeed almost all symmetric functions, do not have such large simple restrictions, they cannot have small formulas. Comparing our results to Hodes' and Specker's in the most interesting case of symmetric functions, we note that their theorem yields nonlinear lower bounds whenever ours does, but their bounds are much smaller.[3] Indeed, our bounds of $\Omega(n \log n)$ are the largest lower bounds on formula length known for *any* symmetric Boolean function. (We remind the reader that $\alpha(n)=\Omega(\beta(n))$ iff $\beta(n)=O(\alpha(n))$ iff $\liminf \alpha(n)/\beta(n)>0$.)

In the next section, we state the main theorem giving lower bounds and apply it to C_k^n and a related example. In § 3 we derive a corollary which is easily applicable to arbitrary symmetric functions and then prove that all but a vanishing fraction of symmetric functions require formulas of length $\Omega(n \log n)$. Section 4 contains the proof of the main theorem. In the final § 5, we compare known upper and lower bounds on formula length and mention some open problems.

2. The lower bound. Boolean formulas over the *full unary-binary basis* are constructed from variables and constants (0 and 1) possibly using any of the unary and binary Boolean connectives ($\wedge$, $\vee$, $\neg$, $\oplus$, NAND, etc.). Let $L(f)$, the *length* of the formula f, be the number of occurrences of variables (not constants) in f. Let var (f) be the set of variables that appear in f. Formulas f and g are *equivalent*, denoted $f \equiv g$, if and only if f and g define the same function on var $(f) \cup$ var (g). Every Boolean formula g is easily shown to be equivalent to a Boolean formula f constructed from variables and constants using *only* the two connectives $\oplus$ ("exclusive or") and $\wedge$ ("and") such that f contains *exactly* the same number of occurrences of each variable as g. In particular, $L(g)=L(f)$, so without loss of generality, we henceforth consider Boolean formulas constructed from variables and constants using only $\oplus$ and $\wedge$.

An *assignment* A over a set of variables V is a partial map from V into $\{0, 1\}$; dom $(A) \subseteq V$ is the set of variables on which A is defined, i.e., the variables which A *fixes*. The *eccentricity* ecc (A) of an assignment A is the excess of 1's over 0's in the assignment, that is, ecc $(A)=|A^{-1}(1)|-|A^{-1}(0)|$. A is *central* if ecc (A) is zero or one. Given a formula f and an assignment A, the *restriction* $f|_A$ is the formula obtained by substituting $A(x)$ for each occurrence of x in f, where x ranges over dom (A). If A is central and dom $(A) \subseteq$ var (f), then $f|_A$ is called a *central restriction* of f.

The *dimension* dim (f) of f is the cardinality $|\text{var}(f)|$, of var (f). The formula f is *affine* if and only if f is equivalent to some formula of the form $\bigoplus W \oplus c$, where $c \in \{0, 1\}$ and $W \subseteq$ var (f). The theorem below shows that any Boolean formula of n variables, all of whose affine central restrictions have small dimension, has length $\Omega(n \log n)$. More precisely, let the *affine diameter* diam (f) of f be the largest dimension of any affine central restriction of f.

LOWER BOUND THEOREM. *There is an $\varepsilon>0$ such that for any Boolean formula f with n variables*

$$L(f) \geq \varepsilon n \log (n/\text{diam}(f)).$$

---

[3] Vilfan [22] notes that the nonlinear lower bounds of Hodes and Specker can be shown to be $O(n \log^* n)$, where $\log^* n$ is the least integer m such that

$$2^{2^{\cdot^{\cdot^{2}}}} \text{(height } m) \geqq n.$$

The theorem immediately applies to formulas for C_k^n. To see this, note that the only affine restrictions of C_k^n either are of dimension one or are equivalent to constant functions of dimension less than k, so diam $(C_k^n) < k$. Therefore

Example 1. $L(C_k^n) > \varepsilon n \log(n/k)$.

As another example, consider $n = km$ variables x_{ij} for $1 \leqq i \leqq k$, $1 \leqq j \leqq m$ and refer to the variables with second index j as the jth block of variables. Let p_j denote the mod 2 sum of the jth block, namely, $p_j = \bigoplus_{i=1}^k x_{ij}$, and let $f^{k,m}$ be the function $C_4^m(p_1, \cdots, p_m)$ of n variables. It is not hard to see that no restriction of a formula for $f^{k,m}$ which contains variables from three or more blocks is affine. Hence diam $(f^{k,m}) \leqq 2k$, so:

Example 2. $L(f^{k,m}) \geqq \varepsilon km \log(m/2)$ for ε as in the lower bound theorem.

We remark that choosing $k = n^{1-\delta}$ still yields $\Omega(n \log n)$ lower bounds on $L(f^{k,m})$ even though $f^{k,m}$ has "large" affine diameter $n^{1-\delta}$. This is an example where Hodes' and Specker's results do not apply.

To establish an upper bound in $L(C_4^n)$, let x denote Boolean variables $x_1, \cdots, x_n$. Construct formulas $D_0^n(x)$ and $D_1^n(x)$ for the low order and second lowest order digits of the binary representation of $\sum_{i=1}^n x_i$ as follows: $D_0^1(x_1) = x_1$ and $D_1^1(x_1) = 0$. Let y denote $x_{n+1}, \cdots, x_{2n}$. Then

$$D_0^{2n}(x, y) = \bigoplus_{i=1}^{2n} x_i$$

and

$$D_1^{2n}(x, y) = D_1^n(x) \oplus D_1^n(y) \oplus (D_0^n(x) \wedge D_0^n(y)).$$

Hence, $L(D_0^n) = n$ and $L(D_1^{2n}) = 2(L(D_1^n) + L(D_0^n))$. This recurrence implies that $L(D_1^n) \leqq n \log_2 n$ when n is a power of two. Now a formula for C_4^n is NOR (D_0^n, D_1^n), so $L(C_4^n) \leqq n(1 + \log_2 n)$ when n is a power of two. For arbitrary n, one can obtain a formula for C_4^n of length $n\lceil \log_2 n \rceil + 2n - 2^{\lceil \log_2 n \rceil}$, so:

PROPOSITION 1. *$L(C_4^n) < n\lceil 1 + \log_2 n \rceil$ for all n.*

Since $L(f^{k,m}) \leqq L(C_4^m) L(p_j)$ and $L(p_j) = k$, we also have

PROPOSITION 2. *$L(f^{k,m})$ is asymptotically at most $km \log_2 m$.*

So the lower bounds on L in Example 1 for $k = 4$ and in Example 2 are achievable to within a multiplicative factor.

3. Lower bounds for symmetric functions. For any Boolean formula f of dimension n which defines a symmetric function, there is by definition a *characteristic function*,

$$\chi_f: \{0, \cdots, n\} \to \{0, 1\}, \text{ such that } f(x_1, \cdots, x_n) = \chi_f\left(\sum_{i=1}^n x_i\right).$$

LEMMA. *If $\chi_f(\lfloor n/2 \rfloor) \neq \chi_f(\lfloor n/2 \rfloor + 2)$, then $L(f) \geqq \varepsilon n \log(n/2)$.*

Proof. The reader can easily verify that no *central* restriction of f which has three or more variables can be affine, viz., diam $(f) \leqq 2$. The bound on $L(f)$ now follows immediately from the lower bound theorem. □

SYMMETRIC FUNCTION LOWER BOUND THEOREM. *There is an $\varepsilon > 0$ such that for every formula f of dimension n which defines a symmetric function, if $\chi_f(k) \neq \chi_f(k+2)$ for some k, $0 \leqq k \leqq n-2$, then*

$$L(f) \geqq \varepsilon n \log \min(k, n-k).$$

Proof. Assume without loss of generality that $k \leqq n/2$. Let A be any assignment such that $|\mathrm{dom}(A) \cap \mathrm{var}(f)| = n - 2k$ and $A(x) = 0$ for all $x \in \mathrm{dom}(A)$. Now $\chi_{f|_A}(j) =$

$\chi_f(j)$ for $0 \leqq j \leqq 2k = \dim(f|_A)$, so applying the lemma above to $f|_A$ yields

$$L(f|_A) \geqq \varepsilon 2k \log(2k/2).$$

Therefore, at least one of the $2k$ variables of $f|_A$ occurs $\varepsilon \log k$ or more times in $f|_A$, and a fortiori also occurs that often in f.

By choosing dom (A) to be the $n-2k$ most frequently occurring variables in f, we conclude that each variable in dom (A) occurs at least $\varepsilon \log k$ times in f, so

$$L(f) \geqq (n-2k)\varepsilon \log k + L(f|_A) \geqq (n-2k)\varepsilon \log k + 2k\varepsilon \log k = \varepsilon n \log k. \quad \square$$

Let T_k^n be the threshold k function of n variables, that is,

$$T_k^n(x_1, \cdots, x_n) = 1 \quad \text{iff} \sum_{i=1}^{n} x_i \geqq k.$$

Since $\chi_{T_k^n}(k) = 0$ and $\chi_{T_k^n}(k+2) = 1$, we have:

Example 3. $L(T_k^n) \geqq \varepsilon n \log \min(k, n-k)$.

More generally, there are exactly $4 \cdot 2^{2b}$ symmetric functions f of n variables such that $\chi_f(k) = \chi_f(k+2)$ for all k, $b \leqq k \leqq n-b$. The preceding theorem implies a bound of $\varepsilon n \log b$ on length of formulas for the remaining $2^{n+1} - 4 \cdot 2^{2b}$ symmetric functions. Choosing any δ, $0 < \delta < 1$, and $b = \delta n$, we have:

COROLLARY. *The minimum formula length for all but* $o(2^{n+1})$ *of the* 2^{n+1} *symmetric functions of n variables is* $\Omega(n \log n)$.

Finally, we note that the symmetric function lower bound theorem also applies to nonsymmetric functions f as long as $\chi_f(k)$ and $\chi_f(k+2)$ are well defined, i.e., as long as

$$\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i = m \quad \text{implies } f(x_1, \cdots, x_n) = f(y_1, \cdots, y_n)$$

for $m = k, k+2$. For example, the length of formulas for *any* function which agrees with $T_{\lfloor n/2 \rfloor}^n$ on arguments of weight $\lfloor n/2 \rfloor$ and $\lfloor n/2 \rfloor + 2$ is $\Omega(n \log n)$.

4. Proof of the lower bound. The lower bound theorem follows directly from the main lemma below. The proof of the main lemma requires four elementary lemmas which are presented first.

Let f, g be formulas. We call g an *affine variant* of f if and only if $f \oplus g$ is affine. A formula f is an *r-formula* if no variable in f occurs more than r times; f is *r-minimal* with respect to some property of formulas if f is an r-formula and $L(f)$ is minimal among the r-formulas with the property. (Note that $L(f) \leqq r \dim(f)$ for any r-formula f, but this condition does not imply that f is necessarily an r-formula.)

AFFINE VARIANT LEMMA. *Let g be an affine variant of f. For all assignments A,*

(i) $f|_A$ *is affine if* $g|_A$ *is affine, and*

(ii) *if for some* $r \geqq 1$, *g is an r-minimal affine variant of f and* dom $(A) \subseteq$ var (g), then $\dim(f|_A) - \dim(g|_A) = \dim(f) - \dim(g)$.

Proof. (i) The formula $f \oplus g$ is affine by hypothesis; hence $f|_A \oplus g|_A$ is affine. If also $g|_A$ is affine, then adding the two together gives the affine function $f|_A$.

(ii) var $(g) \subseteq$ var (f), for if not, substituting constants for the variables in g which do not appear in f yields a shorter affine variant which is an r-formula, contradicting the r-minimality of g. The result follows easily. $\square$

An assignment B is an *extension* of A if B extends the partial function A. Let dom (B, A) denote dom (B) − dom (A), the set of new variables fixed by B.

CONJUNCTION LEMMA. *Given a central assignment A and a formula f such that $f|_A \equiv g \wedge h$, where g and h are affine, there is a central extension B of A such that* $\operatorname{dom}(B, A) \subseteq \operatorname{var}(f|_A)$, *$f|_B$ is affine, and* $\dim(f|_B) \geqq \dim(f|_A)/3$.

Proof. We have

$$g \equiv \bigoplus P \oplus \bigoplus R \oplus c \quad \text{and} \quad h \equiv \bigoplus Q \oplus \bigoplus R \oplus d,$$

where P, Q, R are disjoint subsets of $\operatorname{var}(f|_A)$ and $c, d \in \{0, 1\}$. Let B_1, B_2, B_3 be central extensions of A, fixing additionally the variables of $Q \cup R$, $P \cup R$, $P \cup Q$, respectively. Each of $f|_{B_i}$ for $i = 1, 2, 3$ is affine, and

$$\operatorname{var}(f|_{B_1}) \cup \operatorname{var}(f|_{B_2}) \cup \operatorname{var}(f|_{B_3}) = P \cup Q \cup R = \operatorname{var}(f|_A).$$

Hence, for some i, $\dim(f|_{B_i}) \geqq \dim(f|_A)/3$. □

PARTITION LEMMA. *Given sets $S_1, S_2, \cdots, S_t$, let T be the elements which occur in two or more of the S_i. That is,*

$$T = \bigcup_{i<j\leqq t} (S_i \cap S_j).$$

Then there exists a partition $\{\lambda, \mu\}$ of $\{1, \cdots, t\}$ such that if $L = \bigcup_{i\in\lambda} S_i$ and $M = \bigcup_{i\in\mu} S_i$, then $|L \cap M| \geqq |T|/2$.

Proof. Suppose $x \in T$; say $x \in S_i \cap S_j$. Then $x \in L \cap M$ for those partitions of $\{1, \cdots, t\}$ where i, j are on different sides, and hence, $x \in L \cap M$ for at least half the partitions. Therefore, the average size of $L \cap M$ over all partitions $\{\lambda, \mu\}$ is at least $|T|/2$, so the result certainly holds for some partition. □

We now contrive a function β with appropriate inductive properties for our main lemma using the well-known Catalan numbers.

BETA LEMMA. *There exist constants $\alpha > 0$, $a > 1$ such that if we define $\beta(r) = (\alpha a^r C_r)^{-1}$, where $C_r = \binom{2r-2}{r-1}/r$ is the Catalan number, then*

(i) $\beta(r) = \alpha / \sum_{s=1}^{r-1} (\beta(s)\beta(r-s))^{-1}$ *for* $r > 1$,

(ii) $\beta(r) \leqq (1 - 15\alpha)/6 < 1$ *for* $r \geqq 1$,

(iii) $\beta(r) \leqq (1 - 5\alpha)/(1 - 5\alpha + 4r)$.

Proof. (i) The Catalan numbers satisfy the convolution property

$$C_r = \sum_{s=1}^{r-1} C_s C_{r-s}$$

[10, § 2.3.4.4] from which the corresponding property (i) of β follows immediately.

(ii), (iii). Moreover, C_r is asymptotic to $dr^{-3/2}4^r$ for some fixed $d > 0$ [10, § 2.3.4.4]. This estimate makes it obvious that for any sufficiently small α one can choose a value for a which guarantees (ii) and (iii). Suitable values are $\alpha = 1/30$ and $a = 360$. □

MAIN LEMMA. *Let f be an r-formula with $r \geqq 1$, and let A_0 be a central assignment. There exists a central extension A of A_0 such that $f|_A$ is affine,* $\operatorname{dom}(A, A_0) \subseteq \operatorname{var}(f)$, *and*

$$\dim(f|_A) \geqq \beta(r) \cdot \dim(f|_{A_0}).$$

Proof of main lemma. The proof is by course-of-values induction on r. Hence, we assume $r \geqq 1$ and that the lemma holds for all r'-formulas with $r' < r$. To show the lemma holds for all r-formulas, we proceed, using a course-of-values subinduction on $L(f)$. Hence, we consider some r-formula f and some central assignment A_0 and, further, assume that the lemma holds for all r-formulas of length less than $L(f)$.

Suppose that g is an r-minimal affine variant of $f|_{A_0}$ and $L(g) < L(f)$. Then by the subinduction hypothesis, there is a central extension A of A_0 satisfying the lemma

for g. $f|_A$ is affine by the affine variant lemma (i). Moreover,

$$\begin{aligned}\dim(f|_A) &= \dim(g|_A) + (\dim(f|_{A_0}) - \dim(g)) && \text{by the affine variant lemma (ii),}\\ &\geqq \beta(r)\cdot\dim(g) + (\dim(f|_{A_0}) - \dim(g)) && \text{by induction,}\\ &\geqq \beta(r)\cdot\dim(f|_{A_0}) && \text{since } \beta(r)<1 \text{ by the beta lemma (ii).}\end{aligned}$$

This shows the lemma holds for f using the same A.

Hence, we can now assume that f itself is already an r-minimal affine variant of $f|_{A_0}$, and so, we have $f = f|_{A_0}$ and $\text{var}(f) \cap \text{dom}(A_0) = \varnothing$.

Since the lemma holds for f if and only if it holds for $1 \oplus f$, we may without loss of generality express f as $\bigoplus_{i=1}^k F_i$ where no F_i is constant or has $\oplus$ as its main connective. Clearly no F_i is affine since otherwise $(\bigoplus_{j\neq i} F_j)$ is an affine variant of f, contradicting the minimality of f. Hence, each F_i equals $G_i \wedge H_i$, and furthermore, the minimality of f ensures that neither of the formulas G_i nor H_i are equivalent to constant functions.

We define a partition of each set $\text{var}(F_i)$ into four sets as follows:

$$\text{global}(F_i) = \text{var}(F_i) \cap \left(\bigcup_{j\neq i} \text{var}(F_j)\right);$$

$$\text{joint}(G_i, H_i) = (\text{var}(G_i) \cap \text{var}(H_i)) - \text{global}(F_i);$$

$$\text{own}(G_i) = \text{var}(G_i) - (\text{joint}(G_i, H_i) \cup \text{global}(F_i));$$

$$\text{own}(H_i) = \text{var}(H_i) - (\text{joint}(G_i, H_i) \cup \text{global}(F_i)).$$

Let **global** $= \bigcup_i \text{global}(F_i)$, **joint** $= \bigcup_i \text{joint}(G_i, H_i)$ and **own** $= \bigcup_i (\text{own}(G_i) \cup \text{own}(H_i))$. So $\text{var}(f)$ is the disjoint union of **global**, **joint** and **own**.

The following four cases, defined solely in terms of the cardinalities of $\text{var}(f)$, **global**, **joint** and **own**, are obviously exhaustive. Let $n = \dim(f)$.

Case 1. $n \leqq 1/\beta(r)$. In this case we can take any central extension A of A_0 such that $\text{dom}(A, A_0) \subseteq \text{var}(f)$ and $\dim(f|_A) = 1$. Any formula in a single variable is necessarily affine.

Case 2. $|\textbf{global}| \geqq 2\alpha n$. Noting that **global** equals the set of variables which occur in two or more of the sets $\text{var}(F_i)$, we apply the partition lemma to the sets $\text{var}(F_i)$, $i = 1, \cdots, k$, and obtain a partition $\{\lambda, \mu\}$ of $\{1, \cdots, k\}$ such that

$$\left|\bigcup_{i\in\lambda} \text{var}(F_i) \cap \bigcup_{i\in\mu} \text{var}(F_i)\right| \geqq |\textbf{global}|/2 \geqq \alpha n.$$

Now f is equivalent to $L \oplus M$, where $L = \bigoplus_{i\in\lambda} F_i$, $M = \bigoplus_{i\in\mu} F_i$. We use the fact that all the variables of $\text{var}(L) \cap \text{var}(M)$ must occur fewer than r times in each of L, M, to invoke the main lemma successively on the two parts. For $1 \leqq s \leqq r-1$, let V_s be the set of those variables of $\text{var}(L) \cap \text{var}(M)$ which occur exactly s times in L (and hence at most $r-s$ times in M). For some t,

$$|V_t| \geqq \beta(r)n/(\beta(t)\beta(r-t))$$

since

$$\sum_{s=1}^{r-1} |V_s| = |\text{var}(L) \cap \text{var}(M)| \geqq \alpha n = \sum_{s=1}^{r-1} [\beta(r)n/(\beta(s)\beta(r-s))]$$

by the beta lemma (i).

Let B be any central extension of A_0 with $\operatorname{dom}(B, A_0) = \operatorname{var}(f) - V_t$, and let $L' = L|_B$, $M' = M|_B$. Thus, $\operatorname{var}(L') = \operatorname{var}(M') = V_t$. The main lemma applied to the t-formula L' yields an extension B' of B such that $L'|_{B'}$ is affine and

$$\dim(M'|_{B'}) = \dim(L'|_{B'}) > \beta(t) \cdot \dim(L') \geqq \beta(r)n/\beta(r-t).$$

The lemma applied to the $(r-t)$-formula $M'|_{B'}$ yields an extension A of B' such that $M'|_A$ is affine and $\dim(M'|_A) \geqq \beta(r-t) \cdot \dim(M'|_{B'}) \geqq \beta(r)n$. Since $(L \oplus M)|_A = ((L'|_{B'})|_A) \oplus (M'|_A)$ and is clearly affine and $\dim((L \oplus M)|_A) = \dim(M'|_A) \geqq \beta(r)n$, we have concluded the proof of Case 2.

Case 3. $|\mathbf{joint}| \geqq 3\alpha n$. Each variable in **joint** occurs in exactly one F_i and at least once but strictly fewer than r times in G_i and in H_i. We will restrict to a subset of these variables and then apply the induction hypothesis for smaller values of r.

Let $u_i = |\operatorname{joint}(G_i, H_i)|$. As in Case 2, there is some t_i, $1 \leqq t_i \leqq r-1$, such that if V_i is the set of variables in joint (G_i, H_i) that occur exactly t_i times in G_i (and, hence, at most $r - t_i$ times in H_i) then

$$|V_i| \geqq u_i \beta(r)/(\alpha\beta(t_i)\beta(r-t_i)).$$

Let B_0 be any central extension of A_0 fixing all the variables in $\operatorname{var}(f) - \bigcup_i V_i$, and let $F'_i = F_i|_{B_0}$, $G'_i = G_i|_{B_0}$, $H'_i = H_i|_{B_0}$. G'_i is a t_i-formula and H'_i is an $(r-t_i)$-formula.

We now proceed in k stages. At the ith stage, we find a central extension B_i of B_{i-1} such that $F'_i|_{B_i}$ is affine and $\dim(F'_i|_{B_i}) \geqq u_i\beta(r)/(3\alpha)$.

Stage i. Since G'_i is a t_i-formula, $1 \leqq t_i < r$, we apply the induction hypothesis to G'_i and B_{i-1} to obtain a central extension B'_i such that $G'_i|_{B'_i}$ is affine and $\dim(G'_i|_{B'_i}) \geqq \beta(t_i) \cdot \dim(G'_i)$. Since H'_i is an $(r-t_i)$-formula, $r - t_i < r$, we apply the induction hypothesis again to $H'_i|_{B'_i}$ to obtain a central extension B''_i such that $H'_i|_{B''_i}$ is affine and $\dim(H'_i|_{B''_i}) \geqq \beta(r-t_i) \cdot \dim(H'_i|_{B'_i})$. The restriction of an affine function is affine, so $G'_i|_{B''_i}$ is affine. Hence, by the conjunction lemma, there exists a central extension B_i of B''_i such that $F'_i|_{B_i}$ is affine and has dimension at least $\dim(F'_i|_{B''_i})/3$. In calculating the dimension of $F'_i|_{B_i}$, we make use of the fact that $\operatorname{var}(F'_i) = \operatorname{var}(G'_i) = \operatorname{var}(H'_i) = V_i$. We have

$$\begin{aligned}\dim(H'_i|_{B''_i}) \geqq \beta(r-t_i) \cdot \dim(H'_i|_{B'_i}) &= \beta(r-t_i) \cdot \dim(G'_i|_{B'_i}) \\ &\geqq \beta(r-t_i)\beta(t_i)|V_i| \geqq u_i\beta(r)/\alpha.\end{aligned}$$

Then

$$\dim(F'_i|_{B_i}) \geqq \dim(H'_i|_{B''_i})/3 \geqq u_i\beta(r)/(3\alpha).$$

Now let $A = B_k$, the central assignment obtained after the final stage. Note that $f|_A = \bigoplus_{i=1}^k F'_i|_{B_i}$, and so $f|_A$ is affine. Moreover,

$$\dim(f|_A) = \sum_i \dim(F'_i|_{B_i}) \geqq \sum_i u_i\beta(r)/(3\alpha) = |\mathbf{joint}| \cdot \beta(r)/3\alpha \geqq \beta(r)n$$

by the defining condition for this case. This concludes the proof of Case 3.

Case 4. $|\mathbf{own}| \geqq (1-5\alpha)n$ *and* $n > 1/\beta(r)$. In this case, we will find a central extension B of A_0 such that $\operatorname{dom}(B) \subseteq \operatorname{var}(f)$ and $f|_B$ is functionally independent of some nonempty subset V of its variables. Let yield $= |V|$ and cost $=$ yield $+ |\operatorname{dom}(B, A_0)|$. If yield $\geqq \beta(r) \cdot$ cost, then we can find a central extension A of B satisfying the lemma for f.

To see this, let g be the restriction of $f|_B$ obtained from some arbitrary assignment to V. Note that g is equivalent to $f|_B$ since $f|_B$ does not depend on V. Also, $\operatorname{var}(f)$ is the disjoint union of $\operatorname{dom}(B, A_0)$, V and $\operatorname{var}(g)$; in particular, $\dim(f) =$

cost+dim (g). Now $L(g)<L(f)$ since V is nonempty, so, by the subinduction hypothesis, there is a central extension A of B such that $g|_A$ is affine, dom $(A, B) \subseteq$ var (g) and dim $(g|_A) \geqq \beta(r) \cdot$ dim (g). But $f|_A$ is equivalent to $g|_A$, so $f|_A$ is also affine. Moreover, var $(f|_A)$ is the disjoint union of var $(g|_A)$ and V since dom $(A) \cap V = \emptyset$. Therefore,

$$\dim (f|_A) = \dim (g|_A) + \text{yield} \geqq \beta(r) \cdot \dim (g) + \beta(r) \cdot \text{cost} = \beta(r) \cdot \dim (f),$$

as required.

Thus, to complete the proof, we need only describe how to determine B and V.

Let $g_i = |\text{own}\,(G_i)|$, $h_i = |\text{own}\,(H_i)|$. Without loss of generality, we can assume $g_i \geqq h_i$. We note that $\sum_i g_i \geqq |\mathbf{own}|/2$.

For each i, we have two strategies which can remove the dependence of f on a subset of either own (G_i) or own (H_i). We will show below that at least one of these always has an adequate yield/cost ratio for some i.

Strategy A. This strategy is applicable only if there is a central extension of A_0 fixing only var (f) and making H_i equivalent to 0. Find a minimal central extension B of A_0 for which $H_i|_B \equiv 0$, var $(H_i) \subseteq$ dom $(B) \subseteq$ var (f) and dom $(B) \cap$ own (G_i) is as small as possible among such extensions. Since $F_i = G_i \wedge H_i$, we have $F_i|_B \equiv 0$, and so, $f|_B$ is independent of any remaining variables of own (G_i). Thus, V is own $(G_i)-$ dom (B).

Strategy B. This strategy is applicable only if there is a central extension of A_0 fixing only var $(H_i) \cup$ dom (A_0) and making H_i equivalent to 1. Find a maximal set $V \subseteq$ own (H_i) for which there is a central extension B of A_0 satisfying $H_i|_B \equiv 1$ and dom $(B, A_0) =$ var $(H_i) - V$. Since $f|_B$ is independent of V, the yield is $|V|$ and the cost is dim (H_i).

We begin our analysis by noting that since f is r-minimal no subformula of f is equivalent to a constant. Hence there is an extension B' of A_0 such that $H_i|_{B'} \equiv 0$. Let $d(H_i)$ be the least integer for which there is an extension B' of A_0 such that dom $(B', A_0) =$ var (H_i), $H_i|_{B'} \equiv 0$ and

$$-d(H_i) \leqq \text{ecc}\,(B') \leqq d(H_i) + 1.$$

Clearly, $d(H_i) \leqq$ dim (H_i).

Suppose Strategy A is applicable and B is the assignment required in the strategy. Let B' be the restriction of the partial function B to dom $(A_0) \cup$ var (H_i). Since B is minimal central such that $H_i|_B \equiv 0$, it must be that ecc (B') equals either $-d(H_i)$ or $d(H_i)+1$, and the variables in dom (B, B') are the minimal number which serve to extend B' to a central assignment. Hence,

$$|\text{dom}\,(B, A_0)| = \dim (H_i) + d(H_i).$$

Since B is defined to fix as few variables from own (G_i) as possible, either dom $(B) \cap$ own $(G_i) = \emptyset$ or own $(G_i) -$ dom $(B) =$ var $(f) -$ dom (B). Therefore,

$$\text{yield}_A = |\text{own}\,(G_i) - \text{dom}\,(B)| = \min\,(g_i, n - \dim (H_i) - d(H_i))$$

and

$$\text{cost}_A = \dim (H_i) + d(H_i) + \text{yield}_A = \min\,(g_i + \dim (H_i) + d(H_i), n).$$

If Strategy A is not applicable, let $\text{yield}_A = 0$ and $\text{cost}_A = n$, so the preceding formulas for cost_A and yield_A always hold.

If $\text{yield}_A/\text{cost}_A \geqq \beta(r)$ for some value of i, then Strategy A succeeds.

In any application of Strategy B, $|V| \geqq \min(h_i, d(H_i)-1)$. To see this, let V' be any subset of own (H_i) such that $|V'| = \min(h_i, d(H_i)-1)$, and let B' be any central extension of A_0 with $\mathrm{dom}(B') = \mathrm{dom}(A_0) \cup (\mathrm{var}(H_i) - V')$. Let C be an arbitrary assignment with $\mathrm{dom}(C) = V'$. Then

$$
\begin{aligned}
-d(H_i) &< -\min(h_i, d(H_i)-1) && \text{trivially,}\\
&= -|\mathrm{dom}(C)| &&\\
&\leqq \mathrm{ecc}(B' \cup C) && \text{since } B' \text{ is central,}\\
&\leqq |\mathrm{dom}(C)| + 1 && \text{since } B' \text{ is central,}\\
&= \min(h_i, d(H_i)-1) + 1 &&\\
&< d(H_i) + 1. &&
\end{aligned}
$$

By the minimality condition in the definition of d, $H_i|_{(B' \cup C)} \equiv 1$. This holds for any such C, so $H_i|_{B'} \equiv 1$, and $H_i|_{B'}$ does not depend on the variables in V'. Since Strategy B chooses V as large as possible, we have $|V| \geqq |V'| \geqq \min(h_i, d(H_i)-1)$ as desired.

Eliminating V from the expression of cost for Strategy B, we get

$$\mathrm{yield}_B \geqq \min(h_i, d(H_i)-1)$$

and

$$\mathrm{cost}_B = \dim(H_i).$$

If Strategy B is inapplicable, let $\mathrm{yield}_B = 0$ and $\mathrm{cost}_B = \dim(H_i)$. Note that in this case $d(H_i) = 0$, so the preceding formulas for yield_B and cost_B always hold.

If $\mathrm{yield}_B/\mathrm{cost}_B \geqq \beta(r)$ for some value of i, then Strategy B succeeds.

We prove by contradiction that there exists an i for which either Strategy A or Strategy B succeeds. Assume neither strategy succeeds for any i. Since Strategy A fails, $\mathrm{yield}_A/\mathrm{cost}_A < \beta$. (We omit the argument r from β in the remainder of this analysis.) So, for all i,

$$(1) \qquad (1-\beta)\min(g_i + \dim(H_i) + d(H_i), n) < \dim(H_i) + d(H_i).$$

Since Strategy B fails, $\mathrm{yield}_B/\mathrm{cost}_B < \beta$, so, for all i,

$$(2) \qquad \min(h_i, d(H_i)-1) < \beta \cdot \dim(H_i).$$

Let $m = |\mathbf{global} \cup \mathbf{joint}| \leqq 5\alpha n$. Counting up the sizes of the various sets and using the conditions for this case, we get

$$(3) \qquad d(H_i) \leqq \dim(H_i) \leqq m + h_i \leqq n - \sum_j g_j \leqq n - |\mathrm{own}|/2 \leqq (1+5\alpha)n/2.$$

From (3) and (2), we get

$$(4) \qquad d(H_i) - m - 1 \leqq \min(h_i, d(H_i)-1) < \beta \cdot \dim(H_i) \leqq \beta n.$$

Using (3), (4), the fact that $\beta n > 1$ and beta lemma (ii), we get

$$
(5) \qquad
\begin{aligned}
\dim(H_i) + d(H_i) &\leqq (1+5\alpha)n/2 + m + 1 + \beta n\\
&< (1+15\alpha)n/2 + 2\beta n \leqq (1-\beta)n.
\end{aligned}
$$

Assuming the "min" in (1) equals its second argument contradicts (5). Hence, the first argument is always the smaller, and (1) gives

$$(6) \qquad (1-\beta)g_i < \beta \cdot (\dim(H_i) + d(H_i)) \leqq 2\beta \cdot \dim(H_i) \quad \text{for all } i.$$

Therefore,

(7) $$(1-\beta)(1-5\alpha)n/2 \leqq (1-\beta)|\text{own}|/2 \leqq (1-\beta)\sum_i g_i < 2\beta \sum_i \dim(H_i) < 2\beta rn$$

since no variable occurs more than r times in all. Now (7) yields an immediate contradiction with beta lemma (iii).

We conclude that Strategy A or Strategy B succeeds for some i, completing this case and the proof of the main lemma. □

Proof of the lower bound theorem. Let f be a Boolean formula on n variables. Let $r=\lfloor 2L(f)/n \rfloor$, and let A_0 be a central assignment with

$$\text{dom}(A_0)=\{x \mid x \text{ occurs more than } r \text{ times in } f\}.$$

Since $f|_{A_0}$ is an r-formula, by the main lemma there is a central extension A of A_0 such that $f|_A$ is affine, $\text{dom}(A) \subseteq \text{var}(f)$ and

(8) $$\dim(f|_A) \geqq \beta(r) \cdot \dim(f|_{A_0}).$$

By the choice of A_0, $(r+1)\cdot|\text{dom}(A_0)| \leqq L(f)$, so

(9) $$\dim(f|_{A_0}) = n - |\text{dom}(A_0)| \geqq n - L(f)/(r+1) \geqq n/2.$$

Also,

(10) $$\beta(r) \geqq 2/K^r$$

for some large $K>1$ using the asymptotic estimate for C_r given in the proof of the beta lemma. Hence, from (8), (9), (10), we get

$$\dim(f|_A) \geqq (2/K^r)(n/2) = n/K^r.$$

Solving for r, we obtain

(11) $$r \geqq \log(n/\dim(f|_A))/\log K.$$

Therefore,

$L(f) \geqq rn/2$	by the choice of r,
$\geqq \varepsilon n \log(n/\dim(f\|_A))$	by (11) where $\varepsilon = 1/(2\log K)$,
$\geqq \varepsilon n \log(n/\text{diam}(f))$	by definition of diam (f). □

5. Conclusions and open problems. The conditions we have developed above for deducing lower bounds on length of formulas apply to many explicit examples but have their most interesting applications in the case of symmetric Boolean functions. Earlier results of Hodes and Specker [4] imply that, except for sixteen functions, the length of formulas for symmetric functions of n variables grows nonlinearly in n.[4] The results in this paper show that all but a vanishing fraction of the symmetric functions require formulas of length $\Omega(n \log n)$. These are the strongest known lower bounds on length of formulas for any symmetric functions.

Polynomial *upper* bounds on the length of formulas for symmetric functions were first obtained by Khrapchenko [7] and Meyer and Vilfan [22]. The smallest currently known upper bound is $o(n^{3.37})$ by Peterson [18], following earlier work of Pippenger

[4] The sixteen functions are all of the form

$$a \oplus b \bigoplus x_i \oplus c \prod x_i \oplus d \prod (1 \oplus x_i)$$

for $a, b, c, d \in \{0, 1\}$. Each of these obviously has a formula of length at most $3n$.

[19] and Paterson [17]. The constructions used to achieve the upper bounds are extensions of the construction given in § 2 of formulas for C_4^n.

It remains an open problem to improve these bounds. We note three particularly challenging instances of this general problem.

The construction of formulas for C_4^n extends in an obvious way to yield formulas of length $O(n(\log n)^{p-1})$ for $C_{2^p}^n$, but even for C_3^n the best upper bound we can obtain is $\Omega(n^2)$.

Problem 1. Is $L(C_3^n) = o(n^2)$?

The lower bound theorem above does not apply to threshold functions with bounded threshold, although Hodes' and Specker's theorem yields very slowly growing nonlinear bounds (cf. footnote 3). For fixed k, Khasin [5] and Pippenger [19], [20] have shown that $L(T_k^n) = O(n \log n)$.

Problem 2. Is $L(T_2^n) = o(n \log n)$?

The best currently known upper bound on length of formulas for the majority function $T_{\lfloor n/2 \rfloor}^n$ is the same as for arbitrary symmetric functions.

Problem 3. Is $n \log n = o(L(T_{\lfloor n/2 \rfloor}^n))$?

REFERENCES

[1] A. Borodin, *On relating time and space to size and depth*, this Journal, 6 (1977), pp. 733–744.

[2] M. J. Fischer, A. R. Meyer and M. S. Paterson, *Lower bounds on the size of Boolean formulas: preliminary report*, Proc. 7th Annual ACM Symposium on Theory of Computing, 1975, pp. 37–44.

[3] L. Hodes, *The logical complexity of geometric properties in the plane*, J. Assoc. Comput. Mach., 17 (1970), pp. 339–347.

[4] L. Hodes and E. Specker, *Lengths of formulas and elimination of quantifiers* I, in Contributions to Mathematical Logic, H. A. Schmidt, K. Schutte and H.-J. Thiele, eds., North-Holland, Amsterdam, 1968, pp. 175–188.

[5] L. S. Khasin, *Complexity bounds for the realization of monotone symmetrical functions by means of formulas in the basis* $\vee, \wedge, \neg$, Dokl. Akad. Nauk SSSR, 189, 4 (1969), pp. 752–755; Soviet Physics Dokl., 14, 12 (1970), pp. 1149–1151.

[6] V. M. Khrapchenko, *On the complexity of the realization of the linear function in the class of* π*-circuits*, Mat. Zametki 9, 1 (1971), pp. 35–40 (in Russian). (A translation appears in [22].)

[7] ———, *The complexity of realization of symmetrical functions by formulae*, Mat. Zametki, 11, 1 (1972), pp. 109–120), Math. Notes on the Academy of Sciences of the USSR, 11 (1972), pp. 70–76.

[8] ———, *Complexity of realisation of symmetric algebraic logic functions on finite bases*, Problemy Kibernet, 31 (1976), pp. 231–234 (in Russian).

[9] M. Kleiman and N. Pippenger, *An explicit construction of short monotone formulas for the monotone symmetric functions*, Theoret. Comput. Sci., 7 (1977), pp. 325–332.

[10] D. E. Knuth, *The Art of Computer Programming, Vol.* 1, *Fundamental Algorithms*, Addison–Wesley, Reading, MA, 1973.

[11] R. E. Krichevskii, *Realizations of functions by superpositions*, in Prob. Cybernetics II, Pergamon Press, New York, 1961, pp. 458–477.

[12] O. B. Lupanov, *Complexity of formula realisation of functions of logical algebra*, Problemy Kibernet, 3 (1960), pp. 61–80; Problems of Cybernetics, 3 (1962), pp. 782–811.

[13] W. F. McColl, *Complexity hierarchies for Boolean functions*, Acta Informatica, 11 (1978), pp. 71–77.

[14] ———, *The circuit depth of symmetric Boolean functions*, J. Comput. System Sci., 17 (1978), pp. 108–115.

[15] E. I. Neciporuk, *A Boolean function*, Soviet Math. Dokl., 2, 4 (1966), pp. 999–1000.

[16] M. S. Paterson, *An introduction to Boolean function complexity*, Stanford Computer Science Report STAN-CS-76-557, Stanford University, Stanford, CA, 1976; also in Asterisque (journal of the French Mathematical Society) 38, 39 (1976), pp. 183–201.

[17] ———, *New bounds on formula size*, Proc. 3rd GI Conf. Informatik, Darmstadt, Lecture Notes in Computer Science, 48, Springer-Verlag, New York, 1977, pp. 17–26.

[18] G. L. Peterson, *An upper bound on the size of formulae for symmetric Boolean functions*, extended abstract, Dept. Computer Science Tech. Report 78-03-01, Univ. of Washington, Seattle, 1978.

[19] N. Pippenger, *Short formulae for symmetric functions*, IBM Research Report RC-5143, Yorktown Heights, NY, 1974.

[20] ———, *The realization of monotone Boolean functions*, Proc. 8th Annual ACM Symposium on Theory of Computing, 1976, pp. 204–209.

[21] J. Riordan and C. E. Shannon, *The number of two-terminal series-parallel networks*, J. Math. and Phys., 21 (1942), pp. 83–93.

[22] B. Vilfan, *The complexity of finite functions*, Ph.D. Thesis, Dept. Electrical Engineering, Tech. Rep. 97, Project MAC, Massachusetts Institute of Technology, Cambridge, MA, 1972.

[23] ———, *Lower bounds or the size of expression for certain functions in d-ary logic*, Theoret. Comput. Sci., 2 (1976), pp. 249–269.

SIAM J. COMPUT.
Vol. 11, No. 3, August 1982
© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1103-0003 $01.00/0

ON THE AVERAGE-CASE COMPLEXITY OF SELECTING THE kTH BEST*

ANDREW C. YAO† AND F. FRANCES YAO‡

Abstract. Let $\bar{V}_k(n)$ be the minimum average number of pairwise comparisons needed to find the kth largest of n numbers ($k \geqq 2$), assuming that all $n!$ orderings are equally likely. D. W. Matula proved that, for some absolute constant c, $\bar{V}_k(n) - n \leqq ck \ln \ln n$ as $n \to \infty$. In the present paper, we show that there exists an absolute constant $c' > 0$ such that $\bar{V}_k(n) - n \geqq c'k \ln \ln n$ as $n \to \infty$, proving a conjecture of Matula.

Key words. algorithm, average-case, binary tree, comparison, complexity, decision tree, selection

1. Introduction. The problem of selecting the kth largest in a set of n numbers by pairwise comparisons has been a subject of considerable interest (e.g., Knuth [6], [8]). Two particularly interesting situations are the fixed-k case ($n \to \infty$) and the median-finding problem ($k = \lceil n/2 \rceil$). Let $V_k(n)$ denote the complexity of selection in the worst case, and $\bar{V}_k(n)$ the average-case complexity assuming that all $n!$ permutations are equally likely. Table 1 summarizes the known results.[1]

TABLE 1
A summary of known results on selection problems.

	fixed k ($n \to \infty$)	median[2]
$V_k(n)$	$V_k(n) - n = (k-1) \lg n + f(k)$ [2] [4] [5] [10]	$3n \geqq V_{n/2}(n) \geqq 1.75n$ [10] [11]
$\bar{V}_k(n)$	$ck \ln \ln n \geqq \bar{V}_k(n) - n \geqq ?$ [9]	$1.5n \geqq \bar{V}_{n/2}(n) \geqq 1.375n$ [1]

As seen from the table, no good lower bound is known for the fixed-k behavior of $\bar{V}_k(n)$. It was not even known whether $\bar{V}_2(n) - n \to \infty$ as $n \to \infty$.

An early exploration on $\bar{V}_2(n)$ was done by Sobel [13]. In 1973, Matula [9] devised an elegant algorithm which finds the kth largest using $n + ck(\ln \ln n)$ comparisons on the average; and he conjectured that the $k(\ln \ln n)$ term cannot be further reduced. In this paper, we prove that $\bar{V}_k(n) - n \geqq c'k(\ln \ln n)$, thus confirming the conjecture. As a result, $\bar{V}_k(n) - n$ is determined to within a constant factor asymptotically.

MAIN THEOREM. *For every integer $k \geqq 2$, there exists a number N_k such that $\bar{V}_k(n) - n \geqq \frac{1}{2}k(\ln \ln n - \ln k - 9)$ for all $n \geqq N_k$.*

In § 2 some basic concepts are introduced. In § 3 we illustrate certain aspects of the proof by showing a weaker form of the theorem in the case $k = 2$, under a severe "regularity" constraint on the class of allowed algorithms. In § 4 we examine the difficulties encountered in extending the discussion to include nonregular algorithms. We then introduce some new concepts and prove a crucial result (the limited-anomaly theorem) to prepare for the proof of the main theorem, which is completed in § 5.

* Received by the editors April 20, 1979, and in final revised form May 1, 1981. This research was supported in part by the National Science Foundation under grants MCS-72-03752 A03 and MCS-77-05313.

† Computer Science Department, Stanford University, Stanford, California 94305.

‡ Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304.

[1] In this paper, we use lg to stand for logarithm with base 2.

[2] These results have generalizations for the case $k = \alpha n$ with any fixed $0 < \alpha < 1$.

2. The accounting schemes. An algorithm for selecting the kth largest of n (distinct) elements $X=\{x_1, x_2, \cdots, x_n\}$ is a binary decision tree T [8]. Associated with each internal node v is a comparison between two elements x_i, x_j. We will say "v compares x_i, x_j", and use the notation comp $(v)=(x_i : x_j)$. The branching at v is determined by whether $x_i<x_j$ or $x_i>x_j$. By analogy with a tennis tournament that selects the kth best of n players, we will freely use in this paper descriptions such as "x_i defeats x_j" (if $x_i>x_j$), "x_i is undefeated (so far)", etc.

Any particular ordering σ satisfied by the input, i.e., $x_{\sigma(1)}>x_{\sigma(2)}>\cdots>x_{\sigma(n)}$, determines a path from the root to a leaf in T. Let $S(\sigma)$ denote the sequence of internal nodes on this path; and let $s(\sigma)=|S(\sigma)|$, the number of comparisons made. The *average cost* of T is

$$\text{COST}\,(T)=\frac{1}{n!}\sum_{\sigma} s(\sigma). \tag{2.1}$$

The *average-case complexity* $\bar{V}_k(n)$ of selecting the kth best of n is the minimum cost COST (T) among all decision trees. Without loss of generality, we consider only algorithms that make no *redundant* comparisons (i.e., comparisons whose results can be deduced from comparisons made previously).

Let T be any algorithm. We consider two types of *noncrucial comparisons*: for each input ordering σ, let $S_1(\sigma)$ be the set of comparisons made by T in which the loser has been defeated previously, and $S_2(\sigma)$ the set of comparisons involving at least one player ranking in the top $k-1$. We shall write $s_i(\sigma)=|S_i(\sigma)|$ $(i=1,2)$. Note that a comparison can be in both $S_1(\sigma)$ and $S_2(\sigma)$. As each player except the top k must encounter a first defeat, we have

$$s(\sigma)\geqq n-k+s_1(\sigma). \tag{2.2}$$

Also, because each player not in the top k must lose to some player ranking below the top $(k-1)$, we have

$$s(\sigma)\geqq n-k+s_2(\sigma). \tag{2.3}$$

Formulas (2.1), (2.2), (2.3) lead to

$$\text{COST}\,(T)\geqq n-k+\frac{1}{n!}\sum_{\sigma} s_1(\sigma), \tag{2.4}$$

and

$$\text{COST}\,(T)\geqq n-k+\frac{1}{2}\frac{1}{n!}\sum_{\sigma}(s_1(\sigma)+s_2(\sigma)). \tag{2.5}$$

We will transform (2.5) into another form. For each internal node v, let $q_i(v)$ $(i=1,2)$ be the probability that comp (v) is a noncrucial comparison of type i. Precisely, if we let $\Gamma(v)=\{\sigma\,|\,S(\sigma)\text{ contains }v\}$ and $\Gamma_i(v)=\{\sigma\,|\,\sigma\in\Gamma(v), \text{comp}\,(v)\in S_i(\sigma)\}$ $(i=1,2)$, then

$$q_i(v)=\frac{|\Gamma_i(v)|}{|\Gamma(v)|}.$$

We define further

$$q(v)=q_1(v)+q_2(v),$$

and

$$\alpha(\sigma)=\sum_{v\in S(\sigma)} q(v).$$

Then

$$(2.6)\quad \sum_{\sigma}(s_1(\sigma)+s_2(\sigma))=\sum_{v\in T}(|\Gamma_1(v)|+|\Gamma_2(v)|)=\sum_{v\in T}|\Gamma(v)|q(v)=\sum_{\sigma}\sum_{v\in S(\sigma)}q(v)=\sum_{\sigma}\alpha(\sigma).$$

We obtain from (2.5) and (2.6),

$$(2.7)\qquad \text{COST}(T)\geqq n-k+\frac{1}{2}\,\frac{1}{n!}\sum_{\sigma}\alpha(\sigma).$$

We collect (2.4) and (2.7) in the following lemma.

LEMMA 2.1.

$$(2.8)\qquad \text{COST}(T)\geqq n-k+\frac{1}{n!}\sum_{\sigma}s_1(\sigma),$$

$$(2.9)\qquad \text{COST}(T)\geqq n-k+\frac{1}{2}\,\frac{1}{n!}\sum_{\sigma}\alpha(\sigma).$$

We can think of the two formulas in the above lemma as two counting methods for the comparisons. The first one is *direct counting*, while the other is *distributive counting* as the cost is "distributed" to the internal nodes of the decision tree. To illustrate the utility of these alternative counting methods, we can combine the two formulas to obtain

$$(2.10)\qquad \text{COST}(T)\geqq n-k+\frac{1}{3}\cdot\frac{1}{n!}\sum_{\sigma}(s_1(\sigma)+\alpha(\sigma)).$$

Our aim will be, roughly speaking, to show that for any permutation σ,

$$(2.11)\qquad s_1(\sigma)+\alpha(\sigma)\geqq \text{const.}\times k\ln\ln n.$$

That is, for any computation sequence $S(\sigma)$, either itself contains a large number $s_1(\sigma)$ of noncrucial comparisons, or it will effect a large number $\alpha(\sigma)=\sum_{v\in S(\sigma)}q(v)$ of noncrucial comparisons distributed over other paths. However, in the proof we shall not be using (2.10) and (2.11), but rather Lemma 2.1 itself, in order to obtain better coefficients of $k\ln\ln n$ in the lower bounds.

Remark. The quantities $s(\sigma), s_i(\sigma), \alpha(\sigma), \cdots$ all depend on T; we have suppressed this dependence in our notations for simplicity.

3. Regular algorithms.

3.1. Introduction. In this section we shall prove a weaker form of the Main Theorem for $k=2$, under certain "regularity" constraints on the algorithms under consideration.

We begin with a discussion about general algorithms. Let T be any decision tree algorithm selecting the kth largest of $X=\{x_1, x_2, \cdots, x_n\}$. One can view the computation process for any input ordering σ as building up successively larger partial orders on X. Formally we associate with each node v in T a partial order $P(v)$, which is the transitive closure of all the relations $x_i>x_j$ obtained on the path from the root of T to v (prior to performing the comparison at v). We call comp $(v)=(x_i:x_j)$ a *joining comparison* if x_i and x_j belong to different connected components in $P(v)$. At each leaf l, $P(l)$ must contain only a single component, otherwise the relative order of

elements in different components can change the identity of the kth largest element. Thus, there are exactly $n-1$ joining comparisons comp (v) in the sequence $v \in S(\sigma)$ for any σ; we denote the subsequence of these nodes v by $S'(\sigma)$.

Clearly x is a maximal element in the partial order $P(v)$ if and only if x is yet undefeated. A component C of a partial order is said to be *anomalous* if C has more than one maximal element. A maximal element x in $P(v)$ is *anomalous* if x is in an anomalous component, and *normal* otherwise. A partial order is *anomalous* if it contains an anomalous component. Fig. 1 shows an anomalous partial order with C_1 being an anomalous component, x_2 a normal element, and x_1, x_3 two anomalous elements.

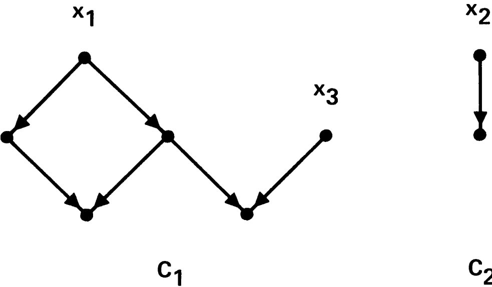


FIG. 1. *An anomalous partial order.*

We now define the notion of regular algorithms, in which the choice of a comparison comp (v) is restricted by the current partial order $P(v)$.

DEFINITION 3.1. An algorithm T is *regular* if no joining comparison can involve an anomalous (maximal) element.

In particular, any algorithm that removes anomalies in partial order as soon as they occur is regular. For instance, suppose the current partial order $P(v)$ is as shown in Fig. 2 and comp $(v) = (x_1 : x_3)$ is performed with result $x_1 > x_3$, thereby creating an anomalous partial order. By choosing the next comparison to be $(x_1 : x_2)$, we can immediately remove the anomaly independent of the outcome. Matula's algorithm [9] for $k = 2$ is of this type.

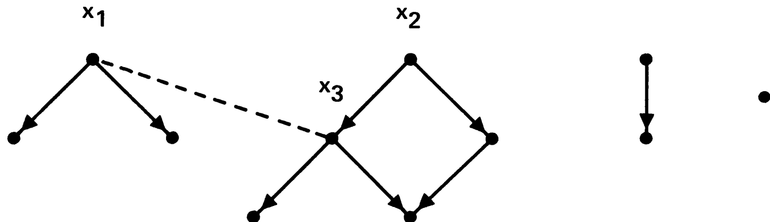


FIG. 2. *Creation and removal of an anomaly.*

The rest of this section is devoted to proving the following result.

THEOREM 3.1. *Let T be a regular algorithm for selecting the second largest element of $\{x_1, x_2, \cdots, x_n\}$. Then*

$$\text{COST}(T) - n \geqq \tfrac{1}{2} \ln \ln n - 6.$$

3.2. Some properties of binary trees. We digress to discuss some useful facts about binary trees.

Let M be a binary tree. We use M_I to denote the set of internal nodes. For each node u, we use notations father $[u]$, brother $[u]$, lson $[u]$, rson $[u]$ for the father, brother, leftson, rightson of u, respectively. Let $D(u)$ be the set of internal-node-descendants of u, and $D_L(u)$ the set of leaf-descendants (u is also considered to be a descendant of u). The *weight* $w(u)$ is the number of leaf-descendants of u; thus $w(u)=|D_L(u)|=|D(u)|+1$, and for any leaf u, $w(u)=1$. The *external path length* is defined as $E(M)=\sum_{u\in M_I} w(u)$.

LEMMA 3.2. *Let M be any binary tree with n leaves, then $E(M)\geqq n(\lg n-1)$.*

Proof. From Knuth [7, §2.3.4.5 Eqs. (3) and (4)], one has $E(M)\geqq n\lfloor \lg n\rfloor -2n+2+2(n-1)\geqq n(\lg n-1)$. □

Let $H_n=\sum_{1\leqq i\leqq n} 1/i$ be the *harmonic numbers* (see [7]). It is clear that

$$H_n-H_{n'}=\frac{1}{n'+1}+\frac{1}{n'+2}+\cdots+\frac{1}{n}\geqq\int_{n'+1}^{n+1}\frac{1}{x}\,dx;$$

therefore

$$H_n-H_{n'}\geqq \ln\left(\frac{n+1}{n'+1}\right),\quad \text{for } n\geqq n'\geqq 0. \tag{3.1}$$

DEFINITION 3.2. Let M be a binary tree. A subset of nodes V is called a *cross section* of M if root $\notin V$ and the following condition is true: For any two distinct $u_i, u_j\in V$, father $[u_i]\neq$ father $[u_j]$ and u_i, u_j have no common descendants.

LEMMA 3.3. *If V is a cross section of a binary tree M with n leaves, then*

$$\sum_{u\in V}\frac{w(u)}{w(\text{brother }[u])}\geqq \ln\left(\frac{n+1}{n-W+1}\right),$$

where $W=\sum_{u\in V} w(u)$.

Proof. For each node u of M, use u' to denote brother $[u]$ when it exists (i.e., when $u\neq$ root). Let depth (u) be the distance from the root to a node u, with depth (root) $=0$. We sort the nodes in the cross section V in decreasing order of the depth as $u_1, u_2, \cdots, u_t$; i.e., $i<j$ implies depth $(u_i)\geqq$ depth (u_j).

FACT A. *For any $i\leqq j$, u_i' and u_j have no common descendants.*

Proof of Fact A. The case $i=j$ is trivial, as u_i' and u_j are brothers. Assume $i<j$, which implies depth $(u_i')=$ depth $(u_i)\geqq$ depth (u_j). If u_i' and u_j have any common descendants, then u_i', and hence u_i, must be a descendant of u_j. But this is ruled out since V is a cross section. □

From Fact A, we have for $1\leqq i\leqq t$,

$$w(u_i')\leqq n-\sum_{i\leqq j\leqq t} w(u_j)=n-W+\sum_{1\leqq j<i} w(u_j).$$

Let $W(i)=\sum_{1\leqq j\leqq i} w(u_j)$; then

$$\frac{w(u_i)}{w(u_i')}\geqq\frac{w(u_i)}{n-W+W(i-1)}\geqq\sum_{1\leqq j\leqq w(u_i)}\frac{1}{n-W+W(i-1)+j}.$$

Therefore,

$$\sum_{u\in V}\frac{w(u)}{w(u')}=\sum_{1\leqq i\leqq t}\frac{w(u_i)}{w(u_i')}\geqq\sum_{1\leqq j\leqq W}\frac{1}{n-W+j}=H_n-H_{n-W}.$$

Lemma 3.3 then follows from formula (3.1). □

For later reference, we also include here the following simple fact.

FACT B. *Let $a_1, a_2, \cdots, a_t$ be positive numbers. Then*

$$\sum_{1 \leqq i \leqq t} a_i \lg a_i \geqq t(\bar{a} \lg \bar{a}),$$

when $\bar{a} = (\sum_i a_i)/t$.

Proof. The function $x \lg x$ is convex for $x > 0$. □

3.3. Merge-trees and the proof of Theorem 3.1. Let T be a regular algorithm that selects the second best of n players. We shall show that, for any σ,

$$\sum_{v \in S'(\sigma)} q(v) \geqq \ln \ln n - 7. \tag{3.2}$$

This immediately implies Theorem 3.1, since by Lemma 2.1,

$$\text{COST}(T) \geqq n - 2 + \tfrac{1}{2}(\ln \ln n - 7) \geqq n + \tfrac{1}{2} \ln \ln n - 6.$$

The basis for proving (3.2) is the following bound on $q(v)$.

LEMMA 3.4. *For a regular algorithm T, let $v \in T$ and* comp $(v) = (x_i : x_j)$ *be a joining comparison between elements in two components of sizes c_1, c_2, respectively. Then*

$$q(v) \geqq \min\left\{\frac{c_1}{c_1 + c_2}, \frac{c_2}{c_1 + c_2}, \frac{c_1 + c_2}{n}\right\}.$$

Proof. Recall that $q(v) = q_1(v) + q_2(v)$. There are four cases. If x_i and x_j are both undefeated, then $q_2(v) \geqq (c_1 + c_2)/n$ as the larger of x_i, x_j will be the largest of all elements with probability $(c_1 + c_2)/n$. If neither is undefeated, then $q_i(v) = 1 > c_1/(c_1 + c_2)$. If x_i is undefeated and x_j is not, then $q_1(v) = (\text{Probability that } x_i > x_j) \geqq c_1/(c_1 + c_2)$. If x_j is undefeated and x_i is not, then $q_1(v) \geqq c_2/(c_1 + c_2)$ by the same token. Thus the lemma is true in all cases. □

We shall now apply the lower bound on $q(v)$ to prove (3.2). We construct an auxiliary binary tree that represents the successive joining operations performed in $S'(\sigma)$, and then use results obtained in § 3.2.

Merge-tree. Let σ be an input ordering to algorithm T. We can construct a binary tree $M(\sigma)$ corresponding to $S'(\sigma)$ with the following properties.

(1) $M(\sigma)$ has n leaves labeled by the n input elements $X = \{x_1, x_2, \cdots, x_n\}$.

(2) Each internal node u of $M(\sigma)$ corresponds to a $v \in S'(\sigma)$; the x_i's that are descendants of lson $[u]$ and rson $[u]$ respectively form the two components that are joined by the comparison at v.

An example of a merge-tree is shown in Fig. 3.

Let $C(u)$ denote the subset of X which label the leaf-descendants of node u in $M(\sigma)$. Define a function φ on $M(\sigma)_I$, the set of internal nodes of $M(\sigma)$, by letting $\varphi(u) = q(v)$ if u corresponds to $v \in S'(\sigma)$. We wish to prove the following equivalent formula of (3.2).

$$\sum_{u \in M(\sigma)_I} \varphi(u) \geqq \ln \ln n - 7. \tag{3.3}$$

By Lemma 3.4, we have for each $u \in M(\sigma)_I$,

$$\varphi(u) \geqq \min\left\{\frac{w_1}{w_1 + w_2}, \frac{w_2}{w_1 + w_2}, \frac{w_1 + w_2}{n}\right\}, \tag{3.4}$$

where $w_1 = w(\text{lson}[u])$ and $w_2 = w(\text{rson}[u])$. Therefore, Theorem 3.1 will follow from the following result.

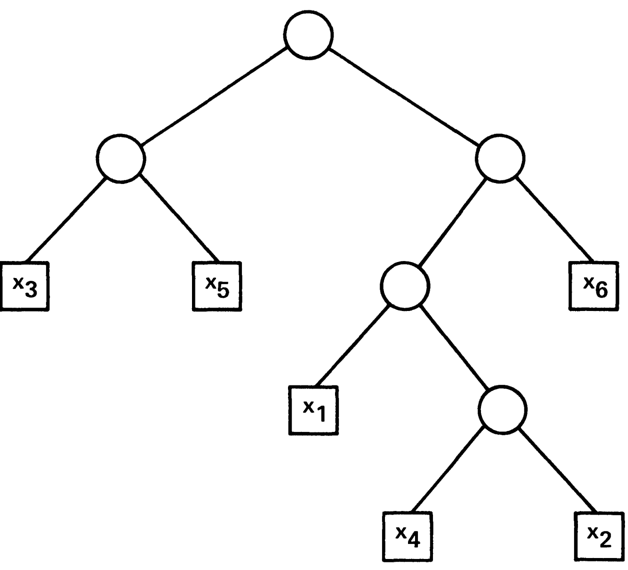


FIG. 3. *The merge-tree $M(\sigma)$ corresponding to the sequence of joining comparisons* $((x_3:x_5), (x_4:x_2), (x_1:x_4), (x_6:x_2), (x_3:x_1))$.

LEMMA 3.5. *Let M be any binary tree with n leaves. For each $u \in M_I$, let $g(u) = \min\{(w_1+w_2)/n,\ w_1/(w_1+w_2),\ w_2/(w_1+w_2)\}$ where $w_1 = w(\text{lson}[u])$ and $w_2 = w(\text{rson}[u])$. Then*

$$\sum_{u\in M_I} g(u) \geqq \ln \ln n - 7.$$

Proof. The proof makes use of the lemmas in § 3.2. It is given in Appendix A because of its length. □

3.4. Remarks. The lower bound given in Theorem 3.1 is only about half as large as the corresponding bound in the main theorem. This is due to the use of a relatively loose bound for $q(v)$ in Lemma 3.4. A stronger bound for $q(v)$ will be used in the general proof in § 5, where the regularity constraint is also dropped.

We also wish to point out that (2.8), the first formula in Lemma 2.1, was not used in the above proof, but will be needed later in the proof for the general case.

4. The limited-anomaly theorem. The arguments in the previous section fail when algorithms are not required to be regular. The important assertion in Lemma 3.4 is no longer true. Consider the partial order $P(v)$ exhibited in Fig. 4, and suppose that the next comparison v is between x_1 and an anomalous maximal element x_2. Assuming that the components C_1, C_2 and C_3 have sizes 5, 102 and n-107 respectively, it is intuitively clear that the probability $q_2(v)$ is less than $(5+102)/n$, as $\max\{x_1, x_2\}$ is unlikely to be the largest among elements in $C_1 \cup C_2$. It will be seen later (§ 5.3) that, in estimating $q_2(v)$, one should use $f(x_2)$, the number of elements in $P(v)$ that are less than (or equal to) x_2 but *not* less than any other maximal elements, in place of the component size $|C_2|$. In this example $f(x_2) = 4$ and thus $q_2(v) \geqq (5+4)/n$, a much weaker lower bound than $(5+102)/n$. Therefore, two complications arise when non-regular algorithms are considered. Firstly, it was previously possible to attach a lower bound to $q(v)$ which depended only on the shape of the associated merge tree; now

more details of the partial order $P(v)$ must be taken into account. Secondly, when comparisons involving anomalous elements x_i occur, we may obtain very weak bounds on $q(v)$, if $f(x_i)$ is small. We shall presently prove a result to overcome the second difficulty, by stating that comparisons involving an anomalous maximal element x_i with a small $f(x_i)$ cannot happen too often unless COST (T) is large anyway.

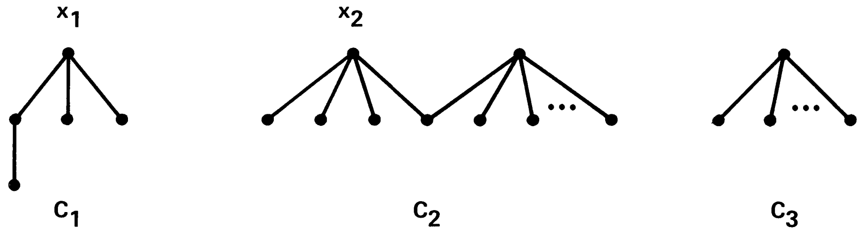


FIG. 4. *Difficulties caused by anomaly.*

Let P be a partial order on $X=\{x_1, x_2, \cdots, x_n\}$. For each x_i, let $H(x_i)$ be the component containing x_i, and $h(x_i)=|H(x_i)|$. For any maximal element x_i, the *fiefdom* of x_i, $F(x_i)$ is the set $\{x_j \mid x_j \leqq x_i$ (in P), and x_j is not less than any other maximal element in $P\}$. We denote $|F(x_i)|$ by $f(x_i)$. Note that $F(x_i) \subseteq H(x_i)$, and the containment is proper if and only if x_i is anomalous. When x_i is anomalous, we call $f(x_i)$ the *anomaly degree* of x_i.

Let T be an algorithm that selects the kth largest of n elements. For any internal node $v \in T$, the comparison at v, $x_i : x_j$, is said to be *anomalous of degree* m if the minimum anomaly degree of x_i and x_j is m. (Equivalently, it means that one of x_i and x_j has an anomaly degree m, and the other one is either not anomalous or anomalous with degree at least m.)

THEOREM 4.1 (the limited-anomaly theorem). *Let T be an algorithm selecting the* kth *largest of* $X=\{x_1, x_2, \cdots, x_n\}$, *and* σ *an input ordering. Then the number of anomalous comparisons of degree* $\leqq m$ *is at most* $(2m+1)s_1(\sigma)$.

Proof. We assign a weight $m+1-i$ to an anomalous element of degree i for $1 \leqq i \leqq m$, and a weight 0 to all other elements. Let E and E' be respectively the total weight of all elements before and after a comparison $x_i > x_j$. Then the following is true.

LEMMA 4.2.

(A) $E' \leqq E+2m$.

(B) *If* $x_i > x_j$ *is a first defeat, then* $E' \leqq E$.

(C) *If* $x_i > x_j$ *is a first defeat and an anomalous comparison of degree* $\leqq m$, *then* $E' < E$.

Proof of Lemma 4.2. Let G be the set of x_l such that, before the comparison, $F(x_l)$ contains at least one of x_i and x_j. Clearly, $|G| \leqq 2$, as any element x_t can be in at most one $F(x_l)$. It is easy to see that only elements in G may be assigned new weights after the comparison. Since the largest increase in weight for an element is from 0 to m, this proves (A).

Suppose $x_i > x_j$ is a first defeat. Then x_j must have been a maximal element before the comparison, and hence $x_j \in G$. It follows that either $G=\{x_j\}$ or $G=\{y, x_j\}$ with $y \neq x_j$.

Note that the weight of x_j cannot increase after the comparison, and will strictly decrease from $m+1-f(x_j)$ to 0 if x_j is anomalous of degree $\leqq m$; this proves (B) and (C) when $G=\{x_j\}$. It remains to prove (B) and (C) assuming that $G=\{y, x_j\}$ with $y \neq x_j$, $x_i \in F(y)$.

After the comparison, x_j is no longer maximal. Let $F'(y)$ be the new fiefdom of y after the comparison, then $F'(y) \supseteq F(y) \cup F(x_j)$; clearly, $|F'(y)| > f(x_j)$. We consider two cases according to whether x_j was anomalous of degree $\leqq m$ before the comparison $x_i > x_j$.

Case (a). *x_j was anomalous of degree* $\leqq m$. The decrease in x_j's weight is from $m+1-f(x_j)$ to 0 while the maximum increase in y's weight is from 0 to $\max\{0, m+1-|F'(y)|\} < m+1-f(x_j)$. This means $E' < E$.

Case (b). *x_j was not anomalous of degree* $\leqq m$. Then x_j's weight does not change; y's weight has two cases:

(b1) y was anomalous of degree $\leqq m$. Then y's weight strictly decreases due to the strict increase in its anomaly degree.

(b2) y was not anomalous of degree $\leqq m$. Then y's weight remains 0.

This proves (B). Statement (C) follows from the analysis of Case (a) and Case (b1) above. This completes the proof of Lemma 4.2. □

We will now complete the proof of Theorem 4.1. Statements (A) and (B) of Lemma 4.2 imply that the total increase in weight along path $S(\sigma)$ is bounded by $2ms_1(\sigma)$. Since the sum of weights of the elements is initially 0 and always nonnegative by definition, the number of comparisons n_3 which fits statement (C) of Lemma 4.2 is at most $2ms_1(\sigma)$. The total number of comparisons along $S(\sigma)$ that are anomalous of degree $\leqq m$ is clearly at most $n_3 + s_1(\sigma)$, and is hence bounded by $(2m+1)s_1(\sigma)$. This proves Theorem 4.1. □

5. Proof of the main theorem.

5.1. Introduction. We will prove the following result in this section.

THEOREM 5.1. *Let k, n be integers with $k \geqq 2$ and $n \geqq N_k = (8k)^{18k}$. Suppose T is an algorithm that selects the kth largest of n elements, and σ an input ordering. If $s_1(\sigma) \leqq n^{0.2}$, then $\alpha(\sigma) \geqq k(\ln \ln n - \ln k - 6)$.*

As defined in § 2, the quantities $\alpha(\sigma)$, $s_1(\sigma)$ depend on T. Also note that, for $n \geqq N_k$, the following inequalities hold, as can be verified by elementary arguments.

$$n^{0.1} \geqq k \ln \ln n, \tag{5.1}$$

$$n^{1/(6k)} \geqq 2 \lg n, \tag{5.2}$$

$$n^{1/12} \geqq k. \tag{5.3}$$

We first demonstrate that Theorem 5.1 implies the main theorem. If there are more than $n! \times n^{-0.1}$ σ satisfying $s_1(\sigma) > n^{0.2}$, then (2.8) implies

$$\text{COST}(T) \geqq n - k + \frac{1}{n!} n! n^{-0.1} n^{0.2} \geqq n - k + k \ln \ln n,$$

in view of (5.1). On the other hand if less than $n! \times n^{-0.1}$ of the σ's satisfy $s_1(\sigma) > n^{0.2}$, then (2.9) and Theorem 5.1 lead to

$$\begin{aligned} \text{COST}(T) &\geqq n - k + \frac{1}{2}\frac{1}{n!}(n! - n! \times n^{-0.1}) k(\ln \ln n - \ln k - 6) \\ &\geqq n + \tfrac{1}{2} k(\ln \ln n - \ln k - 6 - n^{-0.1} \ln \ln n - 2). \end{aligned}$$

Again, using (5.1), we obtain

$$\text{COST}(T) \geqq n + \tfrac{1}{2} k(\ln \ln n - \ln k - 9).$$

Thus, the main theorem is true in both cases.

5.2. Some results on partial orders. Let P be a partial order on a set $X=\{x_1, x_2, \cdots, x_n\}$. Assume that all orderings on X consistent with P are equally likely. We are interested in bounds on the probability of some element x_i being greater than another element x_j (or all elements in some subset). For instance, if x_i is the unique maximal element in a component (in P) of size m, then the probability that x_i is the maximum of all n elements in X is clearly m/n, and it is also not difficult to show that $\Pr(x_i > x_j)$ is at least $m/(m+r)$, if x_j is a nonmaximal element in a different component of size r. A generalization of these facts is given below in two lemmas.

LEMMA 5.2. *If x_i is a maximal element, then*

$$\Pr(x_i \text{ is the largest element in } X) \geqq \frac{f(x_i)}{n}.$$

LEMMA 5.3. *If x_i is a maximal element, and x_j a nonmaximal element in a different component, then*

$$\Pr(x_i > x_j) \geqq \frac{f(x_i)}{f(x_i)+h(x_j)}.$$

Intuitively, the above lemmas must be true, since knowing that some elements in $F(x_i)$ are greater than some elements outside $F(x_i)$ should not lower the rank of x_i. However, the proofs are not trivial, and will be derived in Appendix B as consequences of a general theorem by Shepp [13]. (For further results of this type, see Graham, Yao and Yao [3] and Shepp [13].)

LEMMA 5.4. *Suppose x_i is the unique element in a component C of size m, and x_j a nonmaximal element in a different component C' of size $\Delta - m$. Assume that $\Delta > 2k$. Define the quantity β to be* ($\Pr(x_i > x_j) + \Pr(\max\{x_i, x_j\}$ *is in the top* $k-1$ *of* X)). *Then*

$$\beta \geqq \min\left\{1-e^{-km/\Delta}, 1-e^{-tm/\Delta}+\left(\frac{\Delta}{2n}\right)^t, \text{for } 1<t<k\right\}.$$

Proof. See Appendix C. □

5.3. Lower bounds on $q_l(v)$. Let v be an internal node in the algorithm T. Suppose v compares x_i, x_j. We will give lower bounds on $q_l(v)$ in terms of component sizes such as $f(x_i)$, $h(x_j)$, etc. defined relative to the partial order $P(v)$. We will assume for the rest of § 5 $k \geqq 2$ and $n \geqq N_k$.

LEMMA 5.5. *If x_i is a nonmaximal element and x_j is in a component not containing x_i, then $q_1(v) \geqq 1/(h(x_i)+1)$.*

Proof. If x_j is also nonmaximal, then $q_1(v)=1$, else by Lemma 5.3,

$$q_1(v) = \Pr(x_j > x_i) \geqq f(x_j)/(f(x_j)+h(x_i)) \geqq 1/(h(x_i)+1). \qquad \square$$

LEMMA 5.6. *If both x_i and x_j are maximal, then $q_2(v) \geqq (f(x_i)+f(x_j))/n$.*

Proof. The properties of x_i, x_j being the largest element in X are mutually exclusive. Hence $q_2(v) \geqq f(x_i)/n + f(x_j)/n$ by Lemma 5.2. □

LEMMA 5.7. *If x_i is a maximal element and x_j a nonmaximal element in a different component, then $q_1(v) \geqq f(x_i)/(f(x_i)+h(x_j))$.*

Proof. It follows directly from Lemma 5.3. □

LEMMA 5.8. *Suppose x_i is the unique maximal element in a component C, and x_j a nonmaximal element in a different component. If $h(x_i) \leqq n^{1/3}$ and $h(x_i)+h(x_j) \geqq n^{1-(1/6k)}$, then*

$$q(v) \geqq k\frac{h(x_i)}{h(x_j)} - 3k^2\frac{1}{n^{7/6}}.$$

Proof. Let $m = h(x_i)$, $m' = h(x_j)$ and $\Delta = m + m'$. Then by assumption

$$m \leqq n^{1/3} \quad \text{and} \quad \Delta \geqq n^{1-(1/6k)}. \tag{5.4}$$

Clearly $\Delta > 2k$. By Lemma 5.4, we need only show that

$$1 - e^{-km/\Delta} \geqq k\frac{m}{m'} - 3k^2\frac{1}{n^{7/6}}, \tag{5.5}$$

and

$$\min_{1<t<k}\left\{1 - e^{-tm/\Delta} + \left(\frac{\Delta}{2n}\right)^t\right\} \geqq k\frac{m}{m'} - 3k^2\frac{1}{n^{7/6}}. \tag{5.6}$$

As $e^{-x} \leqq 1 - x + \frac{1}{2}x^2$ for $x \geqq 0$, we have

$$1 - e^{-km/\Delta} \geqq \frac{km}{\Delta} - \frac{1}{2}\left(\frac{km}{\Delta}\right)^2 = k\frac{m}{m'} - k\frac{m^2}{\Delta m'} - \frac{1}{2}\left(\frac{km}{\Delta}\right)^2 \tag{5.7}$$

Now, from (5.4),

$$\frac{m}{\Delta} \leqq n^{-(2/3-1/6k)}. \tag{5.8}$$

This implies $m/\Delta < \frac{1}{2}$ and hence

$$m' > \tfrac{1}{2}\Delta. \tag{5.9}$$

Using (5.8) and (5.9) in (5.7), we obtain

$$1 - e^{-km/\Delta} \geqq k\frac{m}{m'} - \left(2k + \frac{k^2}{2}\right)\left(\frac{m}{\Delta}\right)^2 \geqq k\frac{m}{m'} - 3k^2 n^{-(4/3-1/3k)} \geqq k\frac{m}{m'} - 3k^2 n^{-7/6}. \tag{5.10}$$

This proves (5.5).

For $1 < t < k$,

$$1 - e^{-tm/\Delta} + \left(\frac{\Delta}{2n}\right)^t \geqq \left(\frac{\Delta}{2n}\right)^{k-1} \geqq n^{-1/6+1/6k} \cdot 2^{-(k-1)} \geqq 2kn^{-(2/3-1/6k)}, \tag{5.11}$$

where we have used (5.4) and the fact $n \geqq N_k > k^2 4^k$. We now use (5.8) and (5.9) to obtain

$$1 - e^{-tm/\Delta} + \left(\frac{\Delta}{2n}\right)^t \geqq 2k\frac{m}{\Delta} \geqq k\frac{m}{m'}.$$

This implies (5.6) immediately. □

5.4. Completing the proof. As in § 3.3, we construct a merge-free $M(\sigma)$ corresponding to the merging process for σ, and assign $\varphi(u) = q(v)$ to each $u \in M(\sigma)_I$. Let σ be any ordering that satisfies $s_1(\sigma) \leqq n^{0.2}$. We will show that

$$\sum_{u \in M(\sigma)_I} \varphi(u) \geqq k(\ln \ln n - \ln k - 6). \tag{5.12}$$

This would prove Theorem 5.1, as

$$\alpha(\sigma) = \sum_{v \in S(\sigma)} q(v) \geqq \sum_{v \in S'(\sigma)} q(v) = \sum_{u \in M(\sigma)_I} \varphi(u).$$

To prove (5.12), we first partition the set of nodes in $M(\sigma)$ into upper and lower parts, $U = \{u \mid w(u) \geqq n^{1/3}\}$ and $L = \{u \mid w(u) < n^{1/3}\}$. Let $V' = \{u \mid u \in U, \text{ lson}[u] \in L,$

rson $[u]\in L\}$, $V''=\{u \mid u\in L, \text{father } [u]\in U-V'\}$, and $V=V'\cup V''$. (These definitions are similar to those used in Appendix A, and properties P1–P5 there remain true.)

We now partition V into seven disjoint parts $V_1, V_2, \cdots, V_7$. For each $u\in V$, we assign u to a unique V_i according to the following procedure, which halts as soon as u is assigned.

Procedure Decompose;

step 1: If there is some $u'\in D(u)$ where the joining comparison is *not* between two maximal elements, then assign u to V_1.

[*comment*: If u is not assigned in step 1, then the joining comparison at u creates a component $C(u)$ with a unique maximal element; recall that $C(u)$ consists of the x_i's that label the leaves in $D_L(u)$.]

step 2: If $u\in V'$, then assign u to V_2.

[*comment*: If u has not been assigned after step 2, then u must be in V'' and father $[u]$ exists.]

step 3: If father $[u]$ compares a nonmaximal element in $C(u)$ with any element, then assign u to V_3.

step 4: If father $[u]$ compares the maximal element of $C(u)$ with another maximal element (in a different component), then

$$\text{assign } u \text{ to } \begin{cases} V_4 \text{ if the comparison is anomalous of degree at most } \lceil n^{1/5}\rceil, \\ V_5 \text{ otherwise.} \end{cases}$$

step 5: If father $[u]$ compares the maximal element of $C(u)$ with some nonmaximal element (in a different component), then

$$\text{assign } u \text{ to } \begin{cases} V_6 \text{ if } w\ (\text{father } [u])\leqq n^{1-1/6k}, \\ V_7 \text{ if } w\ (\text{father } [u])> n^{1-1/6k} \end{cases}$$

end *Decompose.*

Let $W_i=\sum_{u\in V_i} w(u)$ $(1\leqq i\leqq 7)$, and

$$A_i=\begin{cases} \sum\limits_{u\in V_i}\sum\limits_{u'\in D(u)} \varphi(u') & \text{if } i\in\{1,2,4\}, \\ \sum\limits_{u\in V_i} \varphi(\text{father } [u]) & \text{if } i\in\{3,6,7\}, \\ \sum\limits_{u\in V_i}\left(\sum\limits_{u'\in D(u)} \varphi(u')+\varphi(\text{father } [u])\right) & \text{if } i\in\{5\}. \end{cases}$$

In analogy with discussions in Appendix A, it is not difficult to see that V_7 is a cross section (recall Definition 3.2), and that

$$\sum_{1\leqq i\leqq 7} W_i=n, \tag{5.13}$$

and

$$\sum_{u\in M(\sigma)_I} \varphi(u)\geqq \sum_{1\leqq i\leqq 7} A_i. \tag{5.14}$$

We will now find lower bounds to the A_i's in terms of the W_i's. We treat first A_i for $i\in\{1,3,6\}$, which are "costly" and thus efficient algorithms should not have large W_i for these values of i.

LEMMA 5.9. $A_1+A_3+A_6\geqq (W_1+W_2+W_6)n^{-(1-1/6k)}$.

Proof. For each $u \in V_1$, some $u' \in D(u)$ has a comparison involving a nonmaximal element. Thus, by Lemma 5.5, $\sum_{u' \in D(u)} \varphi(u') \geqq 1/(2n^{1/3})$. We have

$$A_1 \geqq \tfrac{1}{2}|V_1| \cdot n^{-1/3}. \tag{5.15}$$

Similarly, by Lemma 4.5, we have

$$A_3 \geqq \tfrac{1}{2}|V_3| \cdot n^{-1/3}. \tag{5.16}$$

As each $u \in V$ has $w(u) \leqq 2n^{1/3}$, we have for $i \in \{1, 3\}$

$$|V_i| \geqq \tfrac{1}{2} W_i n^{-1/3}. \tag{5.17}$$

Formulas (5.15)–(5.17) lead to

$$A_i \geqq \tfrac{1}{4} W_i \cdot n^{-2/3} \geqq W_i \cdot n^{-(1-1/6k)}, \quad \text{for } i \in \{1, 3\}. \tag{5.18}$$

For each $u \in V_6$, we apply Lemma 5.7 to father $[u]$ and obtain

$$\varphi(\text{father }[u]) \geqq \frac{w(u)}{w(\text{father }[u])} \geqq w(u) n^{-(1-1/6k)}.$$

Thus,

$$A_6 \geqq \sum_{u \in V_6} w(u) n^{-(1-1/6k)} = W_6 n^{-(1-1/6k)}. \tag{5.19}$$

Combining (5.18) and (5.19), we obtain the lemma. □

LEMMA 5.10. $W_4 \leqq 8n^{11/15}$.

Proof. By the limited-anomaly theorem (Theorem 4.1),

$$|V_4| \leqq (2\lceil n^{1/5} \rceil + 1) s_1(\sigma) \leqq 8n^{0.4},$$

since $s_1(\sigma) \leqq n^{0.2}$ by assumption. As each $u \in V_4$ has $w(u) \leqq n^{1/3}$, we have

$$W_4 \leqq |V_4| n^{1/3} \leqq 8n^{11/15},$$

□

LEMMA 5.11. $A_2 \geqq (W_2/3n) \lg n - 1$.

Proof. Let $u \in V_2$. For each $u' \in D(u)$, $\varphi(u') \geqq w(u')/n$ by Lemma 5.6, as the corresponding comparison is between normal maximal elements. This gives, by Lemma 3.2,

$$\sum_{u' \in D(u)} \varphi(u') \geqq \frac{1}{n} \sum_{u' \in D(u)} w(u') \geqq \frac{1}{n} w(u)(\lg w(u) - 1).$$

As $w(u) \geqq n^{1/3}$, we have

$$\sum_{u' \in D(u)} \varphi(u') \geqq \frac{1}{n} w(u) \left(\frac{1}{3} \lg n - 1 \right).$$

Therefore,

$$A_2 \geqq \frac{1}{n} \sum_{u \in V_2} w(u) \left(\frac{1}{3} \lg n - 1 \right) \geqq \frac{W_2}{3n} \lg n - 1.$$

□

LEMMA 5.12. $A_5 \geqq (W_5/5n) \lg n - 1$.

Proof. If $|V_5| = 0$ then $W_5 = 0$ and the lemma is clearly true. We thus assume that $|V_5| > 0$. For each $u \in V_5$,

$$\sum_{u' \in D(u)} \varphi(u') \geqq \frac{1}{n} w(u)(\lg w(u) - 1).$$

Thus, using Fact B in § 3.2, we have

$$\sum_{u\in V_5}\sum_{u'\in D(u)}\varphi(u')\geqq\frac{1}{n}\Big(\sum_{u\in V_5}w(u)\lg w(u)-W_5\Big)\geqq\frac{1}{n}W_5\lg\frac{W_5}{|V_5|}-1. \tag{5.20}$$

Now, for each $u\in V_5$, let the comparison at father $[u]$ be between x_i and x_j, where x_i is the maximal element of $C(u)$. By Lemma 5.6,

$$\varphi(\text{father}\,[u])\geqq\frac{f(x_i)+f(x_j)}{n}\geqq\begin{cases}\dfrac{w(\text{father}\,[u])}{n}\geqq\dfrac{1}{n}n^{1/3} & \text{if } x_j \text{ is normal,}\\[2ex] \dfrac{1}{n}\lceil n^{1/5}\rceil & \text{if } x_j \text{ is anomalous.}\end{cases}$$

Thus,

$$\sum_{u\in V_5}\varphi(\text{father}\,[u])\geqq|V_5|n^{-4/5}. \tag{5.21}$$

Formulas (5.20) and (5.21) lead to

$$A_5\geqq\frac{1}{n}W_5\lg\frac{W_5}{|V_5|}+|V_5|n^{-4/5}-1. \tag{5.22}$$

By standard minimization technique (e.g., see the proof of Fact E in Appendix A), (5.22) yields

$$A_5\geqq\frac{1}{n}W_5\lg((\ln 2)\cdot n^{1/5})+\frac{1}{n}W_5\frac{1}{\ln 2}-1.$$

The lemma follows, noting that $\lg\ln 2+1/\ln 2>0$. □

LEMMA 5.13. $A_7\geqq k\ln[(n+1)/(n-W_7+1)]-3$.

Proof. Let $u\in V_7$, we write $u'=\text{brother}\,[u]$. By Lemma 5.8 and (5.3), we have

$$\varphi(\text{father}\,[u])\geqq k\frac{w(u)}{w(u')}-3k^2\frac{1}{n^{7/6}}\geqq k\frac{w(u)}{w(u')}-3\frac{1}{n}.$$

As V_7 is a cross section, we obtain from Lemma 3.3 that

$$A_7\geqq k\sum_{u\in V_7}\frac{w(u)}{w(u')}-3\geqq k\ln\frac{n+1}{n-W_7+1}-3. \qquad □$$

We are now ready to prove (5.12), and hence Theorem 5.1. Using Lemmas 5.9, 5.11, 5.12, 5.13 and formula (5.14), we have

$$\begin{aligned}\sum_{u\in M(\sigma)_I}\varphi(u)&\geqq\sum_i A_i\\ &\geqq(W_1+W_3+W_6)n^{-(1-1/6k)}+\frac{\lg n}{3n}W_2+\frac{\lg n}{5n}W_5+k\ln\frac{n+1}{n+1-W_7}-5.\end{aligned}$$

Making use of (5.2) and (5.13)

$$\begin{aligned}\sum_{u\in M(\sigma)_I}\varphi(u)&\geqq\frac{\lg n}{5n}(W_1+W_2+W_3+W_5+W_6)+k\ln\frac{n+1}{n-W_7+1}-5\\ &=(n-W_7)\frac{\lg n}{5n}+k\ln\frac{n+1}{n-W_7+1}-5-\frac{W_4}{5n}\lg n.\end{aligned} \tag{5.23}$$

From Lemma 5.10 and (5.2),

$$\frac{W_4}{5n}\lg n \leqq \frac{8}{5}\frac{n^{11/15}}{n}\lg n \leqq 2\frac{\lg n}{n^{4/15}} < 1. \tag{5.24}$$

Therefore, (5.23) leads to

$$\sum_{u \in M(\sigma)_I} \varphi(u) \geqq x\frac{\lg n}{5n} + k \ln\frac{n+1}{x+1} - 6,$$

for some x, $0 \leqq x \leqq n$.

A standard minimization gives

$$\sum_{u \in M(\sigma)_I} \varphi(u) \geqq k \ln\left(\frac{\lg n}{5k}\right) - 6 \geqq k(\ln \ln n - \ln k - 6);$$

which is (5.12).

This completes the proof of the main theorem. □

Appendix A: proof of Lemma 3.5. The lemma is clearly true when $n \leqq 8$. We shall thus assume that $n > 8$. Note that, in this range,

$$n^{1/3} > \max\{\tfrac{1}{3}\lg n, \tfrac{1}{2}\ln \ln n\}. \tag{A.1}$$

We say a node $u \in M_I$ to be of *category* 1 if $g(u) = \min\{w_1, w_2\}/(w_1 + w_2)$, and of *category* 2 otherwise. For a node u to be of category 1, we must have

$$\frac{\min\{w_1, w_2\}}{w_1 + w_2} \leqq \frac{w_1 + w_2}{n},$$

implying

$$w(u) = w_1 + w_2 \geqq \sqrt{n}. \tag{A.2}$$

Let us divide the set of nodes of M into an upper part U and a lower part L according to whether or not $w(u) \geqq n^{1/3}/2$. As $n > 8$, the root must be in U and all leaves are in L. Now consider the set V' of lowest nodes in U, i.e.,

$$V' = \{u \mid u \in U, \text{lson}[u] \in L, \text{rson}[u] \in L\},$$

and the set V'' defined by

$$V'' = \{u \mid u \in L, \text{father}[u] \in U - V'\}.$$

An alternative characterization of V'' is given by

$$V'' = \{u \mid u \in L, \text{father}[u] \in U, \text{brother}[u] \in U\}.$$

Let $V = V' \cup V''$. The following simple properties are easy to check.

P1: V' and V'' are disjoint.

P2: Any two distinct nodes in V have no common descendants.

P3: Any two distinct nodes in V'' have distinct fathers; furthermore, the set $\{\text{father}[u] \mid u \in V''\}$ is disjoint from the union of descendants of nodes in V.

P4: V'' is a cross section of M.

P5: The family of sets $\{D_L(u) \mid u \in V\}$ forms a partition of the leaves of M.

We partition $V = V' \cup V''$ into V_i $(1 \leqq i \leqq 4)$ as follows. The set V_1 is simply V'. Sets V_2, V_3, V_4 are given by

$$V_2 = \{u \mid u \in V'', \text{father}[u] \text{ is of category } 2\},$$

$$V_3 = \{u \mid u \in V'', \text{father}[u] \text{ is of category } 1,\ w(\text{father}[u]) < n^{2/3}\},$$

$$V_4 = \{u \mid u \in V'', \text{father}[u] \text{ is of category } 1,\ w(\text{father}[u]) \geqq n^{2/3}\}.$$

The definitions are illustrated in Fig. 5.

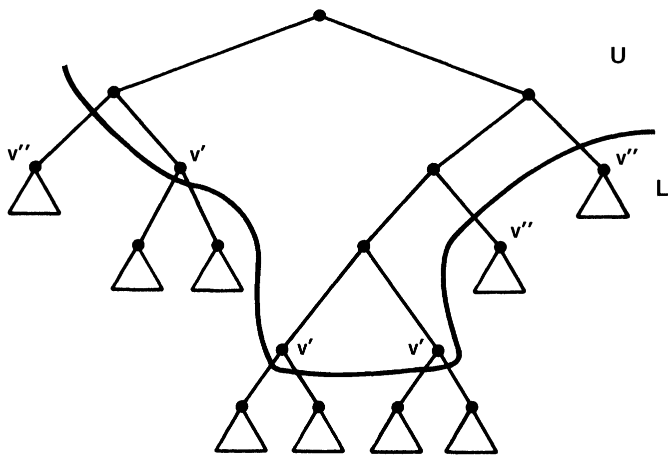


FIG. 5. *A schematic illustration of sets U, L,* $V' = V_1$, $V'' = V_2 \cup V_3 \cup V_4$; *nodes in* V', V'' *are labeled as* v', v'', *respectively.*

Let $W_i = \sum_{u \in V_i} w(u)$ for $1 \leqq i \leqq 4$. Define

$$A_1 = \sum_{u \in V_1} \sum_{u' \in D(u)} g(u'),$$

$$A_2 = \sum_{u \in V_2} \left(\sum_{u' \in D(u)} g(u') + g(\text{father}[u]) \right),$$

$$A_i = \sum_{u \in V_i} g(\text{father}[u]), \qquad i = 3, 4.$$

As an immediate consequence of property P5, we have

$$\text{(A.3)} \qquad \sum_{1 \leqq i \leqq 4} W_i = n.$$

Now, from properties P1–P3, we have

$$\text{(A.4)} \qquad \sum_{u \in M_I} g(u) \geqq \sum_{1 \leqq i \leqq 4} A_i.$$

Our plan is to first derive lower bounds to A_i in terms of W_i, and then apply (A.4) to prove Lemma 3.5.

FACT C. *If* $w(u) < \sqrt{n}$, *then* $\sum_{u' \in D(u)} g(u') \geqq (1/n) w(u)(\lg w(u) - 1)$.

Proof. We may assume that $u \in M_I$, as the assertion is clearly true when u is a leaf. Now each $u' \in D(u)$ must be of category 2 $(w(u') < \sqrt{n})$, and hence $g(u') =$

$w(u')/n$. Using Lemma 3.2, we have

$$\sum_{u' \in D(u)} g(u') = \frac{1}{n} \sum_{u' \in D(u)} w(u') \geqq \frac{1}{n} w(u)(\lg w(u) - 1). \quad \square$$

FACT D. $A_1 \geqq (W_1/3n) \lg n - 2$.

Proof. Each $u \in V_1$ satisfies $w(u) < 2(n^{1/3}/2) \leqq \sqrt{n}$, and hence from Fact C,

$$A_1 = \sum_{u \in V_1} \sum_{u' \in D(u)} g(u') \geqq \sum_{u \in V_1} \frac{1}{n} w(u)(\lg w(u) - 1).$$

As $w(u) \geqq n^{1/3}/2$ (since $u \in U$), we have

$$A_1 \geqq \frac{1}{n} \sum_{u \in V_1} w(u)\left(\frac{1}{3} \lg n - 2\right) \geqq \frac{W_1}{3n} \lg n - 2. \quad \square$$

FACT E. $A_2 \geqq (W_2/3n) \lg n - 3$.

Proof. The statement is obviously true when $|V_2| = 0$. We shall thus assume that $|V_2| > 0$. For each $u \in V_2$, $g(\text{father}[u]) = w(\text{father}[u])/n \geqq 1/(2n^{2/3})$, since father $[u]$ is of category 2 and is in U. Making use of Fact C, we have

$$A_2 = \sum_{u \in V_2} \sum_{u' \in D(u)} g(u') + \sum_{u \in V_2} g(\text{father}[u])$$

$$\geqq \sum_{u \in V_2} \frac{w(u)}{n} (\lg w(u) - 1) + |V_2| \frac{1}{2n^{2/3}}.$$

We now use Fact B to obtain

$$A_2 \geqq \frac{W_2}{n} \lg \frac{W_2}{|V_2|} - 1 + \frac{|V_2|}{2n^{2/3}}. \tag{A.5}$$

The right-hand side expression $d(|V_2|)$ achieves its absolute minimum over $|V_2| \in [0, \infty)$ at $|V_2| = 2W_2/(n^{1/3} \ln 2)$, where

$$d(|V_2|) = \frac{W_2}{n} \lg\left(\frac{\ln 2}{2} n^{1/3}\right) - 1 + \frac{1}{\ln 2} \frac{W_2}{n} \geqq \frac{W_2}{3n} \lg n - 3.$$

Thus, formula (A.5) implies

$$A_2 \geqq \frac{W_2}{3n} \lg n - 3, \tag{A.6}$$

proving Fact E. $\square$

The derivation of (A.6) from (A.5) is a standard argument, and similar derivations will henceforth be referred to as "by standard minimization technique" with details omitted.

For each $u \in V_3 \cup V_4$, $w(\text{brother}[u]) \geqq n^{1/3}/2 > w(u)$, and father $[u]$ is of category 1. Thus,

$$g(\text{father}[u]) = \frac{w(u)}{w(\text{father}[u])}. \tag{A.7}$$

FACT F. $A_3 \geqq W_3/n^{2/3}$.

Proof. For each $u \in V_3$, $w(\text{father}[u]) < n^{2/3}$. Using (A.7), we have

$$A_3 = \sum_{u \in V_3} g(\text{father}[u]) = \sum_{u \in V_3} \frac{w(u)}{w(\text{father}[u])} \geqq \frac{W_3}{n^{2/3}}. \quad \square$$

FACT G. $A_4 \geqq (1-1/2n^{1/3}) \ln [(n+1)/(n-W_4+1)]$.

Proof. For each $u \in V_4$, $w(u) < n^{1/3}/2$ and $w(\text{father}[u]) \geqq n^{2/3}$. Using (A.7), we have

$$g(\text{father}[u]) = \frac{w(u)}{w(\text{father}[u])} = \frac{w(u)}{w(\text{brother}[u])}\left(1 - \frac{w(u)}{w(\text{father}[u])}\right)$$
$$\geqq \frac{w(u)}{w(\text{brother}[u])}\left(1 - \frac{1}{2n^{1/3}}\right).$$

Thus,

$$A_4 = \sum_{u \in V_4} g(\text{father}[u]) \geqq \left(1 - \frac{1}{2n^{1/3}}\right) \sum_{u \in V_4} \frac{w(u)}{w(\text{brother}[u])}. \tag{A.8}$$

As V'' is a cross section of M by property P4, so is V_4. Fact G then follows from (A.8) and Lemma 3.3. □

We will now finish the proof of Lemma 3.5. Using Facts D–G, we obtain from (A.4)

$$\sum_{u \in M_I} g(u) \geqq \frac{W_1 + W_2}{3n} \lg n + \frac{W_3}{n^{2/3}} + \left(1 - \frac{1}{2n^{1/3}}\right) \ln \frac{n+1}{n - W_4 + 1} - 5.$$

Using (A.1) and (A.3), we obtain then

$$\begin{aligned} \sum_{u \in M_I} g(u) &\geqq \frac{W_1 + W_2 + W_3}{3n} \lg n + \left(1 - \frac{1}{2n^{1/3}}\right) \ln \frac{n+1}{W_1 + W_2 + W_3 + 1} - 5 \\ &\geqq \left(1 - \frac{1}{2n^{1/3}}\right)\left(\frac{x}{3n} \lg n + \ln \frac{n+1}{x+1}\right) - 5, \end{aligned} \tag{A.9}$$

where $x = W_1 + W_2 + W_3$.

By standard minimization technique, we obtain from (A.9)

$$\sum_{u \in M_I} g(u) \geqq \left(1 - \frac{1}{2n^{1/3}}\right)(\ln \ln n - 1) - 5 \geqq \ln \ln n - 7,$$

where (A.1) was used in the last step. This proves Lemma 3.5. □

Appendix B: proof of Lemmas 5.2 and 5.3. Let $A = \{a_1, a_2, \cdots, a_m\}$, $B = \{b_1, b_2, \cdots, b_{m'}\}$, $X = A \cup B$, and $(Q_0, <)$ be a partial order on X which contains no relation of the form $a_i > b_j$ or $a_i < b_j$. Let J be a set of inequalities $\{a_{i_1} > b_{j_1}, a_{i_2} > b_{j_2}, \cdots, a_{i_p} > b_{j_p}\}$, and let Q_1 be the partial order generated by $Q_0 \cup J$. The following result is proved by Shepp [13, Theorem 2].

THEOREM B1 (Shepp [13]). *Let E be an event of the form* $(a_{s_1} > b_{t_1}) \wedge (a_{s_2} > b_{t_2}) \wedge \cdots \wedge (a_{s_q} > b_{t_q})$. *Then* $\Pr(E|Q_1) \geqq \Pr(E|Q_0)$.

Remark. As usual, the probability $\Pr(E|Q_i)$ is defined with the assumption that all linear orderings of X consistent with Q_i are equally likely.

We will now use the above theorem to prove our lemmas. Recall that, in Lemmas 5.2 and 5.3, a partial order P on $X = \{x_1, x_2, \cdots, x_n\}$ is given and x_i is a maximal element. Let $A = F(x_i)$, $B = X - A$, and $J = \{x_r > x_l \mid x_r \in A, x_l \in B, (x_r > x_l) \in P\}$. Define $Q_0 = P - J$. It is easy to see that Q_0 and J satisfy the requirements for Theorem B1. The partial order Q_1 is clearly just P.

To prove Lemma 5.2, let E be the event $\wedge_{x_l \in B} (x_i > x_l)$. Then

$$\Pr(E|Q_0) = \frac{|A|}{|X|} = \frac{f(x_i)}{n}.$$

Using Theorem B1, we obtain

$$\Pr\{x_i \text{ is the largest element in } X \text{ under } P\} = \Pr(E \mid Q_1) \geqq \Pr(E \mid Q_0) = \frac{f(x_i)}{n}.$$

This establishes Lemma 5.2.

To prove Lemma 5.3, let E denote the event $\wedge_{x_l \in H(x_j)} (x_i > x_l)$. As the event E implies $x_i > x_j$, we have

$$\Pr\{x_i > x_j \text{ under } P\} \geqq \Pr(E \mid P) = \Pr(E \mid Q_1).$$

Using Theorem B1, we obtain

$$\Pr(E \mid Q_1) \geqq \Pr(E \mid Q_0) = \frac{|A|}{|A| + |H(x_j)|} = \frac{f(x_i)}{f(x_i) + h(x_j)}.$$

Lemma 5.3 follows. □

Appendix C: proof of Lemma 5.4. Let $\beta(t)$ be the quantity β when the component C' has been sorted and x_j is the tth largest in it. Then, denoting by $p(t)$ the probability that x_j is the tth largest in C' under partial order P, we have with $m' = \Delta - m$,

$$\beta = \sum_{1 \leqq t \leqq m'} p(t)\beta(t).$$

As x_j is not a maximal element, $p(1) = 0$. Therefore, the lemma would follow, if we can show that for all $1 < t \leqq m'$,

$$\beta(t) \geqq \min\left\{1 - e^{-km/\Delta},\ 1 - e^{-t'm/\Delta} + \left(\frac{\Delta}{2n}\right)^{t'} \quad \text{for } 1 < t' < k\right\}. \tag{C.1}$$

Let $\beta(t) = a_1 + a_2$, where

a_1 = probability that $x_i > x_j$,

a_2 = probability that $\max\{x_i, x_j\}$ is in the top $k - 1$.

Clearly,

$$a_1 = 1 - (\text{probability } x_i < x_j) = 1 - \frac{\binom{\Delta - t}{m}}{\binom{\Delta}{m}}$$

$$= 1 - \left(1 - \frac{t}{D}\right)\left(1 - \frac{t}{\Delta - 1}\right) \cdots \left(1 - \frac{t}{\Delta - m + 1}\right)$$

$$\geqq 1 - \left(1 - \frac{t}{\Delta}\right)^m.$$

But,

$$\left(1 - \frac{t}{\Delta}\right)^m = e^{m \ln(1 - t/\Delta)} \leqq e^{m(-t/\Delta)}.$$

Thus,

$$a_1 \geqq 1 - e^{-tm/\Delta}. \tag{C.2}$$

Formula (C.2) proves (C.1) for the case $k \leqq t \leqq m'$. We shall now restrict our attention to the case $1 < t < k' = \min\{k, m'+1\}$. In this range,

$$\begin{aligned} a_2 &= \Pr(\max\{x_i, x_j\} \text{ is in the top } k-1 \text{ of } X) \\ &\geqq \Pr(\text{the } t\text{th largest element in } C \cup C' \text{ is in the top } k-1 \text{ of } X) \\ &= \sum_{t \leqq l < k} \Pr(\text{the } t\text{th largest element in } C \cup C' \text{ is the } l\text{th largest in } X) \\ &= \sum_{t \leqq l < k} \frac{\binom{n-l}{\Delta-t}\binom{l-1}{t-1}}{\binom{n}{\Delta}}. \end{aligned}$$

Taking only the term $l = t$ and using the assumption $\Delta > 2k$, we obtain

$$\text{(C.3)} \qquad a_2 \geqq \frac{\Delta}{n}\frac{\Delta-1}{n-1}\cdots\frac{\Delta-t+1}{n-t+1} \geqq \left(\frac{\Delta-k}{n}\right)^t \geqq \left(\frac{\Delta}{2n}\right)^t \quad \text{when } 1 < t < k'.$$

From (C.2) and (C.3) we see that for $1 < t < k'$

$$\beta(t) = a_1 + a_2 \geqq 1 - e^{-tm/\Delta} + \left(\frac{\Delta}{2n}\right)^t.$$

Thus, (C.1) is also true in this case.

This completes the proof of Lemma 5.4. □

Acknowledgments. We wish to thank David Matula for helpful suggestions and a careful reading of the manuscript.

REFERENCES

[1] R. W. FLOYD AND R. L. RIVEST, *Expected time bounds for selection*, Comm. ACM, 18 (1975), pp. 165–172.

[2] F. FUSSENEGGER AND H. N. GABOW, *A counting approach to lower bounds for selection problems*, J. Assoc. Comput. Mach., 26 (1979), pp. 227–238.

[3] R. L. GRAHAM, A. C. YAO AND F. F. YAO, *Some monotonicity properties of partial orders*, SIAM J. Alg. Disc. Meth., 1 (1980), pp. 251–258.

[4] L. HYAFIL, *Bounds for selection*, this Journal, 5 (1976), pp. 109–144.

[5] D. G. KIRKPATRICK, *Topics in the complexity of combinatorial algorithms*, Computer Science Department Technical Report TR 74, University of Toronto, Toronto, Canada, 1974.

[6] D. E. KNUTH, *Mathematical analysis of algorithms*, in Information Processing 71 Proceedings of the 1971 IFIP Congress, North-Holland, Amsterdam, 1972, pp. 19–27.

[7] ———, *The Art of Computer Programming, vol.* 1, *Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1968.

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

[9] D. W. MATULA, *Selecting the* tth best in average $n + O(\log\log n)$ *comparisons*, TR 73-9, Washington University, St. Louis, 1973.

[10] V. R. PRATT AND F. F. YAO, *On lower bounds for computing the* ith *largest element*, Proc. 14th IEEE Symposium on Switching and Automata Theory (1973), pp. 70–81.

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

[12] L. A. SHEPP, *The* FKG *inequality and some monotonicity properties of partial orders*, SIAM J. Alg. Disc. Meth., 1 (1980), pp. 295–299.

[13] M. SOBEL, Technical Reports 113, 114 (Nov. 1968), Department of Statistics, University of Minnesota, Minneapolis.

SIAM J. COMPUT.
Vol. 11, No. 3, August 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1103-0004 $01.00/0

THE EQUIVALENCE PROBLEM FOR DETERMINISTIC TWO-WAY SEQUENTIAL TRANSDUCERS IS DECIDABLE*

EITAN M. GURARI†

Abstract. The equivalence problem for deterministic two-way sequential transducers is a long time open problem which is known to be decidable for some restricted cases. Here, the problem is shown to be decidable also for the general case. In fact, the result holds even when the devices are allowed to make some finite number of nondeterministic moves.

Key words. transducer, decidability, PSPACE-completeness

1. Introduction. The problem of deciding the equivalence of deterministic two-way sequential transducers was first posed in [2]. Since then the problem has been shown to be decidable for the restricted cases in which the input heads are allowed to make only some finite number of reversals [5] or when the input strings are over some bounded language (i.e., a language of the form $a_1^* \cdots a_k^*$, where $a_1, \cdots, a_k$ are distinct symbols) [6]. These results are known to hold even when the devices are allowed to make some finite number of nondeterministic moves [5]. In this paper the above decidable results are shown to hold also for the general case, i.e., for deterministic two-way sequential transducers that are allowed to make some finite number of nondeterministic moves. On the other hand, it should be noted that the equivalence problem is known to be undecidable for nondeterministic one-way sequential transducers [4], [9]. The reader is referred to, e.g., [1] for the applicability of transducers in defining translations.

The results of this paper are given in the next section. The remainder of this section is devoted to recalling the (informal) definitions of the devices used in this paper.

A *two-way sequential transducer*, e.g., [2] is a two-way finite-state automaton which is augmented by an output tape. At the start of each computation, the two-way sequential transducer is set to a specific initial state, the input head is on the leftmost character of the input string and the output tape contains blanks only. An *atomic move* consists of changing the state of the finite-state control, moving the input head -1, 0 or 1 positions to the right and writing 0 or 1 nonblank characters on the output tape. Two such devices are said to be *equivalent* if they agree on all their input-output relations defined by their corresponding accepting computations. (Accepting configurations are assumed to be halting configurations.) A two-way sequential transducer is said to be *deterministic* if in each of its configurations it has at most one choice of next move. A *reversal-bounded m-counters machine*, e.g., [8] is a one-way finite-state automaton which is augmented by m ($\geqq 1$) counters. Each of the counters is capable of storing any nonnegative integer. On each atomic move, at most one of the counters is incremented by -1 or $+1$, while in every computation, a counter can alternately increase and decrease its value by no more than some finite number of times.

2. The results. Theorem 1 below is the main result of this paper. The proof of Theorem 1 generalizes an idea in [5].

THEOREM 1. *The equivalence problem is decidable for deterministic two-way sequential transducers.*

* Received by the editors December 19, 1979 and in revised form May 13, 1981. This research was supported in part by the National Science Foundation under grant MCS 79-09967.

† Department of Computer Science, State University of New York at Buffalo, Amherst, New York 14226.

Proof. Let M_1 and M_2 be any two given deterministic two-way sequential transducers. Without loss of generality it is assumed that M_1 and M_2 have a (same) distinguished symbol that they output when and only when they enter a halting configuration. Then a (nondeterministic) reversal-bounded 2-counters machine M is constructed such that M halts on some input if and only if M_1 and M_2 are inequivalent. (Note that M can be simulated by a pushdown automaton whose pushdown store behaves like a counter.) The result then follows from the decidability of the emptiness problem for reversal-bounded m-counters machines [5], [8].

M, when introduced with an input string, starts its computation by nondeterministically determining some positive integer, say v, and putting it into its counters, say C_1 and C_2. Then M nondeterministically simulates (in parallel) the computations of M_1 and M_2 on such an input. However, whenever M_i, $i=1$ or 2, is to output a symbol, M instead decreases C_i by 1 or leaves C_i unchanged depending on whether C_i contains a nonzero or a zero value, respectively. In addition, M records in its finite-state control the vth symbols in the outputs of M_1 and M_2 if and when they are encountered. M halts (on the given input) if and only if M_1 halts in an accepting configuration while M_2 does not halt in an accepting configuration, M_2 halts in an accepting configuration while M_1 does not halt in an accepting configuration, or M_1 and M_2 halt in accepting configurations but the symbols recorded in the finite-state control of M are distinct. The simulation of an accepting computation of M_i, $i=1, 2$, is given below.

In every accepting computation of M_i, its input head visits each of the symbols of the input strings at most s_i times, where s_i is the number of states of M_i. This is because M_i is deterministic and a repetition of a state on a symbol of a given input string implies a nonhalting computation. In addition, every such halting computation of M_i can be described by a time-input graph which shows the sequence of the transition rules used during the computation (see Fig. 1(a)). A node is at coordinate (ζ, μ) in the graph if and only if it corresponds to the transition rule associated with the ζth move in the computation and just before this move the input head of M_i was at the μth symbol of the input string.

Now, consider any accepting computation of M_i. Then a linear tree, say T, which describes the computation can also be constructed (see Fig. 1(b)). Each node in T corresponds to an ordered set of transition rules with a separator dividing the set into two. The ith node in T is associated with the ith symbol of the input string, say a_i, where

(i) the corresponding set of transition rules includes exactly those being used on the moves involving a_i and this in their order of being used; and

(ii) the separator divides the set of the transition rules into those used till (and including) the instant in which the vth output symbol is written (e.g., a subset of $\alpha_1, \cdots, \alpha_{11}$ in Fig. 1) and those used after the vth output symbol is written.

Thus, in simulating an accepting computation of M_i the device M needs just nondeterministically determine a sequence of ordered sets of distinct transition rules with a separator dividing each such set into two, where the sequence of these sets corresponds to the linear tree which describes the desired computation.

From the discussion above, the following algorithm can be given to describe the computation of the counter machine M on a given input.

$v \leftarrow$ (nondeterministically determine a positive integer); $C_1 \leftarrow v$; $C_2 \leftarrow v$;
M_1accept $\leftarrow$ 'false'; M_2accept $\leftarrow$ 'false';
initialize V_1; **initialize** V_2;
while both V_1 and V_2 contain at least one transition rule **do**
begin $V_1 \leftarrow$ **next** V_1; $V_2 \leftarrow$ **next** V_2; **end**;

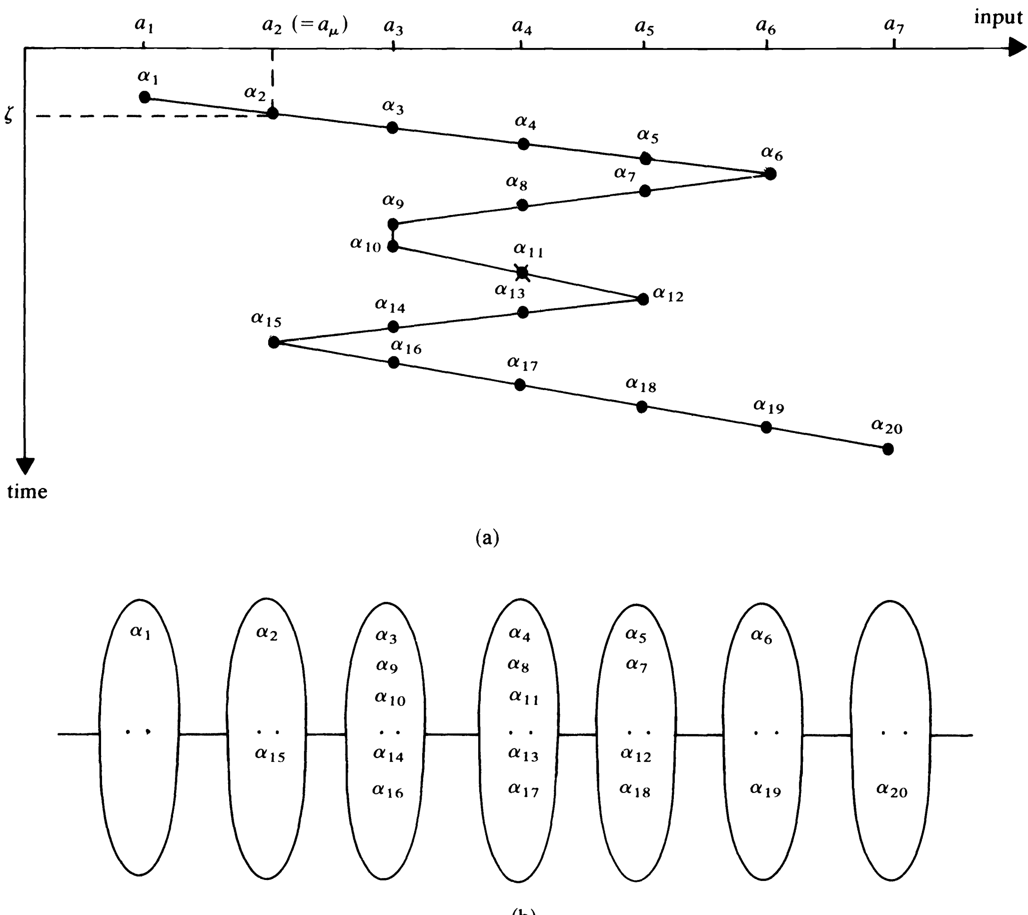


FIG. 1. *A description of an accepting computation of a deterministic two-way sequential transducer by* (a) *a graph and* (b) *a linear tree.*

> **if** $M_i$accept **and** V_i contains at least one transition rule **then reject;** (*improper simulation*)
> **if not** $M_i$accept **and** V_i contains more than one transition rule **then reject;** (*improper simulation*)
> $\Big\}\ i = 1, 2$
>
> **if** M_1accept $\neq M_2$accept **then accept;** (*exactly one of M_1 and M_2 accept the input*)
>
> **if** M_1accept **and** M_2accept **then**
>   **if** $C_1 = C_2 = 0$ **then**
>   **if** $sym_1 \neq sym_2$ **then accept;** (*both M_1 and M_2 accept the input but their vth output symbols are distinct*)
>
> **reject**

At any given instant of the computation, V_i holds the ordered set of the transition rules used by M_i in the moves from the input symbol under consideration, $i = 1, 2$. However, if the ordered set consists of more than s_i elements (i.e. M_i enters an infinite loop), then only the (at most $s_i + 1$) transition rules till and including the first repetition of transition rule are held in V_i. In addition, V_i holds a separator which divides the

set into two. Moreover, associated with each transition rule in V_i is the direction of the input head movement (i.e. $-1, 0$ or 1) just before the transition rule is reached. (In Fig. 1 the corresponding values of V_i are $((\alpha_1, 0), *)$, $((\alpha_2, 1), *, (\alpha_{15}, -1))$, $((\alpha_3, 1), (\alpha_9, -1), (\alpha_{10}, 0), *, (\alpha_{14}, -1), (\alpha_{16}, 1)), \cdots, (*, (\alpha_{20}, 1))$.) Whenever M (nondeterministically) determines a new value for V_i it also:

(a) reads an input symbol and checks that all the transition rules in the previous value of V_i are on such a symbol. This step, however, is not performed when V_i is initialized;

(b) decreases C_i by the value which equals the number of symbols contributed to the output of M_i by the transition rules that precede the separator in V_i;

(c) sets $M_i$accept to equal *true* if the last transition rule in V_i corresponds to a move to an accepting state;

(d) checks that the (new) value of V_i is compatible with its previous value;

(e) searches the simulated portion of the computation (given by the (new) value of V_i and the previous value of V_i) for a pair of consecutive transition rules, where the first transition rule immediately precedes a separator while the second transition rule immediately succeeds a separator. If such a pair is found, then M sets sym$_i$ to equal the output symbol of the first transition rule in this pair. □

For every two given deterministic sequential transducers M_1 and M_2 which are allowed to make some finite number, say k, of nondeterministic moves, two corresponding equivalent two-way sequential transducers $\hat{M}_1$ and $\hat{M}_2$ can be constructed with each of them being a union of 2^k deterministic two-way transducers. (At most two choices of next moves are assumed in each configuration of M_i, $i = 1, 2$). On a given input, $\hat{M}_i$, $i = 1, 2$, simulates its jth deterministic two-way sequential transducer. (The choice of j, $1 \leqq j \leqq 2^k$, is made nondeterministically.) The deterministic two-way sequential transducer, in turn, simulates the computation of M_i on the given input. However, whenever M_i is to make its lth, $1 \leqq l \leqq k$, nondeterministic move, the next move of the deterministic two-way sequential transducer is determined according to the value of the lth bit in the binary representation of j (zero corresponds to one choice and one corresponds to the other choice). Thus, M_1 and M_2 are inequivalent if and only if there exists an input-output relation definable by an accepting computation of a deterministic two-way sequential transducer which constitutes $\hat{M}_i$ but which is defined by none of the accepting computations of the deterministic two-way sequential transducers constituting $\hat{M}_{3-i}$, $i = 1$ or 2. Hence, the proof to Theorem 1 can be generalized to show also the next result.

THEOREM 2. *The equivalence problem is decidable for deterministic two-way sequential transducers which are allowed to make some finite number of nondeterministic moves.*

The nonemptiness problem for deterministic two-way finite-state automata is known to be PSPACE-complete [7]. On the other hand, the problem is solvable in t^{cm} time for reversal-bounded m-counters machines whose number of transition rules is t [5]. Thus, from [5], [7] and the proofs to Theorems 1 and 2, it follows that the inequivalence problem is PSPACE-complete for the transducers considered in Theorems 1 and 2. (See, e.g., [3] for the definition of PSPACE-complete.)

REFERENCES

[1] A. AHO AND J. ULLMAN, *The Theory of Parsing, Translation and Compiling*, vol. 1, Prentice-Hall, Englewood Cliffs, NJ, 1972.

[2] R. EHRICH AND S. YAU, *Two-way sequential transductions and stack automata*, Inform. and Control, 18 (1971), pp. 404–446.

[3] M. GAREY AND D. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, 1978.

[4] T. GRIFFITHS, *The unsolvability of the equivalence problem for ε-free nondeterministic generalized machines*, J. Assoc. Comput. Mach., 15 (1968), pp. 409–413.

[5] E. GURARI, *Transducers with decidable equivalence problem*, TR-CS-79-4, Univ. of Wisconsin Milwaukee, 1979; revised, TR-CS-81-182, State University of New York at Buffalo (1981).

[6] E. GURARI AND O. IBARRA, *Some decision problems concerning sequential transducers and checking automata*, J. Comput. and System Sci., 18 (1979), pp. 18–34.

[7] H. HUNT, *On the time and tape complexity of languages* I, Proc. of the Fifth Annual ACM Symposium on Theory of Computing, 1973, pp. 10–19.

[8] O. IBARRA, *Reversal-bounded multicounter machines and their decision problems*, J. Assoc. Comput. Mach., 25 (1978), pp. 116–133.

[9] ———, *The unsolvability of the equivalence problem for ε-free NGSM*'s *with unary input (output) alphabet and applications*, this Journal, 4 (1978), pp. 524–532.

SIAM J. COMPUT.
Vol. 11, No. 3, August 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1103-0005 $01.00/0

THE WORKING SET SIZE DISTRIBUTION FOR THE MARKOV CHAIN MODEL OF PROGRAM BEHAVIOR*

M. HOFRI† AND P. TZELNIC‡

Abstract. The history of modelling of the address sequences generated by computer programs (often termed "program behavior") follows a familiar pattern: the better a hypothetical model fits experimental evidence, the less amenable it is for calculation. In this paper programs that generate successive page references that can be described by a first order Markov chain are considered. We produce a closed form expression for the distribution and usable expressions for the first moments of the steady state size of their working set of pages. These expressions are also specialized for the independent reference model and the Easton model. Only standard Markov chain theory is used.

Key words. program behavior, paging algorithms, Markov chain model, working set, miss rate function, reverse chain, taboo probabilities

1. Introduction.

1.1. It is well known that the performance of virtual memory management policies depends significantly on certain properties of the sequence of addresses generated by the active programs—the so-called *program behavior* (PB).

Since the late sixties, people have attempted to model (analytically) program behavior. Early efforts concerned the *independent reference model*, IRM (Denning and Schwartz [1]), and the *last recently used stack model*, LRUSM (Denning, Savage and Spirn [28]). Both these models proved to be only partly effective in capturing the most prominent feature of real life behavior of programs known as *locality of reference* (Spirn [20]). The main redeeming quality of these models is that both are zero-order Markov chains (albeit with different state spaces), and as such they lend themselves easily to analysis.

Standing out among the analytical results that have been obtained for these models is the characterization of the working set by its mean for the IRM [1], as well as for the LRUSM (Lenfant [22]). The pmf of the working set size has also been obtained for the IRM (Vantilborgh [5]). Another important result for the IRM is the miss rate of the popular paging algorithm LRU (Gelenbe [23]).

1.2. The search for better models naturally leads to the representation of PB by stochastic processes more general than the zero order Markov chain. A prime candidate is the *first-order Markov chain model* (MCM). Higher order Markov chains can be reduced, by conventional techniques, to first order.

A Markov transition matrix has been used to characterize the evolution of a program's "states" (Aho, Denning and Ullman [24]), in order to find an optimal paging policy. This elusive target was sought using various optimization techniques (Aven and Kogan [7], Ingargiola and Korsch [25]). Special cases of the MCM have been investigated (Freiberger, Grenander and Sampson [11]). Several other papers investigate the validation of the MCM (Bogott and Franklin [8], Shedler and Tung [26]).

The interest in Markov chain modelling has been revived by the realization that PB is best modelled as a "phase process", evolving at two levels (Spirn [20]). At the higher level, a so-called *macro mechanism* governs the transitions among the *macro states*, or

* Received by the editors February 18, 1980, and in final revised form June 3, 1981.

† Department of Computer Science, Technion-Israel Institute of Technology, Haifa, Israel.

‡ Department of Computer Science, The Pennsylvania State University, University Park, Pennsylvania. Present address: M/S 1218, WANG Laboratories, One Industrial Avenue, Lowell, Massachusetts 01851.

phases. Lower level *micro mechanisms*, specific to each phase, control the page reference generation (transitions among the *micro states*—page references).

While such a stochastic complexity is beyond the IRM, the LRUSM and their variations, a MCM (or, as proposed in [11] and [20], a semi-Markov model) can do the job. A nearly decomposable Markov transition matrix, where diagonal subblocks are approximated by the IRM or other simple models and the migration between subblocks represents phase behavior, has been suggested (Courtois [3]). Another approach based on a MCM is that of Tzelnic and Gertner [19], where a Markov phase generator is overlaid on intraphase IRMs. A description of the clustering of page faults around phase transitions is thereby obtained. The miss rate for the LRU algorithm for the MCM is obtained by Tzelnic [18], [27]. This is computationally better than previous results (Franklin and Gupta [9], Glowacki [10]).

Lehman [15] has recently proposed a system modelling methodology, based on a special workload model suitable for the description and generation of PB. This is an MCM displaying phase behavior. Wen-Te K. Lin [16] uses a higher order Markov chain to model various activities of an operating system (CP/67), among them PB, and presents a statistical inference method to detect "regime" (phase) processes.

1.3. In this paper the program behavior is modelled by a general MCM defined over the same state space as the IRM (the address space of a program, reduced to its pages). The main result is the pmf of the working set size (Proposition 1). Its mean is obtained as a computationally manageable closed formula (Proposition 2).

The technique used in § 2 consists of a stochastic characterization of the *number of distinct states* visited by a Markov chain on a certain path. Proposition 1, which produces the distribution of this random variable, is a novel result. This random variable might be termed *length of path* (Tzelnic [18], [27]), to distinguish it from the well-known *first passage time* (the number of steps taken to reach for the first time a given state (Kemeny and Snell [13])).

The length of path (related to a notion of capacity in the context of potential theory, as applied to Markov chains) appears not to have received any formal consideration. It has considerable conceptual interest for applications where probabilistic walks on graphs might be beneficially described not only in terms of the number of steps taken, but also by the number of distinct nodes traversed (probabilistic automata, computational complexity, etc.).

The tools used to establish Proposition 1 are routine probabilistic techniques (taboo probability, effective conditioning, the inclusion-exclusion principle of combinatorial analysis). What renders them methodologically important in the present context is the demonstration of how an essentially path counting procedure leads to closed expressions.

1.4. The working set concept and its important implications for optimal multiprogramming control have been extensively presented (Denning [2]). The working set (WS) is defined as a set process $W(t, \tau)$ comprising at time t all the distinct page references through the course of the last τ references. It is suggested that $W(t, \tau)$ is a good estimator of the locality of reference at time t and the near future. Here τ—the WS "window"—is a parameter of the model, and its choice determines the goodness of the estimation. We denote the size of $W(t, \tau)$ by $w(t, \tau)$. The "pure" WS *memory management policy* is implemented by keeping in memory only the $w(t, \tau)$ pages of the current WS. To regulate the multiprogramming load only those tasks whose working sets are loaded in main memory are eligible to receive processor service.

The object of analysis usually is the limit of $w(t, \tau)$ as $t \to \infty$, if it exists. In § 2 the pmf of $w(\tau)$ and its first two moments are computed for the MCM. This extension of similar results for the IRM comes at a nonnegligible cost: first, whereas the IRM uses $n-1$ parameters to completely describe the evolution of the reference strings, just as the LRUSM does [20], and Easton's model makes do with n [12], the MCM requires $n(n-1)$. This makes the process of estimating the required parameters a longer, more expensive operation. Second, the calculations of the quantities of interest mentioned above are more complex in the face of the more elaborate dynamics. Still, using the procedure in § 2.7, based on Proposition 3, the mean size of the working set can be computed at a reasonably low expenditure.

1.5. In § 3 we show that Denning and Schwartz's formula [1] for the mean WS size for the IRM is indeed a special case of our result. We also apply our result to Easton's special MCM [12].

In § 4 we give a numerical example. A MCM displaying phase model characteristics is stipulated and the aforementioned quantities of interest are calculated. The WS size distribution we thus obtain is *multimodal*, in striking agreement with the empirical observations.

2. Working set size distribution.

2.1. Let the page reference generator be a first order time homogeneous Markov chain over the set of states $N = \{1, \cdots, n\}$. Further, assume it is ergodic (in particular, irreducible and aperiodic). Let its transition matrix be P. We identify the state of the chain at time t, $X(t)$, with the page referenced at time t. Conditioned on the page reference generated at time t, the probabilities of the next—$(t+1)$st—page reference are then given by

$$\Pr[X(t+1) = j | X(t) = i] = P_{ij}. \tag{2.1}$$

Let $\vec{\pi}$ be the steady state distribution of this chain ($\vec{\pi}P = \vec{\pi}$). Let $D(\vec{\pi})$ denote the diagonal matrix whose elements are π_i, $i = 1, \cdots, n$; and let $\tilde{P}$ denote the matrix transpose of P.

2.2. Following Kemeny and Snell [13], we define the reverse process of $X(t)$ as the process that corresponds to the evolution in reverse order of the original chain. It is further assumed that the process $X(t)$ has as initial distribution (at $t = 0$) its stationary distribution. This last assumption makes the reverse process thus defined a Markov chain, with the transition matrix

$$\hat{P} = [D(\vec{\pi})]^{-1} \tilde{P} D(\vec{\pi}), \tag{2.2}$$

(or termwise, $\hat{P}_{ij} = \pi_j P_{ji} / \pi_i$).

Furthermore, the matrix $\hat{P}$ possesses the same steady state distribution, $\vec{\pi}$, as the direct chain [13].

From the matrix $\hat{P}$ we introduce n matrices $\hat{P}_j$, each being the zero matrix, with the exception of column j which is column j of $\hat{P}$. We also find use for the n matrices $\underline{\hat{P}}_j$, defined by

$$\underline{\hat{P}}_j = \hat{P} - \hat{P}_j, \qquad j \in N, \tag{2.3}$$

i.e., identical to $\hat{P}$ with the exception of a zero column j.

The reverse chain is a natural tool to use when the dynamics of the WS of the program are considered, since given $X(t)$ it is $\hat{P}$ that governs its characteristics directly (in other words, through a backward looking window).

2.3. In the following propositions we assume that the initial memory state (= the set of program pages that is in main memory) is not important or, alternatively, that t is "large enough". Note that it is useless to say the process is in "steady state" since what concerns us requires observing the chain as it evolves.

The WS at time t, $W(t, \tau)$, is defined in terms of the process $\{X(t)\}$ as $W(t, \tau) = \{i | i \in \{X(t-\tau+1), X(t-\tau+2), \cdots, X(t-1), X(t)\}\}$, and $w(t, \tau)$ is the number of members of this set.

PROPOSITION 1. *The conditional probability that the working set size at time t is k, given that page i is referenced at time* $t-1$, *is*

$$\text{(2.4)} \qquad \begin{aligned} &\Pr[w(t, \tau) = k | X(t+1) = i] \\ &= \sum_{l=1}^{k} (-1)^{k-l} \binom{n-l}{k-l} \sum_{1 \leq j_1 < \cdots < j_l \leq n} \sum_{j=1}^{n} (\hat{P}_{j_1} + \cdots + \hat{P}_{j_l})^{\tau}_{ij}. \end{aligned}$$

(See the note below on the perhaps peculiar conditioning.)

Proof. Briefly presented the argument runs along the following lines:

1) Given the state of the memory reference process at time $t+1$, τ transitions governed by $\hat{P}$ determine $W(t, \tau)$.

2) It is convenient to interpret the event $\{w(t, \tau) = k\}$ as "the program avoids precisely $n-k$ pages during $t-\tau+1$ to t".

3) Denote by $A_{I|i}$ the event: $\{X(t-\tau+1), \cdots, X(t)\}$ avoids the set of pages I, given that $X(t+1) = i$.

4) Let $S_{r|i}$ denote the sum of probabilities $\sum_I \Pr[A_{I|i}]$, the summation being carried over all sets I s.t. $|I| = r$.

5) Then by 2) above and the inclusion-exclusion principle [14],

$$\Pr[w(t, \tau) = k | i] = \sum_{v=0}^{k} (-1)^v \binom{n-k+v}{v} S_{n-k+v|i}.$$

6) The probability that starting from i the next (under $\hat{P}$) τ references are all within a set of l pages J_l is given by

$$\sum_{r=1}^{n} (\hat{P}_{j_1} + \cdots + \hat{P}_{j_l})^{\tau}_{ir}, \qquad J_l = \{j_1, \ldots, j_l\}.$$

7) Using 4) and 6) we obtain for the value of S_r conditioned on $X(t+1) = i$

$$S_{r|i} = \sum_{J_{n-r}} \sum_{j=1}^{n} (\hat{P}_{j_1} + \cdots + \hat{P}_{j_{n-r}})^{\tau}_{ij}, \qquad J_{n-r} = \{j_1 \cdots j_{n-r}\},$$

the summation being on all sets of size $n-r$.

8) Substituting the last result into 5) and putting $k - v = l$ yields (2.4) (when we note that $S_n = 0$). □

Using Proposition 1, we obtain, by removing the conditioning on $X(t+1)$;

COROLLARY 1. *The steady state* pmf *of the working set size is*

$$\text{(2.5)} \qquad \Pr[w(\tau) = k] = \sum_{l=0}^{k} (-1)^{k-l} \binom{n-l}{k-l} \sum_{(1 \leq j_1 < \cdots < j_l \leq n)} \sum_{i,j=1}^{n} \pi_i (\hat{P}_{j_1} + \cdots + \hat{p}_{j_l})^{\tau}_{ij}.$$

Note. Proposition 1 was stated using an "unnatural" conditioning—normally $W(t, \tau)$ is of interest at time t when $X(t+1)$ is not known. Indeed, $\Pr[w(t, \tau) =$

$k|X(t)=i]$ can be evaluated as well, and one obtains via essentially the same route,

$$\Pr[w(t,\tau)=k|X(t)=i]$$
$$=\sum_{l=1}^{k}(-1)^{k-l}\binom{n-l}{k-l}\sum_{J_l}\binom{k-l}{n-l}^{I_{i\notin J_l}}\sum_{j=1}^{n}(\hat{P}_{j_1}+\cdots+\hat{P}_{j_l})_{ij}^{\tau},$$

where I_A is 1 when A holds and 0 otherwise.

This, however, is a more complex expression, and its derivation is less intuitive. Inasmuch as our main interest lies with the *unconditional* distribution $\Pr[w(t)=k]$, and clearly

$$\Pr[w(t,\tau)=k]=\sum_{i=1}^{n}\Pr[w(t,\tau)=k|X(t)=i]\Pr[X(t)=i]$$
$$=\sum_{j=1}^{n}\Pr[w(t,\tau)=k|X(t+1)=j]\Pr[X(t+1)=j]$$

and, further, in the limit $t\to\infty$, both randomizations use the same pmf $\vec{\pi}$, Corollary 1 is indeed the result we need.

2.4. Using (2.5) the moments of the steady state WS size can now be computed. Denote the mean by $S(\tau)$:

$$S(\tau)=E[w(\tau)]=\sum_k k\Pr[w(\tau)=k].$$

PROPOSITION 2.

$$S(\tau)=n-\sum_{k=1}^{n}\sum_{i,j=1}^{n}\pi_i(\hat{P}_k)_{ij}^{\tau}. \tag{2.6}$$

Proof.

$$S(\tau)=\sum_{k=1}^{n}k\Pr[w(\tau)=k]=\sum_{k=1}^{n}k\sum_{l=1}^{k}(-1)^{k-l}\binom{n-l}{k-l}S_l, \tag{2.7}$$

where

$$S_l=\sum_{i,j=1}^{n}\pi_i\sum_{1\leqq j_1<\cdots<j_l\leqq n}(\hat{P}_{j_1}+\cdots+\hat{P}_{j_l})_{ij}^{\tau}.$$

By changing the order of summation over k and l in (2.7) and summing over $u=k-l$ from zero to n, we obtain

$$S(\tau)=\sum_{l=1}^{n}S_l\sum_{u=0}^{n-l}(-1)^u\binom{n-l}{u}(u+l).$$

The sum over u can be split into two components

$$l\sum_{u=0}^{n-l}(-1)^u\binom{n-l}{u}=l\delta_{l,n},\qquad \sum_{u=0}^{n-l}(-1)^u u\binom{n-l}{u}=-\delta_{l,n-1}, \tag{2.8}$$

where $\delta_{i,j}$ is one when $i=j$ and zero otherwise. Therefore

$$S(\tau)=nS_n-S_{n-1}=n\sum_{i=1}^{n}\pi_i\sum_{j=1}^{n}(\hat{P})_{ij}^{\tau}-\sum_{i,j=1}^{n}\pi_i\sum_{1\leqq j_1<\cdots<j_{n-1}\leqq n}(\hat{P}_{j_1}+\cdots+\hat{P}_{j_{n-1}})_{ij}^{\tau}$$
$$=n-\sum_{i,j=1}^{n}\pi_i\sum_{k=1}^{n}(\hat{P}_k)_{ij}^{\tau}.$$

(See § 2.8 concerning the evaluation of these quantities.) □

In an analogous manner, we also obtain

$$S^{(2)}(\tau)=E[w^2(\tau)]=n^2S_n-(2n-1)S_{n-1}+2S_{n-2}$$
(2.9)
$$=n^2(2n-1)\sum_{i,j=1}^{n}\pi_i\sum_{k=1}^{n}(\hat{\underline{P}}_k)^{\tau}_{ij}+2\sum_{i,j=1}^{n}\pi_i\sum_{1\leq j_1<\cdots<j_{n-2}\leq n}(\hat{P}_{j_1}+\cdots+\hat{P}_{j_{n-2}})^{\tau}_{ij},$$

whence the variance is immediate.

2.5. A related interesting result is an expression for the miss rate at equilibrium, $M(\tau)$, in the present model, under the WS policy. $M(\tau)$ is defined as $\lim_{t\to\infty}\Pr[X(t)\notin W(t-1,\tau)]$. This is derived by observing that the present model satisfies the assumptions of [1][1], by virtue of which the following relation was shown to hold:

$$M(\tau)=S(\tau+1)-S(\tau). \tag{2.10}$$

COROLLARY 2.

$$M(\tau)=\sum_{k=1}^{n}\sum_{i,j=1}^{n}\pi_i[(\hat{\underline{P}}_k)^{\tau}(I-\hat{\underline{P}}_k)]_{ij}. \tag{2.11}$$

2.6. We now present alternative expressions for $S(\tau)$ and $M(\tau)$. These perhaps are not as intuitive in derivation as the preceding ones, but are computationally superior.

PROPOSITION 3.

$$S(\tau)=\sum_{s=0}^{\tau-1}\sum_{j,k=1}^{n}\pi_k(\hat{\underline{P}}_k)^{s}_{kj}, \tag{2.12}$$

$$M(\tau)=\sum_{k,j=1}^{n}\pi_k(\hat{\underline{P}}_k)^{\tau}_{kj}. \tag{2.13}$$

Proof (directly from Proposition 2).

$$S(\tau)=n-\sum_{k,j=1}^{n}U_{jk}(\tau), \tag{2.14}$$

where $U_{jk}(\tau)=\sum_{i=1}^{n}\pi_i(\hat{\underline{P}}_k)^{\tau}_{ij}$.

A recursive evaluation for these $U(\tau)$ follows:

$$U_{jk}(\tau)=\sum_{i=1}^{n}\pi_i\sum_{h=1}^{n}(\hat{\underline{P}}_k)_{ih}(\hat{\underline{P}}_k)^{\tau-1}_{hj}=\sum_{h=1}^{n}\left\{\sum_{i=1}^{n}\pi_i\hat{P}_{ih}-\sum_{i=1}^{n}\pi_i(\hat{P}_k)_{ih}\right\}(\hat{\underline{P}}_k)^{\tau-1}_{hj}$$
$$=\sum_{h=1}^{n}\left\{\pi_h-\sum_{i=1}^{n}\pi_iP_{ik}\delta_{kh}\right\}(\hat{\underline{P}}_k)^{\tau-1}_{hj} \tag{2.15}$$
$$=U_{jk}(\tau-1)-\pi_k(\hat{\underline{P}}_k)^{\tau-1}_{kj}.$$

This difference equation for $U(\tau)$ has the initial value

$$U_{jk}(1)=\pi_j-\pi_k\delta_{jk} \tag{2.16}$$

[1] These are: the considered reference strings are infinitely long; the stochastic generator is time-homogeneous; the correlation between the references $X(t)$ and $X(t+k)$ vanishes in the limit $k\to\infty$.

and thus yields

(2.17)
$$U_{jk}(\tau)=\pi_j-\pi_k\sum_{s=0}^{\tau-1}(\hat{\underline{P}}_k)^s_{kj}.$$

Substituting in (2.14) we obtain

$$S(\tau)=n-\sum_{k,j=1}^{n}\pi_j+\sum_{k,j=1}^{n}\pi_k\sum_{s=0}^{\tau-1}(\hat{\underline{P}}_k)^s_{kj}=\sum_{s=0}^{\tau-1}\sum_{j,k=1}^{n}\pi_k(\hat{\underline{P}}_k)^s_{kj},$$

and therefore

$$M(\tau)=S(\tau+1)-S(\tau)=\sum_{j,k=1}^{n}\pi_k(\hat{\underline{P}}_k)^\tau_{kj}. \qquad \square$$

2.7. Remarks.

1. In evaluating $M(\tau)$, it is not necessary to evaluate all of $(\hat{\underline{P}}_k)^\tau$; it suffices to evaluate for each k only the kth row of these matrices via the obvious $\text{row}_k(\hat{\underline{P}}_k)^i=[\text{row}_k(\hat{\underline{P}}_k)^{i-1}]\hat{\underline{P}}_k$.
2. The last relations suggest the following calculation scheme:
   (a) $S(1)=1$;
   (b) $M(\tau)$ as given in (2.13);
   (c) $S(\tau+1)=S(\tau)+M(\tau)$.
3. Using a familiar representation of the mean WS size [1],

(2.18)
$$S(\tau)=\sum_{s=0}^{\tau-1}\left(1-\sum_{k=1}^{n}\pi_k F_k(s)\right),$$

where $F_k(\cdot)$ is the distribution function for the inter-reference interval for page k, we get, by comparison with (2.12),

(2.19)
$$F_k(s)=1-\sum_{j=1}^{n}(\hat{\underline{P}}_k)^s_{kj}.$$

(This is actually a compact way to represent the summation over all sample paths.)

2.8. Complexity of calculations. It is of some interest to evaluate the number of operations involved in the various expressions obtained for $S(\tau)$, $M(\tau)$, etc. (essentially additions and multiplications of elements of $\hat{P}$ and $\vec{\pi}$; no account is taken of loop control variables, the calculation of $\vec{\pi}$ itself and various "bookkeeping" chores). Equation (2.4) requires, for each k and i, $n\tau\sum_{l=1}^{k} l\binom{n}{l}$ and for the complete (conditioned) pmf, for every i, $n^2\tau(n+1)2^{n-2}$ operations. Covering all i adds a factor of n. The computation of the steady state pmf of the working set size (2.5) requires therefore about the same number of operations as that needed for (2.4) reduced by a factor of n. A straightforward calculation of (2.6) requires $n^2(n-1)^2\tau$ operations (some elaboration indicates that this figure can be rather simply reduced by a factor of n by explicitly using $\vec{\pi}=\vec{\pi}\hat{P}$. The calculation is more complex to control, however). Equation (2.12), using the procedure of Remark 1, § 2.7, can be done in $n^3\tau$ operations (approximately the same as the elaborate calculation of (2.6)). Some of these complexities can be somewhat reduced by employing the like of Straßen's algorithm.

3. Special cases. As an illustration let us evaluate the above expressions for two simple models of program behavior.

3.1. Independent reference model. Given that $P_{ij}=p_j$, $\sum_{j=1}^n p_j=1$, the following results for the IRM follow:

$$\pi_i=p_i,\quad \hat{P}_{ij}=P_{ij}=p_j,\quad (\underline{\hat{P}}_k)_{ij}^{\tau}=(1-p_k)^{\tau-1}(1-\delta_{jk})p_j \tag{3.1}$$

for $i, j=1,\cdots,n$, $\tau\geqq 1$. Hence

$$\begin{aligned} S(\tau)&=n-\sum_{k=1}^{n}\sum_{i,j=1}^{n}\pi_i(\underline{\hat{P}}_k)_{ij}^{\tau}=n-\sum_k\sum_{ij}p_ip_j(1-p_k)^{\tau-1}(1-\delta_{jk})\\ &=n-\sum_{k=1}^{n}(1-p_k)^{\tau}, \end{aligned}\tag{3.2}$$

consistent with the results in [1].

3.2. Easton's model [12]. This model makes use of a Markov chain of special structure. It is claimed to be a good description of some interactive data base reference strings [12]. The transition probabilities are given by

$$P_{ii}=\alpha_i=1-r+r\lambda_i,\qquad P_{ij}=\beta_j=r\lambda_j,\qquad i\neq j, \tag{3.3}$$

where $0<r\leqq 1$, $\sum_{i=1}^n\lambda_i=1$ and $\lambda_i>0$ for $i=1,\cdots,n$. For this model, we can easily verify that:

$$\pi_i=\lambda_i,\qquad \hat{P}_{ij}=P_{ij},\qquad i,j=1,\cdots,n. \tag{3.4}$$

In order to compute $S(\tau)$ for Easton's model, we use the following relation (which is proved below):

$$\sum_j(\underline{\hat{P}}_k)_{ij}^m=(1-\beta_k)^{m-1}[(1-\beta_k)(1-\delta_{ik})+\delta_{ik}(1-\alpha_k)]. \tag{3.5}$$

With this it follows that

$$\begin{aligned} \sum_{i,j=1}^{n}\pi_i(\underline{\hat{P}}_k)_{ij}^{\tau}&=(1-\beta_k)^{\tau-1}\sum_i\lambda_i[(1-\beta_k)(1-\delta_{ik})+\delta_{ik}(1-\alpha_k)]\\ &=(1-\beta_k)^{\tau-1}\sum_i\lambda_i[(1-\beta_k)+\delta_{ik}(r-1)]\\ &=(1-\beta_k)^{\tau-1}[1-\beta_k+\lambda_k(r-1)]\\ &=(1-\beta_k)^{\tau-1}(1-\beta_k+r\lambda_k-\lambda_k)\\ &=(1-\beta_k)^{\tau-1}(1-\lambda_k). \end{aligned}$$

Hence,

$$S(\tau)=n-\sum_{k,i,j=1}^{n}\pi_i(\underline{\hat{P}}_k)_{ij}^{\tau}=n-\sum_{k=1}^{n}(1-\lambda_k)(1-r\lambda_k)^{\tau-1}, \tag{3.6}$$

which is identical with Easton's result.

To prove the relation (3.5), one may argue as follows: $\sum_j(\underline{\hat{P}}_k)_{ij}^m$ is actually the taboo probability of not entering state k, following an exit from state i, through m successive steps (along any sample path of the reverse Markov chain, $\hat{X}(t)$).

Two cases here have to be considered, corresponding to $i\neq k$ or $i=k$ in the initial state. Due to the sample structure of the Markov chain, the above probabilities are $(1-\beta_k)^m$ in the first case and $(1-\beta_k)^{m-1}(1-\alpha_k)$ in the second one.

A computational proof, which has the side benefit of yielding an explicit representation of the powers of $\hat{P}_k$, is given in [17][2] and [18].

Note. Both examples above happen to possess self-dual transition matrices (i.e., $\hat{P}=P$). This is neither a requirement of any of the procedures developed here nor does it help in the calculation.

4. Example.

4.1. We present here a special MCM that exhibits phase behavior. The locality sets are disjoint in this example. Whereas this assumption limits the use of near decomposability approximation techniques, it is not required for the exact calculations.

This example is not a full validation of the MCM. We do not attempt to estimate a Markov transition matrix from an actual program reference trace and then to compare the theoretical and the empirical distributions of the WS size. The example merely illustrates the capability of the MCM to produce multimodal distributions of the WS size, in agreement with well-established empirical evidence.

It also shows how affordable our computational procedure is when combined with the "near-decomposition" approximation method (Courtois [3]), even for larger address spaces than considered here. Thus we present the exact as well as an approximate pmf. Both are bimodal and almost identical (the maximum error is 7%). While to obtain the exact pmf (for an address space of 10 pages) about 300,000 multiplications are necessary, fewer than 12,000 suffice for the calculation of the approximate pmf.

4.2. We assume that the 10×10 page transition matrix P is given as displayed in Fig. 1. Note that the two square diagonal blocks corresponding to pages 1, 2, 3 and 4 to

.1	.545	.33	.00287	.0043	.00646	.000717	.000717	.0043	.00502
.64	.261	.0592	.00118	.00474	.00118	.00592	.00829	.00711	.0118
.29	.29	.401	.00384	.00256	.000426	.00298	.00213	.00341	.00426
.0231	.00865	.00288	.0692	.0807	.104	.049	.141	.259	.262
.02	.00998	.0175	.0175	.227	.127	.13	.0798	.247	.125
.0286	.0357	.0143	.0964	.0357	.339	.0286	.182	.139	.1
.0064	.0213	.0171	.196	.113	.1	.203	.0128	.164	.166
.0197	.0172	.00246	.204	.0319	.00491	.17	.214	.155	.182
.0165	.0212	.00708	.172	.236	.21	.0566	.0731	.0840	.123
.0121	.0194	.0121	.145	.206	.102	.206	.0654	.102	.131

FIG. 1. *The page transition matrix P.*

10, respectively, have much larger elements than the offdiagonal blocks. This suggests that this model has two phases—let these be called Phase I and Phase II—with the above locality sets. Using a computational procedure based on (2.5), the limit pmf of the working set size

$$D(\tau,k)=\Pr[w(\tau)=k],\quad k=1,\cdots,10,\quad \tau=1,\cdots,10,$$

is obtained as displayed in Fig. 2.

From this (an easier way would be to use the procedure outlined in § 2.7, without first calculating the pmf) the mean WS size, $S(\tau)$, and the miss rate, $M(\tau)$, are obtained as displayed in Fig. 3 for all $\tau=1,\cdots,10$. Figure 4 exhibits graphically the pmf $D(\tau,k)$ for $\tau=4$, 6, 8 and 10.

[2] An earlier version of this paper.

τ \ k	1	2	3	4	5	6	7	8	9	10
1	1.0									
2	.215	.785								
3	.0587	.56	.381							
4	.0184	.365	.472	.145						
5	.00625	.246	.471	.21	.0672					
6	.00223	.172	.458	.206	.138	.0247				
7	.00082	.124	.449	.173	.179	.0665	.00719			
8	.00031	.00914	.444	.137	.191	.110	.0024	.0017		
9	.00012	.00684	.440	.107	.184	.145	.048	.00688	.00031	
10	.00004	.0517	.436	.0853	.167	.168	.0747	.0161	.00153	.000028

FIG. 2. *The* WS *size probability mass function*, $D(t, k)$, $\tau, k = 1, \cdots, 10$.

τ	1	2	3	4	5	6	7	8	9	10
$S(\tau)$	1.0	1.78	2.32	2.74	3.09	3.38	3.63	3.66	4.07	4.26
$M(\tau)$	.785	.538	.421	.344	.292	.255	.228	.207	.19	—
$S'(\tau)$	1.0	1.78	2.31	2.7	3.0	3.23	3.43	3.58	3.71	3.82
$M'(\tau)$	.782	.523	.391	.3	.237	.192	.157	.131	.11	—

FIG. 3. *The* WS *mean size and the* WS *miss rate.* $S(\tau)$, $M(\tau)$, $\tau = 1, \cdots, 10$—*exact solution*; $S'(\tau)$, $M'(\tau)$, $\tau = 1, \cdots, 10$—*approximate solution.*

A global maximum at $k = 3$ is common to all (corresponding to the size of Phase I locality set); a marked knee, at $k = 5$, appears for $\tau > 4$, becomes a second maximum for $\tau \geqq 7$ and drifts to $k = 7$ as τ becomes 10 (corresponding to the size of Phase II locality set).

About 280,000 multiplications were performed to obtain these quantities.

4.3. Following Courtois [3] we derive from the matrix P a completely decomposable matrix P' consisting of two strictly disjoint localities corresponding to Phases I and II above:

$$P = P' + \varepsilon C,$$

where $\varepsilon = 0.0786$, and the error matrix C satisfies

$$\max_{i,j=1,\cdots 10} |c_{ij}| = 1.$$

Thus, the micro-level models are again MCM as specified by the transition matrices P_1 (Phase I) and P_2 (Phase II) displayed in Fig. 5. The steady state probabilities that the macro model is in Phases I and II are

$$\pi_1 = 0.622 \quad \text{and} \quad \pi_2 = 0.378,$$

respectively. The exact pmf's of the WS size for the micro models P_1 and P_2 can now be calculated as in § 4.2,

$$\begin{aligned} D_1(\tau, k) &= \Pr[w_1(\tau) = k], \qquad k = 1, \cdots 3, \quad \tau \leqq 10 \\ D_2(\tau, k) &= \Pr[w_2(\tau) = k], \qquad k = 1, \cdots 7, \quad \tau \leqq 10, \end{aligned} \tag{4.1}$$

and the approximate pmf of the WS size for the phase model is then obtained as

$$D'(\tau, k) = \pi_1 D_1(\tau, k) + \pi_2 D_2(\tau, k), \qquad k \leqq 10, \quad \tau \leqq 10. \tag{4.2}$$

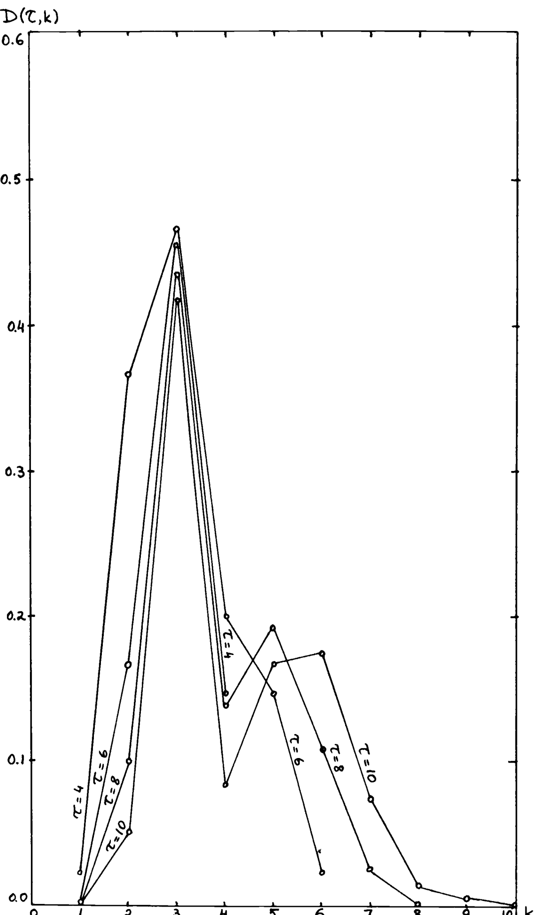


FIG. 4. *The* WS *size probability mass function.* $S(\tau, k), k=1, \cdots, 10, \tau=4, 6, 8, 10.$

In (4.2) the RHS terms corresponding to those values of k that are not covered in (4.1) must be taken as 0.

Figure 6 displays $D'(\tau, k)$ in tabular form, and Fig. 7 displays them graphically for the same window sizes as in § 4.2. It can be seen that the relative error

$$\max \frac{|D'(\tau, k)-D(\tau, k)|}{D(\tau, k)}$$

P_1

.1	.545	.354
.64	.261	.099
.309	.29	.401

P_2

.104	.0807	.104	.049	.141	.259	.262
.0648	.227	.127	.13	.0798	.247	.125
.175	.0357	.339	.0286	.182	.139	.1
.241	.113	.1	.203	.0128	.164	.166
.243	.0319	.00491	.17	.214	.155	.182
.217	.236	.21	.0566	.0731	.0849	.123
.189	.206	.102	.206	.0654	.102	.131

FIG. 5. *The micro-model page transition matrices,* P_1 *and* P_2.

τ \ k	1	2	3	4	5	6	7	8	9	10
1	1.0									
2	.0218	.782								
3	.006	.575	.365							
4	.0019	.379	.489	.113						
5	.0065	.256	.518	.175	.0451					
6	.00235	.178	.529	.176	.102	.0118				
7	.00087	.128	.504	.148	.145	.0369	.00153			
8	.00033	.0934	.553	.113	.165	.0696	.00629	0.0		
9	.00013	.0693	.565	.081	.166	.103	.0152	0.0	0.0	
10	.00005	.052	.576	.056	.155	.132	.0282	0.0	0.0	0.0

FIG. 6. *The* WS *size probability mass function (approximation).* $D'(\tau, k)$, τ, $k = 1, \cdots, 10$.

increases with τ and is less than 7%. The approximate mean WS size and miss rate ($S'(\tau)$ and $M'(\tau)$, respectively) are displayed in Fig. 3.

The number of multiplications required for the approximate solution is about 12,000 (more than 20 times fewer than for the exact computation).

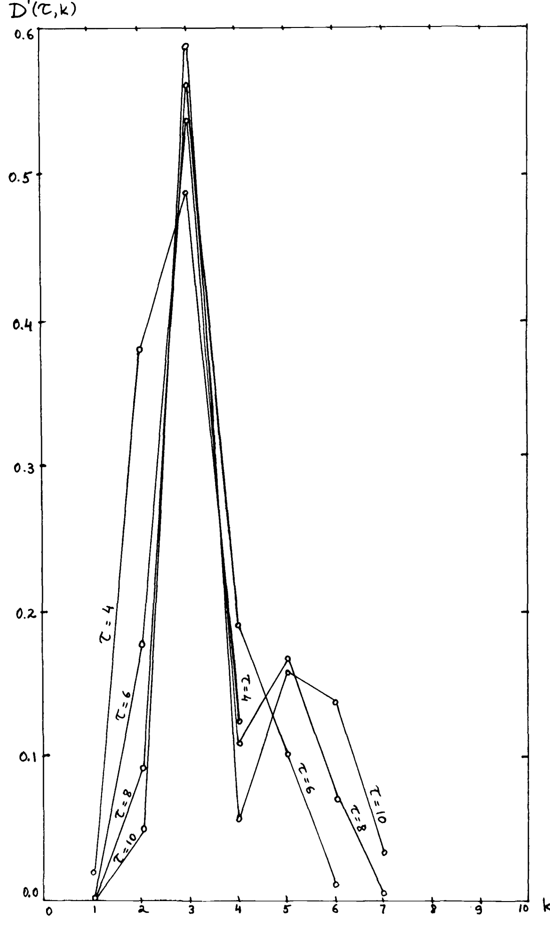


FIG. 7. *The* WS *size probability mass function.* $D'(\tau, k)$, $k = 1, \cdots, 10$, $\tau = 4, 6, 8, 10$.

5. Conclusion.

5.1. In this paper it was shown how quantities related to the WS of a program can be handled when the program obeys the Markov chain model. Specifically, we produced expressions for the mean and variance of the WS size at equilibrium.

Although the calculations are inherently of the path counting type, closed expressions were obtained for two special models of program behavior. As is clear on examining the complexities calculated in § 2.8—even for the closed expressions—and on considering the difficulty of estimating the matrix P, even if a program did exist for which all the MCM assumptions held, using our expressions to evaluate the optimal τ, say, would be foolhardy—and this even before external effects of the computing system have been taken care of. How the MCM can serve is as a test-bed on which different memory management policies can be compared. Obviously, it is desirable that such a model display "real world" characteristics, and this the MCM can do.

The following fact is also worth a remark. When it comes to computing the pmf of $w(\tau)$, IRM [5] requires about as much work as the MCM does. Moreover, as shown in [18], the same holds when computing the miss rate of the least recently used paging algorithm. These measures all present an essentially exponential computational complexity, no matter which program behavior model is used.

5.3. A promising approach towards the validation of the MCM is by using the WS size pmf. MCM parameters (transition matrix) are estimated from an actual program trace. From this, the WS size pmf is computed and then compared to the empirical distribution. However, for programs with large address spaces, the exact computation is not feasible. Using the decomposition approximation instead limits the validation to a class of special cases. Nevertheless, this class is large enough to include the programs of interest: those for which the transition matrix has a nearly block-diagonal form with small size blocks (small, nearly disjoint locality sets). Such a program of validation is currently being undertaken.

REFERENCES

[1] P. J. Denning and S. C. Schwartz, *Properties of the working set model*, Comm. ACM, 15 (1972), pp. 191–198.

[2] P. J. Denning, *Working sets past and present*, IEEE Trans. Soft. Eng., SE-6 (1980), pp. 64–84.

[3] P. J. Courtois, *Decomposability*, Academic Press, New York, 1977.

[4] P. J. Courtois and H. Vantilborgh, *A decomposable model of program paging behavior*, Acta Inf., 6 (1976), pp. 251–275.

[5] H. Vantilborgh, *On the working set size distribution and its normal approximation*, BIT, 14 (1974), pp. 240–251.

[6] Arvind, R. Y. Kain and E. Sadeh, *On reference string generation process*, Proc. 4th ACM Symposium on Operating System Principles, 1973, pp. 80–87.

[7] O. I. Aven and Ya. A. Kogan, *Stochastic control of paging in a two-level computer memory*, Automatica, 11 (1975), pp. 309–313.

[8] R. P. Bogott and M. A. Franklin, *Evaluation of Markov program models in virtual memory systems*, Software-Pract. Exp., 5 (1975), pp. 337–346.

[9] M. A. Franklin and R. K. Gupta, *Computation of* PF *probabilities from program transition diagrams*, Comm. ACM, 17 (1974), pp. 186–191.

[10] C. Glowacki, *A closed form expression of the page fault rate for the* LRU *algorithm in a Markovian reference model of program behavior*, Proc. International Computing Symposium, E. Morlet and D. Ribbens, eds., North-Holland, Amsterdam, 1977, pp. 315–318.

[11] W. F. Freiberger, U. Grenander and P. D. Sampson, *Patterns in program behavior*, IBM J. Res. Dev., 19 (1975), pp. 230–243.

[12] M. C. Easton, *Model for interactive data base reference strings*, IBM J. Res. Dev., 19 (1975), pp. 550–557.

[13] J. G. KEMENY AND J. L. SNELL, *Finite Markov Chains*, D. Van Nostrand, Princeton, NJ, 1960.
[14] W. FELLER, *An Introduction to Probability Theory and Its Applications*, vol. I, 3rd ed., John Wiley, New York, 1968.
[15] A. LEHMAN, *Performance evaluation and prediction of storage hierarchies*, Proc. 7th IFIP W.G.7.3 International Symposium on Computing Performance Modeling, Measurement and Evaluation, May 1980, Toronto.
[16] WEN-TE K. LIN, *Analysis of a* VM *operating system*, 4th International Symposium on Modeling and Performance Evaluation of Computer Systems, Vienna, Feb. 1979.
[17] M. HOFRI AND P. TZELNIC, *On the working set size for the Markov chain model of program behavior*, in Performance of Computer Systems, M. Arato et al., eds., North-Holland, Amsterdam, 1979, pp. 393–405.
[18] P. TZELNIC, *Stochastic models of program behavior in paged virtual memory systems*, Ph.D. Thesis, Israel Institute of Technology, 1979.
[19] P. TZELNIC AND I. GERTNER, *An approach to program behavior modelling and optimal memory control*, J. ACM, to appear.
[20] J. R. SPIRN, *Program Behavior: Models and Measurements*, Elsevier-Holland, Amsterdam, 1976.
[21] W. F. KING III, *Analysis of demand paging algorithms*, Proc. IFIP Congress, Ljubljana, Aug., 1971, TA-3-155-TA-3-159.
[22] J. LENFANT, *Evaluation sur des modeles des comportement de programme de la taille d'un ensemble de travail*, Proc. International Symposium Rocquencourt, Apr., 1974, pp. 218–236.
[23] E. GELENBE, *A unified approach to the evaluation of a class of replacement algorithms*, IEEE Trans. Comp., C-226 (1973), pp. 611–618.
[24] A. V. AHO, P. J. DENNING AND J. D. ULLMAN, *Principles of optimal page replacement*, J. ACM, 18 (1971), pp. 80–93.
[25] G. INGARGIOLA AND J. F. KORSH, *Finding optimal demand paging algorithms*, J. ACM, 21 (1974), pp. 40–53.
[26] G. S. SHEDLER AND C. TUNG, *Locality in page reference strings*, this Journal, 1 (1972), pp. 218–241.
[27] P. TZELNIC, *The length of path for finite Markov chains, and its application to modelling program behaviour and interleaved memory systems*, ORSA-TIMS Meeting, Jan., 1981, Boca Raton, Florida.
[28] P. J. DENNING, J. E. SAVAGE AND J. R. SPIRN, *Models for locality in program behavior*, Comp. Sci. Rept. TR-107, Princeton Univ., Princeton, NJ, 1972.

SIAM J. COMPUT.
Vol. 11, No. 3, August 1982

© 1982 Society for Industrial and Applied Mathematics

0097-5397/82/1103-0006 $01.00/0

RAPID MULTIPLICATION OF RECTANGULAR MATRICES*

D. COPPERSMITH†

Abstract. The number of essential multiplications required to multiply matrices of size $N \times N$ and $N \times N^{0.172}$ is bounded by $CN^2 \log^2 N$.

Key words. matrix multiplication, tensor rank, algebraic complexity

Introduction. Let Rank$\langle K, M, N\rangle$ denote the number of essential multiplications required to multiply a $K \times M$ matrix by an $M \times N$ matrix, i.e., the rank of the 3-dimensional tensor defining this matrix multiplication. We show here, by two different routes, the existence of a positive number α such that Rank $\langle N, N, N^\alpha\rangle \leqq CN^2 \log^2 N$.

THEOREM. *There is a positive constant* $\alpha = 2 \log 2/5 \log 5 = 0.17227$ *such that*

$$\text{Rank}\,(\langle N, N, N^\alpha\rangle) = O(N^2(\log N)^2).$$

Remark. This agrees well with the trivial lower bound

$$\text{Rank}\,(\langle N, N, N^\alpha\rangle) \geqq N^2.$$

Proof. The proof may be done in two ways. Each relies on existing basic constructions (each due to Schönhage), and minor modifications to existing techniques for combining basic constructions (i.e., the exponential direct sum theorem, partial matrix multiplication and approximate algorithms). The modifications involve (1) selecting a binomial coefficient to maximize an "area" rather than a "volume" (which allows the agreement between upper and lower bounds) and (2) doing two arguments at once (e.g., the exponential direct sum theorem and approximate algorithms) which serves only to improve the "error bound" from N^ε to $C(\log N)^2$.

Proof version 1 (*partial matrix multiplication*). Begin with the following construction, due to Schönhage [3]:

$$\begin{aligned}&(a_{11}+x^2a_{12})(b_{21}+x^2b_{11})(c_{11})+(a_{11}+x^2a_{13})(b_{31})(c_{11}-xc_{21})\\&\quad+(a_{11}+x^2a_{22})(b_{21}-xb_{12})(c_{12})\\&\quad+(a_{11}+x^2a_{23})(b_{31}+xb_{12})(c_{12}+xc_{21})-(a_{11})(b_{21}+b_{31})(c_{11}+c_{12})\\&=x^2(a_{11}b_{11}c_{11}+a_{11}b_{12}c_{21}+a_{12}b_{21}c_{11}+a_{13}b_{31}c_{11}+a_{22}b_{21}c_{12}+a_{23}b_{31}c_{12})\\&\quad+x^3P(a,b,c,x).\end{aligned}$$

This construction performs an approximate evaluation of the partial matrix product

$$\text{trace}\begin{pmatrix}a_{11} & a_{12} & a_{13}\\ 0 & a_{22} & a_{23}\end{pmatrix}\begin{pmatrix}b_{11} & b_{12}\\ b_{21} & 0\\ b_{31} & 0\end{pmatrix}\begin{pmatrix}c_{11} & c_{12}\\ c_{21} & c_{22}\end{pmatrix}$$

with five multiplications. Notice that five is also the trivial lower bound for this matrix product, obtained by counting independent elements of the matrix A. It is this equality

* Received by the editors July 17, 1980, and in final form October 16, 1981.

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

of lower and upper bounds which will allow the tight agreement between lower and upper bounds in the theorem, namely $N^2 \leqq \text{Rank}\langle N, N, N^\alpha\rangle \leqq CN^2 \log^2 N$.

The matrix product $D = AB$ can be thought of as the sum, as j goes through 1 to 3, of the outer products $a_{*,j}b_{j,*}$; then the indicated trace is the sum, over i and k, of $d_{i,k}c_{k,i}$. Among these three indicated outer products, we have two of dimension (2, 1) and one of dimension (1, 2).

Now iterate (tensorize) this construction M times. The formation of the product of the new, large matrices A and B, now involves 3^M outer products, of which (for each m between 0 and M) we have $\binom{M}{m}2^m$ outer products of dimension $(2^m, 2^{M-m})$.

Fix values of M and m. Then, mimicking Schönhage's proof of partial matrix multiplication [3], we may create a constant matrix A' of dimension $(2^m, 2^M)$ and a matrix B' of dimension $(2^M, 2^{M-m})$, such that each maximal minor of A' or B' has nonvanishing determinant. (Here we require that our underlying field is large enough; the rationals will do.) Suppose we want to multiply matrices A'' of dimension $(2^m, \binom{M}{m}2^m)$ and B'' of dimension $(\binom{M}{m}2^m, 2^{M-m})$. We start with a 1-1 mapping of the columns of A'' onto those columns of A'' with exactly 2^m nonzero entries. For each such column of A, compute the inverse of the $(2^m, 2^m)$ minor of A' whose rows correspond to the nonzero entries in this column of A. Multiply this inverse by the appropriate column of A''. Fill in the rest of A with zeros. Then we have $A'' = A'A$. Similarly create B such that $BB' = B''$. These matrices are created without essential multiplications, since A' and B' are constant matrices of scalars, and scalar multiplications don't count in the rank of a matrix multiplication problem. Then finally, $A''B'' = (A'A)(BB') = A'(AB)B'$. The multiplication AB can be done in 5^M multiplications in the ring of polynomials in x, and the left multiplication by A' and the right multiplication by B' are again scalar multiplications which don't enter into the calculation of rank.

Thus we have, so far, that

$$\text{Border Rank}\,(\langle 2^m, \tbinom{M}{m}2^m, 2^{M-m}\rangle) \leqq 5^M,$$

where Border Rank is the number of essential x-polynomial multiplications required to perform a given matrix multiplication.

The new wrinkle in our proof lies in our choice of m. Rather than choose m to maximize the "volume" of the left-hand side (the product of all three dimensions), which would enable us to minimize the exponent β for symmetric matrix multiplication (rank $\langle N, N, N\rangle = O(N^\beta)$), we instead select m to maximize the "area" of the projection onto the first two dimensions. Namely, we choose $m = (4M/5)$ as that value of m which maximizes the product of (2^m) and $(\binom{M}{m}2^m)$. This product is just $\binom{M}{m}4^m$, which is a term in the binomial expansion of $(4+1)^M$; thus it is maximized when $m/M = 4/(4+1)$, and for that value of m we have $\binom{M}{m}4^m = 5^M M^{-1/2}K$ for some constant K, by Stirling's formula. (Naively, the largest term in the binomial expansion of $(4+1)^M$ must be at least $(4+1)^M/(M+1)$, and Stirling's formula just gives a tighter bound.)

This gives us that

$$\text{Border Rank}\left(\left\langle 2^{4M/5}, \binom{M}{4M/5}2^{4M/5}, 2^{M/5}\right\rangle\right) \leqq 5^M,$$

or

$$\text{Border Rank}\,(\langle 2^{4M/5}, K5^M 2^{-4M/5}M^{-1/2}, 2^{M/5}\rangle) \leqq 5^M.$$

Now do the same arguments, with first and second dimensions reversed, to get

$$\text{Border Rank}\,(\langle K5^M 2^{-4M/5} M^{-1/2}, 2^{4M/5}, 2^{M/5}\rangle) \leqq 5^M.$$

Multiply (tensorize) to get

$$\text{Border Rank}\,(\langle K5^M M^{-1/2}, K5^M M^{-1/2}, 2^{2M/5}\rangle) \leqq 5^{2M}.$$

Letting $N = K5^M M^{-1/2}$ we get

$$\text{Border Rank}\,(\langle N, N, N^{\alpha}\rangle) \leqq K' N^2 (\log N),$$

where $\alpha = 2 \log 2/5 \log 5 = 0.17227$, and K' is some constant.

As usual, each multiplication in the ring of polynomials in x can be done as a convolution, via Fourier transforms, and since the degree of the product polynomials does not exceed $8M$ (each basic construction entails polynomials of degree 4, and the degrees are additive in the $2M$ iterations, yielding a total overall degree of $8M$), each such polynomial multiplication involves only $8M+1$ essential multiplications. Combining these results, we have that

$$\text{Rank}\,(\langle N, N, N^{\alpha}\rangle) \leqq K'' N^2 (\log N)^2,$$

as desired.

Remarks. If we were more careful, we would probably get a bound of $K'' N^2 (\log N)^{3/2}$.

Schönhage's construction involves two parameters k and n, each of which must be an integer greater than 1. The present theorem goes through exactly for each choice of k and n, and the value of α so obtained is

$$\alpha = \frac{2 \log ((k-1)(n-1)+1)}{(kn+1) \log (kn+1)}.$$

The present theorem selects $k = n = 2$ to maximize α at $2 \log 2/5 \log 5$.

The technique of partial matrix multiplication is due to Schönhage [3], and is valid in more generality than presented here. Here we are specializing his results (particularly by choice of m) to make possible the agreement between upper and lower bounds in our theorem.

Proof version 2 (*via exponential direct sum theorem*). Begin with Schönhage's construction which performs two completely disjoint matrix multiplications, of sizes $\langle k, n, 1\rangle$ and $\langle 1, 1, (k-1)(n-1)\rangle$, in $kn+1$ multiplications over the ring of polynomials in x. The construction is similar to that given above for partial matrix multiplication, and will not be repeated. His specialization to $k = n = 4$ gives the exponent $2.54\cdots$ for symmetric matrix multiplication.

Note again that $kn+1$ is the best possible result for this case, since the number of independent variables in the first two dimensions is kn for the first matrix and 1 for the second, thus $kn+1$ in all. Again this is the fact which will allow the close agreement between upper and lower bounds in the theorem.

Represent this construction as

$$(*) \qquad \text{Border Rank}\,(\langle k, n, 1\rangle + \langle 1, 1, (k-1)(n-1)\rangle) = kn+1.$$

Again we fix values of k and n which will maximize the eventual value of α', namely $k = n = 3$, and go through the proof for these fixed values, bearing in mind that the proof works for the general values as well.

Then $(*)$ becomes

$$\text{Border Rank } (\langle 3,3,1\rangle + \langle 1,1,4\rangle) = 10,$$

or in other words,

$$(\langle 3,3,1\rangle + \langle 1,1,4\rangle) \leftarrow 10\langle 1,1,1\rangle,$$

which can be tensorized by $\langle a, b, c\rangle$ to obtain

$$(**) \qquad (\langle 3a, 3b, c\rangle + \langle a, b, 4c\rangle) \leftarrow 10\langle a, b, c\rangle.$$

Here $\leftarrow$ means "homomorphic image by approximating algorithm", and summation of several matrices implies direct sum (the matrices are completely disjoint).

Suppose we are allowed to do M arbitrary multiplications in the ring of polynomials in x. That is, we have at our disposal $M\langle 1,1,1\rangle$. We wish to apply $(**)$ as often as possible; thus we divide M into groups of 10, with possibly some left over, and from $M\langle 1,1,1\rangle$ we get at least $(M/10-1)\langle 3,3,1\rangle + (M/10-1)\langle 1,1,4\rangle$. That is, there are at least $(M/10-1)$ disjoint groups of 10 among the M multiplications we are allowed, and each group will yield a $\langle 3,3,1\rangle$ and a $\langle 1,1,4\rangle$, all disjoint. Apply $(**)$ to the $(M/10-1)\langle 3,3,1\rangle$ to get at least $((M/10-1)/10-1)\langle 9,9,1\rangle +$ $((M/10-1)10-1)\langle 3,3,4\rangle$. Similarly applying $(**)$ to $(M/10-1)\langle 1,1,4\rangle$ we get $((M/10-1)/10-1)\langle 3,3,4\rangle + ((M/10-1)/10-1)\langle 1,1,16\rangle$. Combining, and doing the implied divisions, we get at least $(M/100-1.1)\langle 9,9,1\rangle + (2M/100-2.2)\langle 3,3,4\rangle +$ $(M/100-1.1)\langle 1,1,16\rangle$. Continue in like fashion. After k iterations we have at least

$$\sum_j \left(\binom{k}{j}\frac{M}{10^k} - 2.5\right)\langle 3^j, 3^j, 4^{k-j}\rangle,$$

as can be proved by induction.

Again nothing is new; this is just a proof of the exponential direct sum theorem [3]. But now we choose our j to maximize, again, the "area" of the resulting expression, i.e., its projection to the first two dimensions, rather than its volume. Indeed, choosing j to maximize the product of the binomial coefficient with the first two dimensions, that is, roughly,

$$\left(\binom{k}{j}\frac{M}{10^k}\right)(9^j),$$

we get the value of $j = 9k/10$. Fixing k we may choose M so that the factor $(\binom{k}{j}M/10^k)$ is just greater than 2.5; this will insure us that we will have at least one piece of size $\langle 3^j, 3^j, 4^{k-j}\rangle$. Thus we choose:

$$j = \frac{9k}{10}, \qquad M = 1 + \frac{2.5(10^k)}{\binom{k}{j}}.$$

With these choices, we get that

$$\text{Border Rank } (\langle 3^j, 3^j, 4^{k-j}\rangle) \leqq M,$$

or, in terms of k,

$$\text{Border Rank } (\langle 3^{9k/10}, 3^{9k/10}, 4^{k/10}\rangle) \leqq \frac{2.5(10^k)}{\binom{k}{9k/10}}.$$

Again we use Stirling's approximation to get that

$$\binom{k}{9k/10} \geqq C' 10^k 9^{-9k/10} k^{-1/2}.$$

Letting $N = 3^{9k/10}$ and substituting, we get

$$\text{Border Rank } (\langle N, N, N^{\alpha'} \rangle) < C'' N^2 (\log N)^{1/2}.$$

Here α' is $(2 \log 4)/(9 \log 9) = .1402$, or in general, $(2 \log ((k-1)(n-1)))/(kn \log (kn))$.

Again, we may replace Border Rank by Rank (i.e., eliminate the x-polynomials) at the price of a factor of N^ε. (The error bound is not as good as before, since we have no nice bound on the degrees of the x-polynomials.) Thus we get

$$\text{Rank } (\langle N, N, N^{\alpha'} \rangle) = O(N^{2+\varepsilon}).$$

Conclusion. We present, by two different routes, means by which matrix multiplication problems of size $\langle N, N, N^\alpha \rangle$ can be done with $N^{2+\varepsilon}$ operations, for numbers α strictly bounded away from 0. We do not see directly how this can be used to accelerate the symmetric matrix multiplication problem, but we present the result as interesting in its own right, and also with the hope that progress may be made in this new and different direction towards the solution of the symmetric matrix multiplication problem.

Perhaps one can find a way of arguing that if this theorem holds for some α between 0 and 1, then it must hold for a larger α, and that the sequence of α's so obtained would converge to 1. But I see no path which such a construction would take.

Another result which would be of interest, and which we cannot seem to obtain, would be the existence of a number $\alpha > 1$ such that Rank $(\langle N, N, N^\alpha \rangle) = O(N^{1+\alpha+\varepsilon})$. We can almost get this result, namely, by similar techniques, for each $\beta > 0$ there is an $\alpha > 1$ such that

$$\text{Rank } (\langle N, N, N^\alpha \rangle) = O(N^{\alpha+1+\beta} (\log N)^{3/2}).$$

A literature search shows that in 1976 Brockett and Dobkin obtained the related result

$$\text{Rank } (\langle N, N, \log N \rangle) = N^2 + o(N^2).$$

The present result is incomparable, in the sense that we do a larger problem and require a larger number of multiplications.

Note. More recent results by Coppersmith and Winograd [4] (this issue, pp. 472–492), combined with these techniques, all yield a better estimate of α: Rank $(\langle N, N, N^\alpha \rangle) = O(N^{2+\varepsilon})$ for $\alpha = 0.197$.

Acknowledgment. The present paper was inspired by a conversation with Victor Pan.

REFERENCES

[1] R. W. Brockett and D. Dobkin, *On the number of multiplications required for matrix multiplication*, this Journal, 5 (1976), pp. 624–628.

[2] V. Ya. Pan, *New combinations of methods for the acceleration of matrix multiplication*, Comput. Math. with Appl., 7 (1981), pp. 73–125.

[3] A. Schönhage, *Partial and total matrix multiplication*, this Journal, 10 (1981), pp. 434–455.

[4] D. Coppersmith and S. Winograd, *On the asymptotic complexity of matrix multiplication*, this Journal, this issue, pp. 472–492.

SIAM J. COMPUT.
Vol. 11, No. 3, August 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1103-0007 $01.00/0

ON THE ASYMPTOTIC COMPLEXITY OF MATRIX MULTIPLICATION*

D. COPPERSMITH† AND S. WINOGRAD†

Abstract. The main results of this paper have the following flavor: Given one algorithm for multiplying matrices, there exists another, better, algorithm. A consequence of these results is that ω, the exponent for matrix multiplication, is a limit point, that is, it cannot be realized by any single algorithm. We also use these results to construct a new algorithm which shows that $\omega < 2.495548$.

Key words. matrix multiplication, complexity, tensor product construction

1. Introduction. In this paper we investigate the number of arithmetic operations needed to compute the product of two "general" matrices, A and B. As is customary, we assume that the entries a_{ij} and b_{jk} of the matrices are indeterminates, and we study the multiplicative complexity of the system of bilinear forms $c_{ik} = \sum_j a_{ij} b_{jk}$.

We will restrict our attention to bilinear noncommutative algorithms, that is, to algorithms in which each multiplication is of the form $(\sum_{ij} \alpha_{ij} a_{ij})(\sum_{jk} \beta_{jk} b_{jk})$ where α_{ij}, β_{jk} are elements of some field F, the field of constants. (The a_{ij}s and b_{jk}s are indeterminates over F.) The advantage of thus restricting the class of algorithms is that we can use the "tensor product" construction. This means that if there exists an algorithm for multiplying two $n \times n$ matrices using K multiplications, then there exists an algorithm for multiplying two $n^2 \times n^2$ matrices using K^2 multiplications. More generally, if there exist two algorithms $\mathfrak{a}$ and $\mathfrak{a}'$, where $\mathfrak{a}$ computes $A_{m\times n} \times B_{n\times p}$ using K multiplications, and $\mathfrak{a}'$ computes $A_{m'\times n'} \times B_{n'\times p'}$ using K' multiplications, then there exists an algorithm $\mathfrak{a}'' = \mathfrak{a} \otimes \mathfrak{a}'$ which computes $A_{mm'\times nn'} \times B_{nn'\times pp'}$ using KK' multiplications. Another advantage of bilinear algorithms is that if there exists an algorithm for computing $A_{m\times n} \times B_{n\times p}$ using K multiplications then there exist five other algorithms, each using K multiplications, which compute $A_{n\times p} \times B_{p\times m}$, $A_{p\times m} \times B_{m\times n}$, $A_{p\times n} \times B_{n\times m}$, $A_{n\times m} \times B_{m\times p}$, $A_{m\times p} \times B_{p\times n}$, respectively [1] (see also [11]). On the other hand, it is known that if there exists an algorithm (not necessarily bilinear) which computes $A_{m\times n} \times B_{n\times p}$ using K multiplications/divisions, then there exists a bilinear algorithm for the product $A_{m\times n} \times B_{n\times p}$ using no more than $2K$ multiplications.

Let $M(n) = n^{\omega_n(F)}$ be the minimum number of multiplications needed to compute $A_{n\times n} \times B_{n\times n}$ (by a bilinear algorithm). We define

$$\omega(F) = \inf \{\omega_n(F) | n \geqq 2\}.$$

The notation serves as a reminder that $\omega_n(F)$ may depend on the field of scalars F. We call $\omega(F)$ the *exponent of matrix multiplication.*

It is well known [2] that $\omega(F)$ is also the exponent when we consider the total number of arithmetic operations. More precisely, let $M'(n) = n^{\omega'_n(F)}$ be the minimum number of arithmetic operations needed to multiply two $n \times n$ matrices. Let $\omega'(F)$ be defined as $\omega'(F) = \inf \{\omega'_n(F) | n \geqq 2\}$, then $\omega'(F) = \omega(F)$.

The properties of bilinear algorithms imply [3] that if there exists an algorithm for computing $A_{m\times n} \times B_{n\times p}$ with K multiplications, then, for every r, there exists an algorithm for computing the product of two $r \times r$ matrices with fewer than Cr^{ω} multiplications, where C is a constant independent of r, and $\omega = 3 \log K / \log (mnp)$.

* Received by the editors April 21, 1981, and in final revised form August 20, 1981.

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

In the rest of this section we will recapitulate some of the previous results which have a bearing on the material of this paper.

In 1969, Strassen [3] showed that it is possible to multiply two 2×2 matrices using 7 multiplications, and therefore that $\omega(F)\leqq \log 7/\log 2\approx 2.807$ for every F. In 1978, Pan [4] showed that it is possible to multiply two $n\times n$ matrices using $\frac{1}{3}n^3+\frac{9}{2}n^2-\frac{1}{3}n$ multiplications, and in particular, that it is possible to multiply two 48×48 matrices using 47216 multiplications. Therefore $\omega(F)\leqq \log 47216/\log 48\approx 2.780$ for every field F. In 1979, Bini et al. [5] considered the computation of the product of matrices over the field $F(\lambda)$, where λ is a new indeterminate. The sense in which they defined the computation will be described in the next section. They showed that over $F(\lambda)$ it is possible to multiply two 12×12 matrices using 1000 multiplications and therefore $\omega(F(\lambda))\leqq \log 1000/\log 12\approx 2.779$. Bini [6], showed that $\omega(F(\lambda))=\omega(F)$ and therefore $\omega(F)\leqq \log 1000/\log 12$ for every field F.

Schönhage [7] showed that the direct sum conjecture [8] does not hold when we consider computations over $F(\lambda)$. He showed that it is possible to compute (over $F(\lambda)$) both $A_{m\times 1}\times B_{1\times n}$ and $A'_{1\times k}\times B'_{k\times 1}$ using $mn+1$ multiplications, where $k=(m-1)(n-1)$. Because this construction of Schönhage is the springboard of the work reported in this paper, we will describe it in more detail in the next section. With each algorithm for computing $\bigoplus_i \boldsymbol{A}_{m_i\times n_i}\times \boldsymbol{B}_{n_i\times p_i}$ using K multiplications, Schönhage defined the associated equation $\sum_i (m_in_ip_i)^\tau=K$. (We use the direct sum notation to emphasize that all the indeterminates, which are the entries of the $\boldsymbol{A}_{m_i\times n_i}$s and $\boldsymbol{B}_{n_i\times p_i}$s, are distinct.) Schönhage then proved that if τ is the zero of the associated equation, then $\omega(F)\leqq 3\tau$. In particular, if we take $m=n=4$ in Schönhage's construction, we obtain that $\omega(F)\leqq 3\tau<2.548$, where τ is the zero of the equation $16^\tau+9^\tau=17$.

Pan [9] gave another construction of algorithms for the direct sum of products of matrices. In particular, Pan's construction yields an algorithm for computing $\boldsymbol{A}_{5\times 1}\times \boldsymbol{B}_{1\times 22}\oplus \boldsymbol{A}_{2\times 11}\times \boldsymbol{B}_{11\times 5}\oplus \boldsymbol{A}_{11\times 10}\times \boldsymbol{B}_{10\times 1}$ using 156 multiplications. As a consequence of this construction we have $\omega(F)\leqq 3\log 52/\log 110\approx 2.522$.

In the next section we will describe what is meant by computing over $F(\lambda)$, and illustrate the power of this concept by describing Schönhage's construction. In § 3, we will generalize the construction of Schönhage. A consequence of this generalization is that $\omega(F)$ is a limit point, i.e., $\omega(F)<\omega(n)$ for all n. In § 4, we will modify the proof of § 3 to show that if we have an algorithm $\mathfrak{a}$ for computing the direct sum of products of matrices having a special property (to be defined in § 4), then there exists an algorithm $\mathfrak{a}'$ for computing the direct sum of products of matrices having the same property. The importance of $\mathfrak{a}'$ is that the zero of the equation associated with $\mathfrak{a}'$ is smaller than the zero of the equation associated with $\mathfrak{a}$. Thus $\mathfrak{a}'$ yields a better estimate of $\omega(F)$. We will iterate this result to show that $\omega(F)\leqq 2.498$. In § 5 we will generalize the tensor product construction. This generalization will enable us to better utilize the result of § 4, and we will show that $\omega(F)\leqq 2.4956$.

The main result of § 6 is that $\omega(F)$ is a limit point in a strong sense. More precisely, we show there that no algorithm for λ-computing the direct sum of matrix multiplications can yield the exact value of $\omega(F)$.

2. $\boldsymbol{\lambda}$-computations. In this section, we will describe the concept of λ-computations which was introduced by Bini et al. [5], [6]. Our use of the term λ-computation is identical with the concept of approximate computations of [5], [6], [7], [9]. We adopt the terminology of λ-computations to emphasize that λ is an indeterminate, and that the computation is exact. That is, the λ-computation is a tool for obtaining the exact product of two large matrices, not an approximation.

We will start this section with the definition of an algorithm (bilinear) for a system of bilinear forms. This will serve the dual purpose of reviewing known results, as well as establishing the notation we will use in the rest of the paper. Next, we will define the concept of λ-computations, and relate it to the ordinary concept of computations. We will end the section with a description of a class of algorithms which was discovered by Schönhage [7]. These algorithms illustrate the power of λ-computations; they also serve to introduce the constructions which we use in the rest of the paper.

Let $\{x_1, x_2, \cdots, x_r, y_1, y_2, \cdots, y_s\}$ be a set of $r+s$ distinct indeterminates, and let G be a field. Let B_k be the bilinear form $B_k = \sum_{j=1}^{s} \sum_{i=1}^{r} b_{ijk} x_i y_j$, $k = 1, 2, \cdots, t$. We will also use B_k to denote the $r \times s$ matrix whose (i, j) entry is b_{ijk}. If we denote the (column) vector $(x_1, x_2, \cdots, x_r)^T$ by $\mathbf{x}$, and the (column) vector $(y_1, y_2, \cdots, y_s)^T$ by $\mathbf{y}$, then the bilinear form B_k can be written as $\mathbf{x}^T B_k \mathbf{y}$. We will use $\mathcal{B}$ to denote the system of bilinear forms $\mathcal{B} = (B_1, B_2, \cdots, B_t)$. To continue our double use of the same symbol to denote both the bilinear form and the matrix which defines it, we use $\mathcal{B}$ to denote the 3-tensor whose (i, j, k) entry is b_{ijk}.

DEFINITION 2.1. A (bilinear) algorithm $\mathfrak{a}$ over G is two sequences $M_1(\mathbf{x})$, $M_2(\mathbf{x}), \cdots, M_L(\mathbf{x})$ and $N_1(\mathbf{y}), N_2(\mathbf{y}), \cdots, N_L(\mathbf{y})$, where $M_l(\mathbf{x}) = \sum_{i=1}^{r} \alpha_{li} x_i$ is a G-linear form of $\{x_1, x_2, \cdots, x_r\}$ and $N_l(\mathbf{y}) = \sum_{j=1}^{s} \beta_{lj} y_j$ is a G-linear form of $\{y_1, y_2, \cdots, y_s\}$, $l = 1, 2, \cdots, L$. We denote L by $\mu(\mathfrak{a})$.

DEFINITION 2.2. An algorithm $\mathfrak{a}$ is said to (be able to) compute the system of bilinear forms $\mathcal{B} = (B_1, B_2, \cdots, B_t)$ if for each $k = 1, 2, \cdots, t$ there exist $L = \mu(\mathfrak{a})$ elements $\gamma_{lk} \in G$ such that $B_k = \sum_{l=1}^{L} \gamma_{lk} M_l(\mathbf{x}) N_l(\mathbf{y})$.

At times it will be convenient to use a matricial form of Definition 2.2. Let α denote the $L \times r$ matrix (α_{li}), and let β denote the $L \times s$ matrix (β_{lj}). The algorithm $\mathfrak{a}$ can compute the system of bilinear forms $\mathcal{B} = (B_1, B_2, \cdots, B_t)$ if for every $k = 1, 2, \cdots, t$ there exists an $L \times L$ diagonal G-matrix C_k such that

$$B_k = \mathbf{x}^T \alpha^T C_k \beta \mathbf{y}, \qquad k = 1, 2, \cdots, t,$$

or in terms of the matrices B_k

$$B_k = \alpha^T C_k \beta, \qquad k = 1, 2, \cdots, t.$$

We can continue Definitions 2.1 and 2.2 and conclude: The system $\mathcal{B} = (B_1, B_2, \cdots, B_t)$ of bilinear forms can be computed by an algorithm $\mathfrak{a}$ with $\mu(\mathfrak{a}) = L$ if there exist elements of G, α_{li}, β_{lj}, γ_{lk} $(1 \leq i \leq r, 1 \leq j \leq s, 1 \leq k \leq t, 1 \leq l \leq L)$ such that

$$B_k = \sum_{l=1}^{L} \gamma_{lk} \left(\sum_{i=1}^{r} \alpha_{li} x_i \right) \left(\sum_{j=1}^{s} \beta_{lj} y_j \right), \qquad k = 1, 2, \cdots, t. \tag{2.1}$$

Let $\boldsymbol{\alpha}_l$ denote the vector $(\alpha_{l1}, \alpha_{l2}, \cdots, \alpha_{lr})$, $\boldsymbol{\beta}_l$ denote the vector $(\beta_{l1}, \beta_{l2}, \cdots, \beta_{ls})$, and $\boldsymbol{\gamma}_l$ the vector $(\gamma_{l1}, \gamma_{l2}, \cdots, \gamma_{lt})$. As was pointed out by Strassen [8], equation (2.1) is equivalent to the requirement on the tensor $\mathcal{B}$:

$$\mathcal{B} = \sum_{l=1}^{L} \alpha_l \otimes \boldsymbol{\beta}_l \otimes \boldsymbol{\gamma}_l. \tag{2.2}$$

Equation (2.2) is called a decomposition of $\mathcal{B}$ into L triads. The smallest integer L such that there exists a decomposition of $\mathcal{B}$ into L triads is called the rank of $\mathcal{B}$ and is denoted by Rk $(\mathcal{B})$.

Let $\mathcal{B} = (\sum_{j=1}^{s} \sum_{i=1}^{r} b_{ijk} x_i y_j, \ k = 1, 2, \cdots, t)$ and $\bar{\mathcal{B}} = (\sum_{j=1}^{\bar{s}} \sum_{i=1}^{\bar{r}} \bar{b}_{ijk} \bar{x}_i \bar{y}_j, \ k = 1, 2, \cdots, \bar{t})$ be two systems of bilinear forms. (The x_is, y_js, $\bar{x}_i$s and $\bar{y}_j$s are all distinct.) The system $\mathcal{B}' = \mathcal{B} \cup \bar{\mathcal{B}}$ is called the direct sum of $\mathcal{B}$ and $\bar{\mathcal{B}}$, and the tensor $\mathcal{B}'$ is denoted by $\mathcal{B}' = \mathcal{B} \oplus \bar{\mathcal{B}}$.

Let $\mathscr{B}$ and $\bar{\mathscr{B}}$ be as before. The $r\bar{r}\times s\bar{s}\times t\bar{t}$ tensor $\mathscr{B}'$ whose $((i,\bar{\imath}),(j,\bar{\jmath}),(k,\bar{k}))$ entry is $b_{ijk}\bar{b}_{\bar{\imath}\bar{\jmath}\bar{k}}$ is called the tensor product of $\mathscr{B}$ and $\bar{\mathscr{B}}$ and is denoted by $\mathscr{B}'=\mathscr{B}\otimes\bar{\mathscr{B}}$.

We will now list, without proof, some well-known facts.

Fact 2.1. Let $\mathfrak{a}$ be the algorithm $M_1(\mathbf{x}),\cdots,M_L(\mathbf{x});N_1(\mathbf{y}),\cdots,N_L(\mathbf{y})$, and let $\mathfrak{a}'$ be the algorithm $a_1M_1(\mathbf{x}),a_2M_2(\mathbf{x}),\cdots,a_LM_L(\mathbf{x});b_1N_1(\mathbf{y}),\cdots,b_LN_L(\mathbf{y})$ for some $2L$ elements $a_l, b_l\in G$, none of which is 0. If $\mathfrak{a}$ can compute the system $\mathscr{B}$ of bilinear forms, then so can $\mathfrak{a}'$.

Fact 2.2. Let $\mathscr{B}=(b_{ijk})$ be an $r\times s\times t$ tensor, and let $U=(U_{ii'})$ be an $r\times r$ nonsingular matrix, $V=(V_{jj'})$ be an $s\times s$ nonsingular matrix, $W=(W_{kk'})$ be a $t\times t$ nonsingular matrix. Let $\mathscr{B}'=(b'_{i'j'k'})$ be the $r\times s\times t$ tensor defined by $b'_{i'j'k'}=\sum_{k=1}^{t}\sum_{j=1}^{s}\sum_{i=1}^{r}U_{ii'}V_{jj'}W_{kk'}b_{ijk}$. Then $\mathrm{Rk}(\mathscr{B}')=\mathrm{Rk}(\mathscr{B})$. Moreover, if $\mathscr{B}=\sum_{l=1}^{L}\boldsymbol{\alpha}_l\otimes\boldsymbol{\beta}_l\otimes\boldsymbol{\gamma}_l$ is a decomposition of $\mathscr{B}$, then $\mathscr{B}'=\sum_{l=1}^{L}(U\boldsymbol{\alpha}_l)\otimes(V\boldsymbol{\beta}_l)\otimes(W\boldsymbol{\gamma}_l)$ is a decomposition of $\mathscr{B}'$.

Fact 2.3. Let $\mathscr{B}=\mathscr{B}'+\mathscr{B}''$ and $\bar{\mathscr{B}}$ be four tensors, then $\bar{\mathscr{B}}\otimes(\mathscr{B}'+\mathscr{B}'')=\bar{\mathscr{B}}\otimes\mathscr{B}'+\bar{\mathscr{B}}\otimes\mathscr{B}''$, and $(\mathscr{B}'+\mathscr{B}'')\otimes\bar{\mathscr{B}}=\mathscr{B}'\otimes\bar{\mathscr{B}}+\mathscr{B}''\otimes\bar{\mathscr{B}}$.

Fact 2.4. Let $\mathscr{B}=\mathscr{B}'\oplus\mathscr{B}''$ and $\bar{\mathscr{B}}$ be four tensors, then $\bar{\mathscr{B}}\otimes(\mathscr{B}'\oplus\mathscr{B}'')=\bar{\mathscr{B}}\otimes\mathscr{B}'\oplus\bar{\mathscr{B}}\otimes\mathscr{B}''$, and $(\mathscr{B}'\oplus\mathscr{B}'')\otimes\bar{\mathscr{B}}=\mathscr{B}'\otimes\bar{\mathscr{B}}\oplus\bar{\mathscr{B}}''\otimes\bar{\mathscr{B}}$.

Fact 2.5. Following [7] we use $\langle m,n,p\rangle$ to denote the system of bilinear forms of the product of an $m\times n$ matrix by an $n\times p$ matrix, that is, of the system $(B_{ik}=\sum_{j=1}^{n}x_{ij}y_{jk}, 1\leqq i\leqq m, i\leqq k\leqq p)$. Then $\langle m,n,p\rangle\otimes\langle m',n',p'\rangle=\langle mm',nn',pp'\rangle$.

Fact 2.6. Let $\mathscr{B}=(b_{ijk})$ and $\mathscr{B}'=(b_{\pi(i),\pi(j),\pi(k)})$, where $\pi:\{i,j,k\}\overset{1:1}{\to}\{i,j,k\}$ is some permutation of the symbols i, j, and k. Then $\mathrm{Rk}(\mathscr{B})=\mathrm{Rk}(\mathscr{B}')$. In particular, $\mathrm{Rk}(\langle m,n,p\rangle)=\mathrm{Rk}(\langle n,p,m\rangle)=\mathrm{Rk}(\langle p,m,n\rangle)=\mathrm{Rk}(\langle n,m,p\rangle)=\mathrm{Rk}(\langle m,p,n\rangle)=\mathrm{Rk}(\langle p,n,m\rangle)$.

Fact 2.7. Let $\mathscr{B}$ and $\mathscr{B}'$ be two tensors. Then $\mathrm{Rk}(\mathscr{B}\otimes\mathscr{B}')\leqq\mathrm{Rk}(\mathscr{B})\,\mathrm{Rk}(\mathscr{B}')$. Moreover, if $\mathscr{B}=\sum_{l=1}^{L}\boldsymbol{\alpha}_l\otimes\boldsymbol{\beta}_l\otimes\boldsymbol{\gamma}_l$ and $\mathscr{B}'=\sum_{l'=1}^{L'}\boldsymbol{\alpha}'_{l'}\otimes\boldsymbol{\beta}'_{l'}\otimes\boldsymbol{\gamma}'_{l'}$ are decompositions of $\mathscr{B}$ and $\mathscr{B}'$ respectively then $\mathscr{B}\otimes\mathscr{B}'=\sum_{l'=1}^{L'}\sum_{l=1}^{L}(\alpha_l\otimes\alpha'_{l'})\otimes(\beta_l\otimes\beta_{l'})\otimes(\gamma_l\otimes\gamma_{l'})$ is a decomposition of $\mathscr{B}\otimes\mathscr{B}'$.

We are now ready to consider λ-computations. Let F be a field, and let λ be an indeterminate over F. Every nonzero element $g\in F(\lambda)$ can be written uniquely as $g=\lambda^dP(\lambda)/(1-Q(\lambda))$, where $P(\lambda)$ and $Q(\lambda)$ are polynomials with coefficients in F, $P(0)\neq 0$, $Q(0)=0$, P and $1-Q$ are relatively prime, and d is an integer. *We call* $-d$ *the deficiency of* g, and denote it by $\mathrm{def}(g)$. We extend the definition of deficiency to vectors, matrices, and tensors. If $\boldsymbol{\alpha}=(\alpha_1,\alpha_2,\cdots,\alpha_n)$ is a vector in $F(\lambda)^n$ we define $\mathrm{def}(\boldsymbol{\alpha})=\max_{1\leqq i\leqq n}\{\mathrm{def}(\alpha_i)\}$. Similarly we define the deficiency of a matrix $M=(M_{ij})$ as $\mathrm{def}(M)=\max_{i,j}\{\mathrm{def}(M_{ij})\}$, and the deficiency of $\mathscr{B}=(b_{ijk})$ as $\mathrm{def}(\mathscr{B})=\max_{i,j,k}\{\mathrm{def}(b_{ijk})\}$. There are obvious relations between deficiencies; for example, $\mathrm{def}(g_1g_2)=\mathrm{def}(g_1)+\mathrm{def}(g_2)$. These relations are very similar to those which hold for degrees of polynomials, and we will not elaborate them.

An element $g=\lambda^dP(\lambda)/(1-Q(\lambda))\in F(\lambda)$ can be viewed as the Laurent power series $\lambda^dP(\lambda)\sum_{i=0}^{\infty}Q(\lambda)^i$. This presentation of g motivates the next definition.

DEFINITION 2.3. Let $g=\lambda^dP(\lambda)/(1-Q(\lambda))$ be an element of $F(\lambda)$. For every integer i we define the mapping cf_i: $F(\lambda)\to F$ by $cf_i(g)=0$ if $i<d$, and $cf_i(g)$ is the $(i-d)$th coefficient of the polynomial $P(\lambda)\sum_{j=0}^{i-d}Q(\lambda)^j$. We call $cf_i(g)$ the *i*th *coefficient of* g. We extend the definition of cf_i to vectors, matrices and tensors. If $\boldsymbol{\alpha}=(\alpha_1,\alpha_2,\cdots,\alpha_n)$ is in $F(\lambda)^n$ then $cf_i(\boldsymbol{\alpha})=(cf_i(\alpha_1),cf_i(\alpha_2),\cdots,cf_i(\alpha_n))\in F^n$. We define $cf_i(M)$ and $cf_i(\mathscr{B})$ in a similar way.

The concept of λ-computation enables us to use algorithms over $G=F(\lambda)$ to compute a system $\mathscr{B}$ of bilinear forms over F.

DEFINITION 2.4. Let $\mathscr{B}=(B_1, B_2, \cdots, B_t)$ be a system of F-bilinear forms, and let $\mathfrak{a}$ be an $F(\lambda)$-algorithm. We say that $\mathfrak{a}$ *can* λ*-compute* $\mathscr{B}$ if there exists a system $\bar{\mathscr{B}}=(\bar{B}_1, \bar{B}_2, \cdots, \bar{B}_t)$ of $F(\lambda)$-bilinear forms such that:

(a) $\mathfrak{a}$ can compute $\bar{\mathscr{B}}$.

(b) $\operatorname{def}(\bar{\mathscr{B}}) \leqq 0$.

(c) $\mathscr{B}=cf_0(\bar{\mathscr{B}})$.

Remark 2.1. Conditions (b) and (c) of Definition 2.4 imply that $\operatorname{def}(\bar{B}_k)=0$ for $k=1, 2, \cdots, t$ whenever none of the B_ks is 0.

Most of the properties of computations are enjoyed by λ-computations. We will now state, and prove, the property analogous to Fact 2.7.

PROPOSITION 2.1. *If* $\mathfrak{a}$ *can* λ*-compute* $\mathscr{B}$ *and* $\mathfrak{a}'$ *can* λ*-compute* $\mathscr{B}'$, *then there exists an algorithm* $\bar{\mathfrak{a}}=\mathfrak{a}\otimes\mathfrak{a}'$, *which can* λ*-compute* $\mathscr{B}\otimes\mathscr{B}'$, *satisfying* $\mu(\bar{\mathfrak{a}})=\mu(\mathfrak{a})\mu(\mathfrak{a}')$.

Proof: Let $\mu(\mathfrak{a})=L$ and $\mu(\mathfrak{a}')=L'$. Let $\bar{\mathscr{B}}$ and $\bar{\mathscr{B}}'$ be the $F(\lambda)$-bilinear forms satisfying the conditions of Definition 2.4. By Fact 2.7 $\bar{\mathscr{B}}\otimes\bar{\mathscr{B}}'$ has a decomposition into LL' triads. Let $\bar{\mathfrak{a}}$ be the algorithm of this decomposition. But $\operatorname{def}(\bar{\mathscr{B}}\otimes\bar{\mathscr{B}}')=\operatorname{def}(\bar{\mathscr{B}})+\operatorname{def}(\bar{\mathscr{B}}')\leqq 0$, and therefore $cf_0(\bar{\mathscr{B}}\otimes\bar{\mathscr{B}}')=cf_0(\bar{\mathscr{B}})\otimes cf_0(\bar{\mathscr{B}}')=\mathscr{B}\otimes\mathscr{B}'$. Thus $\bar{\mathscr{B}}\otimes\bar{\mathscr{B}}'$ satisfy the conditions of Definition 2.4.

Our next task is to connect λ-computations with ordinary computations. More precisely, given an $F(\lambda)$-algorithm $\mathfrak{a}$ which can λ-compute the system $\mathscr{B}$ of F-bilinear forms, we will construct an F-algorithm $\mathfrak{a}'$ which can also compute $\mathscr{B}$.

Let $\mathfrak{a}$ be an $F(\lambda)$-algorithm which can λ-compute $\mathscr{B}$, and let $\bar{\mathscr{B}}$ be the system of $F(\lambda)$-bilinear forms of Definition 2.4. Then $\bar{B}_k=\sum_{l=1}^{L}\gamma_{l,k}M_l(\mathbf{x})N_l(\mathbf{y})$, $k=1, 2, \cdots, t$, where $L=\mu(\mathfrak{a})$. By Fact 2.1 we may assume that $M_l(\mathbf{x})$ and $N_l(\mathbf{y})$ are polynomials in λ $(l=1, 2, \cdots, L)$ with coefficients F-linear forms in the x_is and y_js respectively, such that λ does not divide $M_l(\mathbf{x})$ or $N_l(\mathbf{y})$. If $\mathfrak{a}$ satisfies this assumption we say that $\mathfrak{a}$ is in λ*-canonical* form. We define the deficiency of $\mathfrak{a}$ as $\operatorname{def}(\mathfrak{a}, \mathscr{B})=\max_{l,k}\{\operatorname{def}(\gamma_{l,k})\}$, and the degree of $\mathfrak{a}$ as $\deg(\mathfrak{a})=\max_l\{\deg(M_l(\mathbf{x})), \deg(N_l(\mathbf{y}))\}$, where $\deg(P(\lambda))$ denotes the degree of the polynomial $P(\lambda)$.

We now have, for each $k=1, 2, \cdots, t$:

$$
\begin{aligned}
B_k=cf_0(\bar{B}_k)&=\sum_{l=1}^{L} cf_0(\gamma_{l,k}M_l(\mathbf{x})N_l(\mathbf{y})) \\
&=\sum_{l=1}^{L}\sum_{j=0}^{\operatorname{def}(\gamma_{l,k})} cf_{-j}(\gamma_{l,k})cf_j(M_l(\mathbf{x})N_l(\mathbf{y})).
\end{aligned}
\tag{2.3}
$$

Recalling that $cf_{-j}(\gamma_{l,k})\in F$ we see that if $\mathfrak{a}'$ is an F-algorithm which can compute the system of F-bilinear forms $\{cf_j(M_l(\mathbf{x})N_l(\mathbf{y}))|1\leqq j\leqq \operatorname{def}(\mathfrak{a}), 1\leqq l\leqq L\}$, then $\mathfrak{a}'$ can compute $\mathscr{B}$. □

It is well known that the coefficients of the polynomial $T(\lambda)=(\sum_{i=0}^{n}x_i\lambda^i)(\sum_{j=0}^{n}y_j\lambda^j) \bmod \lambda^{n+1}$ can be computed by an algorithm $\mathscr{C}$ satisfying $\mu(\mathscr{C})=\mu_F(n)$ where $2n+1\leqq\mu_F(n)\leqq\frac{1}{2}(n+1)(n+2)$. Therefore there exists an F-algorithm $\mathfrak{a}'$ which can compute $\{cf_j(M_l(\mathbf{x})N_l(\mathbf{y}))|0\leqq j\leqq \operatorname{def}(\mathfrak{a}), 1\leqq l\leqq L\}$ satisfying $\mu(\mathfrak{a}')\leqq\mu_F(\operatorname{def}(\mathfrak{a}, \mathscr{B}))\mu(\mathfrak{a})$. We have thus proved:

PROPOSITION 2.2. *If* $\mathfrak{a}$ *is an* $F(\lambda)$*-algorithm which can* λ*-compute* $\mathscr{B}$, *and* $\mathfrak{a}$ *is in* λ*-canonical form, then there exists an F-algorithm* $\mathfrak{a}'$ *which can compute* $\mathscr{B}$ *satisfying* $\mu(\mathfrak{a}')\leqq\mu_F(\operatorname{def}(\mathfrak{a}, \mathscr{B}))\mu(\mathfrak{a})$.

Examination of (2.3) shows that we have also proved the following proposition:

PROPOSITION 2.3. *Let* $\mathfrak{a}$ *be the* $F(\lambda)$*-algorithm* $(M_1(\mathbf{x}), \cdots, M_L(\mathbf{x}); N_1(\mathbf{y}), \cdots, N_L(\mathbf{y}))$ *which is in* λ*-canonical form. If* $\mathfrak{a}$ *can* λ*-compute* $\mathscr{B}$, *then so can the* $F(\lambda)$*-algorithm* $\mathfrak{a}' = (M_1(\mathbf{x}) + M_1'(\mathbf{x}), \cdots, M_L(\mathbf{x}) + M_L'(\mathbf{x}); N_1(\mathbf{y}) + N_1'(\mathbf{y}), \cdots, N_L(\mathbf{y}) + N_L'(\mathbf{y}))$ *whenever* $\max_l \{\text{def } M_l'(\mathbf{x}), \text{ def } N_l'(\mathbf{y})\} < -\text{def}(\mathfrak{a}, \mathscr{B})$. *Moreover,* $\text{def}(\mathfrak{a}', \mathscr{B}) = \text{def}(\mathfrak{a}, \mathscr{B})$.

DEFINITION 2.5. Let $\mathscr{B}$ be a system of bilinear forms. We define $\text{Rk}_\lambda(\mathscr{B}) = \min \mu(\mathfrak{a})$, where the minimization is over all algorithms which can λ-compute $\mathscr{B}$. In [7] $\text{Rk}_\lambda(\mathscr{B})$ is called the *border rank* of $\mathscr{B}$.

Let $A(\mathbf{x}; \lambda)$ be an $m \times n$ matrix whose entries are linear forms in the indeterminates $\{x_1, x_2, \cdots, x_r\}$ with coefficients in $F(\lambda)$. We assume further that $\text{def}(A(\mathbf{x}; \lambda)) = 0$. Let $A_0(\mathbf{x})$ be the $m \times n$ matrix $A_0(\mathbf{x}) = cf_0(A(\mathbf{x}; \lambda))$. We will use $\rho_r(A(\mathbf{x}; \lambda))$, respectively $\rho_c(A(\mathbf{x}; \lambda))$, to denote the maximum number of rows, respectively columns, of $A(\mathbf{x}; \lambda)$ which are linearly independent over $F(\lambda)$. By a slight abuse of notation we use $\rho_r(A_0(\mathbf{x}))$ and $\rho_c(A_0(\mathbf{x}))$ to denote the maximum number of rows, respectively columns, of $A_0(\mathbf{x})$ which are linearly independent over F. It is easily verified that $\rho_r(A_0(\mathbf{x})) \leqq \rho_r(A(\mathbf{x}; \lambda))$ and that $\rho_c(A_0(\mathbf{x})) \leqq \rho_c(A(\mathbf{x}; \lambda))$. We thus obtain:

PROPOSITION 2.4. *Let* $\mathscr{B}$ *be the system of bilinear forms* $A(\mathbf{x})\mathbf{y}$, *where* $\mathbf{y}$ *is the column vector* $\mathbf{y} = (y_1, y_2, \cdots, y_s)^T$, *then* $\text{Rk}_\lambda(\mathscr{B}) \geqq \max(\rho_r(A(\mathbf{x})), \rho_c(A(\mathbf{x})))$.

The usefulness of λ-computations for studying the complexity of matrix multiplication comes from the fact that the deficiencies add under tensor product. This is, $\text{def}(\mathfrak{a} \otimes \mathfrak{a}', \mathscr{B} \otimes \mathscr{B}') = \text{def}(\mathfrak{a}, \mathscr{B}) + \text{def}(\mathfrak{a}', \mathscr{B}')$. Thus if an algorithm $\mathfrak{a}$ can λ-compute $\langle n, n, n\rangle$, with deficiency d, there exists an $F(\lambda)$-algorithm $\mathfrak{a}'$ for computing $\langle n^s, n^s, n^s\rangle$ with deficiency sd satisfying $\mu(\mathfrak{a}') = \mu(\mathfrak{a})^s$. Therefore there exists an F-algorithm $\bar{\mathfrak{a}}$ which can compute $\langle n^s, n^s, n^s\rangle$ satisfying $\mu(\bar{\mathfrak{a}}) \leqq \mu_F(sd)\mu(\mathfrak{a})^s \leqq (ds+1)^2 \mu(\mathfrak{a})^s$. Therefore, using the terminology of the introduction we have $\omega(F) \leqq \log \mu(\mathfrak{a})/\log n$. The same reasoning shows that if $\mathfrak{a}$ can λ-compute $\langle m, n, p\rangle$ then $\omega(F) \leqq 3 \log \mu(\mathfrak{a})/\log mnp$.

This last relation between ω and $\mu(\mathfrak{a})$ was generalized by Schönhage [7]. We will now describe Schönhage's algorithms and state his generalization.

CONSTRUCTION 2.1. Let $m > 1$ and $n > 1$ be two integers. Consider the following $F(\lambda)$-algorithm $\mathfrak{a}(m, n)$, given by

(a) $M_{ij}(\mathbf{x}, \boldsymbol{\xi}) = (x_i + \lambda \xi_{ij}), \quad N_{ij}(\mathbf{y}, \boldsymbol{\eta}) = (y_j + \lambda \eta_{ij}), \quad 1 \leqq i \leqq m, \quad 1 \leqq j < n.$

(b) $M_{mj}(\mathbf{x}, \boldsymbol{\xi}) = \left(x_m - \sum_{i=1}^{m-1} \lambda \xi_{ij}\right), \quad N_{mj}(\mathbf{y}, \boldsymbol{\eta}) = y_j, \quad 1 \leqq j < n.$

(c) $M_{in}(\mathbf{x}, \boldsymbol{\xi}) = x_i, \quad N_{in}(\mathbf{y}, \boldsymbol{\eta}) = \left(y_n - \sum_{j=1}^{n-1} \lambda \eta_{ij}\right), \quad 1 \leqq i < m.$

(d) $M_{mn}(\mathbf{x}, \boldsymbol{\xi}) = x_m, \quad N_{mm}(\mathbf{y}, \boldsymbol{\eta}) = y_n.$

(e) $M_{0,0}(\mathbf{x}, \boldsymbol{\xi}) = \sum_{i=1}^{m} x_i, \quad N_{0,0}(\mathbf{y}, \boldsymbol{\eta}) = \sum_{j=1}^{n} y_j.$

We see immediately that $\mathfrak{a}(m, n)$ can λ-compute $x_i y_j$, $1 \leqq i \leqq m$, $1 \leqq j \leqq n$, and also that $\lambda^{-2}(\sum_{j=1}^{n} \sum_{i=1}^{m} M_{ij}(\mathbf{x}, \boldsymbol{\xi}) N_{ij}(\mathbf{y}, \boldsymbol{\eta}) - M_{0,0} N_{0,0}) = \sum_{j=1}^{n-1} \sum_{i=1}^{m-1} \xi_{ij} \eta_{ij}$. That is, $\mathfrak{a}(m, n)$ can λ-compute $\langle m, 1, n\rangle \oplus \langle 1, (m-1)(n-1), 1\rangle$, while $\mu(\mathfrak{a}(m, n)) = mn + 1$.

The usefulness of this construction for studying the value of ω rests on the following theorem, due to Schönhage [7].

THEOREM 2.1. *Let* $\mathfrak{a}$ *be an* $F(\lambda)$*-algorithm which can* λ*-compute* $\bigoplus_{i=1}^{N} \langle m_i, n_i, p_i \rangle$, *so that for some i we have* $m_i + n_i + p_i > 3$, *and let* τ *be the root of the associated equation*

$$\sum_{i=1}^{N} (m_i n_i p_i)^\tau = \mu(\mathfrak{a}).$$

Then $\omega \leqq 3\tau$.

We apply Theorem 2.1 to $\mathfrak{a}(4, 4)$ of Construction 2.1. We obtain that $\omega(F) \leqq 3\tau \approx 2.548$, where τ is the root of the associated equation $16^\tau + 9^\tau = 17$.

COROLLARY 2.1. *Let* $\bigoplus_{i=1}^{N} \langle m_i, n_i, p_i \rangle$ *be as in Theorem* 2.1. *Then* $\omega \leqq 3\tau$ *where* τ *is the root of the equation*

$$\sum_{i=1}^{N} (m_i n_i p_i)^\tau = \mathrm{Rk}_\lambda\left(\bigoplus_{i=1}^{N} \langle m_i, n_i, p_i \rangle\right). \tag{2.4}$$

We call (2.4) the equation associated with $\bigoplus_{i=1}^{N} \langle m_i, n_i, p_i \rangle$.

3. Generalization of Schönhage's construction. We will now generalize the construction of Schönhage which was described in the end of the last section. The main result of this section is that if $\mathfrak{a}$ is an algorithm which can λ-compute a system $\mathscr{B}$ of bilinear forms and if $\mathfrak{a}$ is a nonminimal algorithm in a way which will be made precise in the statement of Theorem A, then there exists an algorithm $\mathfrak{a}'$ which can λ-compute $\mathscr{B} \oplus \langle 1, R, 1 \rangle$ satisfying $\mu(\mathfrak{a}') = \mu(\mathfrak{a})$. (The quantity R will be defined later in this section.) Indeed, Theorem A is the basic technical result of this paper, and most of the other results are consequences of this theorem. Before stating, and proving, Theorem A we have to establish some notation and terminology.

DEFINITION 3.1. Let $\mathfrak{a} = (M_1(\mathbf{x}), M_2(\mathbf{x}), \cdots, M_L(\mathbf{x}); N_1(\mathbf{y}), N_2(\mathbf{y}), \cdots, N_L(\mathbf{y}))$ be an algorithm which can λ-compute $\langle 1, R, 1 \rangle \oplus \mathscr{B}$. We say that $\langle 1, R, 1 \rangle$ is *isolated* (relative to $\mathfrak{a}$) if

$$\langle 1, R, 1 \rangle = \sum_{i=1}^{R} u_i v_i = \sum_{j=1}^{L} c_j M_j(\mathbf{x}) N_j(\mathbf{y})$$

for some $c_1, c_2, \cdots, c_L \in F(\lambda)$. That is, if $\mathfrak{a}$ not only λ-computes $\langle 1, R, 1 \rangle$ but also computes it. In the case that $c_j \neq 0$ for all $j = 1, 2, \cdots, L$ we say that $\langle 1, R, 1 \rangle$ is *full and isolated* (relative to $\mathfrak{a}$). In the tensorial notation, if $\tilde{\mathscr{B}}$ is the tensor for $\langle 1, R, 1 \rangle \oplus \mathscr{B}$, and $\bar{\mathscr{B}} = \sum_{l=1}^{L} \boldsymbol{\alpha}_l \otimes \boldsymbol{\beta}_l \otimes \boldsymbol{\gamma}_l$ is the decomposition of $\bar{\mathscr{B}}$ by $\mathfrak{a}$, then $\langle 1, R, 1 \rangle$ being isolated means that we have the matrical equality $\tilde{\mathscr{B}}(i, j, 1) = \bar{\mathscr{B}}(i, j, 1)$, and $\langle 1, R, 1 \rangle$ is full and isolated means that we further require that $\gamma_{l1} \neq 0$ for all $l = 1, 2, \cdots, L$.

In this paper we will either prove, or demand, that $\langle 1, R, 1 \rangle$ be full and isolated. That is, the algorithm $\mathfrak{a}$ will satisfy

$$\langle 1, R, 1 \rangle = \sum_{i=1}^{R} u_i v_i = \sum_{j=1}^{L} c_j M_j(\mathbf{x}) N_j(\mathbf{y})$$

where $0 \neq c_j \in F(\lambda)$ for all $j = 1, 2, \cdots, L$.

As in the last section, we use r to denote the number of x indeterminates, and s to denote the number of y indeterminates. We will characterize an $F(\lambda)$ algorithm $\mathfrak{a} = (M_1(\mathbf{x}), M_2(\mathbf{x}), \cdots, M_L(\mathbf{x}); (N_1(\mathbf{y}), N_2(\mathbf{y}), \cdots, N_L(\mathbf{y}))$ by the two matrices α and β, where α is the $L \times r$ matrix defined by

$$(M_1(\mathbf{x}), M_2(\mathbf{x}), \cdots, M_L(\mathbf{x})) = \mathbf{x}^T \alpha^T,$$

and β is the $L \times s$ matrix defined by

$$(N_1(\mathbf{y}), N_2(\mathbf{y}), \cdots, N_L(\mathbf{y})) = \mathbf{y}^T \beta^T.$$

In the rest of the paper we will assume that $\mathrm{Rk}(\alpha)=r$ and $\mathrm{Rk}(\beta)=s$. This assumption means that we cannot reduce the number of indeterminates appearing in $\mathfrak{a}$ by replacing the x_is by an invertible linear combination of new indeterminates, and similarly for the y_js.

THEOREM A. *Let $\mathfrak{a}$ be an $F(\lambda)$ algorithm which can λ-compute a system $\mathcal{B}$ of F-bilinear forms. If there exists an $L\times L$ nonsingular diagonal matrix C with entries in $F(\lambda)$, where $L=\mu(\mathfrak{a})$, so that $\alpha^TC\beta=0$, then there exists an algorithm $\mathfrak{a}'$ which can λ-compute $\mathcal{B}\oplus\langle 1,R,1\rangle$ satisfying*:

(a) $\mu(\mathfrak{a}')=\mu(\mathfrak{a})=L$,

(b) $R=L-r-s\geqq 0$,

(c) $\langle 1,R,1\rangle$ *is full and isolated* (*relative to* $\mathfrak{a}'$).

Proof. Before proving the theorem we should note that the assumption that $\alpha^TC\beta=0$ means that $\sum_{j=1}^L c_jM_j(\mathbf{x})N_j(\mathbf{y})=0$, where c_j, $j=1,2,\cdots,L$, are the diagonal entries of C. That means that $\mathfrak{a}$ is *not* a minimal algorithm, and therefore $\mathrm{Rk}_\lambda(\mathcal{B})<L$.

One more comment: We will prove that $R=L-r-s\geqq 0$. In case $R=0$ then $\langle 1,R,1\rangle$ is vacuous, and then what we mean by "$\langle 1,R,1\rangle$ is full and isolated (relative to $\mathfrak{a}'$)" is that there exist $0\neq c_j'\in F(\lambda)$, $1\leqq j\leqq L$, so that

$$0=\sum_{j=1}^{L} c_j'M_j'(\mathbf{x})N_j'(\mathbf{y})$$

where $\mathfrak{a}'=(M_1'(\mathbf{x}),M_2'(\mathbf{x}),\cdots,M_L'(\mathbf{x});N_1'(\mathbf{y}),N_2'(\mathbf{y}),\cdots,N_L'(\mathbf{y}))$.

And now to the proof of the theorem. By the assumptions that $\mathrm{Rk}(\alpha)=r$ and $\mathrm{Rk}(C)=L$ we have that $\mathrm{Rk}(\alpha^TC)=r$. Therefore there exists an $L\times(L-r)$ matrix $\hat V$ so that $\alpha^TC\hat V=0$, and $\mathrm{Rk}(\hat V)=L-r$. But $\alpha^TC\beta=0$ and $\mathrm{Rk}(\beta)=s$ so we may assume that $\hat V=(V\,|\,\beta)$, where V is an $L\times(L-r-s)$ matrix satisfying $\mathrm{Rk}(V)=L-r-s$. This shows that $L-r-s\geqq 0$, as stated by the theorem.

Similarly there exists an $L\times(L-s)$ matrix $\hat U$ so that $\hat U^TC\beta=0$, and $\mathrm{Rk}(\hat U)=L-s$. We may also assume that $\hat U=(U\,|\,\alpha)$, where U is an $L\times(L-r-s)$ matrix satisfying $\mathrm{Rk}(U)=L-r-s$.

Consider the $(L-s)\times(L-r)$ matrix $\hat W=\hat U^TC\hat V$. Because $\mathrm{Rk}(\hat U^T)=L-s$, $\mathrm{Rk}(C)=L$, $\mathrm{Rk}(\hat V)=L-s$, we have $\mathrm{Rk}(\hat W)\geqq L-r-s$. On the other hand,

$$\hat W=\hat U^TC\hat V=\left(\frac{U^T}{\alpha^T}\right)C(V\,|\,\beta)=\left(\begin{array}{c|c}U^TCV & U^TC\beta\\ \hline \alpha^TCV & \alpha^TC\beta\end{array}\right)=\left(\begin{array}{c|c}W&0\\ \hline 0&0\end{array}\right)$$

where W is the $(L-r-s)\times(L-r-s)$ matrix U^TCV. Because $L-r-s\geqq\mathrm{Rk}(W)=\mathrm{Rk}(\hat W)\geqq L-r-s$, we have that $\mathrm{Rk}(W)=L-r-s=R$. With no loss of generality we may assume that $W=I(R)$, the $R\times R$ identity matrix; for if not we can replace $\hat V$ by

$$\hat V\begin{pmatrix}W^{-1}&0\\0&I(s)\end{pmatrix}$$

($I(s)$ being the $s\times s$ identity matrix), that is, we can replace V by VW^{-1}.

We will now construct the algorithm $\mathfrak{a}'$. Let $\xi_1,\xi_2,\cdots,\xi_R$ and $\eta_1,\eta_2,\cdots,\eta_R$ be new indeterminates. Let $\mathbf{u}_1^T,\mathbf{u}_2^T,\cdots,\mathbf{u}_L^T$ be the L rows of U and $\mathbf{v}_1^T,\mathbf{v}_2^T,\cdots,\mathbf{v}_L^T$ be the L rows of V. We define $\mathfrak{a}'=(M_1'(\boldsymbol{\xi},\mathbf{x}),\ M_2'(\boldsymbol{\xi},x),\cdots,M_L'(\boldsymbol{\xi},\mathbf{x});\ N_1'(\boldsymbol{\eta},\mathbf{y}),\ N_2'(\boldsymbol{\eta},\mathbf{y}),\cdots,\ N_L'(\boldsymbol{\eta},\mathbf{y}))$ by $M_j'(\boldsymbol{\xi},\mathbf{x})=\lambda^du_j^T\boldsymbol{\xi}+M_j(\mathbf{x})$, $1\leqq j\leqq L$, and $N_j'(\boldsymbol{\eta},\mathbf{y})=\lambda^dv_j^T\boldsymbol{\eta}+N_j(\mathbf{y})$, $1\leqq j\leqq L$. Here $\boldsymbol{\xi}$ is the column vector $\boldsymbol{\xi}=(\xi_1,\xi_2,\cdots,\xi_R)^T$ and $\boldsymbol{\eta}$ is the column vector $\boldsymbol{\eta}=(\eta_1,\eta_2,\cdots,\eta_R)^T$. The integer d is chosen so large that $\mathfrak{a}'$ can

λ-compute $\mathscr{B}$. In other words, the algorithm $\mathfrak{a}'$ is specified by $\alpha' = (\lambda^d U|\alpha)$ and $\beta' = (\lambda^d V|\beta)$. All that is left to show is that $\mathfrak{a}'$ can λ-compute $\langle 1, R, 1\rangle = \sum_{i=1}^R \xi_i\eta_i$ so that $\langle 1, R, 1\rangle$ is full and isolated.

Let $c_1, c_2, \cdots, c_L$ be the diagonal entries C, and let $c_j' = \lambda^{-2d}c_j$, $1 \leqq j \leqq L$. Then

$$\sum_{j=1}^L c_j' M_j'(\boldsymbol{\xi}, \mathbf{x})N_j'(\boldsymbol{\eta}, \mathbf{y}) = \lambda^{-2d}(\boldsymbol{\xi}^T|\mathbf{x}^T)(\alpha')^T C\beta'(\boldsymbol{\eta}^T|\mathbf{y}^T)^T$$

$$= \lambda^{-2d}(\boldsymbol{\xi}^T|\mathbf{x}^T)\left(\begin{array}{c|c} \lambda^{2d}I(R) & 0 \\ \hline 0 & 0\end{array}\right)(\boldsymbol{\eta}^T|\mathbf{y}^T)^T = \boldsymbol{\xi}^T\boldsymbol{\eta} = \sum_{i=1}^R \xi_i\eta_i.$$

This proves the theorem. □

Remark 3.1. An examination of the proof shows that we can replace the requirement $\mathrm{Rk}(C) = 1$ by $\mathrm{Rk}(C) = L - p$. In this case the algorithm $\mathfrak{a}'$ λ-computes $\mathscr{B} \oplus \langle 1, R, 1\rangle$, where $R = L - r - s - p$. The term $\langle 1, R, 1\rangle$ is still isolated but is not necessarily full anymore.

We will now state, and prove, several consequences of Theorem A.

COROLLARY 3.1. *Let $\mathfrak{a}$ be an $F(\lambda)$ algorithm which can λ-compute a system $\mathscr{B}$ of F-bilinear forms. Let C be an $L \times L$ nonsingular diagonal matrix with entries in $F(\lambda)$, where $L = \mu(\mathfrak{a})$. There exists an $F(\lambda)$-algorithm $\mathfrak{a}'$ which can λ-compute $\mathscr{B} \oplus \langle 1, R, 1\rangle$ satisfying $\mu(\mathfrak{a}') = \mu(\mathfrak{a}) + \rho$, where $\rho = \mathrm{Rk}(\alpha^T C\beta)$, and $R = \mu(\mathfrak{a}) - r - s + \rho$.*

Proof. Let $c_1, c_2, \cdots, c_L$ be the diagonal entries of C. Let $\mathfrak{a} = (M_1(\mathbf{x}), \cdots, M_L(\mathbf{x}); N_1(\mathbf{y}), \cdots, N_L(\mathbf{y}))$. Then because $\mathrm{Rk}(\alpha^T C\beta) = \rho$ we obtain that $\sum_{j=1}^L c_j M_j(\mathbf{x})N_j(\mathbf{y}) = \mathbf{x}^T\alpha^T C\beta\mathbf{y} = \sum_{i=1}^\rho M_{L+i}(\mathbf{x})N_{L+i}(\mathbf{y})$. Define the algorithm $\tilde{\mathfrak{a}}$ by $\tilde{\mathfrak{a}} = (M_1(\mathbf{x}), M_2(\mathbf{x}), \cdots, M_L(\mathbf{x}), M_{L+1}(\mathbf{x}), \cdots, M_{L+\rho}(\mathbf{x}); N_1(\mathbf{y}), N_2(\mathbf{y}), \cdots, N_L(\mathbf{y}), N_{L+1}(\mathbf{y}), \cdots, N_{L+\rho}(\mathbf{y}))$. The algorithm $\tilde{\mathfrak{a}}$ can clearly λ-compute $\mathscr{B}$, and it also satisfies $0 = \sum_{j=1}^L c_j M_j(\mathbf{x})N_j(\mathbf{y}) - \sum_{i=1}^\rho M_{L+i}(\mathbf{x})N_{L+i}(\mathbf{y})$. Apply Theorem A to $\tilde{\mathfrak{a}}$. □

COROLLARY 3.2. *For every m, n, p there exists an algorithm $\mathfrak{a}'$ which can λ-compute $\langle m, n, p\rangle \oplus \langle 1, R, 1\rangle$ such that $\mu(\mathfrak{a}') = \mathrm{Rk}(\langle m, n, p\rangle) + n$, where $R = \mathrm{Rk}(\langle m, n, p\rangle) - n(m + p - 1)$. Here $\mathrm{Rk}(\langle m, n, p\rangle)$ is the rank of the tensor $\langle m, n, p\rangle$,* i.e., *the smallest L so that an algorithm $\mathfrak{a}$ for computing $\langle m, n, p\rangle$ exists, with $\mu(\mathfrak{a}) = L$.*

Proof. Let $\mathfrak{a}$ be an algorithm for computing $\langle m, n, p\rangle$ with $\mu(\mathfrak{a}) = L$. That is, there exist elements γ_{lik} in F such that

$$\sum_{j=1}^n x_{ij}y_{jk} = \sum_{l=1}^L \gamma_{lik}M_l(\mathbf{x})N_l(\mathbf{y}), \qquad 1 \leqq i \leqq m, \quad 1 \leqq k \leqq p. \tag{3.1}$$

Consider the bilinear form $\psi = \sum_{j=1}^n (\sum_{i=1}^m a_i x_{ij})(\sum_{k=1}^p b_k y_{jk}) = \sum_{i=1}^m \sum_{j=1}^n \sum_{k=1}^p a_i b_k x_{ij} y_{jk}$. By equation (3.1) we have

$$\psi = \sum_{i=1}^m \sum_{k=1}^p \sum_{l=1}^L a_i b_k \gamma_{lik} M_l(\mathbf{x})N_l(\mathbf{y}) = \sum_{l=1}^L \left(\sum_{i=1}^m \sum_{k=1}^p a_i b_k \gamma_{lik}\right) M_l(\mathbf{x})N_l(\mathbf{y}).$$

From the assumption that $\mathrm{Rk}(\langle m, n, p\rangle) = L$ we conclude that for every $l = 1, 2, \cdots, L$ there exist i, k such that $\gamma_{lik} \neq 0$. It is well known [10, vol. 1, p. 86] that because $F(\lambda)$ has infinitely many elements, there exist $a_i \in F(\lambda)(1 \leqq i \leqq m)$ and $b_k \in F(\lambda)(1 \leqq k \leqq p)$ such that $0 \neq c_l = \sum_{i=1}^m \sum_{k=1}^p a_i b_k \gamma_{lik}$, $l = 1, 2, \cdots, L$. Choosing $C = \mathrm{diag}(c_1, c_2, \cdots, c_L)$, we now have $\mathrm{Rk}(C) = L$ and $\mathrm{Rk}(\alpha^T C\beta) = n$. The corollary now follows from Corollary 3.1. □

COROLLARY 3.3 (Schönhage's construction). *For every $m \geqq 1$, $n \geqq 1$ there exists an algorithm $\mathfrak{a}$ which can λ-compute $\langle m, 1, n\rangle \oplus \langle 1, (m-1)(n-1), 1\rangle$ such that $\mu(\mathfrak{a}) = mn + 1$.*

Proof. It is known that $\mathrm{Rk}(\langle m, 1, n\rangle) = mn$. Apply Corollary 3.2. □

COROLLARY 3.4. *For any $n>1$, if* $\mathrm{Rk}(\langle n,n,n\rangle)=n^{\omega_0}$ *then* $\omega<\omega_0$.

Proof. It was shown in [12] that $\mathrm{Rk}(\langle n,n,n\rangle)\geqq 2n^2-1$, and therefore $\omega_0>2$ and $\omega_0^2/3>1$. Let $\mathfrak{a}$ be an algorithm which computes $\langle n,n,n\rangle$ satisfying $\mu(\mathfrak{a})=n^{\omega_0}$. Therefore, for any positive integer s there exists an algorithm $\mathfrak{a}_s$ which computes $\langle n^s,n^s,n^s\rangle$ and satisfies $\mu(\mathfrak{a}_s)=n^{s\omega_0}$. By Corollary 3.2 there exists an algorithm $\mathfrak{a}_s$ which can λ-compute $\langle n^s,n^s,n^s\rangle\oplus\langle 1,R_s,1\rangle$, satisfying $\mu(\mathfrak{a}_s)=n^{s\omega_0}+n^s$, where $R_s=n^{s\omega_0}-2n^{2s}+n^s$. Since $\omega_0>2$, the $n^{s\omega_0}$ term dominates. Now choose s so large that $R_s^{\omega_0/3}>n^s$. Such an s exists because $\omega_0^2/3>1$. The equation associated with this $\mathfrak{a}_s'$ is $n^{3s\tau}+R_s^\tau=n^{s\omega_0}+n^s$. If we take $\tau=\omega_0/3$ we have $n^{s\omega_0}+R_s^{\omega_0/3}>n^{s\omega_0}+n^s$, so the root τ_1 of the equation satisfies $3\tau_1<\omega_0$. By Theorem 2.1, $\omega\leqq 3\tau_1<\omega_0$.

Corollary 3.4 says that no single algorithm for computing $\langle n,n,n\rangle$, like the algorithm of Strassen [3] for $\langle 2,2,2\rangle$, can give us ω. The next corollary says that under certain conditions, the root of the associated equation does not yield ω either. Before stating this result we need some notation.

Let $\mathfrak{a}$ be an algorithm. We use $\rho(\mathfrak{a})$ to denote $\rho(\mathfrak{a})=\min_C\{\mathrm{Rk}(\alpha^TC\beta)\}$, where C ranges over all $\mu(\mathfrak{a})\times\mu(\mathfrak{a})$ nonsingular diagonal matrices.

COROLLARY 3.5. *Let $\mathfrak{a}$ be an algorithm which can λ-compute $\bigoplus_{i=1}^t\langle m_i,n_i,p_i\rangle$, and let τ_0 be the root of the associated equation. If $\mu(\mathfrak{a})>\max\{\sum_{i=1}^t m_in_i,\sum_{i=1}^t n_ip_i\}$ and $\mu(\mathfrak{a})^{\tau_0}>\rho(\mathfrak{a})$, then $\omega<3\tau_0$.*

Proof. We will apply Corollary 3.1 to the algorithm $\mathfrak{a}_d=(\mathfrak{a})^{\otimes d}$. (We use $X^{\otimes d}$ to denote the d-fold tensor product $X\otimes X\otimes\cdots\otimes X$). The algorithm $\mathfrak{a}_d$ can λ-compute $\mathscr{B}_d=(\bigoplus_{i=1}^t\langle m_i,n_i,p_i\rangle)^{\otimes d}$, and therefore (using the notation of Theorem A) $r_d=(\sum_{i=1}^t m_in_i)^d=r^d$, and $s_d=(\sum_{i=1}^t n_ip_i)^d=s^d$. It is also easy to verify that $\rho(\mathfrak{a}_d)\leqq\rho(\mathfrak{a})^d$. Because $\mu(\mathfrak{a})>\max(r,s)$, and $\mu(\mathfrak{a})^{\tau_0}>\rho(\mathfrak{a})$, we can choose d so large that $(\mu(\mathfrak{a})^d-r^d-s^d+\rho(\mathfrak{a})^d)^{\tau_0}>\rho(\mathfrak{a})^d$. By Corollary 3.1, applied to $\mathfrak{a}_d$, there exists an algorithm $\mathfrak{a}'$ which can λ-compute $(\bigoplus_{i=1}^t\langle m_i,n_i,p_i\rangle)^{\otimes d}\oplus\langle 1,R,1\rangle$ where $R=\mu(\mathfrak{a})^d-r^d-s^d+\rho(\mathfrak{a})^d$, and $\mu(\mathfrak{a}')=\mu(\mathfrak{a})^d+\rho(\mathfrak{a})^d$. The equation associated with $\mathfrak{a}'$ is:

$$\left(\sum_{i=1}^t(m_in_ip_i)^\tau\right)^d+R^\tau=\mu(\mathfrak{a})^d+\rho(\mathfrak{a})^d. \tag{3.2}$$

But, by the choice of d, and by the fact that $\sum_{i=1}^t(m_in_ip_i)^{\tau_0}=\mu(\mathfrak{a})$, we obtain

$$\left(\sum_{i=1}^t(m_in_ip_i)^{\tau_0}\right)^d+R^{\tau_0}>\mu(\mathfrak{a})^d+\rho(\mathfrak{a})^d.$$

So if we denote the root of (3.2) by τ_1, Theorem 2.1 yields $\omega\leqq 3\tau_1<3\tau_0$. □

We can apply Corollary 3.5 to Schönhage's construction. Consider Schönhage's algorithm, $\mathfrak{a}$, which λ-computes $\langle 4,1,4\rangle\oplus\langle 1,9,1\rangle$ using 17 multiplications. It can be shown that $\rho(\mathfrak{a})\leqq 9$. Direct calculation shows that $17^\tau>9$, where τ is the root of $16^\tau+9^\tau=17$. Corollary 3.5 then states that $\omega<3\tau$. More than that, the proofs of Corollaries 3.5 and 3.1 and Theorem A give us a method for constructing the algorithm whose associated equation has a root smaller than τ.

4. Further consequences of Theorem A. In the end of § 3 we used Theorem A to show that a certain class of algorithms is not optimal. That is, if τ is the root of the equation associated with algorithm $\mathfrak{a}$, we used Theorem A to construct an algorithm $\mathfrak{a}'$, having $\tau'<\tau$ as the root of its associated equation. The main result of this section, Theorem B, will serve to show that the algorithms constructed by Theorem A and Corollary 3.1 cannot be optimal either. If $\mathfrak{a}_1$ is the algorithm constructed by Theorem A, then Theorem B will enable us to construct a sequence of algorithms $\mathfrak{a}_2,\mathfrak{a}_3,\cdots$, with the associated roots $\tau_2,\tau_3,\cdots$ satisfying $\tau_{i+1}<\tau_i$, $i=1,2,\cdots$.

In this section, we will concentrate on algorithms which can λ-compute $\langle 1, R, 1\rangle \oplus \mathscr{B}$. For the sake of definiteness we will assume that $\langle 1, R, 1\rangle$ is the first bilinear form in the system $\langle 1, R, 1\rangle \oplus \mathscr{B}$. We will begin the section by proving that "fullness" and "isolation" are closed under tensor product construction.

PROPOSITION 4.1. *Let* $\mathfrak{a}$ *λ-compute* $\mathscr{C} = \langle 1, R, 1\rangle \oplus \mathscr{B}$ *and* $\mathfrak{a}'$ *λ-compute* $\mathscr{C}' = \langle 1, R', 1\rangle \oplus \mathscr{B}'$. *If* $\langle 1, R, 1\rangle$ *and* $\langle 1, R', 1\rangle$ *are isolated (relative to* $\mathfrak{a}$ *and* $\mathfrak{a}'$*, respectively) then* $\langle 1, RR', 1\rangle$ *is isolated relative to* $\mathfrak{a} \otimes \mathfrak{a}'$. *Furthermore, if* $\langle 1, R, 1\rangle$ *and* $\langle 1, R', 1\rangle$ *are full and isolated, then so is* $\langle 1, RR', 1\rangle$.

Proof. Because the $((i, i'), (j, j'), (1, 1))$ entries of $\mathscr{C} \otimes \mathscr{C}'$ satisfy $(\mathscr{C} \otimes \mathscr{C}')((i, i'), (j, j'), (1, 1)) = \mathscr{C}(i, j, 1)\mathscr{C}'(i', j', 1)$; and similarly for $\bar{\mathscr{C}} \otimes \bar{\mathscr{C}}'$, we have $(\mathscr{C} \otimes \mathscr{C}')((i, i'), (j, j'), (1, 1)) = (\bar{\mathscr{C}} \otimes \bar{\mathscr{C}}')((i, i'), (j, j'), (1, 1))$. So $\langle 1, RR', 1\rangle$ is isolated if $\langle 1, R, 1\rangle$ and $\langle 1, R', 1\rangle$ are isolated.

Let $\mathfrak{a}$ decompose $\bar{\mathscr{C}} = \overline{\langle 1, R, 1\rangle \oplus \mathscr{B}}$ as $\bar{\mathscr{C}} = \sum_{l=1}^{L} \boldsymbol{\alpha}_l \otimes \boldsymbol{\beta}_l \otimes \boldsymbol{\gamma}_l$, and let $\mathfrak{a}'$ decompose $\bar{\mathscr{C}}' = \overline{\langle 1, R', 1\rangle + \mathscr{B}'}$ as $\bar{\mathscr{C}}' = \sum_{l'=1}^{L'} \boldsymbol{\alpha}'_{l'} \otimes \boldsymbol{\beta}'_{l'} \otimes \boldsymbol{\gamma}'_{l'}$. Then $\mathfrak{a} \otimes \mathfrak{a}'$ decomposes $\overline{\mathscr{C} \otimes \mathscr{C}'}$ as $\overline{\mathscr{C} \otimes \mathscr{C}'} = \sum_{l'=1}^{L'} \sum_{l=1}^{L} (\boldsymbol{\alpha}_l \otimes \boldsymbol{\alpha}'_{l'}) \otimes (\boldsymbol{\beta} \otimes \boldsymbol{\beta}'_{l'}) \otimes (\boldsymbol{\gamma}_l \otimes \boldsymbol{\gamma}'_{l'})$. The $(1, 1)$ entry of $\boldsymbol{\gamma}_l \otimes \boldsymbol{\gamma}'_{l'}$ is $\gamma_{l,1}\gamma'_{l',1}$, which is not zero if $\gamma_{l,1} \neq 0$ and $\gamma'_{l',1} \neq 0$ for all $l = 1, 2, \cdots, L$ and $l' = 1, 2, \cdots, L'$. $\square$

It will be convenient, in the rest of this section, to give a separate designation to the indeterminates which appear in $\langle 1, R, 1\rangle$. We will therefore denote the x-indeterminates of $\langle 1, R, 1\rangle \oplus \mathscr{B}$ by $\xi_1, \xi_2, \cdots, \xi_R, x_1, x_2, \cdots, x_r$, and its y-indeterminates by $\eta_1, \eta_2, \cdots, \eta_R, y_1, y_2, \cdots, y_s$. (This notation is somewhat in conflict with the one we have used up to now. The "old" meaning of r is now replaced by $R + r$, and the "old" meaning of s is now replaced by $R + s$. We trust that this change of notation will not cause confusion.) We will also denote the (column) vector $(\xi_1, \xi_2, \cdots, \xi_R, x_1, x_2, \cdots, x_r)^T$ by $(\boldsymbol{\xi}, \mathbf{x})$, and the (column) vector $(\eta_1, \eta_2, \cdots, \eta_R, y_1, y_2, \cdots, y_s)^T$ by $(\boldsymbol{\eta}, \mathbf{y})$.

Even though the next theorem, Theorem B, is an immediate consequence of Theorem A we will not label it as a corollary. The central role it plays in this section justifies calling it a theorem. But before we state the theorem we need a lemma.

LEMMA 4.1. *Let* $\mathfrak{a}$ *be an algorithm which* λ*-computes* $\langle 1, R, 1\rangle \oplus \mathscr{B}$ *so that* $\langle 1, R, 1\rangle$ *is full and isolated. Let* r *and* s *denote the number of* x_i*'s and* y_i*'s of* $\mathscr{B}$, *then*

$$\mu(\mathfrak{a}) - r - s \geqq R.$$

Proof. By assumption that $\langle 1, R, 1\rangle$ is full and isolated we have that $\langle 1, R, 1\rangle = \sum_{i=1}^{R} \xi_i\eta_i = \sum_{j=1}^{L} c_j M_j(\boldsymbol{\xi}, \mathbf{x}) N_j(\boldsymbol{\eta}, \mathbf{y})$ for some $0 \neq c_j \in F(\lambda)$, $1 \leqq j \leqq L$, where $L = \mu(\mathfrak{a})$. Construct the algorithm $\tilde{\mathfrak{a}} = (M_1(\boldsymbol{\xi}, \mathbf{x}), M_2(\boldsymbol{\xi}, \mathbf{x}), \cdots, M_L(\boldsymbol{\xi}, \mathbf{x}), \xi_1, \xi_2, \cdots, \xi_R; N_1(\boldsymbol{\eta}, \mathbf{y}), N_2(\boldsymbol{\eta}, \mathbf{y}), \cdots, N_L(\boldsymbol{\eta}, \mathbf{y}), \eta_1, \eta_2, \cdots, \eta_R)$. The algorithm $\tilde{\mathfrak{a}}$ can also compute $\langle 1, R, 1\rangle \oplus \mathscr{B}$ and satisfies $\mu(\tilde{\mathfrak{a}}) = \mu(\mathfrak{a}) + R$. But $\tilde{\mathfrak{a}}$ also has the property that $\sum_{j=1}^{L} c_j M_j(\boldsymbol{\xi}, \mathbf{x}) N_j(\boldsymbol{\xi}, \mathbf{x}) - \sum_{i=1}^{R} \xi_i\eta_i = 0$. Theorem A, applied to $\tilde{\mathfrak{a}}$, states that $\mu(\mathfrak{a}) + R = \mu(\tilde{\mathfrak{a}}) \geqq r + s + 2R$. The statement of the lemma follows. $\square$

THEOREM B. *Let* $\mathfrak{a}$ *be an* $F(\lambda)$*-algorithm which can* λ*-compute* $\langle 1, R, 1\rangle \oplus \mathscr{B}$. *If* $\langle 1, R, 1\rangle$ *is full and isolated relative to* $\mathfrak{a}$, *then there exists an* $F(\lambda)$*-algorithm* $\mathfrak{a}'$ *which can* λ*-compute* $\langle 1, R^*, 1\rangle \oplus \mathscr{B}$ *and satisfies* $\mu(\mathfrak{a}') = \mu(\mathfrak{a})$, *where* $R^* = \mu(\mathfrak{a}) - r - s \geqq R$. *Moreover,* $\langle 1, R^*, 1\rangle$ *is full and isolated (relative to* $\mathfrak{a}'$*).*

Proof. Let $\mathfrak{a} = (M_1(\boldsymbol{\xi}, \mathbf{x}), M_2(\boldsymbol{\xi}, \mathbf{x}), \cdots, M_L(\boldsymbol{\xi}, \mathbf{x}); N_1(\boldsymbol{\eta}, \mathbf{y}), N_2(\boldsymbol{\eta}, \mathbf{y}), \cdots, N_L(\boldsymbol{\eta}, \mathbf{y}))$. By the assumption that $\langle 1, R, 1\rangle$ is full and isolated we have $\sum_{i=1}^{R} \xi_i\eta_i = \sum_{j=1}^{L} c_j M_j(\boldsymbol{\xi}, \mathbf{x}) N_j(\boldsymbol{\eta}, \mathbf{y})$ where $0 \neq c_j \in F(\lambda)$ for all $j = 1, 2, \cdots, L$. Let $\tilde{\mathfrak{a}}$ be the algorithm $\tilde{\mathfrak{a}} = (\tilde{M}_1(\mathbf{x}), \tilde{M}_2(\mathbf{x}), \cdots, \tilde{M}_L(\mathbf{x}); \tilde{N}_1(\mathbf{y}), \tilde{N}_2(\mathbf{y}), \cdots, \tilde{N}_L(\mathbf{y}))$ obtained by setting $\xi_i = \eta_i = 0$, $1 \leqq i \leqq R$. We thus obtain $0 = \sum_{j=1}^{L} c_j \tilde{M}_j(\mathbf{x}) \tilde{N}_j(\mathbf{y})$. The algorithm $\tilde{\mathfrak{a}}$ can, clearly, λ-compute $\mathscr{B}$. Apply Theorem A to $\tilde{\mathfrak{a}}$ to obtain an algorithm $\mathfrak{a}'$ which can λ-compute

$R^*, 1\rangle \oplus \mathscr{B}$ so that $\mu(\mathfrak{a}') = \mu(\tilde{\mathfrak{a}}) = \mu(\mathfrak{a})$, $\langle 1, R^*, 1\rangle$ is full and isolated, and $R^* = \mu(\mathfrak{a}) - r - s$. The fact that $R^* \geqq R$ is a restatement of Lemma 4.1. □

If an algorithm $\mathfrak{a}$ λ-computes $\langle 1, R, 1\rangle \oplus (\bigoplus_{i=1}^t \langle m_i, n_i, p_i\rangle)$ so that $\langle 1, R, 1\rangle$ is full and isolated, and if, furthermore, $R < R^* = \mu(\mathfrak{a}) - r - s = \mu(\mathfrak{a}) - \sum_{i=1}^t m_i n_i - \sum_{i=1}^t n_i p_i$, then Theorem B guarantees the existence of an algorithm $\mathfrak{a}'$, with associated root τ', which satisfies $\tau' < \tau$, where τ is the associated root of $\mathfrak{a}$. But what if $R = \mu(\mathfrak{a}) - r - s$? In this case, we can always find another algorithm, namely, $\mathfrak{a}' = \mathfrak{a} \otimes \mathfrak{a}$, which also has τ as its associated root, and which satisfies $\mu(\mathfrak{a}') - r' - s' > R'$. We will now state and prove this assertion.

COROLLARY 4.1. *Let $\mathfrak{a}$ be an algorithm which can λ-compute $\langle 1, R, 1\rangle \oplus (\bigoplus_{i=1}^t \langle m_i, n_i, p_i\rangle)$ so that $\langle 1, R, 1\rangle$ is full and isolated. There exists an infinite sequence of algorithms $\mathfrak{a} = \mathfrak{a}_1, \mathfrak{a}_2, \mathfrak{a}_3, \cdots$ so that $\mathfrak{a}_j$ can λ-compute $\langle 1, R(j), 1\rangle \oplus aQ_{i=1}^{t(j)} \langle m_i(j), n_i(j), p_i(j)\rangle)$, and such that $\tau_1 > \tau_2 > \tau_3, \cdots$ where τ_j is the root associated with $\mathfrak{a}_j$.*

Proof. We will break the proof into two parts. If $\mathfrak{a}$ can λ-compute $\langle 1, R, 1\rangle \oplus \mathscr{B}$ where $\langle 1, R, 1\rangle$ is full and isolated, then $\mathfrak{a} \otimes \mathfrak{a} = \mathfrak{a}'$ can λ-compute $(\langle 1, R, 1\rangle \oplus \mathscr{B}) \otimes (\langle 1, R, 1\rangle \oplus \mathscr{B}) = \langle 1, R^2, 1\rangle \oplus [(\langle 1, R, 1\rangle \otimes \mathscr{B}) \oplus (\mathscr{B} \otimes \langle 1, R, 1\rangle) \oplus (\mathscr{B} \otimes \mathscr{B})] = \langle 1, R^2, 1\rangle \oplus \mathscr{B}'$ so that $\langle 1, R^2, 1\rangle$ is full and isolated. Let r' and s' denote the number of x and y indeterminates of $\mathscr{B}'$, and, respectively, r and s those of $\mathscr{B}$. Then $r' = 2Rr + r^2$ and $s' = 2Rs + s^2$. Let L denote $\mu(\mathfrak{a})$; then simple calculations show that

$$L^2 - r' - s' = L^2 - r^2 - s^2 - 2(r+s)R = (L - r - s)^2 + 2(r+s)(L - r - s - R) + 2rs. \tag{4.1}$$

But, by Theorem B, $L - r - s \geqq R$, and therefore (4.1) implies $L^2 - r' - s' > R^2$.

The second part of the proof starts with the observation that the roots associated with $\mathfrak{a}$ and $\mathfrak{a} \otimes \mathfrak{a}$ are the same. We therefore define $\mathfrak{a}_{i+1} = (\mathfrak{a}_i \otimes \mathfrak{a}_i)^*$, where the $*$ indicates the application of Theorem B. □

Example 4.1. It will be convenient to establish some "shorthand" convention. We use

$$L \to \langle m_1, n_1, p_1\rangle \oplus \langle m_2, n_2, p_2\rangle \oplus \cdots \oplus \langle m_t, n_t, p_t\rangle$$

to denote that there exists an algorithm $\mathfrak{a}$ which can λ-compute $\bigoplus_{i=1}^t \langle m_i, n_i, p_i\rangle$ such that $\mu(\mathfrak{a}) = L$. By applying Corollary 3.2 to the regular computation of $\langle 3, 1, 3\rangle$, or alternatively by appealing directly to Schönhage's construction (Construction 2.1), we obtain

$$10 \to \langle 1, 4, 1\rangle \oplus \langle 3, 1, 3\rangle$$

where $\langle 1, 4, 1\rangle$ is full and isolated. The root of the associated equation, τ_1, satisfies

$$3\tau_1 = 2.5938833.$$

Using the tensor product construction we get

$$100 \to \langle 1, 16, 1\rangle \oplus 2\langle 3, 4, 3\rangle \oplus \langle 9, 1, 9\rangle \tag{4.2}$$

where $2\langle 3, 4, 3\rangle$ denotes $\langle 3, 4, 3\rangle \oplus \langle 3, 4, 3\rangle$. The root of the associated equation is

$$\tau_2' = \tau_1.$$

Applying Theorem B we obtain

$$100 \to \langle 1, 34, 1\rangle \oplus 2\langle 3, 4, 3\rangle \oplus \langle 9, 1, 9\rangle$$

with the associated root, τ_2, satisfying

$$3\tau_2 = 2.5198543.$$

We again use the tensor product construction and get

$$10000 \to \langle 1, 1156, 1\rangle \oplus 4\langle 9, 16, 9\rangle \oplus \langle 81, 1, 81\rangle \oplus 4\langle 3, 136, 3\rangle \oplus 4\langle 27, 4, 27\rangle \oplus 2\langle 9, 34, 9\rangle$$

with the associated root

$$\tau_3' = \tau_2.$$

If we apply Theorem B we obtain

$$10000 \to \langle 1, 3334, 1\rangle \oplus 4\langle 9, 16, 9\rangle \oplus \langle 81, 1, 81\rangle \oplus 4\langle 3, 136, 3\rangle \oplus 4\langle 27, 4, 27\rangle \oplus 2\langle 9, 34, 9\rangle \tag{4.3}$$

whose associated root, τ_3, obeys

$$3\tau_3 = 2.4998847.$$

When we apply Theorem B to the "tensor" square of the algorithm of (4.3) we obtain the associated root, τ_4, where $3\tau_4 = 2.4977718$. The size of the algorithms as well as the fact that τ_4 is not much smaller than τ_3 dictated that we stop the iterative process here.

Example 4.1, as well as the proof of Corollary 4.1, used a straightforward application of Theorem B. Yet the effects of applying Theorem B are a little subtle. For example, when we start with Schönhage's construction for $\langle 1, 9, 1\rangle \oplus \langle 4, 1, 4\rangle$, which gave Schönhage the smallest value of τ, we did not get as much improvement in the values of the associated root as in Example 4.1. The reason was that $r+s$ for $\langle 1, 4, 1\rangle \oplus \langle 3, 1, 3\rangle$ was smaller than the value of $r+s$ for $\langle 1, 9, 1\rangle \oplus \langle 4, 1, 4\rangle$. We will illustrate this phenomenon in the next example, where we deliberately make the algorithm "worse," but reduce the value of $r+s$. But by applying Theorem B we more than get back what we sacrificed by reducing the efficiency of the algorithm.

Example 4.2. We start with the algorithm of (4.2). By identifying indeterminates we can turn $2\langle 3, 4, 3\rangle$ into $\langle 6, 4, 3\rangle$. We now have

$$100 \to \langle 1, 16, 1\rangle \oplus \langle 6, 4, 3\rangle \oplus \langle 9, 1, 9\rangle,$$

whose associated root, σ_2', satisfies

$$3\sigma_2' = 2.6230934 > 3\tau_2'.$$

Applying Theorem B we get

$$100 \to \langle 1, 46, 1\rangle \oplus \langle 6, 4, 3\rangle \oplus \langle 9, 1, 9\rangle \tag{4.4}$$

whose associated root, σ_2, satisfies

$$3\sigma_2 = 2.5104566 < 3\tau_2.$$

The "tensor product" construction now yields

$$10000 \to \langle 1, 2116, 1\rangle \oplus \langle 36, 16, 9\rangle \oplus \langle 81, 1, 81\rangle \oplus 2\langle 6, 184, 3\rangle \oplus 2\langle 9, 46, 9\rangle \oplus 2\langle 54, 4, 27\rangle \tag{4.5}$$

whose associated root is

$$\sigma_3' = \sigma_2,$$

and the application of Theorem B yields

$$10000 \to \langle 1, 3502, 1\rangle \oplus \langle 36, 16, 9\rangle \oplus \langle 81, 1, 81\rangle \oplus 2\langle 6, 184, 3\rangle \oplus 2\langle 9, 46, 9\rangle \oplus 2\langle 54, 4, 27\rangle$$

with the associated root, σ_3, satisfying

$$3\sigma_3 = 2.4993366. \tag{4.6}$$

However, we could have modified (4.5) by replacing $2\langle 6, 184, 3\rangle$ by $\langle 6, 184, 6\rangle$ and would have obtained

$$10000 \to \langle 1, 2116, 1\rangle \oplus \langle 36, 16, 9\rangle \oplus \langle 81, 1, 81\rangle \oplus \langle 6, 184, 6\rangle \oplus 2\langle 9, 46, 9\rangle \oplus 2\langle 54, 4, 27\rangle$$

with associated root, $\sigma_3^{*\prime}$, satisfying

$$3\sigma_3^{*\prime} = 2.5171476 > 3\sigma_3'.$$

But by applying Theorem B we get

$$10000 \to \langle 1, 4606, 1\rangle \oplus \langle 36, 16, 9\rangle \oplus \langle 81, 1, 8\rangle \oplus \langle 6, 184, 6\rangle \oplus 2\langle 9, 46, 9\rangle \oplus 2\langle 54, 4, 27\rangle \tag{4.7}$$

whose associated root, σ_3^*, satisfies

$$3\sigma_3^* = 2.4978379 < 3\sigma_3.$$

Another application of "tensor product" construction and Theorem B yields the associated root, σ_4^*, where $3\sigma_4^* = 2.4966600$.

This last construction of σ_4^* is not the best possible. The following easy-to-verify fact explains why.

Fact 4.1. Let $\mathfrak{a}$ λ-compute $\langle 1, R, 1\rangle \oplus (\bigoplus_{i=1}^t \langle m_i, n_i, p_i\rangle)$. Then there exists an algorithm $\mathfrak{a}^T$ which can λ-compute $\langle 1, R, 1\rangle \oplus \bigoplus_{i=1}^t \langle p_i, n_i, m_i\rangle$ such that:

(a) $\mu(\mathfrak{a}^T) = \mu(\mathfrak{a})$.

(b) If $\langle 1, R, 1\rangle$ is full and isolated relative to $\mathfrak{a}$, then $\langle 1, R, 1\rangle$ is full and isolated relative to $\mathfrak{a}^T$.

Moreover, if we denote by τ the root associated with $(\mathfrak{a} \otimes \mathfrak{a})^*$ (i.e., the application of Theorem B to $\mathfrak{a} \otimes \mathfrak{a}$), and by τ' the root associated with $(\mathfrak{a} \otimes \mathfrak{a}^T)^*$, then $\tau' \leqq \tau$ with equality if and only if $\sum_{i=1}^t m_i n_i = \sum_{i=1}^t n_i p_i$.

If we use Fact 4.1 on the algorithm of (4.7) we get $3\sigma_4^{*\prime} = 2.4966271$.

5. Generalized tensor product construction. The construction of the algorithm $\mathfrak{a} \otimes \mathfrak{a}'$ from the two algorithms $\mathfrak{a}$ and $\mathfrak{a}'$ was one of the main tools which enabled us to exploit Theorems A and B. In this section we will generalize this construction. The generalized construction will enable us to obtain a slightly better estimate of ω than the one given in the end of § 4. In addition, the construction may be of interest in its own right. Rather than subject the reader to a maze of notation, we will give only an informal description of the construction.

THEOREM 5.1. *Let $\mathfrak{a}$ be an algorithm which can λ-compute the system of bilinear forms $\mathcal{B} = \bigoplus_{r=1}^n \mathcal{B}_r$, with $\mu(\mathfrak{a}) = L$. For each $r = 1, 2, \cdots, n$, let $\mathfrak{a}^{(r)}$ be an algorithm which λ-computes the system of bilinear forms $\mathcal{B}^{(r)}$, with $\mu(\mathfrak{a}^{(r)}) = L'$. Then we can construct an algorithm $\mathfrak{c}$, called the* generalized tensor product *of $\mathfrak{a}$ and* $(\mathfrak{a}^{(1)},$

$\mathfrak{a}^{(2)}, \cdots, \mathfrak{a}^{(n)}$) *and denoted by* $\mathfrak{c} = \mathfrak{a} \otimes (\mathfrak{a}^{(1)}, \mathfrak{a}^{(2)}, \cdots, \mathfrak{a}^{(n)})$, *satisfying*:

$$\mu(\mathfrak{c}) = LL',$$

$$\mathfrak{c} \textit{ can } \lambda\textit{-compute } \bigoplus_{i=1}^{n} (\mathcal{B}_i \otimes \mathcal{B}^{(i)}).$$

Further, if $\mathcal{B}_1 = \langle 1, R, 1 \rangle$ *is a full isolated inner product* (*relative to* $\mathfrak{a}$), *and* $\mathcal{B}_1^{(1)} = \langle 1, R', 1 \rangle$ *is a full isolated inner product* (*relative to* $\mathfrak{a}^{(1)}$), *then* $\mathcal{B}_1 \otimes \mathcal{B}_1^{(1)} = \langle 1, RR', 1 \rangle$ *is a full isolated inner product* (*relative to* $\mathfrak{c}$).

Sketch of construction. Choose a large integer d, and let $\mathfrak{a}''$ be the algorithm obtained from $\mathfrak{a}$ by substituting λ^d for λ. Then $\mathfrak{a}''$ uses L multiplications to λ-compute $\bigoplus_{r=1}^{n} \mathcal{B}_r$. Further, each bilinear form in $\bigoplus_{r=1}^{n} \mathcal{B}_r$ is computed correctly mod (λ^d).

Take L' copies of algorithm $\mathfrak{a}''$, thus using LL' multiplications to λ-compute $L' \cdot \mathcal{B} = \bigoplus_{r=1}^{n} L' \cdot \mathcal{B}_r$. (Here $L' \cdot \mathcal{B}$ denotes $\mathcal{B} \oplus \mathcal{B} \oplus \cdots \oplus \mathcal{B}$, the direct sum of L' copies of $\mathcal{B}$.)

Let us develop some notation here. Denote by $\xi_r = (\xi_{ri'})$ the system of x-variables in $\mathcal{B}_r$. Similarly $\eta_r = (\eta_{rj'})$ is the system of y-variables in $\mathcal{B}_r$. The product $\xi_r \eta_r$ will denote the system of bilinear forms $\mathcal{B}_r$. When we have several copies of $\mathcal{B}_r$, indexed by i or j, we will denote by $(\xi_r)_i = (\xi_{ri'i})$ the system of x-variables in the ith copy of $\mathcal{B}_r$.

Consider the algorithm $\mathfrak{a}^{(r)}$, which uses L' multiplications to λ-compute $\mathcal{B}^{(r)}$. In $\mathfrak{a}^{(r)}$, formally replace each x-variable $x_i^{(r)}$ of $\mathcal{B}^{(r)}$ by the system $(\xi_r)_i$ of x-variables in the ith copy of $\mathcal{B}_r$. Similarly, replace each y-variable $y_j^{(r)}$ in $\mathfrak{a}^{(r)}$ by the system $(\eta_r)_j$. Wherever $\mathfrak{a}^{(r)}$ requires a linear combination of x-variables $\sum_i \alpha_{li}(\lambda) x_i$, take the corresponding linear combination of systems, component by component: $(\sum_i \alpha_{li}(\lambda) \xi_{ri'i})$, and similarly for y-variables. Interpret each product in $\mathfrak{a}^{(r)}$ as a system of bilinear forms similar to $\mathcal{B}_r$. This reinterpretation of algorithm $\mathfrak{a}^{(r)}$ tells us how to use L' copies of the *exact* computation of the system of bilinear forms $\mathcal{B}_r$, to λ-compute $\mathcal{B}_r \otimes \mathcal{B}^{(r)}$. Unfortunately, we only have available λ-computations of these L' copies of $\mathcal{B}_r$. However, these λ-computations are correct modulo (λ^d), and if d is chosen large enough ($d > \operatorname{def}(\mathfrak{a}^{(r)}, \mathcal{B}^{(r)})$), the inexactness introduced by using these λ-computations instead of exact computations will not affect our algorithm's ability to λ-compute $\mathcal{B}_r \otimes \mathcal{B}^{(r)}$.

We summarize the construction. Use LL' multiplications, arranged according to algorithm $\mathfrak{a}''$, to λ-compute $L' \cdot \mathcal{B} = \bigoplus_{r=1}^{n} L' \cdot \mathcal{B}_r$. Then for each r, use the L' copies of the λ-computation of $\mathcal{B}_r$, arranged according to a suitable interpretation of algorithm $\mathfrak{a}^{(r)}$, to λ-compute $\mathcal{B}_r \otimes \mathcal{B}^{(r)}$. This completes the desired construction: algorithm $\mathfrak{c}$ uses LL' multiplications to λ-compute $\bigoplus_{r=1}^{n} \mathcal{B}_r \otimes \mathcal{B}^{(r)}$. We omit the proof of the statement about full isolated inner products.

Remark. This construction is not canonical, in that it depends on the choice of an integer d. However, we choose to refer to it as "the generalized tensor product" because of its relation to the usual tensor product. This relation is as follows: if $\mathfrak{a}^{(r)} = \mathfrak{a}'$ and $\mathcal{B}^{(r)} = \mathcal{B}'$ for $r = 1, 2, \cdots, n$, then $\mathfrak{a} \otimes (\mathfrak{a}', \mathfrak{a}', \cdots, \mathfrak{a}')$ can λ-compute $\mathcal{B} \otimes \mathcal{B}'$ and requires LL' multiplications, just as $\mathfrak{a} \otimes \mathfrak{a}'$ can λ-compute $\mathcal{B} \otimes \mathcal{B}'$ and requires LL' multiplications. However, the actual algorithms are slightly different, and the exact bilinear forms they compute (as polynomials in λ) are different, although they agree mod (λ).

Now let us use the generalized tensor product construction to improve our matrix exponent ω.

Example 5.1. We will start with the algorithm of (4.4) of Example 4.2, that is,

$$100 \rightarrow \langle 1, 46, 1 \rangle \oplus \langle 6, 4, 3 \rangle \oplus \langle 9, 1, 9 \rangle. \tag{5.1}$$

That means that we also have algorithm

$$100 \to \langle 1, 46, 1\rangle \oplus \langle 3, 4, 6\rangle \oplus \langle 9, 1, 9\rangle. \tag{5.2}$$

We now construct $\mathfrak{a}' = \mathfrak{a} \otimes (\mathfrak{a}, \mathfrak{a}^T, \mathfrak{a})$, where $\mathfrak{a}$ is the algorithm of (5.1) and $\mathfrak{a}^T$ is the algorithm of (5.2). When we apply Theorem B to $\mathfrak{a}'$ we obtain

$$\begin{aligned} 10000 \to \langle 1, 3646, 1\rangle &\oplus 2\langle 6, 184, 3\rangle \oplus 2\langle 9, 46, 9\rangle \\ &\oplus 2\langle 54, 4, 27\rangle \oplus \langle 18, 16, 18\rangle \oplus \langle 81, 1, 81\rangle \end{aligned} \tag{5.3}$$

and the root of the associated equation, σ_3^+, satisfies

$$3\sigma_3^+ = 2.4982509 < 3\sigma_3$$

where σ_3 is given by (4.6).

We can modify (5.3) by replacing $2\langle 6, 184, 3\rangle$ with $\langle 6, 184, 6\rangle$, and applying Theorem B to this modified algorithm yields

$$\begin{aligned} 10000 \to \langle 1, 4750, 1\rangle &\oplus \langle 6, 184, 6\rangle \oplus 2\langle 9, 46, 9\rangle \\ &\oplus 2\langle 54, 4, 27\rangle \oplus \langle 18, 16\ 18\rangle \oplus \langle 81, 1, 81\rangle \end{aligned} \tag{5.4}$$

and the associated root, σ_3^{++}, satisfies

$$3\sigma_3^{++} = 2.4968212 < 3\sigma_3^+.$$

Because the algorithm $\mathfrak{a}$ of (5.4) does not satisfy $r = s$, Fact 4.1 suggests that the next step is to compute $(\mathfrak{a} \otimes \mathfrak{a}^T)^*$. We will not describe the resulting algorithm, but only mention that its associated root, σ_4^+, satisfies

$$3\sigma_4^+ = 2.4957018.$$

One more application of the construction of the proof of Corollary 4.1 yields an associated root, σ^+, which satisfies

$$3\sigma^+ = 2.4956631.$$

Another use of the generalized tensor product construction is to increase the number of direct summands, even at the expense of $r + s$. We will illustrate that in the next example.

Example 5.2. The starting point of this example is the algorithm $\mathfrak{a}$ of (5.4). Using the generalized tensor product to obtain $\mathfrak{a}' = \mathfrak{a} \otimes (\mathfrak{a}, \mathfrak{a}, \mathfrak{a}, \mathfrak{b}, \mathfrak{a}, \mathfrak{b})$, where $\mathfrak{b}$ is the trivial algorithm which computes $10000\langle 1, 1, 1\rangle$, we obtain an algorithm whose associated root, τ_4^*, satisfies

$$3\tau_4^* = 2.4956168 < 3\sigma_4^+.$$

Using Fact 4.1, and then the construction of the proof of Corollary 4.1 we obtain

$$\omega < 2.4955640. \tag{5.5}$$

The estimate of (5.5) is not the best that can be gotten. By a rather intricate and lengthy construction we can obtain

$$\omega < 2.4955480. \tag{5.6}$$

We will not describe this construction. It does not use any new idea beyond the ones described in the last two sections. Our only justification for presenting (5.6) is to indicate to the reader the extent to which we succeeded in reducing the estimate of ω.

6. ω as a limit point. In § 3 (Corollary 3.4) we saw that if $\mathrm{Rk}(\langle n, n, n\rangle) = n^{\omega_0}$ then $\omega < \omega_0$. In § 4 (Corollary 4.1) we saw that if $\mathfrak{a}$ is an algorithm which λ-computes $\langle 1, R, 1\rangle \oplus \bigoplus_{r=1}^{N} \langle m_r, n_r, p_r\rangle$, so that $\langle 1, R, 1\rangle$ is full and isolated, then $\omega < 3\tau$, where τ is the root of the associated equation. In this section we will generalize these two results. We will show that for every system of bilinear forms $\mathscr{B} = \bigoplus_{r=1}^{N} \langle m_r, n_r, p_r\rangle$, $\omega < 3\tau$ where τ is the root of the equation associated with $\mathscr{B}$; i.e., τ is the root of the equation

$$\sum_{r=1}^{N} (m_r n_r p_r)^{\tau} = \mathrm{Rk}_\lambda(\mathscr{B}).$$

In order to prove this result we will need to prove the following fact about $\mathrm{Rk}_\lambda(\mathscr{B})$, which may be of interest independently. Let $\mathscr{B} = \bigoplus_{r=1}^{N} \langle m_r, n_r, p_r\rangle$ so that for some r we have $m_r + n_r + p_r > 3$, then $3\mathrm{Rk}_\lambda(\mathscr{B}) > \sum_{r=1}^{N} m_r n_r + \sum_{r=1}^{N} n_r p_r + \sum_{r=1}^{N} p_r m_r$.

We now turn our attention to this last result. In order to prove it we will need some well known facts.

Fact 6.1. Let $\mathscr{B} = \bigoplus_{r=1}^{N} \langle m_r, n_r, p_r\rangle = A(\mathbf{x})\mathbf{y}$. Then $\rho_r(A(\mathbf{x})) = \sum_{r=1}^{N} m_r p_r$ and $\rho_c(A(\mathbf{x})) = \sum_{r=1}^{N} n_r p_r$. Therefore, by Proposition 2.4, $\mathrm{Rk}_\lambda(\mathscr{B}) \geqq \max(\sum_{r=1}^{N} m_r p_r, \sum_{r=1}^{N} n_r p_r)$.

Fact 6.2. $\mathrm{Rk}_\lambda(\bigoplus_{r=1}^{N} \langle m_r, n_r, p_r\rangle) = \mathrm{Rk}_\lambda(\bigoplus_{r=1}^{N} \langle n_r, p_r, m_r\rangle) = \mathrm{Rk}_\lambda(\bigoplus_{r=1}^{N} \langle p_r, m_r, n_r\rangle)$, and therefore $3\mathrm{Rk}_\lambda(\mathscr{B}) \geqq t_a + t_b + t_c$, where $\mathscr{B} = \bigoplus_{r=1}^{N} \langle m_r, n_r, p_r\rangle$, $t_a = \sum_{r=1}^{N} m_r n_r$, $t_b = \sum_{r=1}^{N} n_r p_r$, $t_c = \sum_{r=1}^{N} p_r m_r$.

Remark 6.1. The fact that $\mathrm{Rk}_\lambda(\mathscr{B}) \geqq t_c$ shows that the algorithms of examples 4.1, 4.2, 5.1, and 5.2 are all minimal.

Fact 6.3. Let $z_{i,k}^{(r)}$, $1 \leqq i \leqq m_r$, $1 \leqq k \leqq p_r$, $1 \leqq r \leqq N$ be the $t_c = \sum_{r=1}^{N} m_r p_r$ bilinear forms of $\bigoplus_{r=1}^{N} \langle m_r, n_r, p_r\rangle$. Let $f_{i,k}^{(r)} \in F$ be t_c elements of F such that for each $r = 1, 2, \cdots, N$ there exists $i = i(r)$, $k = k(r)$, so that $f_{i,k}^{(r)} \neq 0$. Then $\mathrm{Rk}_\lambda(\sum_{r=1}^{N} \sum_{k=1}^{p_r} \sum_{i=1}^{m_r} f_{i,k}^{(r)} z_{i,k}^{(r)}) \geqq \sum_{r=1}^{N} n_r$.

The first theorem of this section is:

THEOREM 6.1. *Let $\mathscr{B}$ be the system $\bigoplus_{r=1}^{N} \langle m_r, n_r, p_r\rangle$. Then $3\mathrm{Rk}_\lambda(\mathscr{B}) = t_a + t_b + t_c$ if and only if for every $r = 1, 2, \cdots, N$, we have $m_r = n_r = p_r = 1$.*

Proof. If $m_r = n_r = p_r = 1$ for every $r = 1, 2, \cdots, N$ then, clearly, $3\mathrm{Rk}_\lambda(\mathscr{B}) = t_a + t_b + t_c = 3N$.

We will assume now that $\mathrm{Rk}_\lambda(\mathscr{B}) = t_c$ and show that if $t_a = t_b = t_c$ then $m_r = n_r = p_r = 1$ for all $r = 1, 2, \cdots, N$. We will prove this assertion by induction on $L = \mathrm{Rk}_\lambda(\mathscr{B})$. If $L = 1$ the assertion is obvious, so we will assume that $L > 1$.

Let $z_{i,k}^{(r)} = \sum_{j=1}^{n_r} x_{i,j}^{(r)} y_{j,k}^{(r)}$, $1 \leqq i \leqq m_r$, $1 \leqq k \leqq p_r$, $1 \leqq r \leqq N$ be the $L = t_c = \sum_{r=1}^{N} m_r p_r$ bilinear forms of $\mathscr{B} = \bigoplus_{r=1}^{N} \langle m_r, n_r, p_r\rangle$. Let $\mathfrak{a} = (M_1(\mathbf{x}; \lambda), M_2(\mathbf{x}; \lambda), \cdots, M_L(\mathbf{x}; \lambda); N_1(\mathbf{y}; \lambda), N_2(\mathbf{y}; \lambda), \cdots, N_L(\mathbf{y}; \lambda))$ be a minimal algorithm which λ-computes $\mathscr{B}$. If we denote by $\mathbf{z}$ the L-dimensional column vector whose entries are the $z_{i,k}^{(r)}$s, and by $\boldsymbol{\pi}$ the L-dimensional column vector whose entries are the products $M_r(\mathbf{x}; \lambda)N_r(\mathbf{y}; \lambda)$, then there exists an $L \times L$ $F(\lambda)$-matrix T so that $\mathbf{z} =^{\lambda} T\boldsymbol{\pi}$, where $=^{\lambda}$ denotes equality modulo λ. Moreover, by Fact 6.1, T is an invertible matrix.

Let $T_1, T_2, \cdots, T_L$ be the L rows of T. We may assume that $\operatorname{def}(T_1) \geqq \operatorname{def}(T_2) \geqq \cdots \geqq \operatorname{def}(T_L)$, for if not we can permute the entries of $\mathbf{z}$ and the rows of T so that the assumption is satisfied.

The key observation of the proof is that if S is an $L \times L$ $F(\lambda)$-matrix satisfying $\operatorname{def}(S) \leqq 0$ then $\mathbf{z} =^{\lambda} T\boldsymbol{\pi}$ implies $S_0\mathbf{z} =^{\lambda} (ST)\boldsymbol{\pi}$, where $S_0 = cf_0(S)$.

Let $T_{i,j}$ denote the (i, j) element of T; then (by possibly permuting the columns of T and the entries of $\boldsymbol{\pi}$) we may assume that $\operatorname{def}(T_{1,1}) = \operatorname{def}(T_1) \geqq \operatorname{def}(T_{i,1})$ for all

$i=2, 3, \cdots, L$. Let S be the triangular $F(\lambda)$-matrix

$$S=\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ t_2 & 1 & 0 & 0 & \cdots & 0 \\ t_3 & 0 & 1 & 0 & \cdots & 0 \\ \vdots & & & & & \\ t_L & 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

where $t_i=-T_{i,1}/T_{1,1}$. The assumption on $T_{1,1}$ implies that def $(S)=0$, and clearly both S and $S_0=cf_0(S)$ are invertible. Therefore,

$$S_0\mathbf{z} \overset{\lambda}{=} (ST)\boldsymbol{\pi}=T'\boldsymbol{\pi}.$$

By construction, $T'_{1,1}=T_{1,1}$, and $T'_{i,1}=0$ for $i=2, 3, \cdots, L$; that is, T' is of the form

$$T'=\begin{bmatrix} T_{1,1} & T_{1,2} & \cdots & T_{1,L} \\ 0 & & & \\ 0 & & T^{(1)} & \\ \vdots & & & \\ 0 & & & \end{bmatrix}.$$

We now permute the rows of $T^{(1)}$ (and the last $n-1$ coordinates of $S_0\mathbf{z}$) so that def $(T_1^{(1)}) \geqq$ def $(T_2^{(1)}) \geqq \cdots \geqq$ def $(T_{L-1}^{(1)})$. We can now repeat the process and define the $F(\lambda)$-matrix $S^{(1)}$, where

$$S^{(1)}=\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & t'_3 & 1 & 0 & \cdots & 0 \\ 0 & t'_4 & 0 & 1 & \cdots & 0 \\ \vdots & & & & & \\ 0 & t'_L & 0 & 0 & \cdots & 1 \end{bmatrix},$$

etc. Because T was nonsingular, we see that there exists a nonsingular $F(\lambda)$-matrix $\tilde{S}$ so that:

(i) $\tilde{S}T=\tilde{T}$ is upper triangular,
(ii) def $(\tilde{S})=0$,
(iii) $\tilde{S}_0=cf_0(\tilde{S})$ is nonsingular,
(iv) $\tilde{S}_0\mathbf{z} =^{\lambda} \tilde{T}\boldsymbol{\pi}$.

Let $\mathbf{f}^T=(f_{i,k}^{(r)})$ be the last row of $\tilde{S}_0$. Because of (i) and (iv) we have $\mathbf{f}^T\mathbf{z} =^{\lambda} \tilde{T}_{L,L}(M_LN_L)$, and therefore Rk_λ $(\mathbf{f}^T\mathbf{z})=1$. By Fact 6.3, $f_{i,k}^{(r)}=0$ for all $r=1, 2, \cdots, N$ except $r=r_0$. With no loss of generality let $r_0=N$; then Fact 6.3 implies that $n_N=1$. Moreover, $\mathbf{f}^T\mathbf{z}$ must be of the form $(\sum_{i=1}^{m_N} a_ix_{i,1}^{(N)})$ $(\sum_{k=1}^{p_N} b_ky_{1,k}^{(N)})$, for some $a_i \in F$, $b_k \in F$, $1 \leqq i \leqq m_N$, $1 \leqq k \leqq p_N$. With no loss of generality assume that def $(M_L)=$ def $(N_L)=0$, and that $cf_0(M_L)=\sum_{i=1}^{m_N} a_ix_{i,1}^{(N)}$, and $cf_0(N_L)=\sum_{k=1}^{p_N} b_ky_{1,k}^{(N)}$.

We will now show that the assumption $L=t_a=t_b=t_c$ implies that $m_N=p_N=1$. Assume $m_N>1$, and with no loss of generality that $a=a_{m_N}\neq 0$. Let $M_L=\sum_{i=1}^{m_N-1} a_ix_{i,1}+ax_{m_N,1}+h$, where def $(h)<0$. We now substitute $t(\mathbf{x};\lambda)=-1/a(\sum_{i=1}^{m_N-1} a_ix_{i,1}+h)$ for $x_{m_N,1}$ in the algorithm $\mathfrak{a}$ to obtain an algorithm $\mathfrak{a}'$ satisfying $\mu(\mathfrak{a}') \leqq L-1$. Because def $(t(\mathbf{x};\lambda)) \leqq 0$ we see that the algorithm $\mathfrak{a}'$ can λ-compute all the $z_{i,k}^{(r)}$, $r=1, 2, \cdots, N-1$, as well as all the $z_{i,k}^{(N)}$, $i=1, 2, \cdots, m_N-1$, $k=1, 2, \cdots,$

p_N. This new system $\mathscr{B}'$ has $t'_a = t_a - 1$, $t'_b = t_b$, $t'_c = t_c - p_N$. By Fact 6.1 $L - 1 \geqq \mu(\mathfrak{a}') \geqq \mathrm{Rk}_\lambda(\mathscr{B}') \geqq t'_b = t_b$, so $L \geqq t_b + 1$, contradicting our assumption that $L = t_b$. Similarly we obtain that $p_N = 1$. Thus we have shown that $m_N = n_N = p_N = 1$.

If we now effect the same substitution $x_{m_N,1} = t(\mathbf{x}; \lambda) = -h/a$ we see that $\mathfrak{a}'$ λ-computes $\mathscr{B}' = \bigoplus_{i=1}^{N-1} \langle m_i, n_i, p_i \rangle$, and therefore $L - 1 \geqq \mu(\mathfrak{a}') \geqq \mathrm{Rk}_\lambda(\mathscr{B}') = t'_a = t'_b = t'_c = L - 1$. By induction hypothesis we also have $m_i = n_i = p_i = 1$ for all $i = 1, 2, \cdots, N-1$. This proves the theorem. □

Remark 6.2. The same argument as in the proof of Theorem 6.1 shows that if $\mathrm{Rk}_\lambda(\mathscr{B}) = t_b = t_c$ then $m_i = n_i = 1$ for all $i = 1, 2, \cdots, N$.

COROLLARY 6.1. *Let* $\mathscr{B} = \bigoplus_{i=1}^{N} \langle m_i, n_i, p_i \rangle$. *If* $t_a = t_b = t_c > N$ *then* $\mathrm{Rk}_\lambda(\mathscr{B}) > t_a$.

The next theorem, Theorem C, is a generalization of Theorem A of § 3. In the statement of Theorem C and the lemma preceding it, we will employ the same terminology as in Theorem A.

LEMMA 6.1. *Let* $\mathfrak{a}$ *be an algorithm which can* λ-*compute* $\mathscr{B} \oplus \langle 1, R, 1 \rangle$ *so that* $\langle 1, R, 1 \rangle$ *is full and isolated. Let* $\mathfrak{a}'$ *be an algorithm which can* λ-*compute* $\mathscr{B}'$, *and let* $L' = \mu(\mathfrak{a}')$. *Let* C' *be an* $L' \times L'$ *nonsingular diagonal* $F(\lambda)$ *matrix so that* $\mathrm{Rk}\,(\alpha'^T C' \beta') = \rho'$. *If* $\rho' \leqq R$ *then there exists an algorithm* $\tilde{\mathfrak{a}}$ *which can* λ-*compute* $\mathscr{B} \oplus \mathscr{B}' \bigoplus \langle 1, R', 1 \rangle$ *so that*:

(i) $\mu(\tilde{\mathfrak{a}}) = \mu(\mathfrak{a}) + \mu(\mathfrak{a}')$,

(ii) $R' = \mu(\tilde{\mathfrak{a}}) - r - s$, *where* r *is the number of* x_i'*s in* $\mathscr{B} \oplus \mathscr{B}'$ *and* s *is the number of* y_i'*s in* $\mathscr{B} \oplus \mathscr{B}'$,

(iii) $\langle 1, R', 1 \rangle$ *is full and isolated.*

Proof. Let $\mathfrak{a} = (M_1, M_2, \cdots, M_L; N_1, N_2, \cdots, N_L)$, $L = \mu(\mathfrak{a})$. Let $\mathfrak{a}' = (M'_1, M'_2, \cdots, M'_{L'}; N'_1, N'_2, \cdots, N'_{L'})$. By assumption there exist $c_1, c_2, \cdots, c_L \in F(\lambda)$, $c_i \neq 0$, $1 \leqq i \leqq L$, so that $\sum_{i=1}^{L} c_i M_i N_i = \sum_{i=1}^{R} u_i v_i$. Also, by assumption, $\sum_{i=1}^{L'} c'_i M'_i N'_i = \sum_{i=1}^{\rho'} \hat{M}_i(\mathbf{x}; \lambda) \hat{N}_i(\mathbf{y}; \lambda)$, where c'_i is the (i, i) term of C'. We may also assume that $\mathrm{def}\,(\hat{M}_i) \leqq 0$, $\mathrm{def}\,(\hat{N}_i) \leqq 0$, $1 \leqq i \leqq \rho'$, for otherwise we can replace c' by hc', with $0 \neq h \in F(\lambda)$ so that $\mathrm{def}\,(h)$ is small enough.

Let $\mathfrak{a}_1 = (\bar{M}_1, \bar{M}_2, \cdots, \bar{M}_L; \bar{N}_1, \bar{N}_2, \cdots, \bar{N}_L)$ be the algorithm obtained from $\mathfrak{a}$ by replacing u_i by $-\hat{M}_i$ and v_i by $\hat{N}_i$, for all $i = 1, 2, \cdots, \rho'$. Let $\mathfrak{a}_2 = \mathfrak{a}_1 + \mathfrak{a}' = (\bar{M}_1, \bar{M}_2, \cdots, \bar{M}_L, M'_1, M'_2, \cdots, M'_{L'}; \bar{N}_1, \bar{N}_2, \cdots, \bar{N}_L, N'_1, N'_2, \cdots, N'_{L'})$ be the algorithm which "concatenates" $\mathfrak{a}_1$ and $\mathfrak{a}'$. By construction $\sum_{i=1}^{L} c_i \bar{M}_i \bar{N}_i + \sum_{i=1}^{L'} c'_i M'_i N'_i = \sum_{i=\rho'+1}^{R} u_i v_i$, so $\mathfrak{a}_2$ can λ-compute $\mathscr{B} \oplus \mathscr{B}' \oplus \langle 1, R_1, 1 \rangle$ where $R_1 = R - \rho' \geqq 0$ and $\langle 1, R_1, 1 \rangle$ is full and isolated. Theorem B, applied to $\mathfrak{a}_2$, guarantees the existence of $\tilde{\mathfrak{a}}$ with the desired properties. □

THEOREM C. *Let* $\mathfrak{a}$ *be an algorithm which can* λ-*compute* $\mathscr{B}$, *where* $\mu(\mathfrak{a}) = L$. *Let* $\mathfrak{a}'$ *be an algorithm which can* λ-*compute* $\mathscr{B}'$, *where* $\mu(\mathfrak{a}') = L'$. *Let* C *be a nonsingular diagonal* $L \times L$ $F(\lambda)$-*matrix so that* $\mathrm{Rk}\,(\alpha^T C \beta) = \rho$. *Let* C' *be a nonsingular diagonal* $L' \times L'$ $F(\lambda)$-*matrix so that* $\mathrm{Rk}\,(\alpha'^T C' \beta') = \rho'$. *Suppose* $L + \rho - r - s \geqq \rho'$, *where* r *and* s *are the number of* x_i'*s and* y_i'*s, respectively, in* $\mathscr{B}$, *and that* $L' - r' - s' \geqq 0$. *Then for every* $k > 0$ *there exists an algorithm* $\tilde{\mathfrak{a}}_k$ *which can* λ-*compute* $\mathscr{B} \oplus k \cdot \mathscr{B}' \oplus \langle 1, R_k, 1 \rangle$ *so that*:

(i) $\mu(\tilde{\mathfrak{a}}_k) = L + kL' + \rho$,

(ii) $R_k = L + \rho - r - s + k(L' - r' - s')$,

(iii) $\langle 1, R_k, 1 \rangle$ *is full and isolated.*

Proof. By Theorem A there exists an algorithm $\mathfrak{a}_1$ which λ-computes $\mathscr{B} \oplus \langle 1, R', 1 \rangle$ satisfying:

(a) $\mu(\mathfrak{a}_1) = L + \rho$,

(b) $R' = L + \rho - r - s$,

(c) $\langle 1, R', 1 \rangle$ is full and isolated.

Apply Lemma 6.1 to $\mathfrak{a}_1$ and $\mathfrak{a}'$ to obtain $\tilde{\mathfrak{a}}_1$. Now proceed by induction on $k > 1$.

By induction hypothesis there exists an algorithm $\tilde{\mathfrak{a}}_{k-1}$ satisfying (i), (ii) and (iii) of the statement of the theorem. Apply Lemma 6.1 to $\tilde{\mathfrak{a}}_{k-1}$ and $\mathfrak{a}'$ to obtain $\tilde{\mathfrak{a}}_k$. □

Let $\mathfrak{a}$, $\mu(\mathfrak{a}) = L$ be an algorithm which can λ-compute a system of bilinear forms $\mathscr{B}$. We define $\rho(\mathfrak{a})$ to be $\rho(\mathfrak{a}) = \min \mathrm{Rk}\,(\alpha^T C\beta)$, where the minimization is done over all possible nonsingular $L \times L$ diagonal $F(\lambda)$-matrices C. We also define $\rho(\mathscr{B})$ to be $\min \rho(\mathfrak{a})$, where the minimization is over all algorithms which can λ-compute $\mathscr{B}$ and satisfy $\mu(\mathfrak{a}) = \mathrm{Rk}_\lambda(\mathscr{B})$. It should be observed that $\rho(\mathfrak{a})$, and therefore $\rho(\mathscr{B})$, cannot exceed r = the number of x_is of $\mathscr{B}$, or s = the number of y_js of $\mathscr{B}$. With this terminology established, we have the following consequences of Theorem C.

COROLLARY 6.2. *Let $\mathscr{B}$ be a system of bilinear forms, and let $\mathfrak{a}$ be an algorithm which λ-computes $\mathscr{B}$. If $\mu(\mathfrak{a}) \geqq r+s$ then for every k there exists an algorithm $\mathfrak{a}_k$, $\mu(\mathfrak{a}_k) = k\mu(\mathfrak{a}) + \rho(\mathfrak{a})$, which can λ-compute $k \cdot \mathscr{B} \oplus \langle 1, R, 1\rangle$, where $R = \rho(\mathfrak{a}) + k(\mu(\mathfrak{a}) - r - s)$, so that $\langle 1, R, 1\rangle$ is full and isolated.*

Proof. Choose C so that $\mathrm{Rk}\,(\alpha^T C\beta) = \rho(\mathfrak{a})$, and let $\mathscr{B}' = \mathscr{B}$, $\mathfrak{a}' = \mathfrak{a}$, $C' = C$. Then $\rho(\mathfrak{a}) + \mu(\mathfrak{a}) - r - s \geqq \rho(\mathfrak{a})$. Apply Theorem C. □

In particular, if $\mathfrak{a}$ of Corollary 6.2 satisfies $\mu(\mathfrak{a}) = \mathrm{Rk}_\lambda(\mathscr{B})$ and $\rho(\mathfrak{a}) = \rho(\mathscr{B})$ then we obtain:

COROLLARY 6.3. *If $\mathrm{Rk}_\lambda(\mathscr{B}) \geqq r+s$ then for every k there exists an algorithm $\mathfrak{a}_k$, $\mu(\mathfrak{a}_k) = k\,\mathrm{Rk}_\lambda(\mathscr{B}) + \rho(\mathscr{B})$, which can λ-compute $k \cdot \mathscr{B} \oplus \langle 1, R, 1\rangle$, $R = \rho(\mathscr{B}) + k(\mathrm{Rk}_\lambda(\mathscr{B}) - r = s)$, so that $\langle 1, R, 1\rangle$ is full and isolated.*

If we apply Corollaries 6.2 and 6.3 to $\mathscr{B} = \bigoplus_{i=1}^N \langle m_i, n_i, p_i\rangle$ we obtain:

COROLLARY 6.4. *Let $\mathscr{B} = \bigoplus_{i=1}^N \langle m_i, n_i, p_i\rangle$, and let $\mathfrak{a}$ be an algorithm which can λ-compute $\mathscr{B}$. If $\mu(\mathfrak{a}) > t_a + t_b$ then $\omega < 3\tau$, where τ is the root of the associated equation $\mu(\mathfrak{a}) = \sum_{i=1}^N (m_i n_i p_i)^\tau$.*

Proof. Let $\mu(\mathfrak{a}) - t_a - t_b = l \geqq 1$. Choose k so large that $(\rho(\mathfrak{a}) + kl)^\tau > \rho(\mathfrak{a})$. By Corollary 6.2 there exists $\mathfrak{a}_k$, $\mu(\mathfrak{a}_k) = k\mu(\mathfrak{a}) + \rho(\mathfrak{a})$, which can λ-compute $k \cdot \bigoplus_{i=1}^N \langle m_i, n_i, p_i\rangle \oplus \langle 1, \rho(\mathfrak{a}) + kl, 1\rangle$. By construction $k \sum_{i=1}^N (m_i n_i p_i)^\tau + (\rho(\mathfrak{a}) + kl)^\tau > k\mu(\mathfrak{a}) + \rho(\mathfrak{a})$, so τ', the root of the equation associated with $\mathfrak{a}_k$, satisfies $\tau' < \tau$. Therefore $\omega \leqq 3\tau' < 3\tau$. □

In particular, if we choose $\mathfrak{a}$ of Corollary 6.4 so that $\mu(\mathfrak{a}) = \mathrm{Rk}_\lambda(\mathscr{B})$, $\rho(\mathfrak{a}) = \rho(\mathscr{B})$ we obtain:

COROLLARY 6.5. *Let $\mathscr{B} = \bigoplus_{i=1}^N \langle m_i, n_i, p_i\rangle$. If $\mathrm{Rk}_\lambda(\mathscr{B}) > t_a + t_b$ then $\omega < 3\tau$ where τ is the root of the associated equation $\mathrm{Rk}_\lambda(\mathscr{B}) = \sum_{i=1}^N (m_i n_i p_i)^\tau$.*

COROLLARY 6.6. *Let $\mathscr{B} = \bigoplus_{i=1}^N \langle m_i, n_i, p_i\rangle$, and let $\mathfrak{a}$ be an algorithm which can λ-compute $\mathscr{B}$. If $\mu(\mathfrak{a}) > \max(t_a, t_b)$ then $\omega < 3\tau$, where τ is the root of the associated equation $\mu(\mathfrak{a}) = \sum_{i=1}^N (m_i n_i p_i)^\tau$.*

Proof. Let $\mathscr{B}' = \mathscr{B} \otimes \mathscr{B} \otimes \cdots \otimes \mathscr{B}$ be the k-fold tensor product of $\mathscr{B}$ with itself. $\mathscr{B}'$ can be λ-computed by $\mathfrak{a}' = \mathfrak{a} \otimes \mathfrak{a} \otimes \cdots \otimes \mathfrak{a}$, where $\mu(\mathfrak{a}') = \mu(\mathfrak{a})^k$. $\mathscr{B}'$ also satisfies $t'_a = t_a^k$, $t'_b = t_b^k$. The equation associated with $\mathscr{B}'$ (and $\mathfrak{a}'$) is $\mu(\mathfrak{a})^k = (\sum_{i=1}^N (m_i n_i p_i)^\tau)^k$ so it has the same root as the equation associated with $\mathscr{B}$ (and $\mathfrak{a}$). Let k be large enough that $\mu(\mathfrak{a}') > t'_a + t'_b$. Apply Corollary 6.4 to $\mathscr{B}'$ and $\mathfrak{a}'$. □

COROLLARY 6.7. *Let $\mathscr{B} = \bigoplus_{i=1}^N \langle m_i, n_i, p_i\rangle$. If $\mathrm{Rk}_\lambda(\mathscr{B}) > \max(t_a, t_b)$ then $\omega < 3\tau$, where τ is the root of the associated equation $\mathrm{Rk}_\lambda(\mathscr{B}) = \sum_{i=1}^N (m_i n_i p_i)^\tau$.*

Our final result combines Theorem 6.1 (more precisely Corollary 6.1) and Theorem C (more precisely Corollary 6.6).

THEOREM D. *Let $\mathscr{B} = \bigoplus_{i=1}^N \langle m_i, n_i, p_i\rangle$, and let $\mathfrak{a}$ be an algorithm which can λ-compute $\mathscr{B}$. If $\sum_{i=1}^N (m_i n_i p_i) > N$ then $\omega < 3\tau$, where τ is the root of the associated equation $\mu(\mathfrak{a}) = \sum_{i=1}^N (m_i n_i p_i)^\tau$.*

Proof. By Fact 2.6 there exists an algorithm $\mathfrak{a}'$ which can λ-compute $\mathscr{B}' = \bigoplus_{i=1}^N \langle n_i, p_i, m_i\rangle$ so that $\mu(\mathfrak{a}') = \mu(\mathfrak{a})$, and there also exists an algorithm $\mathfrak{a}''$ which can λ-compute $\mathscr{B}'' = \bigoplus_{i=1}^N \langle p_i, m_i, n_i\rangle$ so that $\mu(\mathfrak{a}'') = \mu(\mathfrak{a}') = \mu(\mathfrak{a})$. Then the algorithm $\tilde{\mathfrak{a}} =$

$\mathfrak{a}+\mathfrak{a}'+\mathfrak{a}''$ can λ-compute $\tilde{\mathscr{B}}=\mathscr{B}\oplus\mathscr{B}'\oplus\mathscr{B}''$. But $\tilde{\mathscr{B}}$ satisfies $\tilde{t}_a=\tilde{t}_b=\tilde{t}_c=t_a+t_b+t_c$. So $\tilde{\mathscr{B}}$ satisfies the condition of Corollary 6.1, and therefore $\mu(\tilde{\mathfrak{a}})\geqq \mathrm{Rk}_\lambda(\tilde{\mathscr{B}})>t_a=t_b$. But the root of the equation associated with $\tilde{\mathscr{B}}$ (and $\tilde{\mathfrak{a}}$) is the same as the root of the equation associated with $\mathscr{B}$ (and $\mathfrak{a}$). Apply Corollary 6.6 to $\tilde{\mathscr{B}}$ and $\tilde{\mathfrak{a}}$. □

COROLLARY 6.8. *Let* $\mathscr{B}=\bigoplus_{i=1}^N \langle m_i, n_i, p_i\rangle$. *If* $\sum_{i=1}^N (m_i, n_i, p_i)>N$ *then* $\omega<3\tau$, *where* τ *is the root of the associated equation* $\mathrm{Rk}_\lambda(\mathscr{B})=\sum_{i=1}^N (m_in_ip_i)^\tau$.

7. Conclusions. The main results of this paper, Theorems A and B, provide us with tools for generating new, and improved, algorithms for old ones. However, we do not have good estimates, except by direct numerical calculations, for the amount of improvement. The examples given in the paper indicate that after a few iterations of Theorem B, the improvement is marginal. We would like to have a "closed-form" formula of the limit of, say, iterating the construction of Corollary 4.1, perhaps an equation whose root is the limit. We leave the investigation of this problem to the reader.

The generalized tensor product construction provided us with further improvement of our estimate of ω. This improvement was small, but we hope that this construction is of interest in and by itself, and could help us in future investigation of the asymptotic complexity of matrix multiplication.

The results which we obtained were by either adding a direct summand $\langle 1, R, 1\rangle$ (Theorem A), or by increasing the value of R (Theorem B). We believe that better results would have been obtained had we succeeded in adding, or improving, more than one direct summand. Another avenue for continuing the line of investigation of this paper, is replacing $\langle 1, R, 1\rangle$ by the more general tensor $\langle R_1, R_2, R_3\rangle$. We leave these suggestions as a challenge to the reader.

The results of the last section (§ 6) show that no algorithm for λ-computation of a direct sum of matrix multiplications can yield a τ so that $\omega=3\tau$. This strongly suggests that the only way of determining ω is by exhibiting an infinite scheme of matrix multiplication algorithms. We leave that, too, as a challenge to the reader.

REFERENCES

[1] V. YA. PAN, *On schemata for the computation of matrix products and for matrix inversion*, Uspekhi Mat. Nauk, XXVII, 5 (167) (1972), pp. 249–250 (in Russian).

[2] A. BORODIN AND I. MUNRO, *The Computational Complexity of Algebraic and Numeric Problems*, American Elsevier, New York, 1975.

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

[4] V. YA. PAN, *Strassen's algorithm is not optimal*, Proc. 19th Annual ACM Symposium on the Foundation of Computer Science, 1978, pp. 166–176.

[5] D. BINI, G. LOTTI AND F. ROMANI, *Suboptimal solutions for the bilinear forms computational problem*, Nota Interna B78-26, November 1978, Istituto di Elaborazione della Informazione, Pisa.

[6] D. BINI, *Relations between exact and approximate bilinear algorithms, applications*, Calcolo, 17 (1980), pp. 87–97.

[7] A. SCHÖNHAGE, *Partial and total matrix multiplication*, this Journal, 10 (1981), pp. 434–455.

[8] V. STRASSEN, *Vermeidung von divisionen*, J. Reine Angew. Math., 264 (1973), pp. 184–202.

[9] V. YA. PAN, *New combinations of methods for the acceleration of matrix multiplication*, Comput. Math. with Appl., 7 (1981), pp. 73–125.

[10] B. L. VAN DER WAERDEN, *Algebra*, vol. 1, 7th ed., Frederic Ungar, New York, 1970.

[11] J. E. HOPCROFT AND J. MUSINSKI, *Duality applied to the complexity of matrix multiplications and other bilinear forms*, this Journal, 2 (1973), pp. 159–173.

[12] L. C. LAFON AND S. WINOGRAD, *A lower bound for the multiplicative complexity of the product of two matrices*, Centre de Calcul de L'Esplanade, U.E.R. de Mathematique, Univ. Louis Pasteur, Strasbourg, France, October 1979.

SIAM J. COMPUT.
Vol. 11, No. 3, August 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1103-0008 $01.00/0

A LUBRICANT FOR DATA FLOW ANALYSIS*

BARRY K. ROSEN†

Abstract. This paper deals with concepts recently introduced to avoid duplication of effort when several global data flow problems share the same underlying control flow. The *flow scheme* is the control flow portion of the input to data flow analysis. By finding a family of formal expressions called a *flow cover* for a given flow scheme, one can be ready to solve any problem with that scheme by interpreting the expressions in light of the remaining portions of the problem. Our main result relates the problem of finding a flow cover to that of finding regular expressions to describe certain sets of paths in the graph. The following is a major consequence of this relationship (slightly oversimplified). If a family of formal expressions acts like a flow cover for one special problem constructed from a given flow scheme, then it is a flow cover. The constructed problem has several algebraic properties that fail in some of the problems encountered in practice but that may be assumed anyway, at least when one wishes to construct flow covers. Applications include updating a given flow cover in light of a small change in the flow scheme, and improving upon the folkloric way to cope with multiple entry nodes.

Key words. data flow analysis, optimizing compilers, control flow, regular expression, lattice

1. Introduction. Because our main result removes friction by letting us assume convenient things without loss of generality, we call it a *lubricant*, and specifically a *wolog* lubricant. A whimsical but instructive example of wolog lubrication will be considered before the technicalities begin. In designing a trap to catch one specific bear, we can use whatever we know about that bear's habits. Park rangers would have much less trouble protecting bears and people from each other, if they could apply a result of the following form:

THEOREM 1.1. *In every national park there is a certain bear, called the canonical bear, such that any trap effective for this one bear is effective for any bear in the park.*

To certify a bear trap, we can assume without loss of generality that the bear to be trapped is canonical. Because they have unusually regular habits, canonical bears are relatively easy to trap. A trap that is certain to catch the canonical bear of Yellowstone Park is also certain to catch any other bear in Yellowstone Park, even if the other bear's habits are different in ways that seem to invalidate the argument certifying the trap. As a principle of wildlife management, Theorem 1.1 is absurd. Yet it becomes true when "national park" and "bear" and "trap" are replaced by appropriate technical terms from data flow analysis. After stating the theorem we will outline what the terms mean.

THEOREM 1.2. *For every flow scheme* $(G, \mathbf{E})$ *there is a certain* 3*-way algebraic context* (L, M)*, called the canonical algebraic context, such that any reduced flow cover for* $(G, \mathbf{E})$ *relative to the class* $\{(L, M)\}$ *is also a reduced flow cover for* $(G, \mathbf{E})$ *relative to the class of all* 3*-way algebraic contexts.*

It takes three ordered pairs to specify a data flow analysis problem. The *flow scheme* $(G, \mathbf{E})$ is a finite directed graph G together with a set $\mathbf{E}$ of nodes in G designated the *entry* nodes. (Flow schemes under various constraints not imposed here are often called "program flowgraphs" and the like.) The *algebraic context* (L, M) is a set L together with a set M of maps $U: L \rightarrow L$, under more or less standard assumptions. (Details are in § 2.) For the present, members of L may be thought of as "assertions" in a special language. The algebraic context of a problem specifies what kind of information is sought (with L) and what kinds of changes in this

* Received by the editors October 14, 1980, and in final form September 15, 1981.

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

information can be readily computed (with M). The third pair, which is not mentioned in Theorem 1.2, links the first two pairs. The *local and entry information* (f, E) is a map f: (arcs of G) $\to M$ together with a map $E: \mathbf{E} \to L$. When control enters the program at m in $\mathbf{E}$, we know that the assertion[1] Em is true. When control flows along an arc c from n to p and x in L is true at n, then we know that $(fc)x$ is true at p. From all this information we try to compute *global information*: a map I: (nodes of G) $\to L$ such that Ip is true whenever control reaches p.

In practice it is very common for several problems to share the same underlying control flow. To avoid duplication of effort in solving such a family of problems, [7, § 2] introduced the other concepts involved in Theorem 1.2. A *flow expression* for a flow scheme $(G, \mathbf{E})$ is a formal expression X that will have a value $[X]$ in M after the algebraic context (L, M) and the local information f have been specified. The specific syntax of flow expressions here departs from [7] along lines suggested by [8], [9]. For the moment, all that matters is the fact that evaluation of these expressions presupposes an algebraic structure involving three operations on M. Flow expressions are only meaningful when the contexts of interest[2] all have these operations. Because three ways to operate on M are required, we speak of 3-*way* contexts.

A *flow cover* is a family of flow expressions whose values can be used to solve flow problems. For each entry node m and each node p there is an expression $X(m, p)$ with value $[X(m, p)]$ in M depending on the problem at hand. Applying this value to the entry assertion Em in L yields $[X(m, p)]Em$ in L as what the flow cover propagates to p from entrance at m, with all paths from m to p in G accounted for. Combining the values $[X(m, p)]Em$ for all m in $\mathbf{E}$ in a suitable way yields the desired global information Ip in L. Some of the uses of flow covers appear in [7]; others will appear later in this paper.

It is often useful to consider flow covers *relative* to some special class of contexts (L, M) or to some special class of local and entry information pairs (f, E). Only for problems with inputs in the special classes does a relative flow cover claim to work. For example, *idempotent* contexts have $U \circ U = U$ for all U in M, where $\circ$ is ordinary composition of maps. The traditional "bit vector" data flow problems all have idempotent contexts, and idempotence is heavily exploited in the literature on such problems. To exploit idempotence when constructing flow covers, we can replace subexpressions of the form $X \cdot X$ by the corresponding subexpressions of the form X. (Here, X is a flow expression and $\cdot$ is a formal operator that will later be interpreted as composition of maps.) Applying this rule to a flow cover yields a simpler family of expressions, but one that is only a flow cover relative to the class of idempotent contexts.

A *reduced* flow cover is one whose flow expressions have all been simplified as much as possible by applications of rules from some given set of rules. Theorem 1.2's use of the word refers to nine rules to be stated in § 2. Like the idempotence rule $X \cdot X \to X$, each of our rules removes an operator from an expression. Ordinarily, application of a rule restricts a flow cover to contexts where the corresponding algebraic equation holds, as when $U \circ U = U$ justifies use of the rule $X \cdot X \to X$. Eight of our rules are universally valid, but one of them fails in some important contexts. Nevertheless, we may use this rule without loss of generality when we construct flow covers. It is valid in the canonical context, and that suffices by Theorem 1.2. (Other algebraic

[1] The value of the map E at the argument m is written Em rather than $E(m)$, with parentheses reserved for grouping. Thus $(fc)x$ indicates that f is applied to c, yielding a map which is then applied to x.

[2] As is explained in [7, § 1], the contexts for which flow expressions are meaningful include all those for which elimination algorithms may be used.

properties of the canonical context can be similarly exploited.) The canonical bears of Theorem 1.1 are relatively easy to trap because they have regular habits. The canonical contexts of Theorem 1.2 are relatively easy to cover because they involve regular sets of strings. Readers of [7] may have already noticed that constructing a flow cover for a given flow scheme is somewhat like constructing a regular expression for the language accepted by a given finite automaton. The analogy is much closer than the author could imagine at the time [7] was written, without the benefit of later work here and in [8].

Some important concepts from data flow analysis have been reviewed in discussing the main result. Proving it will require more detail, to appear in § 2. For a given flow scheme $(G, \mathbf{E})$, § 3 constructs the canonical context and establishes some preliminary lemmas. Given $(G, \mathbf{E})$ and a family of reduced flow expressions that is a flow cover relative to the canonical context, § 4 proves Theorem 1.2 by showing that the family is indeed a reduced flow cover relative to all 3-way contexts. In the process a stronger result emerges; this is stated as Theorem 4.4. Devised independently some months after the April 1979 report that became [8], our proof is much like that of a narrower result [8, Thm. 6] in the same spirit. The heart of Theorem 4.4 is the fact that any reduced "path cover" is a flow cover, where a path cover is a family of flow expressions that behaves nicely when the expressions are interpreted as sets of strings of arcs. Specifically, the set of strings of arcs associated with $X(m, p)$ should be the set of all paths from m to p in G. If this holds for all choices of m in $\mathbf{E}$ and p in $\mathbf{N}_G$, then the family of flow expressions is a *path cover* for $(G, \mathbf{E})$. Data flow analysis is one of several applications for path covers; others appear in [8]. Section 5 explains some applications of Theorem 4.4, especially to the problem of updating a given path cover to reflect a small change in a flow scheme. Section 6 considers finite approximations to the information in the canonical context and shows that this context is equivalent to an infinite family of contexts, each with L finite. Finally, the historical remarks in § 7 relate this paper to some folklore and to [3], [7], [8], [9].

2. Background. Throughout this paper a *graph* is a finite directed graph. A graph G has a set $\mathbf{N}_G$ of *nodes* and a set $\mathbf{A}_G$ of *arcs*. Each arc c runs from a node n to a node p. Certain strings (i.e., finite sequences) of arcs are *paths*. The null string λ is a path from any node to itself. A nonnull string $\mathbf{c} = (c_1 \cdots c_r)$ for $r > 0$ is a path if and only if c_i is an arc to the same node that c_{i+1} is from, for all i with $1 \leqq i \leqq r-1$. If c_1 runs from n while c_r runs to p, then $\mathbf{c}$ is a path from n to p and we write $\mathbf{c}: n \to p$. Suppose a map $fc: L \to L$ has been associated with each arc c in G. Then f is extended to paths in the obvious way:

$$f\mathbf{c} = (fc_r) \circ \cdots \circ (fc_1) \quad \text{for } \mathbf{c} = (c_1 \cdots c_r). \tag{2.1}$$

In particular $f\lambda = \mathbf{1}_L$, where the identity map $\mathbf{1}_L$ takes everything in L to itself. Many sets L are important in data flow analysis. The properties of L presupposed by the usual algorithms are discussed in [7, § 1], which should be consulted for motivation and historical remarks. Here we will briefly review some key points and introduce some simplifications. The intuition in [7, § 1] is more or less standard, but no two authors use quite the same machinery. (Earlier formulations like [4], [6] are subsumed under [7].)

An *algebraic context* for data flow analysis is a pair (L, M), where L is a complete lattice and M is a set of isotone maps $U: L \to L$. The partial order on L is denoted $\leqq$, the top of L is denoted $\top_L$, and the greatest lower bound operation (mapping subsets of L to L) is denoted $\bigwedge$. (As usual, we abbreviate $\bigwedge\{x, y\}$ to $x \wedge y$.) The set

M_{all} of all isotone maps from L to L is also a complete lattice, and similar notation is used. Thus $\bigwedge$ and $\wedge$ denote operations on M_{all} also. What have come to be called *elimination* algorithms presuppose that M is closed under some of the operations on M_{all}. The *closed* contexts of [7] satisfy

(2.2.1) M contains $\mathbf{1}_L$ and the constant $\top_M$ with value $\top_L$,

(2.2.2) M is closed under $\circ$ and $\wedge$ as operations on M_{all}.

A third operation @ on M is also presupposed. This operation need not be defined on M_{all}, but its values must be bounded above by those of an operation $@_{\text{all}}$ defined on M_{all}. Let $\boldsymbol{N}$ be the set $\{0, 1, 2, 3, \cdots\}$. For U, V in M_{all} let $V @_{\text{all}} U$ in M_{all} be defined by

$$V @_{\text{all}} U = \bigwedge \{V \circ U^r \mid r \in \boldsymbol{N}\}. \tag{2.3.1}$$

The set M is required to have a binary operation $@: M \times M \to M$ such that

$$V @ U \leqq V @_{\text{all}} U \quad \text{for all } U, V \text{ in } M. \tag{2.3.2}$$

For all x in L the value $(V @ U)x$ is bounded above by the preceding condition; for some x in L the value $(V @ U)x$ is bounded below by the following condition:

$$(\forall x \in L)(x \leqq Ux \text{ implies } Vx \leqq (V @ U)x). \tag{2.3.3}$$

A closed context with this third operation @ will be called a 3-*way* context here. What are called *rapid* closed contexts in [7] are 3-way contexts that satisfy additional conditions, to be discussed at the end of § 7. Rapid closed contexts seem to be the 3-way contexts that arise in practice,[3] but working within the broader class of all 3-way contexts is much more convenient.

A *strongly distributive map* $U: L \to L$ has $U(\bigwedge X) = \bigwedge \{Ux \mid x \in X\}$ for all nonempty $X \subseteq L$. A *strongly distributive context* (L, M) has each U in M strongly distributive, in which case an operation @ on M satisfies (2.3.2) and (2.3.3) if and only if it is the restriction of $@_{\text{all}}$ to the subset $M \times M$ of $M_{\text{all}} \times M_{\text{all}}$. (Details are in the proof of Lemma 2.8.) Therefore a strongly distributive closed context is a 3-way context if and only if M is closed under $@_{\text{all}}$. A great convenience when it holds, strong distributivity is known to fail in some important contexts.

Given a graph G, a *flow expression* for G is a regular expression over $\mathbf{A}_G$ considered as a set of symbols, with the understanding that the syntax of regular expressions has been slightly enriched for reasons to emerge later. Specifically, the simplest flow expressions are the *constants* ("ϕ" and "λ") and the *variables* (each arc c in $\mathbf{A}_G$). More complicated expressions are built by applying the usual *Kleene operators*: if Y and Z are flow expressions, then so are the formal concatenation $Y \cdot Z$, the formal union $Y \cup Z$, and the formal star Y^*. We add another binary operator $\otimes$ and rules

$$Y \otimes Z \to Y^* \cdot Z \ \& \ Y^* \to Y \otimes \lambda. \tag{2.4}$$

We could consider one of the operators $\otimes$ and * to be merely an abbreviation, defined by one of the rules (2.4), but it will be more convenient to admit the operators on an equal footing. Either one can be eliminated as the need arises.

---

[3] In practice, vectors have fewer than 10^{99} components. It is nevertheless unwise to write $N < 10^{99}$ into the definition of N-dimensional vector spaces. The situation here is analogous.

Once a 3-way algebraic context (L, M) and local information $f: \mathbf{A}_G \to M$ have been specified, any flow expression X has a value $[X]$ in M. Specifically,

(2.5.1) $$[\phi] = \top_M \;\&\; [\lambda] = \mathbf{1}_L \;\&\; [c] = fc, \quad \text{all } c \text{ in } \mathbf{A}_G,$$

(2.5.2) $$[Y \cdot Z] = [Z] \circ [Y] \;\&\; [Y \cup Z] = [Z] \wedge [Y],$$

(2.5.3) $$[Y^*] = \mathbf{1}_L @ [Y] \;\&\; [Y \otimes Z] = [Z] @ [Y].$$

As in [7], we will usually honor the role of f in evaluating flow expressions by writing $[X:f]$ rather than just $[X]$. This will resolve the ambiguity in "the value of" when more than one choice of local information is to be considered. Where there is no chance of confusion, the briefer $[X]$ will be used within proofs.

It is possible to have $V @ U \neq V \circ (\mathbf{1}_L @ U)$ and hence $[Y \otimes Z] \neq [Y^* \cdot Z]$. The first rule in (2.4) is therefore not universally valid for the calculations in data flow analysis. Use of invalid rules is rightly frowned upon in most situations, but here it is permissible. In the canonical context we will never have $V @ U \neq V \circ (\mathbf{1}_L @ U)$. Valid for the canonical context, the rules (2.4) may be used freely in constructing flow covers.

A flow expression is *reduced* if and only if none of the rules listed below are applicable to it, even after any uses of $\otimes$ have been replaced by * according to (2.4). The nine rules defining reduced flow expressions are conveniently divided into two groups. The first group consists of rules for simplifying expressions with the constant λ. There are four rules:

(2.6.1) $$Y \cdot \lambda \to Y \;\&\; \lambda \cdot Z \to Z \;\&\; \lambda \cup \lambda \to \lambda \;\&\; \lambda^* \to \lambda.$$

The second group consists of rules for simplifying expressions with the constant ϕ. For ϕ on the right of an operator symbol we have

(2.6.2) $$Y \cdot \phi \to \phi \;\&\; Y \cup \phi \to Y.$$

For ϕ on the left of an operator symbol we have

(2.6.3) $$[\phi \cdot Z \to \phi] \;\&\; \phi \cup Z \to Z \;\&\; \phi^* \to \lambda.$$

The bracketed rule is only valid in contexts with $U\top_L = \top_L$ for all U in M. This fails in the traditional bit vector contexts and many other naturally occurring contexts, but the rule may still be used in constructing flow covers. When all five rules for expressions with ϕ are used, we find that ϕ itself is the only reduced flow expression involving ϕ. Other advantages to using all five rules will emerge in due course.

LEMMA 2.7. *Let a nontrivial flow expression be one that contains variables. Every reduced flow expression is either ϕ or λ or nontrivial.*

Proof. By structural induction we show that any trivial flow expression X can be reduced to either ϕ or λ. This is immediate for X a constant or variable. For X formed by a binary operator, we have subexpressions Y and Z with $X = Y \boxtimes Z$. Note that Y and Z are both trivial and that $\boxtimes$ can be assumed to be either $\cdot$ or $\cup$. By the induction hypothesis, we can assume without loss of generality that Y is ϕ or λ. Likewise for Z. If Y is ϕ, then (2.6.3) reduces X to ϕ or Z and hence to ϕ or λ. Now suppose Y is λ. If $\boxtimes$ is $\cdot$, then (2.6.1) reduces X to Z and hence to ϕ or λ. If $\boxtimes$ is $\cup$, then (2.6.1) or (2.6.2) reduces X to λ. For X formed by the star operator, we have a subexpression Y with $X = Y^*$. By the induction hypothesis, we can assume without loss of generality that Y is ϕ or λ. Now (2.6.3) or (2.6.1) reduces X to λ. □

LEMMA 2.8. *Let (L, M) be any 3-way context and let $f: \mathbf{A}_G \to M$. Let X be a flow expression. Then the value $[X:f]$ can be approximated by values of X in 3-way contexts*

with isotone @ operations. Specifically, let $[X:_{\text{all}} f]$ be the value of X when f is considered as a map into M_{all} (the set of all isotone maps from L to L) and the @ operation on M_{all} is $@_{\text{all}}$ from (2.3.1). *Then $[X:f] \leqq [X:_{\text{all}} f]$, and they are equal if (L, M) is strongly distributive.*

Proof. Induction on the structure of X is used. Note that $\circ$ and $\wedge$ are isotone while (2.3.2) generalized to $V @ U \leqq V' @_{\text{all}} U'$ for all U, V in M and all U', V' in M_{all} with $U \leqq U'$ and $V \leqq V'$. In the strongly distributive case, it will suffice to show that

(1) $\quad$ @ is the restriction of $@_{\text{all}}$ to $M \times M$ in $M_{\text{all}} \times M_{\text{all}}$.

For U, V in M we will show that $V @ U = V @_{\text{all}} U$. Consider any z in L and let $x = \bigwedge \{U^r z \mid r \in \boldsymbol{N}\}$. Then strong distributivity implies

$$x \leqq \bigwedge \{U^r z \mid r \in \boldsymbol{N} - \{0\}\} = U(\bigwedge \{U^s z \mid s \in \boldsymbol{N}\}) = Ux.$$

Therefore $x \leqq Ux$ and (2.3.3) implies $Vx \leqq (V @ U)x \leqq (V @ U)z$ because $x \leqq z$ and $V @ U$ is isotone. But strong distributivity implies $(V @_{\text{all}} U)z = Vx$, so $(V @_{\text{all}} U)z \leqq (V @ U)z$. The reverse inequality holds by (2.3.2), so $V @ U = V @_{\text{all}} U$ and (1) holds. □

As in § 1, a flow scheme $(G, \mathbf{E})$ together with an algebraic context (L, M) and local information (f, E) are what we need to specify a data flow analysis problem. A problem is *solved* by any $I: \mathbf{N}_G \to L$ with values small enough for Ip to be a safe assertion whenever control reaches p [7, Def. 1.2]. The *maximum solution* (often called "MOP" for "meet over (all) paths") is of course

$$I_{\max} p = \bigwedge \{(f\mathbf{c})Em \mid m \in \mathbf{E} \;\&\; \mathbf{c}: m \to p \text{ in } G\},$$

but finding $I_{\max}$ is sometimes impossible [6, Thm. 7]. A *good* solution is one large enough to dominate any *fixpoint* [7, Def. 1.3]. One way to find a good solution is to find an appropriate family of flow expressions and then evaluate them. Rather than restate the definition [7, Def. 2.3] of *flow covers*, we will restate what they do for us.

LEMMA 2.9 [7, Lemma 2.4]. *Let $(G, \mathbf{E})$ and (L, M) and (f, E) specify a data flow problem, with (L, M) being a 3-way context. Suppose $(G, \mathbf{E})$ has a flow cover $\mathscr{X}$, assigning a flow expression $X(m, p)$ to each (m, p) in $\mathbf{E} \times \mathbf{N}_G$. Then a good solution is obtained by setting, for each node p in G,*

$$Ip = \bigwedge \{[X(m, p): f]Em \mid m \in \mathbf{E}\}. \quad \square$$

3. The canonical context. Throughout this section, a flow scheme $(G, \mathbf{E})$ is given. Let $\mathscr{L}$ be the set of all subsets of $\mathbf{A}_G^*$, the set of all strings (i.e., finite sequences) of arcs. The formula (2.1) for composing the maps encountered along a path makes sense for any string $\mathbf{c}$ in $\mathbf{A}_G^*$, even if $\mathbf{c}$ is not a path. Given local information $f: \mathbf{A}_G \to M$ in some algebraic context (L, M), we can map $\mathscr{L}$ into M_{all} (all isotone maps from L to L) by defining $\mu: \mathscr{L} \to M_{\text{all}}$ as follows:

$$\mu\xi = \bigwedge \{f\mathbf{c} \mid \mathbf{c} \in \xi\} \quad \text{for all } \xi \text{ in } \mathscr{L}. \tag{3.1}$$

Given a flow expression X, a 3-way context (L, M), and local information $f: \mathbf{A}_G \to M$, we can interpret X in M_{all} by considering $[X:f]$ in $M \subseteq M_{\text{all}}$. On the other hand, we can interpret X in M_{all} by an indirect route. Treating X like any other regular expression, we will define $X_{\mathscr{L}}$ in $\mathscr{L}$ and then compare $\mu X_{\mathscr{L}}$ with $[X:f]$. Unequal in general, they will be close enough for us.

The constants "ϕ" and "λ" are interpreted as the corresponding sets of strings:

$$\phi_{\mathscr{L}} = \phi \;\&\; \lambda_{\mathscr{L}} = \{\lambda\}. \tag{3.2.1}$$

Each variable c is interpreted as the corresponding set with one member, the string (c):

$$c_{\mathscr{L}} = \{(c)\}. \tag{3.2.2}$$

For $X = Y \mathbin{\square} Z$ with formal operator $\square$ in $\{\cdot, \cup, \otimes\}$ and for $X = Y^*$, we use the Kleene operations on $\mathscr{L}$:

$$(Y \cdot Z)_{\mathscr{L}} = Y_{\mathscr{L}} \cdot Z_{\mathscr{L}}, \tag{3.3.1}$$

$$(Y \cup Z)_{\mathscr{L}} = Y_{\mathscr{L}} \cup Z_{\mathscr{L}}, \tag{3.3.2}$$

$$(Y \otimes Z)_{\mathscr{L}} = Y^*_{\mathscr{L}} \cdot Z_{\mathscr{L}} \;\&\; (Y^*)_{\mathscr{L}} = (Y_{\mathscr{L}})^*. \tag{3.3.3}$$

Before comparing $[X:f]$ with $\mu X_{\mathscr{L}}$ in general, we need to study $X_{\mathscr{L}}$ more closely when X is reduced.

LEMMA 3.4. *Let X be a reduced flow expression. Then*

$$X = \phi \quad \textit{iff} \quad X_{\mathscr{L}} = \phi, \tag{1}$$

$$X = \lambda \quad \textit{iff} \quad X_{\mathscr{L}} = \{\lambda\}, \tag{2}$$

$$X \textit{ is nontrivial} \quad \textit{iff} \quad (\exists \mathbf{c} \neq \lambda)(\mathbf{c} \in X_{\mathscr{L}}). \tag{3}$$

Proof. Looking only to the right of "iff" in (1), (2), (3), we find three conditions on X that are mutually exclusive and exhaust all possibilities. These conditions therefore partition the reduced flow expressions into three sets $R1$, $R2$, $R3$. By Lemma 2.7, the conditions to the left of "iff" in (1), (2), (3) also partition the reduced flow expressions into three sets, call them $L1, L2, L3$. Moreover, $L1 \subseteq R1$ and $L2 \subseteq R2$ by (3.2.1). To show that the two partitions are equal, it will suffice to show $L3 \subseteq R3$. Induction on the structure of X will be used to show that any nontrivial reduced X has a nonnull path in $X_{\mathscr{L}}$.

If X is a constant or variable, then X is an arc c and has the string (c) in $X_{\mathscr{L}}$ by (3.2.2). To continue the induction, suppose $X = Y \mathbin{\square} Z$. (The case $X = Y^*$ is like that of $X = Y \otimes Z$.) Because X is reduced, the subexpressions Y, Z are reduced and cannot be ϕ.

Suppose X is $Y \cdot Z$. Then Y, Z cannot be λ. By Lemma 2.7 and the induction hypothesis, we can choose $\mathbf{c} \neq \lambda$ in $Y_{\mathscr{L}}$ and $\mathbf{d} \neq \lambda$ in $Z_{\mathscr{L}}$. Then $\mathbf{c} \cdot \mathbf{d} \neq \lambda$ in $Y_{\mathscr{L}} \cdot Z_{\mathscr{L}} = X_{\mathscr{L}}$.

Suppose X is $Y \cup Z$. Then Y, Z cannot both be λ. By Lemma 2.7 and the induction hypothesis, we can choose $\mathbf{c} \neq \lambda$ in $Y_{\mathscr{L}} \cup Z_{\mathscr{L}} = X_{\mathscr{L}}$.

Suppose X is $Y \otimes Z$. Then Y cannot be λ. By Lemma 2.7 and the induction hypothesis, we can choose $\mathbf{c} \neq \lambda$ in $Y_{\mathscr{L}}$ and $\mathbf{d}$ (possibly λ) in $Z_{\mathscr{L}}$. Then $\mathbf{c} \cdot \mathbf{d} \neq \lambda$ in $Y^*_{\mathscr{L}} \cdot Z_{\mathscr{L}} = X_{\mathscr{L}}$. □

LEMMA 3.5. *Let X and Y be reduced flow expressions. If Y is a subexpression of X then each $\mathbf{d}$ in $Y_{\mathscr{L}}$ is a substring of some $\mathbf{c}$ in $X_{\mathscr{L}}$.*

Proof. Without loss of generality we can assume that some Z has either $X = Y \mathbin{\square} Z$ or $X = Z \mathbin{\square} Y$ for one of the binary operators $\square$. Note that $Z = \phi$ is impossible because X is reduced. Therefore $Z_{\mathscr{L}} \neq \phi$. By (3.3) and familiar properties of the Kleene operations used to build $X_{\mathscr{L}}$ from $Y_{\mathscr{L}}$ and $Z_{\mathscr{L}}$, each $\mathbf{d}$ in $Y_{\mathscr{L}}$ is a substring of some $\mathbf{c}$ in $X_{\mathscr{L}}$. □

LEMMA 3.6. *Let (L, M) be a 3-way context. Let X be any flow expression for $(G, \mathbf{E})$. Let $f: \mathbf{A}_G \to M$. Let μ be as in* (3.1). *Then $[X:f] \leqq \mu X_{\mathscr{L}}$, and they are equal if (L, M) is strongly distributive and X is reduced.*

Proof. By Lemma 2.8, we can assume without loss of generality that @ is isotone. Let "SD&R" be a local abbreviation for "(L, M) is strongly distributive and X is reduced" in the following induction on the structure of X. For X a constant or variable

we have $[X]=\mu X_{\mathscr{L}}$ directly from the definitions. Otherwise we may assume X is $Y \mathbin{\square} Z$ for a binary operator $\square$ in $\{\cdot, \cup, \otimes\}$. By the induction hypothesis, $[Y] \leqq \mu Y_{\mathscr{L}}$ and $[Z] \leqq \mu Z_{\mathscr{L}}$ (with = if SD&R).

Suppose X is $Y \cdot Z$. Let $\eta = Y_{\mathscr{L}}$ and $\zeta = Z_{\mathscr{L}}$. If SD&R, then $Y \neq \phi$ and Lemma 3.4(1) implies $\eta \neq \phi$. We calculate:

$$
\begin{aligned}
[X] &= [Z] \circ [Y] & \\
&\leqq \mu\zeta \circ \mu\eta & \text{(with = if SD\&R)} \\
&= (\bigwedge \{f\mathbf{d} \mid \mathbf{d} \in \zeta\}) \circ (\bigwedge \{f\mathbf{c} \mid \mathbf{c} \in \eta\}) & \\
&\leqq \bigwedge \{(f\mathbf{d}) \circ (f\mathbf{c}) \mid \mathbf{d} \in \zeta \;\&\; \mathbf{c} \in \eta\} & \text{(with = if SD\&R)} \\
&= \bigwedge \{f(\mathbf{c} \cdot \mathbf{d}) \mid \mathbf{d} \in \zeta \;\&\; \mathbf{c} \in \eta\} & \\
&= \bigwedge \{f\mathbf{b} \mid \mathbf{b} \in (\eta \cdot \zeta)\} = \mu(\eta \cdot \zeta) & \\
&= \mu X_{\mathscr{L}} & \text{(by (3.3.1))}.
\end{aligned}
$$

A similar but simpler calculation handles the case $X = Y \cup Z$, with only = needed after the first $\leqq$ because $\bigwedge$ is associative. We use (3.3.2) in place of (3.3.1).

Finally, suppose $X = Y \otimes Z$. We proceed essentially as above, with isotonicity of @ needed for the first $\leqq$. Skipping some intermediate steps now, we calculate:

$$
\begin{aligned}
[X] &\leqq \mu\zeta \mathbin{@} \mu\eta & \text{(with = if SD\&R)} \\
&\leqq \bigwedge \{(f\mathbf{d}) \circ (f\mathbf{c}_r) \circ \cdots \circ (f\mathbf{c}_1) \mid \mathbf{d} \in \zeta \;\&\; r \in \boldsymbol{N} \;\&\; \mathbf{c}_1 \in \eta \;\&\; \cdots \;\&\; \mathbf{c}_r \in \eta\} & \\
& & \text{(with = if SD\&R)} \\
&= \mu(\eta^* \cdot \zeta) = \mu X_{\mathscr{L}} & \text{(by (3.3.3))}.
\end{aligned}
$$

In all cases, $[X] \leqq \mu X_{\mathscr{L}}$ by a chain of inequalities that become equalities if SD&R. □

This section has still not justified its title. It is time to construct the canonical context for $(G, \mathbf{E})$. For all ξ, η in $\mathscr{L}$ let $\xi \leqq \eta$ if and only if $\eta \subseteq \xi$. Then $\mathscr{L}$ is a complete lattice with $\top_{\mathscr{L}} = \phi$. Why did we not define $\leqq$ to be $\subseteq$ (which would make $\top_{\mathscr{L}}$ be $\mathbf{A}_G^*$)? Intuitively, if $\eta \subseteq \xi$ then we know more about a string $\mathbf{c}$ if we know $\mathbf{c} \in \eta$ than if we only know $\mathbf{c} \in \xi$. Sets that are small according to $\subseteq$ correspond to strong assertions and hence should be large in $\mathscr{L}$.

For each η in $\mathscr{L}$ there is an isotone map $\Delta\eta : \mathscr{L} \to \mathscr{L}$ defined by

$$(\Delta\eta)\xi = \xi \cdot \eta = \{\mathbf{c} \cdot \mathbf{d} \mid \mathbf{c} \in \xi \text{ and } \mathbf{d} \in \eta\}. \tag{3.7.1}$$

This map is strongly distributive and has $(\Delta\eta)\top_{\mathscr{L}} = \top_{\mathscr{L}} = (\Delta\top_{\mathscr{L}})\xi$. Let

$$\mathscr{M} = \{\Delta\eta \mid \eta \in \mathscr{L}\}. \tag{3.7.2}$$

Then $\mathscr{M}$ includes the identity map $\mathbf{1}_{\mathscr{L}} = \Delta\{\lambda\}$ as well as the constant map $\top_{\mathscr{M}} = \Delta\top_{\mathscr{L}}$ with value $\top_{\mathscr{L}}$. Thanks to the $\cdot$ and $\cup$ operations of $\mathscr{L}$, the set $\mathscr{M}$ is closed under the $\circ$ and $\wedge$ operations on isotone maps:

$$\Delta\zeta \circ \Delta\eta = \Delta(\eta \cdot \zeta) \quad \text{and} \quad \Delta\zeta \wedge \Delta\eta = \Delta(\eta \cup \zeta).$$

Thanks to the * and $\cdot$ operations on $\mathscr{L}$, the set $\mathscr{M}$ is closed under @ defined by restricting $@_{\text{all}}$ from (2.3.1) to $\mathscr{M} \times \mathscr{M}$:

$$\Delta\zeta \mathbin{@} \Delta\eta = \Delta(\eta^* \cdot \zeta).$$

In short, $(\mathcal{L}, \mathcal{M})$ is a strongly distributive 3-way context with $U \circ \top_{\mathcal{M}} = \top_{\mathcal{M}}$ for all U in $\mathcal{M}$. This is the *canonical context* for $(G, \mathbf{E})$.

Let $f_{\text{can}} : \mathbf{A}_G \to \mathcal{M}$ have

$$f_{\text{can}} c = \Delta\{(c)\} \quad \text{for each arc } c, \tag{3.8.1}$$

while each m in $\mathbf{E}$ determines $E_m : \mathbf{E} \to \mathcal{L}$ with

$$E_m n = (\textbf{if } n = m \textbf{ then } \{\lambda\} \textbf{ else } \phi). \tag{3.8.2}$$

Linking $(G, \mathbf{E})$ and $(\mathcal{L}, \mathcal{M})$ with the local and entry information (f_{can}, E_m) specifies a data flow analysis problem, the *canonical problem with entry at m.*

LEMMA 3.9. *Let $\mathcal{I}_m : \mathbf{N}_G \to \mathcal{L}$ be the maximum solution to the canonical problem with entry at m in* $\mathbf{E}$. *For each node p, $\mathcal{I}_m p$ is the set of all paths from m to p in G. Moreover, this is the only good solution to the problem.*

Proof. Direct calculation shows that $\mathcal{I}_m p$ is the set of all paths from m to p. Because $(\mathcal{L}, \mathcal{M})$ is strongly distributive and has $U \top_{\mathcal{L}} = \top_{\mathcal{L}}$ for all U in $\mathcal{M}$, the maximum solution is a fixpoint and is therefore the only good solution for any problem with context $(\mathcal{L}, \mathcal{M})$. □

Finally, we note two useful identities verified by direct calculation. All ξ in $\mathcal{L}$ satisfy

$$(\Delta\xi)\{\lambda\} = \xi, \tag{3.10}$$

and this implies that all flow expressions X satisfy

$$[X : f_{\text{can}}]\{\lambda\} = X_{\mathcal{L}}. \tag{3.11}$$

4. Proof of the main theorem. Throughout this section, a flow scheme $(G, \mathbf{E})$ is given. The canonical context $(\mathcal{L}, \mathcal{M})$ is as in § 3. A family $\mathcal{X} = \{X(m, p) \mid m \in \mathbf{E} \ \&\ p \in \mathbf{N}_G\}$ of flow expressions is a *path cover* if and only if all m in $\mathbf{E}$ and p in $\mathbf{N}_G$ have $X(m, p)_{\mathcal{L}}$ equal to the set $\mathcal{I}_m p$ of all paths from m to p in G. All of the lemmas in this section assume that $\mathcal{X}$ is a reduced path cover (i.e., a path cover whose flow expressions are reduced in the sense of § 2). The lemmas will establish that any reduced path cover is also a reduced flow cover relative to the class of all 3-way contexts. It will be helpful to consider the family $\mathcal{F}$ of all flow expressions that are subexpressions of expressions in $\mathcal{X}$. Note that members of $\mathcal{F}$ are also reduced.

Whenever nodes u, v are such that $X_{\mathcal{L}}$ is a set of paths from u to v, let (u, v) be called a *source/target pair* for X. For example, (m, p) is a source/target pair for $X = X(m, p)$ in the path cover $\mathcal{X}$ because $X(m, p)_{\mathcal{L}} = \mathcal{I}_m p$. All pairs (u, v) are source/target pairs for $X = \phi$ by Lemma 3.4(1). All pairs (u, v) with $u = v$ are source/target pairs for $X = \lambda$ by Lemma 3.4(2). What are the source/target pairs for general nontrivial X in $\mathcal{F}$?

LEMMA 4.1. *Let X be in $\mathcal{F}$ and suppose X is nontrivial. Then X has a unique source/target pair (u, v). If $X = Y \cdot Z$, then Y and Z have source/target pairs (u, w) and (w, v) for some w. If $X = Y \cup Z$, then Y and Z both have (u, v), as a source/target pair. If $X = Y^*$ or $X = Y \otimes Z$, then Y has a source/target pair (u, u) and Z has a source/target pair (u, v).*

Proof. By Lemma 3.4(3) and the fact that a nonnull path runs from a unique u to a unique v, X can have at most one source/target pair. To prove existence of a source/target pair we use induction on the structure of X, but with downward rather than the usual upward motion. Let $\xi = X_{\mathcal{L}}$.

The basis for the induction is the case in which X is $X(m, p)$ in $\mathcal{X}$, and in this case (m, p) is a source/target pair. To continue the induction, we consider any X in

$\mathscr{F}$ that has a source/target pair (u, v) and has $X = Y \boxtimes Z$ for one of the formal operators $\boxtimes$. In the course of showing that Y and Z have source/target pairs, we will also show that these pairs relate to (u, v) in the asserted way. Let $\eta = Y_{\mathscr{L}}$ and $\zeta = Z_{\mathscr{L}}$.

Suppose $\boxtimes$ is $\cdot$. Then Y and Z are nontrivial. By Lemma 3.4(3) there are $\mathbf{c} \neq \lambda$ in η and $\mathbf{d} \neq \lambda$ in ζ with $\mathbf{c}\cdot\mathbf{d}$ in $\eta\cdot\zeta = \xi$. Therefore $\mathbf{c}\cdot\mathbf{d}: u \rightarrow v$ in G and some node w must have $\mathbf{c}: u \rightarrow w$ and $\mathbf{d}: w \rightarrow v$. Because any $\mathbf{c}'$ in η and any $\mathbf{d}'$ in ζ have $\mathbf{c}'\cdot\mathbf{d}$ and $\mathbf{c}\cdot\mathbf{d}'$ in ξ also, (u, w) and (w, v) are the desired source/target pairs.

Suppose $\boxtimes$ is $\cup$. Then $\eta \subseteq \xi$ and $\zeta \subseteq \xi$, so Y and Z have (u, v) as a source/target pair.

Suppose $\boxtimes$ is $\otimes$. Then Y is nontrivial and Z is either λ or nontrivial. Any $\mathbf{c}$ in η and any $\mathbf{d}$ in ζ have $\mathbf{c}\cdot\mathbf{d}$ and $\mathbf{d}$ both in $\eta^*\cdot\zeta = \xi$, so $\mathbf{c}: u \rightarrow u$ and $\mathbf{d}: u \rightarrow v$. □

LEMMA 4.2. *Let (L, M) be a 3-way context. Let $J: \mathbf{N}_G \rightarrow L$ be a fixpoint for a problem with this context and with local information $f: \mathbf{A}_G \rightarrow M$. Let X be in $\mathscr{F}$ with source/target pair (u, v). Then $Jv \leqq [X:f]Ju$. In particular, all m in $\mathbf{E}$ and p in $\mathbf{N}_G$ have*

$$(1) \qquad Jp \leqq [X(m, p):f]Jm.$$

Proof. Induction on the structure of X is used. If X is ϕ (resp. λ), then the claim reduces to $Jv \leqq \top_L$ (resp. $Ju \leqq Ju$). If X is an arc, then the claim holds by definition of fixpoints. In all other cases, X is nontrivial by Lemma 2.7 and can be assumed to have the form $Y \boxtimes Z$. Source/target pairs for Y and Z are as in Lemma 4.1.

Suppose $\boxtimes$ is $\cdot$, with (u, w) and (w, v) as in Lemma 4.1. By the induction hypothesis, $Jv \leqq [Z]Jw \leqq [Z][Y]Ju = [X]Ju$.

Suppose $\boxtimes$ is $\cup$. By the induction hypothesis, $Jv \leqq [Z]Ju$ and $Jv \leqq [Y]Ju$. Therefore $Jv \leqq ([Z] \wedge [Y])Ju = [X]Ju$.

Suppose $\boxtimes$ is $\otimes$. By the induction hypothesis, $Jv \leqq [Z]Ju$ and $Ju \leqq [Y]Ju$. We apply (2.3.3) for $x = Ju$ to infer that $Jv \leqq [Z]Ju \leqq ([Z]@[Y])Ju = [X]Ju$. □

LEMMA 4.3. *Let (L, M) be a 3-way context. Let $f: \mathbf{A}_G \rightarrow M$. Each path $\mathbf{c}: m \rightarrow p$ with m in $\mathbf{E}$ and p in $\mathbf{N}_G$ has $[X(m, p):f] \leqq f\mathbf{c}$.*

Proof. By Lemma 3.6, $[X(m, p):f] \leqq \mu X(m, p)_{\mathscr{L}} = \bigwedge \{f\mathbf{c} \mid \mathbf{c} \in \mathscr{I}_m p\}$. □

To be a flow cover *is* to be a family of flow expressions with the properties claimed in Lemma 4.3 and Lemma 4.2(1), so we have shown that any reduced path cover is also a flow cover. A little more work leads to the main theorem, with Theorem 1.2 as a corollary. To be a flow cover adequate for *any* problem solvable by elimination, $\mathscr{X}$ only needs to be a flow cover relative to the class $\{(\mathscr{L}, \mathscr{M})\}$ of contexts *and* the class $\{(f_{\text{can}}, E_m) \mid m \in \mathbf{E}\}$ of local and entry information pairs.

THEOREM 4.4. *For any family $\mathscr{X} = \{X(m, p) \mid m \in \mathbf{E} \;\&\; p \in \mathbf{N}_G\}$ of reduced flow expressions, the following three conditions are equivalent:*

(1) *$\mathscr{X}$ is a path cover,*

(2) *$\mathscr{X}$ is a flow cover relative to the class of all 3-way contexts,*

(3) *$\mathscr{X}$ is a flow cover relative to the classes $\{(\mathscr{L}, \mathscr{M})\}$ and $\{(f_{\text{can}}, E_m) \mid m \in \mathbf{E}\}$.*

Proof. The lemmas in this section have already accomplished the difficult part, which is the proof that (1) implies (2). That (2) implies (3) is trivial. Finally, Lemma 3.9 and (3.11) show that (3) implies (1). □

The phrase "relative to the class of all 3-way contexts" is becoming tiresome. From now on, "flow cover" without any explicit qualification will be used.

COROLLARY 4.5. *Let $\mathscr{X}$ be a reduced flow cover. For all m in $\mathbf{E}$ and all p in $\mathbf{N}_G$, there is a path from m to p in G if and only if $X(m, p)$ is not ϕ.*

Proof. By (2) implies (1) in Theorem 4.4, $\mathscr{X}$ is a path cover. There is a path from m to p in G if and only if $\mathscr{I}_{mp} \neq \phi$ if and only if $X(m, p)_{\mathscr{L}} \neq \phi$. Lemma 3.4(1) completes the proof. □

COROLLARY 4.6. *Suppose the family $\mathscr{X}$ is the result of transforming the expressions in a flow cover by applying rules valid for regular expressions, then reducing with the rules* (2.6). *Then $\mathscr{X}$ is a flow cover.*

Proof. The original flow cover $\mathscr{Y}$ is also a flow cover relative to the canonical context. Rules valid for regular expressions are valid for data flow analysis in this context, so $\mathscr{X}$ is a flow cover relative to the canonical context and hence (by (3) implies (2) in Theorem 4.4) a flow cover. □

5. Applications. Flow covers for looping structures (e.g., the **while** statement in [7, Lemma 5.9]) are derived by a somewhat laborious process in [7]. The flow covers produced by general procedures are optimized by hand. Thanks to Theorem 4.4, the flow covers claimed in results like [7, Lemma 5.9] are obtained with hardly any effort. They are obviously reduced path covers, and *therefore* they are flow covers.

Tarjan [8] shows that various *path problems* (e.g., finding shortest paths) can be solved by a path cover. For each problem an appropriate interpretation of regular expressions can be provided, as when (2.5) is the interpretation appropriate for data flow analysis. The problem of finding a (reduced) path cover for a given flow scheme $(G, \mathbf{E})$ is the *canonical* path problem because its solution leads to solutions for the others. The special cases $\mathbf{E} = \mathbf{N}_G$ and $|\mathbf{E}| = 1$ arise frequently. The general case can be treated either by assuming $\mathbf{E} = \mathbf{N}_G$ (and then discarding superfluous expressions) or by iteration over $\mathbf{E}$ (with each entry node in turn treated as "the" entry node by an algorithm assuming $|\mathbf{E}| = 1$). Another way to treat multiple entries with algorithms that assume single entries will emerge from the results in this section. Our results do not logically depend on Theorem 4.4, but we call them "applications" because they owe their significance to Theorem 4.4 and the similar results for other path problems in [8].

Known algorithms for the canonical path problem range from [1, Alg. 5.5] (simple and directly applicable to $\mathbf{E} = \mathbf{N}_G$, but with $O(|\mathbf{N}_G|^3)$ cost) to the fast method of [9] (complicated and directly applicable only if $|\mathbf{E}| = 1$, but with essentially linear cost on reducible[4] flow schemes). The slower-but-simpler method described in [9] has $O(|\mathbf{A}_G| \log |\mathbf{N}_G|)$ cost on reducible flow schemes. Both methods from [9] also work on single-entry schemes that are not reducible, but the cost bounds are not so simple to state. The simpler method from [9] is likely to work well in practice, especially when $|\mathbf{E}| \ll |\mathbf{N}_G|$.

The path cover algorithms just discussed are designed for use in situations where a large flow scheme is the input. Another kind of situation is also of interest. The large flow scheme is the result of a *small change* in a previously processed scheme. The algorithm *retains* the previous scheme and its path cover. The input is the small change, not the entire large scheme, and the task is to *update* the previous path cover without redoing all the work. We begin with the easiest kind of updating. If H is the result of deleting some arcs from G, it should be possible to quickly derive a path cover for $(H, \mathbf{E})$ from one for $(G, \mathbf{E})$.

THEOREM 5.1. *Let $\mathscr{X}$ be a path cover for $(G, \mathbf{E})$ and let H be the result of deleting some arcs from G. For all (m, p) in $\mathbf{E} \times \mathbf{N}_H$, let $Y(m, p)$ be derived from $X(m, p)$ in two stages. First, replace each occurrence of a deleted arc by an occurrence of the*

[4] Reducibility is a property of many single-entry flow schemes. It has several characterizations [5]. Most of the flow schemes arising in data flow analysis are reducible.

expression ϕ. *Second, reduce the resulting expressions. Then* $\mathscr{Y}$ *is a reduced path cover for* $(H, \mathbf{E})$.

Proof. Let D be the set of arcs deleted from G to form H. By structural induction it is easy to show that

$$Y(m, p)_{\mathscr{L}} = X(m, p)_{\mathscr{L}} \cap (\mathbf{A}_G - D)^* = X(m, p)_{\mathscr{L}} \cap \mathbf{A}_H^*.$$

The paths from m to p in H are exactly the paths from m to p in G that use only arcs in H, so $\mathscr{Y}$ is a path cover because $\mathscr{X}$ is a path cover. □

The process by which $Y(m, p)$ is derived from $X(m, p)$ in Theorem 5.1 will be called "ϕ-ing out" the deleted arcs hereafter. Theorem 5.1 is both stronger and simpler than an earlier result [7, Lemma 2.10] because it uses the rule $\phi \cdot Z \to \phi$ despite the fact that many 3-way contexts (L, M) have $V \circ \top_M \neq \top_M$ for most choices of V.

Theorem 5.1 deals with the easiest aspect of updating a given path cover to reflect a change in the flow scheme. It is more difficult to add arcs than to delete them, of course, but addition of an arc running to an entry node can be handled without much trouble.

THEOREM 5.2. *Let* $\mathscr{X}$ *be a path cover for* $(G, \mathbf{E})$ *and let* H *be the result of adding a new arc* c *to* G, *such that the source node* $\mathbf{s}c$ *is already in* $\mathbf{N}_G$ *and the target node* $\mathbf{t}c$ *is already in* $\mathbf{E}$. *Let* Z *be the result of reducing the flow expression* $[c \cdot X(\mathbf{t}c, \mathbf{s}c)] \otimes c$. *For all* (m, p) *in* $\mathbf{E} \times \mathbf{N}_H$ *let* $Y(m, p)$ *be the result of reducing the flow expression*

$$X(m, p) \cup [(X(m, \mathbf{s}c) \cdot Z) \cdot X(\mathbf{t}c, p)].$$

Then $\mathscr{Y}$ *is a reduced path cover for* $(H, \mathbf{E})$.

Proof. To simplify notation, we deliberately confuse expressions W with their values $W_{\mathscr{L}}$ in $\mathscr{L}$. Because $\mathscr{X}$ is a path cover and the reduction rules preserve values in $\mathscr{L}$, we find that Z considered in $\mathscr{L}$ satisfies

$$\begin{aligned} Z &= [\{(c)\} \cdot X(\mathbf{t}c, \mathbf{s}c)]^* \cdot \{(c)\} \\ &= [\{(c)\} \cdot (\text{Paths from } \mathbf{t}c \text{ to } \mathbf{s}c \text{ in } G)]^* \cdot \{(c)\} \\ &= (\text{Paths from } \mathbf{s}c \text{ to } \mathbf{t}c \text{ in } H \text{ that begin and end with } c). \end{aligned}$$

Therefore, as each path from m to p involving c has a first and a last occurrence of c,

$$\begin{aligned} Y(m, p) &= X(m, p) \cup [(X(m, \mathbf{s}c) \cdot Z) \cdot X(\mathbf{t}c, p)] \\ &= (\text{Paths from } m \text{ to } p \text{ in } H \text{ that do not involve } c) \\ &\quad \cup (\text{Paths from } m \text{ to } p \text{ in } H \text{ that involve } c) \\ &= (\text{Paths from } m \text{ to } p \text{ in } H). \end{aligned}$$

Therefore $\mathscr{Y}$ is a path cover. □

This theorem is reminiscent of the induction step in the proof of [7, Lemma 2.8], but here the proof is both shorter and more detailed because the construction is simpler. Ease of updating when the target of an added arc is already in $\mathbf{E}$ motivates an interesting departure from traditional ways of applying data flow analysis in compilers. Traditionally, the theory assumes that $\mathbf{E}$ has only one node. If necessary, a new node and some arcs (from it to the actual entry nodes) are added to enforce this assumption before analysis begins. When the ability to update flow covers for added arcs is wanted, it may instead be advantageous to pretend that $\mathbf{E}$ is larger than it really is. (When the flow covers are *used*, only the real entries should be considered.)

The combinatorics of letting $\mathbf{E}$ be most of $\mathbf{N}_G$ may seem unfavorable at first glance: $|\mathbf{E}\times\mathbf{N}_G|$ is close to $|\mathbf{N}_G|^2$. However, the flow schemes usually encountered in syntax-directed analysis [7, §§ 4–6] represent very small fractions of the total control flow. It is easy (and advisable, for the sake of readability) to write programs wherein all flow schemes encountered have very few potential entry nodes, no matter how large and complex the programs themselves have to be.

Real additions to the set of entry nodes can occur when programs are changed by the introduction of **goto** statements, so we need to cope with them anyway. With the understanding that $\mathbf{E}$ is maintained somewhat larger than the real set of entry nodes, it becomes possible to cover each addition of a node n_1 to $\mathbf{E}$ by a variation on what is already known about a node n_0 already in $\mathbf{E}$. Together with Theorem 5.1, the next two theorems provide all the updating capability needed to cope with arbitrary changes in programs with unrestricted **goto** statements. As in [7], a rigorously correct general statement is somewhat complicated, yet what actually happens in typical situations is quite simple and intuitive. To provide a little motivation for one of the concepts to be used, let us momentarily think of the arcs in G as bridges in the New York metropolitan area. An arc string without repetitions could list the bridges to be crossed along a certain route, say from Yorktown Heights to Kennedy Airport. If the Whitestone Bridge is closed for repairs, then an arc string that begins with this bridge can be *detoured* to some alternate bridge, perhaps the Throg's Neck Bridge. (For present purposes only the first bridge along a route might be closed for repairs.) This situation is a very special case of the following definition.

DEFINITION 5.3. A *detour scheme* for a flow scheme $(G, \mathbf{E})$ consists of a set $\mathcal{D}$ of arcs and a map $\delta:\mathcal{D}\rightarrow\mathbf{A}_G\cup\{\lambda\}$. An arc string $\mathbf{c}=(c_1\cdots c_K)$ is *detourable* if and only if it is nonnull and the first arc c_1 is in $\mathcal{D}$. The *detour* of $\mathbf{c}$ is the string like $\mathbf{c}$ but with c_1 replaced by δc_1. (Replacement by λ is deletion, which shortens the string.)

In the motivating example with the Whitestone Bridge (WS) and the Throg's Neck Bridge (TN), we have $\mathcal{D}=\{\text{WS}\}$ and $\delta(\text{WS})=\text{TN}$. In general there is a curious situation when a repeated arc b is in $\mathcal{D}$. The first time we try to follow b we are detoured along δb, but thereafter we can follow b. Driving to the airport is confusing enough without attempting to interpret the general case in terms of the motivating example. Suffice it to say, the general case does arise in our approach for updating path covers. The following lemma will be useful.

LEMMA 5.4. *Let $(\mathcal{D}, \delta)$ be a detour scheme and let X be a flow expression for a flow scheme $(G, \mathbf{E})$. Let ξ' be the set of all detours of detourable strings in $\xi=X_{\mathcal{L}}$. Then there is a reduced flow expression X' with $\xi'=X'_{\mathcal{L}}$, such that X' can be constructed from X by structural induction.*

Proof. The basis for the induction is the case where X has no proper subexpressions. If X is an arc c in $\mathcal{D}$, then X' is taken to be δc. Otherwise X' is taken to be ϕ.

To continue the induction, suppose X is $Y\,\square\,Z$, with subexpressions Y, Z that have been detoured by reduced flow expressions Y', Z'. Let Y_{none} be the result of ϕ-ing out all arcs in Y, so that Y_{none} is either ϕ or λ. If $\square$ is $\cdot$, then X' is taken to be the result of reducing

$$(Y'\cdot Z)\cup(Y_{\text{none}}\cdot Z'). \tag{i}$$

If $\square$ is $\cup$, then X' is taken to be the result of reducing

$$Y'\cup Z'. \tag{ii}$$

If $⌑$ is $\otimes$, then X' is taken to be the result of reducing

$$[Y' \cdot (Y \otimes Z)] \cup Z'. \tag{iii}$$

In all three cases, we calculate that $\xi' = X'_{\mathscr{L}}$. □

Because a copy of Z and a modified copy of Z both occur in $(Y ⌑ Z)'$ when $⌑$ is or $\otimes$, a complex expression X can in general have a much larger detoured expression X'. In practice, however, expressions will not be represented as strings or trees but as acyclic list structures (also called "dags" or "collapsed trees") wherein a subexpression may be simultaneously the ith operand of one operator occurrence and the jth operand of another operator occurrence. With such sharing, the size of X' is bounded by a linear function of the size of X in even the worst case. Typically, the many ϕ and λ subexpressions introduced by the construction of X' will leave X' smaller than X after reduction.

THEOREM 5.5. *Let $\mathscr{X}$ be a path cover for $(G, \mathbf{E})$ and let n_0, n_1 be nodes in G such that $n_0 \in \mathbf{E}$ and there is an arc i from n_0 to n_1 in G. A reduced path cover $\mathscr{Y}$ for $(G, \mathbf{E} \cup \{n_1\})$ can be constructed from $\mathscr{X}$ by structural induction.*

Proof. We must choose $Y(n_1, p)$ for arbitrary p in $\mathbf{N}_G$. The flow scheme $(G, \mathbf{E} \cup \{n_1\})$ has a detour scheme $(\{i\}, \delta)$, where $\delta(i) = \lambda$. The given expression $X = X(n_0, p)$ from $\mathscr{X}$ has a convenient property: the set ξ' from Lemma 5.4 is precisely the set of all paths from n_1 to p in G. Therefore the reduced flow expression X' provided by the lemma may be used as $Y(n_1, p)$ in $\mathscr{Y}$. □

THEOREM 5.6. *Let $\mathscr{X}$ be a path cover for $(G, \mathbf{E})$ and let H be the result of adding a new node n_1 and some arcs, as follows. There is a set* COPY *of arcs out from a node n_0 in $\mathbf{E}$, such that each arc in* COPY *is copied to an outarc of n_1 with the same target as in G. A reduced path cover $\mathscr{Y}$ for $(H, \mathbf{E} \cup \{n_1\})$ can be constructed from $\mathscr{X}$ by structural induction.*

Proof. Because n_1 has no inarcs in H, paths from m to p in H are the same as paths from m to p in G whenever $m \neq n_1$ and $p \neq n_1$. We must choose $Y(m, p)$ when n_1 is one of the nodes m, p. For $p = n_1$ we use λ if $m = n_1$ and ϕ otherwise. For $p \neq n_1$ and $m = n_1$ we apply Lemma 5.4. The flow scheme $(H, \mathbf{E} \cup \{n_1\})$ has a detour scheme (COPY, δ), where δ maps each copied outarc of n_0 to the corresponding outarc of n_1. The given expression $X = X(n_0, p)$ from $\mathscr{X}$ has a convenient property: the set ξ' from Lemma 5.4 is precisely the set of all paths from n_1 to p in H. Therefore the reduced flow expression X' provided by the lemma may be used as $Y(n_1, p)$ in $\mathscr{Y}$. □

To avoid maintaining $\mathbf{E}$ much larger than the set of nodes that could easily become real entry nodes (if they are not already), it will be helpful to handle certain changes with another theorem. The updating accomplished by the next theorem could be accomplished by the previous ones, but only by treating an inconveniently large number of nodes as entries.

THEOREM 5.7. *Let $\mathscr{X}$ be a path cover for $(G, \mathbf{E})$ and let H be the result of adding a new node n_1 and two arcs, as follows. There are distinct nodes n_0 and n_2 (not necessarily in $\mathbf{E}$), such that n_2 has no outarcs in G. An arc i from n_0 to n_1 and an arc e from n_1 to n_2 are added in H. A reduced path cover $\mathscr{Y}$ for $(H, \mathbf{E})$ can be constructed from $\mathscr{X}$ by using the following formal equations for all m in $\mathbf{E}$, then reducing the expressions $Y(m, p)$ so defined.*

$$Y(m, p) = X(m, p) \quad \text{if} \quad p \in \mathbf{N}_G \text{ and } p \neq n_2, \tag{1}$$

$$Y(m, n_1) = Y(m, n_0) \cdot i, \tag{2}$$

$$Y(m, n_2) = X(m, n_2) \cup [Y(m, n_1) \cdot e]. \tag{3}$$

Proof. Because n_2 has no outarcs in H, paths from m to p in H are the same as paths from m to p in G whenever $p \neq n_1$ and $p \neq n_2$. The new paths in H from m to n_1 or to n_2 are accounted for by (2) and (3). □

This theorem and Theorem 5.1 can be applied several times to yield [7, Lemma 6.6], whose greater complexity is due partly to the simultaneous handling of several nodes n_1 and partly to the intrusion of deletion. Indeed, the five theorems in this section are so simple that the reader may wonder why they are called "theorems" in the first place. Here the word honors utility, not difficulty. As [7, § 6] and work cited there point out, practical structured programming sometimes goes beyond the classical control structures with escapes (like **leave** or **return** statements) and jumps (general **goto** statements). It is useful to maintain accurate control flow information despite the perturbations due to occasional escapes and jumps in an evolving large program. Our theorems are tools for this task, and perhaps for updating tasks associated with other path problems [8].

Consider the problem of applying a path cover algorithm that assumes $|\mathbf{E}| = 1$ to schemes with multiple entries. As in the traditional way to force $|\mathbf{E}| = 1$, we add a new node n_0 and arcs from n_0 to the actual entries, transforming the original scheme $(G, \mathbf{E})$ to a scheme $(H, \{n_0\})$ with $|\mathbf{N}_H| = |\mathbf{N}_G| + 1$ and $|\mathbf{A}_H| = |\mathbf{A}_G| + |\mathbf{E}|$. After finding a path cover for the new scheme, we get one for $(H, \{n_0\} \cup \mathbf{E})$ by $|\mathbf{E}|$ applications of Theorem 5.5. For m in $\mathbf{E}$ and p in $\mathbf{N}_G$, the paths from m to p in G are exactly the paths from m to p in H, so this path cover is also good for the original scheme $(G, \mathbf{E})$ as soon as we discard the now superfluous expressions for paths from n_0 or to n_0. It is likely that the cost of applying a single-entry algorithm once and then applying Theorem 5.5 several times will be less than the cost of applying the single-entry algorithm several times, especially when "cost" includes the space occupied by acyclic list structures. Unlike several independent applications of the single-entry algorithm, our approach leads to shared subexpressions among expressions for different entries. Unlike the folkloric addition of n_0 to G, our approach does not require that the tables associated with each data flow problem be augmented with special entry information at n_0 and special local information along each outarc of n_0. In compilers and other systems that use data flow analysis, such augmentation is undesirable because it introduces several opportunities for error in an enterprise that already has more than enough of them. We only augment G while constructing the path cover; the artificial node and its outarcs have disappeared by the time local and entry information needs to be manipulated.

6. Finite approximations. Lemma 2.8 illustrates the technical convenience of working with all 3-way contexts rather than just with rapid closed contexts. Without loss of generality, we can sometimes assume that @ is isotone. Perhaps, however, the flow scheme $(G, \mathbf{E})$ has a flow cover $\mathscr{X}$ relative to the class of all rapid closed contexts that is not a flow cover relative to the class of all 3-way contexts. Though inappropriate for the canonical context (which is 3-way but not rapid), $\mathscr{X}$ may be adequate for all practical purposes and may be more efficient to use than any of the flow covers general enough to cope with any 3-way context. We introduced 3-way contexts for theoretical convenience, not to model a practical need. Have we been too clever?

This section provides a strong negative answer. Covering relative to the canonical context is no harder than covering relative to a certain countably infinite set of rapid closed contexts, each of which has a finite lattice. Any flow cover general enough for all members of this modest set of rapid closed contexts is general enough for all 3-way contexts. Intuitively, the rth context in our set corresponds to symbolic execution with the understanding that only paths of length at most r are to be traced. The symbolic

execution system knows about longer paths, but it cannot distinguish one from another. The canonical context $(\mathscr{L}, \mathscr{M})$ considers sets of strings of arcs. So does the rth context $(\mathscr{L}^{(r)}, \mathscr{M}^{(r)})$, but it can only distinguish between strings of length at most r. Fundamentally, the reason that the family $\{(\mathscr{L}^{(r)}, \mathscr{M}^{(r)}) \mid r \in \boldsymbol{N}\}$ is equivalent to $(\mathscr{L}, \mathscr{M})$ is quite simple. A string $\mathbf{c}$ is a path if and only if all its prefixes are paths, and all its prefixes have length at most r for r the length of $\mathbf{c}$. Questions about $\mathbf{c}$ may depend on questions about shorter strings but not on questions about longer strings. The technical working out of this insight is somewhat subtle, but readers familiar with projection pairs and limits from denotational semantics will recognize old friends.

For each r in $\boldsymbol{N}$ let $\alpha^{(r)}$ be the set of all strings in $\mathbf{A}_G^*$ that have length at most r. Let $\omega^{(r)}$ be the set of all strings in $\mathbf{A}_G^*$ that have length greater than r. Finally, let

$$\mathscr{L}^{(r)} = \{\xi \in \mathscr{L} \mid \xi \subseteq \alpha^{(r)} \text{ or } \omega^{(r)} \subseteq \xi\}. \tag{6.1.1}$$

Thus $\mathscr{L}^{(r)}$ is a finite subset of $\mathscr{L}$, with $2|\alpha^{(r)}|$ members. Any set η in $\mathscr{L}$ can be approximated by a set $\Pi^{(r)}\eta$ in $\mathscr{L}^{(r)}$ that knows the short paths in η and also whether η contains long paths:

$$\Pi^{(r)}\eta = (\textbf{if } \eta \subseteq \alpha^{(r)} \textbf{ then } \eta \textbf{ else } \eta \cup \omega^{(r)}). \tag{6.1.2}$$

LEMMA 6.2. *Let η be in $\mathscr{L}$. Then $\eta = \bigcap \{\Pi^{(r)}\eta \mid r \in \boldsymbol{N}\}$.*

Proof. In (6.1.2) we have $\eta \subseteq \Pi^{(r)}\eta$ for all r. Conversely, consider any $\mathbf{c}$ in $\mathbf{A}_G^*$ such that $\mathbf{c} \in \Pi^{(r)}\eta$ for all r. For r large enough to have $\mathbf{c} \in \alpha^{(r)}$ we have $\neg(\mathbf{c} \in \omega^{(r)})$ and conclude that $\mathbf{c} \in \eta$ in (6.1.2). □

Because $\mathscr{L}^{(r)} \subseteq \mathscr{L}$, the Kleene operations are defined on $\mathscr{L}^{(r)}$, with the understanding that $\cdot$ and $*$ will often take arguments from $\mathscr{L}^{(r)}$ to values outside $\mathscr{L}^{(r)}$. The map $\Pi^{(r)}$ nearly commutes with the Kleene operations:

$$\Pi^{(r)}(\xi \cdot \eta) = \Pi^{(r)}(\Pi^{(r)}\xi \cdot \Pi^{(r)}\eta), \tag{6.3.1}$$

$$\Pi^{(r)}(\xi \cup \eta) = \Pi^{(r)}\xi \cup \Pi^{(r)}\eta, \tag{6.3.2}$$

$$\Pi^{(r)}(\xi^*) = \Pi^{(r)}((\Pi^{(r)}\xi)^*). \tag{6.3.3}$$

The map $\Pi^{(r)}$ is also isotone when $\mathscr{L}^{(r)}$ is made into a complete lattice in the obvious way, with $\xi \leqq \eta$ if and only if $\eta \subseteq \xi$.

For an algebraic context we need isotone maps on $\mathscr{L}^{(r)}$ as well as the lattice itself. The construction of $\mathscr{M}$ from (3.7) can be adapted to $\mathscr{L}^{(r)}$ by using $\Pi^{(r)}$. For each η in $\mathscr{L}^{(r)}$ there is an isotone map $\Delta^{(r)}\eta : \mathscr{L}^{(r)} \to \mathscr{L}^{(r)}$ defined by

$$(\Delta^{(r)}\eta)\xi = \Pi^{(r)}[(\Delta\eta)\xi] \quad \text{for all } \xi \in \mathscr{L}^{(r)}. \tag{6.4.1}$$

This map is strongly distributive and has $(\Delta^{(r)}\eta)\phi = \phi = (\Delta^{(r)}\phi)\xi$ for all ξ, η in $\mathscr{L}^{(r)}$. Let

$$\mathscr{M}^{(r)} = \{\Delta^{(r)}\eta \mid \eta \in \mathscr{L}^{(r)}\}. \tag{6.4.2}$$

Then $\mathscr{M}^{(r)}$ includes the identity map on $\mathscr{L}^{(r)}$ as well as the constant map with value ϕ. As in § 3, we find that

$$\Delta^{(r)}\zeta \circ \Delta^{(r)}\eta = \Delta^{(r)}\Pi^{(r)}(\eta \cdot \zeta), \tag{6.5.1}$$

$$\Delta^{(r)}\eta \wedge \Delta^{(r)}\zeta = \Delta^{(r)}(\eta \cup \zeta), \tag{6.5.2}$$

$$\Delta^{(r)}\zeta \,@\, \Delta^{(r)}\eta = \Delta^{(r)}\Pi^{(r)}(\eta^* \cdot \zeta). \tag{6.5.3}$$

In short, $(\mathscr{L}^{(r)}, \mathscr{M}^{(r)})$ is a strongly distributive 3-way context with top-preserving maps. This is the *r-bounded canonical context* for $(G, \mathbf{E})$.

Let $f^{(r)}: \mathbf{A}_G \to \mathcal{M}^{(r)}$ have

$$f^{(r)}c = \Delta^{(r)}\Pi^{(r)}\{(c)\} \quad \text{for each arc } c, \tag{6.6.1}$$

while each m in $\mathbf{E}$ determines $E_m^{(r)}: \mathbf{E} \to \mathcal{L}^{(r)}$ with

$$E_m^{(r)}n = (\mathbf{if}\ n = m\ \mathbf{then}\ \{\lambda\}\ \mathbf{else}\ \phi). \tag{6.6.2}$$

Linking $(G, \mathbf{E})$ and $(\mathcal{L}^{(r)}, \mathcal{M}^{(r)})$ with the local and entry information $(f^{(r)}, E_m^{(r)})$ specifies a data flow analysis problem, the *r-bounded canonical problem with entry at m.*

LEMMA 6.7. *Let $\mathcal{I}_m^{(r)}: \mathbf{N}_G \to \mathcal{L}^{(r)}$ be the maximum solution to the r-bounded canonical problem with entry at m in* $\mathbf{E}$. *For each node p, $\mathcal{I}_m^{(r)}p \cap \alpha^{(r)}$ is the set of all paths of length at most r from m to p in G, while $\omega^{(r)} \subseteq \mathcal{I}_m^{(r)}p$ if and only if there is at least one longer path from m to p in G. Moreover, this is the only good solution to the problem.*

Proof. This is similar to the proof of Lemma 3.9, but now the calculation of $\mathcal{I}_m^{(r)}p$ is not so direct. Consider any path $\mathbf{c} = (c_1 \cdots c_s)$ from m to p. By (2.1) and (6.6.1) and (6.5.1),

$$(f^{(r)}\mathbf{c})\{\lambda\} = [\Delta^{(r)}\Pi^{(r)}\{\mathbf{c}\}]\{\lambda\} = \Pi^{(r)}\{\mathbf{c}\} = \mathbf{if}\ s \leqq r\ \mathbf{then}\ \{\mathbf{c}\}\ \mathbf{else}\ \omega^{(r)}.$$

Because $\bigwedge$ in $\mathcal{L}^{(r)}$ is $\bigcup$, this implies that $\mathcal{I}_m^{(r)}p \cap \alpha^{(r)}$ is the set of all paths of length at most r from m to p while $\omega^{(r)} \subseteq \mathcal{I}_m^{(r)}p$ if and only if there is at least one longer path from m to p. □

LEMMA 6.8. *Let X be a flow expression. For all r in $\boldsymbol{N}$ let $\xi^{(r)}$ be $[X : f^{(r)}]\{\lambda\}$. Then $X_{\mathcal{L}} = \bigcap \{\xi^{(r)} \mid r \in \boldsymbol{N}\}$.*

Proof. By Lemma 6.2, it will suffice to show that $\eta = X_{\mathcal{L}}$ satisfies $\Pi^{(r)}\eta = \xi^{(r)}$ for all r. From (3.10) and (6.4.1) it follows that $\Pi^{(r)}\eta = (\Delta^{(r)}\eta)\{\lambda\}$, so it will suffice to show that $\Delta^{(r)}\eta = [X : f^{(r)}]$. But this follows from (3.3) and (6.5) by structural induction on X (with (6.6.1) starting the induction). □

THEOREM 6.9. *Let $\mathcal{X}$ assign a reduced flow expression $X(m, p)$ to each pair (m, p) in $\mathbf{E} \times \mathbf{N}_G$. If $\mathcal{X}$ is a flow cover relative to the r-bounded canonical problem for each $r \in \boldsymbol{N}$, then $\mathcal{X}$ is a flow cover.*

Proof. By Theorem 4.4, it will suffice to show that all (m, p) have $X(m, p)_{\mathcal{L}} = \mathcal{I}_m p$, where $\mathcal{I}_m p$ is the set of all paths $\mathbf{c}: m \to p$. Consider $\xi^{(r)}$ from Lemma 6.8 for $X = X(m, p)$. By hypothesis and Lemma 6.7, we already have $\xi^{(r)} = \mathcal{I}_m^{(r)}p$. By Lemma 6.8,

$$X_{\mathcal{L}} = \bigcap \{\xi^{(r)} \mid r \in \boldsymbol{N}\} = \bigcap \{\mathcal{I}_m^{(r)}p \mid r \in \boldsymbol{N}\}. \tag{6.10}$$

Lemma 6.7 also relates $\mathcal{I}_m^{(r)}p$ to $\mathcal{I}_m p$ by implying that any arc string $\mathbf{c}$ satisfies

$$(\forall r \in \boldsymbol{N})(\mathbf{c} \in \mathcal{I}_m^{(r)}p) \quad \text{iff} \quad \mathbf{c} \in \mathcal{I}_m p.$$

By (6.10), this implies $X_{\mathcal{L}} = \mathcal{I}_m p$. □

7. Historical remarks. The notion that a path cover might assist in solving data flow problems has been in the folklore for years. Early attempts to exploit regular expressions were unsatisfactory because they had no discernable advantages over known algorithms. The folklore is subsumed under rigorous formulations like [3, Thm. 3.0] and [8, Thm. 5]. Standing alone, the folkloric results suffer from having too few computational complexity bounds and too many presuppositions. Thanks to other results in [3], [7], [8], [9] and the present paper, the folklore becomes more interesting. Good complexity bounds are obtained for low-level analysis [9] and for syntax-directed high-level analysis that nevertheless copes with arbitrary **goto** statements [7]. Presuppositions like idempotence or strong distributivity, though common in folklore, have been found to be unnecessary.

Even problems with nondistributive contexts can be accurately solved by finding a flow cover $\mathscr{X}$ for $(G, \mathbf{E})$ and then evaluating the formal expressions in $\mathscr{X}$ when local information becomes available. This was first shown in [7], which introduced the flow cover concept. When earlier work like [4] is read with flow covers in mind, the same conclusion can be reached for very different ways of choosing $\mathscr{X}$. Written before there had been a satisfactory unification of data flow analysis with other path problems, [7] was very cautious with flow expressions. Properties of regular expressions were deliberately deemphasized by using formal $\circ$, $\wedge$, and @ operators rather than the Kleene operators in building the expressions. In nondistributive algebraic contexts it is possible to have

$$(V \circ U_1) \wedge (V \circ U_2) \neq V \circ (U_1 \wedge U_2),$$

so it was prudent to avoid a syntax wherein the temptation to use rules like

$$(Y_1 \cdot Z) \cup (Y_2 \cdot Z) \rightarrow (Y_1 \cup Y_2) \cdot Z$$

would be irresistable. Thanks to Theorem 4.4, we now know that the temptation need not be resisted. The more intuitive syntax of regular expressions is now preferable and has been used here, but with $\otimes$ added to retain an option for seeking very sharp information [7, p. 190].

The "(1) implies (2)" part of Theorem 4.4 says that every reduced path cover is a flow cover, a result that was first obtained in [8, Thm. 6] under the assumptions (a) that each node is reachable from the (unique) entry node m; (b) that $Em = \perp_L$ (the bottom of the complete lattice L); (c) that @ has the form $V @ U = V \circ \alpha U$ for a unary operator α (whose value at U is written $U^@$ in [8]). Though popular in the literature, assumptions (a) and (b) fail in some applications. For example, one of the criteria for variable propagation (criterion (c) on p. 915 of [2]) is readily expressed as a data flow problem where m is *not* the actual program entry point and (a) fails. On the other hand, (b) fails in interprocedural analysis. Of course there are technical tricks by which (a) and (b) can be forced to hold in any situation, but the tricks require more effort than they are worth. At best, they obfuscate applications wherein (a) and (b) do not naturally hold. At worst, they introduce errors when someone uses an incorrect trick or a trick incompatible with the one assumed by a coworker. It is really simpler to refrain from assuming (a) and (b) until they are needed, which is seldom. No need arises in [8]. The assumption (c) simplifies the calculations slightly, but our use of binary rather than unary @ sometimes yields sharper information [7, p. 190]. The practical significance of binary @ is an open question, but it is prudent for a general theory to err on the side of allowing more sharpness than is known to be worthwhile. Binary @ is much more difficult to motivate than unary @, but the technical difficulties are slight.

Casadei, Righi and Teolis [3, Thm. 3.1] independently showed that every reduced path cover is a flow cover, under assumptions like (a) and (b). They also assumed that certain infinite meets in M_{all} can be computed exactly within M. Their proof was quite unlike ours, with one large induction on the number of nodes in G where we have used several small inductions on the structure of regular expressions. Here and in [8], the expressions are forced by reduction rules to have properties like Lemma 3.4(1). The smaller set of rules in [8] lacks (2.6.1) and therefore is enough for the immediate task but not for the neat characterization in Lemma 2.7. Here and in [8], subexpressions of expressions in an arbitrary reduced path cover are shown to have source/target pairs by downward induction, and then domination of fixpoints can be shown by upward induction. The key ideas in our proof are thus the same as for [8, Thm. 6].

The two papers complement each other. Thanks to the reductions of other path problems to the canonical path problem by [8], our § 5 has some hope of being applicable beyond the original motivating application to data flow analysis of evolving large programs. Thanks to our § 6 (and whatever similar results can be obtained for other path problems), solving the canonical path problem can with some justice be considered necessary (as well as sufficient) for solving a variety of practical problems in a uniform way.

We close this section by comparing the conditions (2.3) on @ in a 3-way context with the conditions on @ in a rapid context [7, Def. 1.6]. The upper bound (2.3.2) is the same in both cases, but the lower bound for rapidity is stronger than (2.3.3):

$$V \circ \bigwedge \{(U \wedge \mathbf{1}_L)^r \mid r \in \boldsymbol{N}\} \leqq V @ U. \tag{7.1.1}$$

Inspection of the proofs in [7] reveals that the weaker condition suffices. Indeed, each use of (7.1.1) in [7] is like a use of (2.3.3) accompanied by a proof that (7.1.1) implies (2.3.3). Easier to verify and easier to use, the weaker condition is a better choice all around. When @ is restricted to have the form $V @ U = V \circ \alpha U$ for some unary $\alpha : M \to M$, condition (2.3.3) is equivalent to the condition used in [8, § 6] for the purposes served by (2.3.3) here.

Rapid contexts are also required to have a constant $t_@$ such that

$$V @ U \text{ can be computed from } U \text{ and } V \text{ in } t_@ \text{ steps,} \tag{7.1.2}$$

where $\circ$ and $\wedge$ operations on maps are counted as single steps. This seems to be true in practice, but there is no point in assuming it before one needs to state time bounds that do not treat evaluations of @ as units. The only nonhistorical interest in the strong conditions (7.1) lies in the possibility that some flow scheme might have a flow cover relative to the class of all rapid closed contexts but unable to cope with some of the 3-way contexts that violate (7.1). Theorem 6.9 shows that this cannot happen.

Acknowledgments. D. Kozen read an early draft of this paper and proposed many improvements. K. W. Kennedy called the author's attention to [8], [9]. The resulting exchange of letters with R. E. Tarjan led to improvements in [8] and major improvements in this paper. J. D. Ullman and the referees also provided helpful comments.

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
[2] J. L. CARTER, *A case study of a new code generating technique for compilers*, Comm. ACM, 20 (1977), pp. 914–920.
[3] G. CASADEI, R. RIGHI AND A. G. B. TEOLIS, *An algebraic view of data flow analysis*, manuscript, April 1980.
[4] S. L. GRAHAM AND M. WEGMAN, *A fast and usually linear algorithm for global flow analysis*, J. Assoc. Comput. Mach., 23 (1976), pp. 172–202.
[5] M. S. HECHT AND J. D. ULLMAN, *Characterizations of reducible flow graphs*, J. Assoc. Comput. Mach., 21 (1974), pp. 367–375.
[6] J. B. KAM AND J. D. ULLMAN, *Monotone data flow analysis frameworks*, Acta Inform., 7 (1977), pp. 305–317.
[7] B. K. ROSEN, *Monoids for rapid data flow analysis*, this Journal, 9 (1980), pp. 159–196.
[8] R. E. TARJAN, *A unified approach to path problems*, J. Assoc. Comput. Mach., 28 (1981), pp. 577–593.
[9] ———, *Fast algorithms for solving path problems*, J. Assoc. Comput. Mach., 28 (1981), pp. 594–614.

SIAM J. COMPUT.
Vol. 11, No. 3, August 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1103-0009 $01.00/0

COMPLEXITY RESULTS FOR SCHEDULING TASKS IN FIXED INTERVALS ON TWO TYPES OF MACHINES*

K. NAKAJIMA,† S. L. HAKIMI,‡ AND J. K. LENSTRA§

Abstract. Suppose that n independent tasks are to be scheduled without preemption on an unlimited number of parallel machines of two types: inexpensive slow machines and expensive fast machines. Each task requires a given processing time on a slow machine or a given smaller processing time on a fast machine. We make two different feasibility assumptions: (a) each task has a specified processing interval, the length of which is equal to the processing time on a slow machine; (b) each task has a specified starting time. For either problem type, we wish to find a feasible schedule of minimum total machine cost. It is shown that both problems are NP-hard in the strong sense. These results are complemented by polynomial algorithms for some special cases.

Key words. parallel machines, tasks, release dates, deadlines, computational complexity, NP-hardness, polynomial algorithm

1. Introduction. We begin by considering the following problem. Suppose there are *n tasks* $T_1, \cdots, T_n$ and an unlimited number of *identical parallel machines*. Each task T_j requires a given *processing time* p_j and is to be executed without interruption between a given *release date* r_j and a given *deadline* $d_j = r_j + p_j$. The tasks are *independent* in the sense that there are no precedence constraints between them. Each machine can execute any task, but no more than one at a time. The problem is to find the minimum number of machines needed to execute all tasks as well as a corresponding schedule of the tasks on the machines.

This problem is known as the "fixed job schedule problem" [6] and as the "channel assignment problem" [8], [9], [10]. It has applications in such diverse areas as vehicle scheduling [2], [15], machine scheduling [6], [8], and computer wiring [8], [9], [10]. As a special case of Dilworth's chain decomposition problem, it is solvable in $O(n^2)$ time by the staircase rule of Ford and Fulkerson [3, p. 65] and by the step-function method of Gertsbakh and Stern [6]. Hashimoto and Stevens [9], [10] presented some interesting graph theoretical approaches to the problem and proposed an $O(n^2)$ algorithm, for which Kernighan, Schweikert and Persky [12] gave an $O(n \log n)$ implementation. Recently, Gupta, Lee and Leung [8] independently developed a different $O(n \log n)$ algorithm and also showed that any solution method for the problem requires $\Omega(n \log n)$ time.

In this paper we will consider a natural generalization of this problem which has potential applications in the scheduling areas mentioned above. Again, there are n independent tasks $T_1, \cdots, T_n$, but there are two types of machines: *slow* machines of cost C^s and *fast* machines of cost $C^f > C^s$. Each task T_j requires a processing time p_j on a slow machine or $q_j (<p_j)$ on a fast machine and is to be executed without interruption between its release date r_j and its deadline $d_j = r_j + p_j$. It is assumed that all numerical problem data are integers. In a feasible schedule, the tasks assigned

* Received by the editors July 30, 1979, and in final revised form September 9, 1981. This research was supported in part by the National Science Foundation under grant ENG79-09724.

† Computer Science Division, Department of Electrical Engineering, Texas Tech University, Lubbock, Texas 79409. Formerly at Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, Illinois 60201.

‡ Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, Illinois 60201.

§ Mathematisch Centrum, Amsterdam, the Netherlands.

to slow machines have to start at their release dates in order to meet their deadlines. For the tasks T_j assigned to fast machines, we make two different feasibility assumptions:

(a) VST (*variable starting times*): T_j may start at any time in the inverval $[r_j, d_j - q_j]$;

(b) FST (*fixed starting times*): T_j has to start at time r_j.

A schedule using m^s slow machine and m^f fast machines has total cost $m^sC^s + m^fC^f$. For either problem type, we wish to find a feasible schedule of minimum total cost.

In § 2 we show that the VST problem is NP-hard [1], [4], [5], [11], even if all release dates are equal. In § 3 we extend our techniques to prove that the FST problem is NP-hard in the case of arbitrary release dates; the case of equal release dates is trivially solvable in $O(n)$ time. The NP-hardness results are "strong" [4], [5] in the sense that they hold even with respect to a unary encoding of the data; this implies that there exists no pseudopolynomial algorithm for these problems unless $\mathcal{P} = \mathcal{NP}$.

In §§ 4 and 5 we consider the special case that $q_j = 1, j = 1, \cdots, n$. We present $O(n \log n)$ algorithms for the VST problem with equal release dates and for the FST problem with arbitrary release dates, respectively.

TABLE 1
Summary of complexity results

p_j arbitrary	VST		FST	
	r_j arbitrary	r_j equal	r_j arbitrary	r_j equal
q_j arbitrary	NP-hard (§ 2)	NP-hard (§ 2)	NP-hard (§ 3)	$O(n)$ (§ 3)
$q_j = 1$	Open	$O(n \log n)$ (§ 4)	$O(n \log n)$ (§ 5)	$O(n)$ (§ 3)

These results represent an almost complete complexity classification of the problem class under consideration, as demonstrated by Table 1.

2. NP-hardness of the VST problem.

THEOREM 1. *The* VST *problem is* NP-*hard in the strong sense, even if all release dates are equal.*

Our proof holds for the case that $C^f/C^s = 3$ and $p_j/q_j = 3, j = 1, \cdots, n$. Theorem 1 dominates a previous result, stating that the VST problem is NP-hard in the strong sense if the release dates are arbitrary, C^f/C^s is an arbitrary constant between 1 and 7, and $p_j/q_j = 4, j = 1, \cdots, n$ [17].

Proof of Theorem 1. We have to show that a problem which is known to be NP-complete in the strong sense is (pseudopolynomially) reducible to the VST problem. Our starting point will be the following problem [5, p. 224, [SP15]]:

3-PARTITION: Given a set $S = \{1, \cdots, 3t\}$ and positive integers $a_1, \cdots, a_{3t}, b$ with $\frac{1}{4}b < a_j < \frac{1}{2}b$, $j \in S$, and $\sum_{j \in S} a_j = tb$, does there exist a partition of S into t disjoint 3-element subsets $S_1, \cdots, S_t$ such that $\sum_{j \in S_i} a_j = b$, $i = 1, \cdots, t$?

Given any instance of 3-PARTITION, we construct, in (pseudo-) polynomial time, a corresponding instance of the VST problem with equal release dates as follows:

1. The cost coefficients are defined by $C^s = 1, C^f = 3$.
2. There are $4t$ tasks:

$$a\text{-tasks } T_j^a, j \in S, \qquad \text{with } r_j^a = 0, p_j^a = 6a_j, q_j^a = 2a_j,$$

$$b\text{-tasks } T_i^b, i \in \{1, \cdots, t\}, \quad \text{with } r_i^b = 0, p_i^b = 3b, q_i^b = b.$$

We claim that 3-PARTITION has a solution if and only if there exists a feasible schedule with total cost at most $C^* = 3t$.

Suppose that 3-PARTITION has a solution $\{S_1, \cdots, S_t\}$. It is possible to construct a feasible schedule for all tasks on t fast machines $M_1^f, \cdots, M_t^f$ as follows (cf. Fig. 1): for each $i \in \{1, \cdots, t\}$, machine M_i^f processes the three tasks $T_j^a, j \in S_i$, in nondecreasing order of q_j^a value in the interval $[0, 2b]$, and the task T_i^b in $[2b, 3b]$; note that the starting time of each task falls within the required interval. The total cost of this schedule is equal to $tC^f = C^*$.

Conversely, suppose that there exists a feasible schedule with total cost at most $C^* = 3t$. No slow machine can process more than one task. No fast machine can process more than four tasks, since the completion time of the fourth task will be larger than $2b$ and the starting time of a fifth task should be no larger than $2b$. Let there be m^s slow machines and m^f fast machines. We have, by the hypothesis, $m^s + 3m^f \leqq 3t$ and, by the above arguments, $m^s \geqq 4t - 4m^f$. The first inequality implies that $m^f \leqq t$ and the two together imply that $m^f \geqq t$. We conclude that $m^f = t$. It follows that there are no slow machines and t fast machines, each processing four tasks.

Instance of 3-PARTITION:

t = 3; b = 25;

j	1	2	3	4	5	6	7	8	9
a_j	7	7	7	8	8	8	9	10	11

Solution: {{1,2,9},{3,4,8},{5,6,7}}

Corresponding VST schedule on t fast machines:

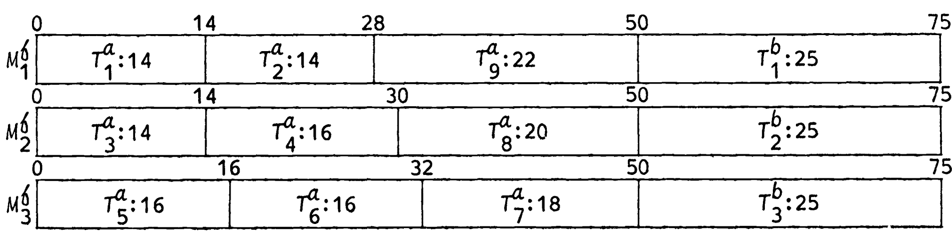


FIG. 1. *Illustration of the transformation in Theorem* 1.

None of these fast machines can process more than one b-task, since otherwise the completion time of the fourth task would be larger than 3b. It follows that the ith fast machine processes exactly one b-task and three a-tasks $T_j^a, j \in S_i$, with $\sum_{j \in S_i} q_j^a \leqq 2b$. Since $\sum_{j \in S} q_j^a = 2tb$, we have $\sum_{j \in S_i} q_j^a = 2b$, $i = 1, \cdots, t$. The collection $\{S_1, \cdots, S_t\}$ constitutes a solution to 3-PARTITION. □

3. NP-hardness of the FST problem.

THEOREM 2. *The* FST *problem is* NP-*hard in the strong sense.*

THEOREM 3. *The* FST *problem is solvable in $O(n)$ time if all release dates are equal.*

Our NP-hardness proof holds for the case that $C^f/C^s = (t+2)/(t+1)$ and $p_j/q_j = z, j = 1, \cdots, n$, where t and z are input variables. Theorem 2 is still true if C^f/C^s is an arbitrary constant between 2 and 3 and $p_j/q_j = 2, j = 1, \cdots, n$ [16]; the proof of this further refinement is quite involved. Theorem 3 shows that the NP-hardness result cannot be extended to the case of equal release dates, unless $\mathcal{P} = \mathcal{NP}$.

Proof of Theorem 2. We will start from the following strongly NP-complete problem [5, p. 224, [SP17]]:

NUMERICAL MATCHING WITH TARGET SUMS. Given a set $S=\{1,\cdots,t\}$ and positive integers $a_1,\cdots,a_t$, $b_1,\cdots,b_t$, $c_1,\cdots,c_t$ with $\sum_{i\in S}(a_i+b_i)=\sum_{i\in S}c_i$, do there exist permutations α and β of S such that $a_{\alpha(i)}+b_{\beta(i)}=c_i$, $i\in S$?

We may assume without loss of generality that $a_1<\cdots<a_t$, $b_1<\cdots<b_t$ and $c_1<\cdots<c_t$. Further, we will assume that for any instance of this problem there exists a positive integer z such that

$$z<a_1<\cdots<a_t<2z<b_1<\cdots<b_t<3z<c_1<\cdots<c_t<5z.$$

(If this does not hold, then define $z=\max\{a_t+1, b_t+1\}$ and set $a_i\leftarrow a_i+z$, $b_i\leftarrow b_i+2z$, $c_i\leftarrow c_i+3z$, $i\in S$.) We will use the notation $S'=\{1,\cdots,t-1\}$.

Given any instance of NUMERICAL MATCHING WITH TARGET SUMS we construct, in (pseudo-) polynomial time, a corresponding instance of the FST problem as follows:

1. The cost coefficients are defined by $C^s=t+1$, $C^f=t+2$.
2. There are $2t^2+t$ tasks:

a-tasks T_i^a, $i\in S$,	with $r_i^a=0$,	$p_i^a=za_i$,	$q_i^a=a_i$,
b-tasks T_{hi}^b, $h\in S$, $i\in S$,	with $r_{hi}^b=a_h$,	$p_{hi}^b=zb_i$,	$q_{hi}^b=b_i$,
c-tasks T_i^c, $i\in S$,	with $r_i^c=c_i$,	$p_i^c=3z^3$,	$q_i^c=3z^2$,
d-tasks T_{hi}^d, $h\in S'$, $i\in S$,	with $r_{hi}^d=2z+zb_i$,	$p_{hi}^d=z^3$,	$q_{hi}^d=z^2$.

We claim that NUMERICAL MATCHING WITH TARGET SUMS has a solution if and only if there exists a feasible schedule with total cost at most $C^*=t^3+t^2+t$.

Suppose that the matching problem has a solution (α,β). It is possible to construct a feasible schedule for all tasks on t fast machines M_i^f, $i\in S$, and t^2-t slow machines M_{hi}^s, $h\in S'$, $i\in S$, as follows (cf. Fig. 2): for each $i\in S$, machine M_i^f processes the tasks $T_{\alpha(i)}^a$, $T_{\alpha(i)\beta(i)}^b$, T_i^c in the intervals $[0, a_{\alpha(i)}]$, $[a_{\alpha(i)}, a_{\alpha(i)}+b_{\beta(i)}]$, $[c_i, c_i+3z^2]$ (note that $a_{\alpha(i)}+b_{\beta(i)}=c_i$), and each of the $t-1$ machines M_{hi}^s, $h\in S'$, processes one of the $t-1$ tasks T_{hi}^b, $h\in S-\{\alpha(\beta^{-1}(i))\}$, in $[a_h, a_h+zb_i]$ and one of the $t-1$ tasks T_{hi}^d, $h\in S'$, in $[2z+zb_i, 2z+zb_i+z^3]$ (note that $a_h<2z$). The total cost of this schedule is equal to $tC^f+(t^2-t)C^s=C^*$.

Conversely, suppose that there exists a feasible schedule with total cost at most C^*. We make the following propositions.

PROPOSITION 1. *Two a-tasks are not assigned to the same machine.*

Proof. Each a-task is processed during the interval $[0, z]$.

PROPOSITION 2. *Two b-tasks are not assigned to the same machine.*

Proof. Each b-task is processed during the interval $[2z, 3z]$.

PROPOSITION 3. *Two c- or d-tasks are not assigned to the same machine.*

Proof. Each c- or d-task is processed during the interval $[3z^2+z, 3z^2+3z]$.

PROPOSITION 4. *An a-task and a b-task are not assigned to the same slow machine.*

Proof. On a slow machine, each a- or b-task is processed during the interval $[2z-1, z^2+z]$.

PROPOSITION 5. *A b-task and a c-task are not assigned to the same slow machine.*

Proof. On a slow machine, each b- or c-task is processed during the interval $[5z-1, 2z^2+2z+1]$.

All tasks are assigned to at most t^2 machines, since $(t^2+1)C^s>C^*$. Propositions 1, 2 and 3 imply that there are exactly t^2 machines, each processing at most one a-task, exactly one b-task and exactly one c- or d-task. These machines include at

Instance of NUMERICAL MATCHING WITH TARGET SUMS:

$t = 3$; $z = 4$;

i	1	2	3
a_i	5	6	7
b_i	9	10	11
c_i	14	16	18

Solution:

$\alpha(i)$	1	2	3
$\beta(i)$	1	2	3

Corresponding FST schedule on t fast machines and t^2-t slow machines:

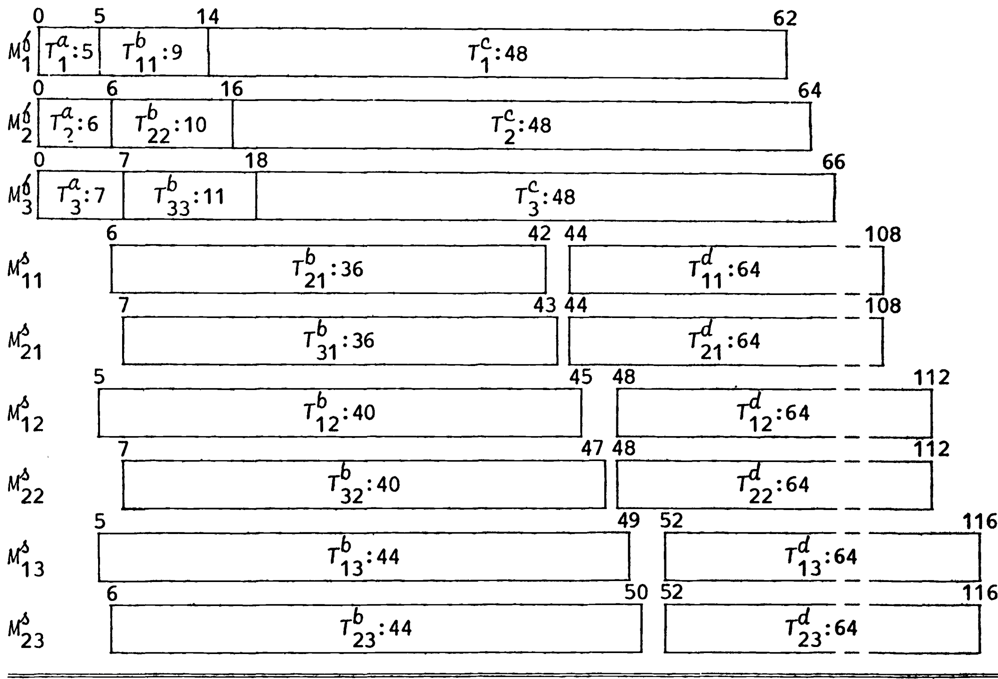


FIG. 2. *Illustration of the transformation in Theorem* 2.

most t fast ones, since $(t^2-t-1)C^s+(t+1)C^f > C^*$. Propositions 4 and 5 imply that there are exactly t fast machines, each processing one a-task, one b-task and one c-task; hence, there are exactly t^2-t slow machines, each processing one b-task and one d-task.

We denote the t fast machines by M_i^f, $i \in S$, and the t^2-t slow machines by M_{hi}^s, $h \in S'$, $i \in S$. It may be assumed that T_i^c is assigned to M_i^f, $i \in S$, and T_{hi}^d to M_{hi}^s, $h \in S'$, $i \in S$. There exists a permutation α of S such that $T_{\alpha(i)}^a$ is assigned to M_i^f, $i \in S$.

Let us define the size of T_{hi}^b as b_i, its processing time on a fast machine. The size of a b-task on M_i^f is at most $c_i - a_{\alpha(i)}$, and the size of a b-task on M_{hi}^s is at most $\lfloor (2z+zb_i-a_1)/z \rfloor = b_i$. The sum of these upper bounds over all machines is equal to $\sum_{i\in S}(c_i-a_{\alpha(i)})+\sum_{h\in S',\, i\in S} b_i = t\sum_{i\in S} b_i$, which is the total size of all b-tasks. It follows that all these upper bounds are actually achieved. More explicitly, for each $i \in S$, there exists an index $\beta(i) \in S$ such that $T_{\alpha(i)\beta(i)}^b$ is assigned to M_i^f, and there exists an index $\gamma(i) \in S$ such that the $t-1$ tasks T_{hi}^b, $h \in S-\{\gamma(i)\}$, are assigned to the $t-1$ machines

M^s_{hi}, $h \in S'$, while $T^b_{\gamma(i)i}$ is assigned to a fast machine. This implies that the functions β and γ are permutations of S with $\gamma(\beta(i)) = \alpha(i)$, $i \in S$.

Since $T^b_{\alpha(i)\beta(i)}$ leaves no idle time between $T^a_{\alpha(i)}$ and T^c_i on M^f_i, we have $a_{\alpha(i)} + b_{\beta(i)} = c_i$, $i \in S$. The pair (α, β) constitutes a solution to the matching problem. □

Proof of Theorem 3. In the FST problem with equal release dates, each task has to start at the same time and therefore each machine can process at most one task. It follows that an optimal schedule uses n slow machines and has total cost nC^s. It is constructed in $O(n)$ time. □

4. A well-solvable case of the VST problem.

THEOREM 4. *In the case that* $q_j = 1, j = 1, \cdots, n$, *the* VST *problem is solvable in* $O(n \log n)$ *time if all release dates are equal.*

The complexity of the VST problem with all $q_j = 1$ and arbitrary release dates remains unresolved (cf. Table 1).

Proof of Theorem 4. In the VST problem with equal release dates, a slow machine can process at most one task but a fast machine may be able to process more than one.

Let us assume that there are m fast machines, with $0 \leqq m \leqq n$, and let X_m denote the maximum number out of the n unit-time tasks that can be completed in time on these machines. A schedule using m fast machines has to use $n - X_m$ slow machines; its total cost is equal to $C_m = mC^f + (n - X_m)C^s$. It follows that an optimal schedule has total cost $\min_{0 \leqq m \leqq n}\{C_m\}$.

For each given value of m, the number X_m and a corresponding schedule on m fast machines can be found by an $O(n \log n)$ algorithm from Lawler [14], [7, p. 295]. Straightforward application of this algorithm for $m = 0, \cdots, n$ would yield an overall optimal schedule in $O(n^2 \log n)$ time.

However, all $X_0, \cdots, X_n$ together can be determined by an $O(n \log n)$ algorithm, which constructs a schedule on n fast machines with the property that, for any value of m, the partial schedule on the first m machines is an optimal schedule on m machines [13]. This algorithm considers the tasks in order of nondecreasing deadlines and assigns each task to the machine with lowest index on which it can be completed in time. A formal statement is as follows.

VST ALGORITHM (*only fast machines, all* $q_j = 1$, *all* $r_j = 0$)

Initialize. Reorder the tasks in such a way that $d_1 \leqq \cdots \leqq d_n$; set $d_0 \leftarrow -\infty$. Introduce an array x of size n and set $x_m \leftarrow 0$, $m = 1, \cdots, n$ [x_m tasks have been assigned to machine M^f_m].
Introduce an array μ of size n [T_j will be assigned to $M^f_{\mu_j}$]. Set $m \leftarrow 1$.

Iterate. **for** $j \leftarrow 1$ **to** n **do**
begin
set $m \leftarrow$ **if** $d_{j-1} < d_j$ **then** 1 **else if** $x_m < d_j$ **then** m **else** $m+1$;
set $\mu_j \leftarrow m$, $x_m \leftarrow x_m + 1$
end.

Finalize. Set $X_0 \leftarrow 0$; **for** $m \leftarrow 1$ **to** n **do** set $X_m \leftarrow X_{m-1} + x_m$.

It can be shown that X_m is the maximum number of tasks that can be completed in time on m fast machines, for $m = 0, \cdots, n$ [13]. The algorithm requires $O(n \log n)$ time to order the tasks, and $O(n)$ time to construct the schedule and to determine the values $X_0, \cdots, X_n$. It follows that an overall optimal schedule is obtained in $O(n \log n)$ time. □

Note. Since $x_m \geqq x_{m+1}$, $m = 1, \cdots, n-1$, X_m is a concave function of m, so that C_m is convex. A similar observation will be exploited in the next section.

5. A well-solvable case of the FST problem.

THEOREM 5. *In the case that* $q_j = 1, j = 1, \cdots, n$, *the* FST *problem is solvable in* $O(n \log n)$ *time.*

The assumption that all $q_j = 1$ is too strong: an analysis of the proof below shows that our algorithm is applicable in the more general situation that the q_j are bounded from above by the minimum length of the interval between two different adjacent release dates. Although this restriction still limits the practical value of our result, we feel that the insight gained might be useful in the design of approximation algorithms for the general FST problem.

Proof of Theorem 5. The development of our algorithm will proceed along the same lines as in the previous section. First, we will assume that there are m fast machines and we will determine an optimal set of tasks to be scheduled on these machines. Next, we will compute the minimum number of slow machines needed to execute the remaining tasks. Finally, we will describe an efficient method to find the optimal value of m.

We start by representing the problem data in a convenient way. Suppose that the release dates assume k different values $\bar{r}_1, \cdots, \bar{r}_k$ with $\bar{r}_1 < \cdots < \bar{r}_k$. For $j = 1, \cdots, k$, there are n_j tasks $T_{1j}, \cdots, T_{n_j j}$ with release dates $r_{ij} = \cdots = r_{n_j j} = \bar{r}_j$ and deadlines $d_{1j} \geqq \cdots \geqq d_{n_j j}$. We have $n = \sum_{j=1}^{k} n_j$ and define $n' = \max_{1 \leqq j \leqq k} \{n_j\}$. This representation can be obtained by sorting the release dates and the deadlines in $O(n \log n)$ time and applying a bucket sort [1] to order the tasks with the same release date according to deadlines in $O(n)$ time.

Let us now assume that there are m fast machines $M_1^f, \cdots, M_m^f$, with $0 \leqq m \leqq n'$. For $j = 1, \cdots, k$, each of these machines can process exactly one of the tasks $T_{1j}, \cdots, T_{n_j j}$. It is obviously optimal to assign T_{ij} to M_i^f for $j = 1, \cdots, k$ and $i = 1, \cdots, \min\{n_j, m\}$, so that the remaining tasks will be as short as possible. Let $\mathcal{T}_m$ denote the set of tasks that are not assigned to the m fast machines, where $\mathcal{T}_0 = \{T_1, \cdots, T_n\}$ and $\mathcal{T}_{n'} = \varnothing$, and let l_m denote the minimum number of slow machines needed to execute these tasks. A schedule using m fast machines has total cost $C_m = mC^f + l_m C^s$. It follows that an optimal schedule uses m^* fast machines, where $C_{m^*} = \min_{0 \leqq m \leqq n'} \{C_m\}$.

For each given value of m, the number l_m and a corresponding schedule of the tasks in $\mathcal{T}_m$ on l_m slow machines can be found in $O(n \log n)$ time. This problem has already been discussed in the first two paragraphs of § 1. The following algorithm is a slight modification of the channel assignment algorithm of Gupta, Lee and Leung [8]; for simplicity, it is stated for the case that $m = 0$.

FST ALGORITHM (*only slow machines*)

Initialize. Reorder the tasks in such a way that $r_1 \leqq \cdots \leqq r_n$; determine a permutation δ of $\{1, \cdots, n\}$ such that $d_{\delta(1)} \leqq \cdots \leqq d_{\delta(n)}$. Introduce a stack S of size n and push machine indices $1, \cdots, n$ onto S in such a way that m is on top of $m+1$, $m = 1, \cdots, n-1$. Introduce an array λ of size n [T_j will be assigned to $M^s_{\lambda_j}$]. Set $j \leftarrow 1$, $i \leftarrow 1$.

Iterate. **while** $j \leqq n$ **do**
if $r_j < d_{\delta(i)}$
then begin set $\lambda_j \leftarrow$ top element of S; pop S; set $j \leftarrow j+1$ **end**
else begin push $\lambda_{\delta(i)}$ onto S; set $i \leftarrow i+1$ **end**.

Finalize. Set $l_0 \leftarrow \max_{1 \leqq j \leqq n} \{\lambda_j\}$.

It can be shown that l_0 is the minimum number of slow machines needed to execute all tasks. The algorithm requires $O(n \log n)$ time to order the tasks, and $O(n)$

time to construct the schedule and to compute the value l_0. Since the release dates and the deadlines have already been sorted, each application of this algorithm requires only $O(n)$ time. Straightforward computation of l_m for $m=0,\cdots,n'$ would yield an overall optimal schedule in $O(n\log n+n'n)=O(n^2)$ time.

However, it will be shown below that C_m is a convex function of m, and this property can be exploited to arrive at an $O(n\log n)$ algorithm. The convexity of C_m implies that, if $C_m<C_{m+1}$, then $m^*\in\{0,\cdots,m\}$, and else $m^*\in\{m+1,\cdots,n'\}$. Thus, m^* can be found by a bisection search as follows: for $m=\lfloor\frac{1}{2}n'\rfloor$, compute C_m and C_{m+1}, reduce the domain of m^* by a factor of two by eliminating either $[0,m]$ or $[m+1,n']$, and repeat the procedure on the remaining interval. The optimal value of m is found in at most $\lceil\log_2(n'+1)\rceil$ iterations.

The entire algorithm requires $O(n\log n)$ time to sort the release dates and the deadlines and, for each of $O(\log n')$ values of m, $O(n)$ time to compute C_m. It follows that an overall optimal schedule is obtained in $O(n\log n)$ time.

It remains to be shown that C_m is a convex function of m. Since $C_m=mC^f+l_mC^s$, we have to prove that l_m is convex, or equivalently that

$$l_{m-1}-l_m\geqq l_m-l_{m+1},\qquad m=1,\cdots,n'-1. \tag{1}$$

We define the *degree of overlap* of the set $\mathcal{S}$ at time t as the number of tasks $T_j\in\mathcal{S}$ such that $t\in[r_j,d_j)$. Let $X_m(t)$ denote the degree of overlap of $\mathcal{T}_m$ at t and $x_{m-1}(t)$ the degree of overlap of $\mathcal{T}_{m-1}-\mathcal{T}_m$ at t, i.e., $X_{m-1}(t)=x_{m-1}(t)-X_m(t)$. It is known [9] that

$$l_m=\max_t\{X_m(t)\},\qquad m=0,\cdots,n'. \tag{2}$$

Since the number of tasks $T_j\in\mathcal{T}_{m-1}-\mathcal{T}_m$ and the lengths of their intervals $[r_j,d_j)$ do not increase as m increases, it is also true that

$$x_{m-1}(t)\geqq x_m(t)\quad \text{all } t,\quad m=0,\cdots,n'-1. \tag{3}$$

Defining t_m such that $X_m(t_m)=\max_t\{X_m(t)\}$, $m=0,\cdots,n'$, and applying (2), we rewrite (1) as

$$X_{m-1}(t_{m-1})-X_m(t_m)\geqq X_m(t_m)-X_{m+1}(t_{m+1}).$$

We have for the left-hand side that

$$X_{m-1}(t_{m-1})-X_m(t_m)=X_{m-1}(t_{m-1})-X_{m-1}(t_m)+x_{m-1}(t_m)\geqq x_{m-1}(t_m).$$

Similarly, we have for the right-hand side that

$$X_m(t_m)-X_{m+1}(t_{m+1})=X_{m+1}(t_m)+x_m(t_m)-X_{m+1}(t_{m+1})\leqq x_m(t_m).$$

Application of (3) for $t=t_m$ now implies the validity of (1). This completes the proof of Theorem 5. □

Note. By means of ingenious counting techniques, the above algorithm for computing a single value l_m can be extended to an $O(n\log n)$ algorithm for computing all $l_0,\cdots,l_n$, together [13]; when the data have already been sorted, it requires only $O(n)$ time, as before. A similar result has been used in the previous section.

Acknowledgment. The authors gratefully acknowledge constructive suggestions by B. J. Lageweg.

REFERENCES

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

[2] G. B. DANTZIG AND D. R. FULKERSON, *Minimizing the number of tankers to meet a fixed schedule*, Naval Res. Logist. Quart., 1 (1954), pp. 217–222.

[3] L. R. FORD, JR. AND D. R. FULKERSON, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962

[4] M. R. GAREY AND D. S. JOHNSON, *"Strong" NP-completeness results: motivation, examples and implications*, J. Assoc. Comput. Mach., 25 (1978), pp. 499–508.

[5] ———, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, 1979.

[6] I. GERTSBAKH AND H. I. STERN, *Minimal resources for fixed and variable job schedules*, Oper. Res., 26 (1978), pp. 68–85.

[7] R. L. GRAHAM, E. L. LAWLER, J. K. LENSTRA AND A. H. G. RINNOOY KAN, *Optimization and approximation in deterministic sequencing and scheduling: A survey*, Ann. Discrete Math., 5 (1979), 287–326.

[8] U. I. GUPTA, D. T. LEE AND J. Y.-T. LEUNG, *An optimal solution for the channel-assignment problem*, IEEE Trans. Comput., C-28 (1979), pp. 807–810.

[9] A. HASHIMOTO AND J. E. STEVENS, *Path cover of acyclic graphs*, ILLIAC IV, Document 239, University of Illinois, Urbana, IL, 1970.

[10] ———, *Wire routing by optimizing channel assignment within large apertures*, in Proc. 8th Design Automation Workshop (1971), pp. 155–169.

[11] R. M. KARP, *Reducibility among combinatorial problems*, in Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–103.

[12] B. W. KERNIGHAN, D. G. SCHWEIKERT AND G. PERSKY, *An optimum channel-routing algorithm for polycell layouts of integrated circuits*, in Proc. 10th Design Automation Workshop, 1973, pp. 50–59.

[13] B. J. LAGEWEG, Personal communication, 1980.

[14] E. L. LAWLER, *Sequencing to minimize the weighted number of tardy jobs*, RAIRO Inform., 10 (1976), 5 Suppl., pp. 27–33.

[15] J. K. LENSTRA AND A. H. G. RINNOOY KAN, *Complexity of vehicle routing and scheduling problems*, Networks, 11 (1981), pp. 221–227.

[16] K. NAKAJIMA, *On nonpreemptive multiprocessor scheduling with discrete starting times*, Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL, 1980.

[17] K. NAKAJIMA AND S. L. HAKIMI, *On the NP-completeness of a real-time scheduling problem with two types of machines*, in Proc. 17th Allerton Conf. Communication, Control, and Computing, University of Illinois, Urbana, IL, 1979, pp. 652–658.

SIAM J. COMPUT.
Vol.11, No. 3, August 1982

© 1982 Society for Industrial and Applied Mathematics

0097-5397/82/1103-0010 $01.00/0

COMPLETIONS OF PARTIALLY ORDERED SETS*

BERNHARD BANASCHEWSKI† AND EVELYN NELSON†

Abstract. We show, for any subset system Z (as defined in Wright, Wagner, and Thatcher, T.C.S. 7 (1978), pp. 57–77) and any order preserving map $f: Q \to P$ of posets, the existence of a universal map $u_f: P \to P_f$ where P_f is Z-complete and $u_f f$ is Z-continuous. This generalizes to arbitrary subset systems the result of Markowsky (T.C.S. 4 (1977), pp. 125–135) for chains, and the completions of Wright, Wagner, and Thatcher for union complete Z; our method, different from theirs, uses the time-honored direct construction of universal maps. Further, we obtain some results on the internal structure of P_f with regard to Z-joins. Finally, we show that each element of the Z-completion of P is a Z-join of elements of P iff Z is union complete.

Key words. partially ordered set, completion, subset system Z, Z-complete, Z-completion, Z-continuous maps

In recent years, partially ordered sets (posets) have become increasingly important to computer science. Of particular significance in this context are those posets which have the kind of completeness properties that reflect general notions of approximation (Markowsky [7], [8], Markowsky and Rosen [9], Wright, Wagner and Thatcher [12]). In the absence of the desired completeness for a given poset P, one often considers extensions $Q \supseteq P$ which remedy this lack, preferably in a universal way. An additional requirement might be that Q should rectify the join deficiencies of P in a computable fashion.

The original result of this type is due to Markowsky [8] who shows that, for any order preserving map $f: Q \to P$ between posets, there exists a universal map $u_f: P \to P_f$ where P_f is *chain*-complete and the composite $u_f f: Q \to P_f$ is *chain*-continuous. Following this, Wright, Wagner and Thatcher [12] introduce a more general concept of completeness and continuity, for posets and maps between them, in which their notion of *subset system* Z replaces the collection of all chains. They prove the existence of a universal Z-complete extension $\bar{P} \supseteq P$, amounting to the counterpart of Markowsky's result in the special case where (i) Q has the same elements as P and is discretely ordered, and f maps the elements of Q identically, and (ii) the subset system Z is union complete. It should be noted that, even under the hypothesis (i), this does not generalize Markowsky's case because the subset system given by all chains is not union complete. However, Meseguer [10] subsequently showed that every subset system Z is equivalent to a union complete one, thereby establishing, for arbitrary Z and the maps as in (i), the existence of a universal map $u_f: P \to P_f$ where P_f is Z-complete and $u_f f: Q \to P$ is Z-continuous.

In this paper, we prove the complete generalization of Markowsky's original result to arbitrary subset systems Z, i.e., the existence of a universal map $u_f: P \to P_f$ as above for *all* posets Q and *all* order preserving maps $f: Q \to P$ (Proposition 1). Our proof is entirely independent of all previous ones, substantially shorter, and much more direct; it uses the construction of universal maps which has long been familiar in algebra and topology (e.g., Bourbaki [4, pp. 43–50]).

Besides the existence of the universal maps $u_f: P \to P_f$, we establish a number of results showing how the elements of P_f are related to the image of u_f in terms of joins, either of Z-sets or of arbitrary sets (Proposition 2). In particular, for $f: Q \to P$ with trivial ZQ, we obtain a characterization of the image of u_f in terms of Z-joins.

* Received by the editors August 8, 1980, and in final revised form September 25, 1981.

† Department of Mathematical Sciences, McMaster University, Hamilton, Ontario, Canada L85 4K1.

Further, as an application of these properties, we obtain a solution to an obvious question, implicit in [12] and highlighted by Meseguer [10]: we show that all the elements of the Z-completions $\bar{P} \supseteq P$ of posets P are actually reached as joins of Z-sets in P if and only if Z is union complete (Proposition 3). Thus, union completeness is precisely the property which ensures that the new elements are all, in a sense, computable from the originally given ones.

The paper concludes with a number of observations concerning the category of Z-complete posets and Z-continuous maps. In particular, we indicate how the results of Meseguer [10] on the tensor multiplication in this category follow from Banaschewski–Nelson [2].

Recall that a *subset system* Z assigns to each poset P a collection ZP of subsets of P such that, for any order preserving map $f: P \to Q$, the direct image $f(A) = \{f(a) | a \in A\}$ belongs to ZQ for each $A \in ZP$. Obvious examples of such Z are given by the following specifications of ZP for each P (see also [12]):

(1) all subsets of P;
(2) all subsets of P which are bounded above in P;
(3) all subsets A of P for which every pair in A has an upper bound in P;
(4) all (up) directed subsets of P;
(5) all chains in P;

and the variants of these obtained by additional cardinality restriction.

Also, for any such Z, $Z'P = ZP - \{\varnothing\}$ is again of this type, as are arbitrary unions or intersections of such Z.

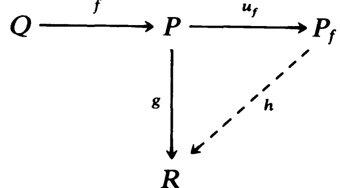
Z is viewed as a parameter of (join) completeness and join preservation as follows: A partially ordered set P is called *Z-complete* if and only if every $A \in ZP$ has a join $\bigsqcup A$ in P, and a map $f: P \to Q$ between posets is called *Z-continuous* if and only if it preserves all existing *Z-joins*, i.e., whenever $A \in ZP$ has a join in P then $f(A)$ has a join in Q and $f(\bigsqcup A) = \bigsqcup f(A)$. We always assume that some ZP contains a set with at least two elements and hence $2 \in Z2$ for the two-element chain 2, which implies that Z-continuous maps are order preserving.

Before proceeding we define some notions which will be used in later arguments. First, for a subset X of a poset P, $\bar{X}$, the *join closure of X in P*, is the set of all joins $\bigsqcup Y$ existing in P for subsets $Y \subseteq X$. X is called *join-closed in P* iff $X = \bar{X}$. Note that

(1) $\operatorname{card}(\bar{X}) \leqq 2^{\operatorname{card}(X)}$,
(2) $\bar{\bar{X}} = \bar{X}$ and
(3) if P is Z-complete then so is $\bar{X}$, and the natural embedding of $\bar{X}$ into P is Z-continuous.

Further, for a subset X of a Z-complete poset P, $\langle X \rangle$, the *Z-join closure of X in P*, is the smallest subset $Y \subseteq P$ such that $X \subseteq Y$, and $\bigsqcup U \in Y$ for each $U \in ZY$, where $\bigsqcup U$ is the join in P. An alternative description of $\langle X \rangle$ is that it is the smallest subset of P containing X which is Z-complete and such that its natural embedding into P is Z-continuous. Consequently it follows from (3) above that $\langle X \rangle \subseteq \bar{X}$.

PROPOSITION 1. *For any order-preserving map $f: Q \to P$ between posets Q and P, there exists a Z-complete poset P_f and an order-preserving map $u_f: P \to P_f$ with $u_f f$*



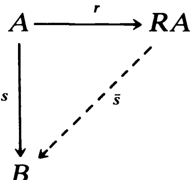
Z-continuous, such that for any order-preserving map $g: P \to R$ with R Z-complete and gf Z-continuous, there exists a unique Z-continuous map $h: P_f \to R$ with $hu_f = g$.

Proof. Let S be a set of representatives of the isomorphism classes of the order-preserving maps $g: P \to P'$ with P' Z-complete, gf Z-continuous and card $(P') \leqq 2^{\text{card}(P)}$. Then for each such g there exists $s: P \to P_s$ in S and an order-isomorphism $k: P_s \to P'$ with $ks = g$.

Let $T = \prod_{s \in S} P_s$ be the product of the codomains (targets) of all the maps in S, with the component-wise ordering; then T is Z-complete, Z-joins in T being formed component-wise, and the projection maps $p_s: T \to P_s$ for $s \in S$ are all Z-continuous. Now, since T is a product, there exists a unique $t: P \to T$ such that $p_s t = s$ for each $s \in S$. Because Z-joins in T are formed component-wise, tf is Z-continuous. Let P_f be the Z-join closure in T of the image of t and let $u_f: P \to P_f$ be the target restriction of t to P_f. Then P_f is Z-complete, and since Z-joins in P_f are just the corresponding joins in T, $u_f f$ is Z-continuous.

Now the usual argument shows that $u_f: P \to P_f$ is the desired map: For any order-preserving $g: P \to R$ with R Z-complete and gf Z-continuous, let R' be the Z-join closure in R of the image of g. Then R' is Z-complete, with Z-joins being formed as they are in R and hence, for the target restriction $g': R \to R'$ of g, $g'f$ is Z-continuous. Also card $(R') \leqq 2^{\text{card}(P)}$ and hence there exists $s: P \to P_s$ in S and an isomorphism $k: P_s \to R'$ with $ks = g'$. Now let $h = ekp_s: T \to R$ where $e: R' \to R$ is the natural embedding; then h is Z-continuous and $hu_f = ekp_s u_f = eks = eg' = g$ as required. The uniqueness of h follows from the fact that any two Z-continuous maps with common domain and codomain which coincide on some set also coincide on the Z-join closure of that set. □

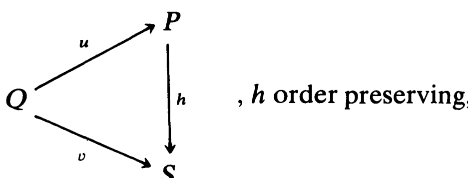
Remark 1. This proposition provides an example of the categorical notion of *reflection*. In general terms, if $\mathbb{L}$ is a subcategory of $\mathbb{K}$ and A an object of $\mathbb{K}$ then a reflection of A in $\mathbb{L}$ is an object RA in $\mathbb{L}$ together with a morphism $r: A \to RA$ which is universal among all morphisms $s: A \to B$, B in $\mathbb{L}$, in the sense that each such s determines a unique $\bar{s}: RA \to B$ for which $s = \bar{s}r$:



In particular, $\mathbb{L}$ is called a *reflective* subcategory of $\mathbb{K}$ iff each object of $\mathbb{K}$ has a reflection in $\mathbb{L}$ (Herrlich–Strecker [6, § 36]). In order to recognize Proposition 1 as a reflectivity assertion, take $\mathbb{K}$ as the category whose objects are all order preserving maps $u: Q \to P$ (for fixed Q) and whose morphisms with domain (source) u and codomain (target) v are the commuting triangles

If $\mathbb{L}$ is now the subcategory of $\mathbb{K}$ consisting of the Z-continuous $u: Q \to P$ with Z-complete P and the triangles (*) with Z-continuous h then direct checking shows

that Proposition 1 expresses the reflectivity of $\mathbb{L}$ in $\mathbb{K}$:



is precisely the reflection of $f: Q \to P$. An immediate consequence of this observation is that, instead of the direct proof given above, we could verify for $\mathbb{L}$ one of the familiar hypotheses known to imply its reflexivity in $\mathbb{K}$. Such hypotheses generally have a first part which here amounts to showing that $\mathbb{L}$ is complete and any limit in $\mathbb{L}$ is also a limit in $\mathbb{K}$, and a second part which expresses a "smallness" condition. As to the latter, the rôle of the set S in the above proof in fact establishes one such, the solution set condition (Herrlich–Strecker [6 § 28]). Alternatively, one could verify the smallness hypothesis of the special adjoint functor theorem [6, § 28.11] for the category $\mathbb{L}$; this would provide the slickest proof of the proposition.

Remark 2. Of particular interest is the case when the above map $u_f: P \to P_f$ is an *embedding*, i.e., $u_f(p) \sqsubseteq u_f(q)$ if and only if $p \sqsubseteq q$ for all $p, q \in P$. This does not always hold. For example, let $f: \omega + 1 \to \omega + 2$ map the elements of ω identically and top to top: if $\omega \in Z(\omega + 1)$ then any map u such that uf is Z-continuous must map the top two elements of $\omega + 2$ the same.

On the other hand, $u_f: P \to P_f$ is indeed an embedding whenever $f: Q \to P$ is itself Z-continuous. To see this, we first recall that there always exists an embedding $g: P \to P'$ such that (i) P' is complete and hence Z-complete, and (ii) g preserves *all* joins existing in P, making it Z-continuous. For example, take P' as the set of all up-sets $A \subseteq P$ (i.e., if $a \in A$ then $x \in A$ for all $x \sqsupseteq a$ in P), ordered by reverse set inclusion, and $g(p) = \{x \mid x \sqsupseteq p\}$. Now, for any such g, gf is also Z-continuous, and hence $g = hu_f$ for some h, making u_f an embedding, too. Consequently, any choice of a Z-continuous map into a given poset P provides a certain *Z-completion*, i.e., Z-complete extension, of P. In particular, one has:

COROLLARY 1. *Any poset P has a Z-completion $\bar{P} \supseteq P$ such that any order preserving map $P \to R$, R Z-complete, uniquely extends to a Z-continuous map $\bar{P} \to R$.*

Proof. Let $|P|$ be the discretely ordered set with the same elements as P and apply Proposition 1 to $f: |P| \to P$ which maps the elements of P identically, using the fact that all maps from a discrete poset are trivially Z-continuous. □

COROLLARY 2. *Any poset P has a Z-completion $\tilde{P} \supseteq P$ such that any Z-continuous map $P \to R$, R Z-complete, uniquely extends to a Z-continuous map $\tilde{P} \to R$. Moreover, the inclusion map $P \to \tilde{P}$ is Z-continuous.*

Proof. Apply Proposition 1 to the identity map $f: P \to P$. □

The above results, for the special case where Z picks out the set of all *chains* in each poset, are given in Markowsky [8]. On the other hand, for the Z which assigns to each poset the collection of *all* its subsets, the completion provided by Corollary 2 is described in Herrlich [5, p. 78], identifying $\tilde{P} \supseteq P$ as the completion given by all down sets S of P (if $x \sqsubseteq y$ and $y \in S$ then $x \in S$).

An explicit construction of the completion $\bar{P} \supseteq P$ is given in Wright, Wagner and Thatcher [12], for union-complete Z, which are defined as follows. Let $\downarrow A = \{x \mid x \sqsubseteq a$ for some $a \in A\}$, and put $\hat{Z}P = \{\downarrow A \mid A \in ZP\}$, partially ordered by set inclusion; then Z is called *union-complete* if and only if $\bigcup A \in \hat{Z}P$ for all $A \in Z(\hat{Z}P)$.

For these Z, $\hat{Z}P$, which is evidently Z-complete, is the completion $\bar{P}$, by the embedding $P \to \hat{Z}P$, which maps x to $\downarrow\{x\}$. Note that without the assumption of union completeness, $\hat{Z}P$ need not be Z-complete. For example, let ZP consist of all chains in P (this Z is not union complete, contrary to [12]). Then, for the poset P of all finite subsets of an uncountable set E, ordered by set inclusion, $\hat{Z}P$ consists of all down sets of P with at most countable union, but the union of a chain in $\hat{Z}P$ need not belong to $\hat{Z}P$; actually $\bar{P}$ is the entire power set of E, as can be seen from Iwamura's lemma (see also Markowsky [7]). On the other hand, Meseguer [10] obtains the completion $\bar{P} \supseteq P$ for arbitrary Z from the construction in [12] by first proving that, for each subset system Z, there is a union-complete $\bar{Z}$ such that Z-completeness and Z-continuity are equivalent to $\bar{Z}$-completeness and $\bar{Z}$-continuity.

Next, we provide some description of the internal structure of P_f for any f. The following notions concerning any poset P will be used for this: An element $x \in P$ is called *Z-join irreducible* if and only if $x = \bigsqcup A$ and $A \in ZP$ implies $x \in A$, and *Z-compact* if and only if $x \sqsubseteq \bigsqcup A$ and $A \in ZP$ implies $x \sqsubseteq a$ for some $a \in A$.

Further, we call ZQ *trivial* for any Q if and only if each $A \in ZQ$ has a top.

PROPOSITION 2. *For any map $f: Q \to P$ and the corresponding $u_f: P \to P_f$ if $I \subseteq P_f$ is the image of u_f then*

(1) *P_f is the Z-join closure of I in P_f.*

(2) *P_f is the join closure of I.*

(3) *Every Z-join irreducible element of P_f belongs to I.*

Further, if ZQ is trivial then

(4) *Every element of I is Z-compact.*

Proof. (1) follows directly from the proof of Proposition 1.

(2) follows directly from (1).

(3) The subposet of P_f consisting of I and all Z-join reducible elements of P_f is closed under Z-joins and hence equal to P_f by (1).

(4) For any $c \in P$, the map $g: P \to 2$ such that $g(x) = 1$ if and only if $x \sqsubseteq c$ is order preserving and gf is Z-continuous since ZQ is trivial. Hence there exists a unique Z-continuous map $h: P_f \to 2$ such that $hu_f = g$. Now, by (1), $P_f = \{x \mid u_f(c) \sqsubseteq x \text{ or } h(x) = 0\}$ since the latter set contains I and is closed under Z-joins. Consequently, if $u_f(c) \sqsubseteq \bigsqcup A$ for some $A \in ZP_f$ then $1 = h(\bigsqcup A) = \bigsqcup h(a)$ $(a \in A)$ and thus $h(a) = 1$ for some $a \in A$, which means $u_f(c) \sqsubseteq a$ by the previous sentence. This shows $u_f(c)$ is Z-compact. □

The following consequence of (3) and (4) in this proposition characterizes the image of u_f inside P_f for certain $f: Q \to P$. It may be viewed as the complete version of which Proposition 3.7 in Wright, Wagner and Thatcher [12] is the forerunner.

COROLLARY 1. *For any map $f: Q \to P$. if ZQ is trivial then the image of u_f consists exactly of the Z-join irreducible elements of P_f, and further, in P_f, Z-join irreducibility and Z-compactness coincide.*

Proof. Since Z-compactness trivially implies Z-join irreducibility, (4) implies that every element of the image of u_f is Z-join irreducible, and (3) then yields the first part of the corollary. In addition, (3) and (4) together show that Z-join irreducibility implies Z-compactness. □

There are situations in which one is interested in Z-complete posets but only Z'-continuous maps, where $Z'P = ZP - \{\varnothing\}$ (Markowsky and Rosen [9]). The following shows that, for appropriate P, the universal map $u_f: P \to P_f$ relative to Z' actually has the corresponding universality property. For the Z which picks out the chains, this type of completion is explicitly constructed in [9], and Markowsky [8] derives it from the corresponding Z-completions.

COROLLARY 2. *For any map $f:Q\to P$ and the associated map $u_f:P\to P_f$ relative to Z', if P has a bottom $\perp$ then P_f is Z-complete.*

Proof. Since P_f is the join closure of the image of u_f, $u_f(\perp)$ is the bottom of P_f. □

Remark. The following observation shows that (4) in the above proposition indeed does not hold in general: if P is Z-complete then $P=\tilde{P}$, the completion given in Corollary 2 of Proposition 1, and hence (4) fails unless all elements of P are Z-compact.

Our last proposition characterizes the desirable property of completions, mentioned in the introduction, that all elements are actually obtained as Z-joins from the given poset.

PROPOSITION 3. *The following are equivalent for any subset system Z:*

(1) *Z is union-complete.*

(2) *For any $f:Q\to P$ with trivial ZQ and the associated $u_f:P\to P_f$, each element of P_f is the join of some $u_f(A)$, $A\in ZP$.*

(3) *For any P, each element of the Z-completion $\bar{P}\supseteq P$ is the join of some $A\in ZP$.*

Proof. (1)⇒(2). Since $\hat{Z}P$ is Z-complete and ZQ trivial, the order preserving map $v:P\to\hat{Z}P$ such that $v(x)=\{y\mid y\sqsubseteq x\}$ determines a unique Z-continuous map $w:P_f\to\hat{Z}P$ such that $wu_f=v$. Further, the Z-completeness of P_f provides the map $s:\hat{Z}P\to P_f$ given by $s(A)=\bigsqcup u_f(A)$. This s is Z-continuous since $B=\bigsqcup Y$, $Y\in Z(\hat{Z}P)$, implies $s(B)=\bigsqcup u_f(B)=\bigsqcup u_f(\bigcup Y)=\bigsqcup_{A\in Y}\bigsqcup u_{f(A)}=s(Y)$. Now, $swu_f=sv=u_f$, hence sw is the identity map on P_f and therefore s is onto.

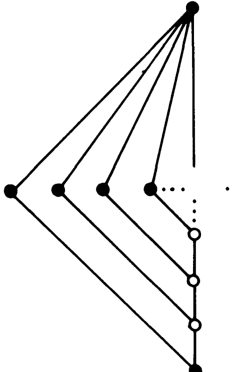
(2)⇒(3). Trivial.

(3)⇒(1). Let $t:\hat{Z}P\to\bar{P}$ be the map such that $t(A)=\bigsqcup A$. Then, t is order preserving, and hence, for any $y\in Z(\hat{Z}P)$, $t(Y)\in Z\bar{P}$ so that $\bigsqcup t(Y)=\bigsqcup\bigcup Y$ exists in $\bar{P}$. Now, by hypothesis, $\bigsqcup\bigcup Y=\bigsqcup B$ for some $B\in ZP$, and by (4) of Proposition 2 it follows that each $a\in\bigcup Y$ is below some $b\in B$. On the other hand, each $b\in B$ is below some $\bigsqcup A\in t(Y)$ and since $A\in\hat{Z}P$ it has a cofinal subset $C\in ZP$; now, again by (4) of Proposition 2, $b\sqsubseteq c$ for some $c\in C$ and hence $b\in A$. This shows that $\bigcup Y\in\hat{Z}P$. □

We conclude with some comments on categorical properties of $Z\mathbb{P}$, the category of Z-complete posets and Z-continuous maps. For this, we first recall that the category $\mathbb{P}$ of posets and *all* order preserving maps is complete and cocomplete. In particular, limits in $\mathbb{P}$ are formed as they are in the category of sets, and it is easy to check that the same holds for $Z\mathbb{P}$, so that $Z\mathbb{P}$ is also complete. Further, the following arguments show that $Z\mathbb{P}$ is cocomplete. The coproduct of a family $(P_i)_{i\in I}$, of Z-complete posets is the completion $\tilde{P}$, described in Corollary 2 of Proposition 1, of the disjoint union P of the $P_i (i\in I)$. The natural embeddings $P_i\to P$ preserve all joins, and the inclusion $P\to\tilde{P}$ is Z-continuous; thus, the embeddings $P_i\to\tilde{P}$ are all Z-continuous. Moreover, if $f_i:P_i\to Q$ are any Z-continuous maps into a Z-complete poset Q, then, because P is the disjoint union of the P_i, there is a unique map $f:P\to Q$ extending all the f_i. Further, since any subset of P which has a join must lie entirely in one of the P_i, this map f is Z-continuous, and hence has a unique Z-continuous extension $\tilde{f}:\tilde{P}\to Q$. It is just as straightforward to show that this $\tilde{f}$ is the only Z-continuous map $\tilde{P}\to Q$ extending all the f_i, and hence $\tilde{P}$ is the required coproduct. Next, concerning coequalizers: for morphisms $f_1,f_2:Q\to R$ in $Z\mathbb{P}$, let $f:R\to P$ be the coequalizer of f_1 and f_2 in $\mathbb{P}$, and let $u_f:P\to P_f$ be the map determined as in Proposition 1. Then P_f is Z-complete, u_ff is Z-continuous, and $u_fff_1=u_fff_2$. Moreover, if $g:R\to P'$ is any Z-continuous map into a Z-complete poset P' with $gf_1=gf_2$, then there is a unique order-preserving $g':P\to P'$ with $g'f=g$. By the properties of u_f, there is a unique Z-continuous $h:P_f\to P'$ with $hu_f=g'$ and hence $hu_ff=g'f=g$. Thus $u_ff:Q\to P_f$ is the coequalizer of f_1 and f_2 in $Z\mathbb{P}$, and this completes the proof that $Z\mathbb{P}$ is cocomplete. In addition, we note that this gives a moderately explicit description of the relevant

constructions. The mere fact of cocompleteness could also be obtained by verifying the smallness hypotheses of the Special Adjoint Functor Theorem for $Z\mathbb{P}$ and then applying the result that any complete category of this sort is also cocomplete (Börger et al. [3], Theorems 2.2 and 2.5).

Regarding injectivity and projectivity: the free Z-complete posets, i.e., the $\bar{P}$ for discretely ordered P, are evidently projective with respect to onto maps, and so every Z-complete poset is the image of a projective one and the projectives are exactly the retracts of free ones. Markowsky [8] shows that there are no nontrivial injectives with respect to monomorphisms in $Z\mathbb{P}$ for the Z which picks out all chains; the same is easily shown for the Z for which ZP consists of all subsets. For some Z, the notion of injectivity with respect to Z-continuous embeddings is more interesting: for example, if all ZP are trivial then $Z\mathbb{P} = \mathbb{P}$, and here the injectives with respect to embeddings are precisely the complete posets (see Banaschewski–Bruns [1]). However, $Z\mathbb{P}$ need *not* have any nontrivial injectives with respect to embeddings. For example, let Z pick out all chains or all countable chains, and let $I \in Z\mathbb{P}$ be any nontrivial injective with respect to embeddings in $Z\mathbb{P}$. Now, as argued in the Remark following Proposition 1, there exists an embedding $e: I \to P$ into a complete P, preserving all joins and hence Z-continuous, and by injectivity there further exists an order preserving map $s: P \to I$ such that se is the identity map on I. This shows I is not discretely ordered, and hence there exist $a, b \in I$ such that $a \sqsubset b$. Let B be the countable poset pictured below, A the subposet represented by the solid discs; then $f: A \to I$ which maps the largest element of A to b and all other elements to a has no extension to a chain-continuous map on B:



Finally, the set $[P, Q]$ of all Z-continuous maps between any Z-complete P and Q, with the usual pointwise partial order, is Z-complete so that $Z\mathbb{P}$ has a functional internal hom-functor in the sense of Banaschewski–Nelson [2]. In addition,by Proposition 4 of [2], $Z\mathbb{P}$ has universal bimorphisms, and Proposition 3 of [2] then shows that these provide a tensor multiplication $\otimes$ for the internal hom-functor in $Z\mathbb{P}$. This means that $[P \otimes Q, R] \cong [P, [Q, R]]$ for any Z-complete P, Q, and R, and this amounts to a λ-conversion for Z-continuous maps. These results are also proved, by somewhat different arguments, in Meseguer [10]. In $\mathbb{P}$, the tensor product is actually the same as the product (Banaschewski–Nelson [2]). In general, by Lemma 1 in Nelson [11], this holds in $Z\mathbb{P}$ if and only if all Z-sets are up-directed.

Acknowledgments. We much appreciate the extensive and patient criticism provided by the referees and some helpful comments from V. Trnkova and J. Reiterman.

REFERENCES

[1] B. BANASCHEWSKI AND G. BRUNS, *Categorical characterization of the MacNeille completion*, Archiv der Math., 18 (1967), pp. 369–377.

[2] B. BANASCHEWSKI AND E. NELSON, *Tensor products and bimorphisms*, Canad. Math. Bull., 19 (1976), pp. 385–402.

[3] R. BÖRGER, W. THOLEN, M. WISCHNEWSKY AND H. WOLFE, *Compact and hypercomplete categories*, J. Pure Appl. Algebra, 21 (1981), pp. 129–143.

[4] N. BOURBAKI, *Theorie des ensembles*, Ch. 4, Structures. Act. Sci. Industr., 1253, Hermann, 1957.

[5] H. HERRLICH, *Topologische Reflexionen und Coreflexionen*, Lecture Notes in Mathematics, 16, Springer, New York, 1968.

[6] H. HERRLICH AND G. STRECKER, *Category Theory*, Allyn and Bacon, Boston, 1973.

[7] G. MARKOWSKY, *Chain-complete posets and directed sets with applications*, Alg. Univ., 6 (1976), pp. 53–68.

[8] G. MARKOWSKY, *Categories of chain-complete posets*, Theoret. Comp. Sci., 4 (1977), pp. 125–135.

[9] G. MARKOWSKY AND B. ROSEN, *Bases for chain-complete posets*, IBM J. Res. Develop., 20 (1976), pp. 138–147.

[10] J. MESEGUER, *Ideal monads and Z-posets*, Manuscript, Berkeley, 1979.

[11] E. NELSON, *Z-continuous algebras*, in Continuous Lattices, B. Banaschewski and R.-E. Hoffmann, eds., Lecture Notes in Mathematics, 871, Springer, New York, 1981.

[12] J. B. WRIGHT, E. G. WAGNER AND J. W. THATCHER, *A uniform approach to inductive posets and inductive closure*, in Mathematical Foundations of Computer Science, G. Goos and H. Hartmanis, eds., Lecture Notes in Computer Science, 53, Springer, New York, 1977, also appeared as T.C.S., 7 (1978), pp. 57–77.

SIAM J. COMPUT.
Vol. 11, No. 3, August 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1103-0011 $01.00/0

THE HAMILTONIAN CIRCUIT PROBLEM IS POLYNOMIAL FOR 4-CONNECTED PLANAR GRAPHS*

D. GOUYOU-BEAUCHAMPS†

Abstract. An algorithm for the determination of an Hamiltonian circuit in a 4-connected planar graph is presented. The timing for this algorithm depends on n^3 (where n is the number of edges in the graph); the storage requirement also depends on n^3. This paper completes the result of Garey, Johnson and Tarjan [SIAM J. Comput., 5 (1976), pp. 704–714] which claims that the problem is NP-complete for 3-connected planar graphs. This algorithm is inspired by the proof of Tutte's theorem which implies the existence of Hamiltonian circuits in 4-connected planar graphs.

Key words. Hamiltonian, planar graph, complexity, bridge, planted plane tree, NP-complete

Introduction. The study of Hamiltonian circuit problems was greatly stimulated by works of Cook [3] and Karp [12] about algorithm complexity in the early 1970's. In fact, this problem is NP-complete (see Corneil [5]). Garey, Johnson and Stockmeyer [7] proved that the more restricted problem of Hamiltonian circuits for graphs with maximum vertex degree 3 was also NP-complete. Garey, Johnson and Tarjan [8] showed that the problem was still NP-complete for cubic 3-connected planar graphs. Then it may be asked whether the Hamiltonian circuit problem is NP-complete for 4-connected planar graphs. Such a question is indeed of interest as we have the result of Tutte [16] which implies the existence of a Hamiltonian circuit for 4-connected planar graphs.

The main purpose of this article is to show that the Hamiltonian problem for 4-connected planar graphs is polynomial. This result enables us to determine more exactly the frontier between the class of NP-complete problems and the class of problems for which a polynomial algorithm is known.

We get this algorithm by using the proof of Tutte's theorem (see Ore [14]). For the computation of the complexity, we use the tree of recursive calls. In § 1, the vertices of this tree are labelled by subsets of the set constituted by edges of the graph. Properties about these labels imply that the number of vertices of the tree depends on n^2 (where n is the number of edges in the graph). In § 2 we give an overview of the proof of Tutte's theorem, by induction on the edges of the graph. Thus we deduce from it a recursive algorithm which is presented in § 3. With the result of § 2, we prove in § 4 that the timing and storage requirements depend on n^3.

1. Preliminary result. To describe an execution of any algorithm, we use a planted plane tree. A *planted plane tree* (also called an ordered tree), is a rooted tree which has been embedded in the plane so that the relative order of subtrees at each branch is part of its structure (see Klarner [13]). In this paper we shall say simply tree instead of planted plane tree, following the custom of computer scientists. If the ordering of the sons of a vertex is $\alpha_1, \alpha_2, \cdots, \alpha_p$ then we say that α_i is on the left of α_j if and only if $i \leqq j$.

We can define a total ordering (called prefix ordering) on S, the set of vertices, as follows:

A vertex α *is on the left of* β if and only if one of two following conditions holds:

- α and β lie on the same branch and the depth of β is greater than that of α.

* Received by the editors February 18, 1980, and in revised form May 11, 1981.

† U.E.R. de Mathematiques et d'Informatique, Université de Bordeaux I, 33405 Talence, France. Present address, Département de Mathématiques, Université de Pau, 64000 Pau, France.

- α and β lie on different branches, s is their minimal common ancestor and the son of s lying on the branches of α is on the left of the son of S lying on the branch of β.

DEFINITION. A *φ-tree A* is a 4-tuple $(T, S, \mathcal{O}, \varphi)$ where:
- T is a tree,
- S is the set of vertices of T,
- $\mathcal{O}$ is a finite set,
- φ is a mapping from S to $\mathcal{P}(\mathcal{O})$-$\{\emptyset\}$ satisfying the 5 following conditions:
  (1) $\varphi(r) = \mathcal{O}$ for the root r of T,
  (2) $\varphi(x) \not\subseteq \varphi(s)$ for any son x of $s \in S$,
  (3) if $F(s)$ denotes the set of sons of s then

$$\bigcup_{x \in F(s)} \varphi(x) = \varphi(s),$$

  (4) for any three sons x_1, x_2 and x_3 of s

$$\varphi(x_1) \cap \varphi(x_2) \cap \varphi(x_3) = \emptyset,$$

  (5) if x_1 and x_2 (x_1 on the left of x_2) belonging to $F(s)$ are such that $\varphi(x_1) \cap \varphi(x_2) \neq \emptyset$ then, for any element 0 of $\varphi(x_1) \cap \varphi(x_2)$, there exist a single pendant vertex y descendant of x_1 such that $0 \in \varphi(y)$.

Let $|X|$ denote the number of elements of a set X. For a given value of $|\mathcal{O}|$, we say that a φ-tree A is *maximal* if it has the greatest possible number of vertices. Figures 1 and 2 give examples of maximal φ-trees for $|\mathcal{O}| = 2$, 3 and 4.

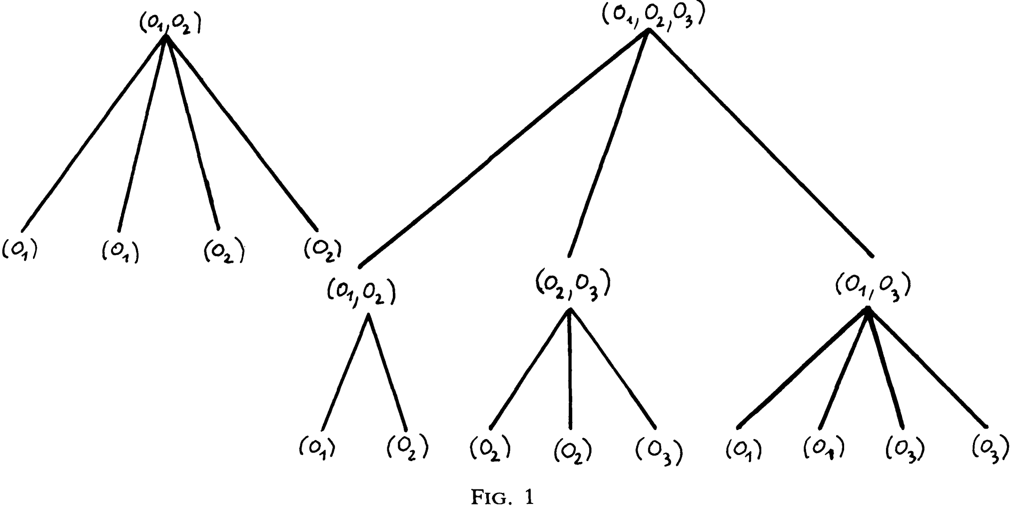


FIG. 1

We need the following result:

THEOREM 1. *If $|\mathcal{O}| = n$, a maximal φ-tree has a maximum of $2n(n-1)+1$ vertices.*

Proof. First, we remark that $|\varphi(s)| = 1$ for every leaf s of a maximal φ-tree.

We prove Theorem 1 by induction on $|\mathcal{O}|$. It is verified for small values of $|\mathcal{O}|$; see Fig. 1. Suppose that the theorem holds for $|\mathcal{O}| < n$. Let $A = (T, S, \mathcal{O}, \varphi)$ be a maximal φ-tree such that $|\mathcal{O}| = n$. We denote $x_1, x_2, \cdots, x_p$ as the p sons ($p \geqq 2$) of the root r. Let $\mathcal{O}'$ be the union of $\varphi(x_1), \varphi(x_2), \cdots, \varphi(x_{p-1})$. Let $\mathcal{O}''$ be $\varphi(x_p)$, x_p being the rightmost son of r. Suppose that $\mathcal{O}' = \mathcal{O}_1 \cup \mathcal{O}_2$ and $\mathcal{O}'' = \mathcal{O}_2 \cup \mathcal{O}_3$, where $\mathcal{O}_1$, $\mathcal{O}_2$ and $\mathcal{O}_3$ are mutually disjoint.

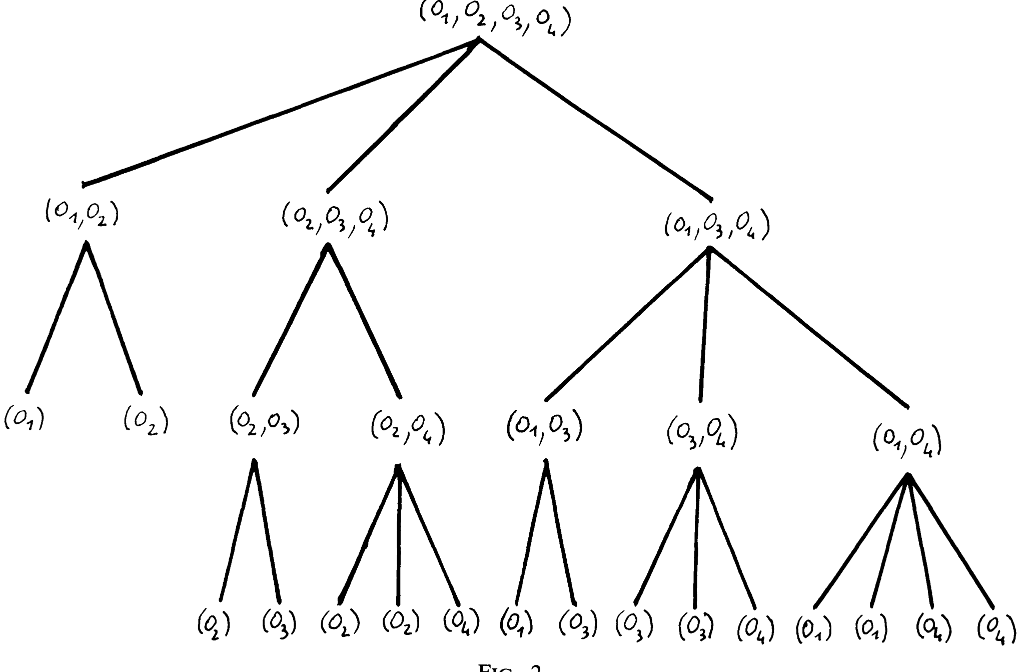


FIG. 2

Let u, v and w be $|\mathcal{O}_1|$, $|\mathcal{O}_2|$ and $|\mathcal{O}_3|$. Condition 2 on φ-trees implies that $u \neq 0$.

The subtree A' including x_p and the descendant of x_p is a tree which satisfies the 5 conditions. By the inductive hypothesis, the number of vertices of A' is less than or equal to $2(v+w)^2 - 2(v+w) + 1$.

Let A'' be the subtree including:

* the root, the vertices $x_1, x_2, \cdots, x_{p-1}$ and their descendants if $p > 2$,
* the vertex x_1 and its descendants if $p = 2$.

By conditions 2 and 3, all vertices of A'' except the leaves have at least two sons.

By condition 5, for every ω in $\mathcal{O}_2$, there exists a single leaf x of A'' containing it. Thus A'' has at most v leaves x such that $\varphi(x) \in \mathcal{O}_2$. Let ω be an element of $\mathcal{O}_1$.

Let A_0 be the subtree of A'' consisting of the vertices x of A'' containing ω. By condition 4, the vertices of A_0 have at most two sons (in A_0). The only vertices which have two sons are on the rightmost branch, and this branch is composed of at most $u+v+1$ vertices because the label of any son of the root of A_0 is bounded by $u+v$. Thus A_0 has a maximum of $u+v+1$ leaves and A'' has a maximum of $u(u+v+1)$ leaves x such that $\varphi(x) \cap \mathcal{O}_1 \neq \emptyset$. Hence A'' has a maximum of $v + u(u+v+1)$ leaves, and a maximum of $2(v + (u(u+v+1))) - 1$ vertices.

But a_n, the number of vertices of A, is at most equal to the sum of vertices of A' and A'' plus eventually one for the root when it is not in A''. Hence

$$a_n \leqq 2(v+w)^2 - 2(v+w) + 1 + 2(v + u(u+v+1)) - 1 + 1$$

and

$$a_n \leqq 2n^2 - 2n + 1 - 2v(u-1) - 4u(w-1).$$

By hypothesis, we have $u > 0$, hence $2v(u-1) \geqq 0$. If w is different from 0, then $u(w-1) \geqq 0$ and $a_n \leqq 2n^2 - 2n + 1$ as u is not equal to 0.

Now, if $w=0$, u and v must be different from zero; by condition 2, we have $p>2$ and $|\varphi(x_i)|\leqq u+v-1$ for $i=1, 2, \cdots, p$. Thus a branch of A'' has a maximum of $u+v$ vertices. In the same way, A'' has a maximum of $v+u(u+v)$ leaves.

Let y be the rightmost leaf of A'' such that $\varphi(y)$ is an element ω of $\mathcal{O}_1$. The maximality of A implies that y has a brother y' such that $\varphi(y')$ contains ω. Condition 2 implies that y has a second brother y'' such that $\varphi(y'')$ contains an element of ω' which is not in $\varphi(y')$. So A'' contains at least one vertex which has more than two sons. Thus A'' has a maximum of $2(v+u(u+v))-2$ vertices. In the special case where $w=0$, the root belongs always to A'' and

$$a_n \leqq 2v^2-2v+1+2(v+u(u+v))-2.$$

Then $a_n \leqq 2n^2-2n+1-2(u-1)(v-1)$. But both u and v are different from zero. Thus we have $(u-1)(v-1)\geqq 0$ and in this case a_n is also lower than or equal to $2n^2-2n+1$.

Hence Theorem 1 is proved.

2. Overview of the proof of Tutte's theorem. For most definitions, e.g., planar graph, circuit, face, we refer to Tutte [17] and Berge [2].

A closed Jordan curve J divides the plane into 2-connected open domains, the *interior domain* ($\operatorname{int} J$) and the *exterior domain* ($\operatorname{ext} J$). For any pair of vertices $j_1 \neq j_2$ on a closed Jordan curve J and any point $u \in \operatorname{int} J$ there exist Jordan curves $T(j_1, u, j_2)$, *having only their endpoints on J.* Such a curve we call an *inner transversal* for *J.* A circuit in a graph G is minimal when it has no inner transversals.

The *vertices of attachment* (or *attachments*) of a subgraph H of a graph G are the common vertices of H and $G\backslash H$.

A *bridge* of a circuit J in a graph G is a subgraph B of G satisfying the 3 following conditions:

i) all the attachments of B are vertices of J,

ii) B is not a subgraph of J,

iii) no proper subgraph of B satisfies conditions (i) and (ii).

An immediate consequence of that definition is that for every circuit J of G, the edges of $G\backslash J$ are partitioned (uniquely) into the bridges of J in G, with two edges belonging to the same bridge iff they can be connected by a path that does not have any vertices in common with J except possibly the end vertices. An *inner bridge* of a circuit J in a planar graph G is a bridge of J which lies in $J+\operatorname{int} J$. The *vertex-connectivity number* $K(G)$ of a connected graph G is the minimal number of vertices of which the suppression separates G or reduces it to a single vertex. G is *h-connected* if its vertex-connectivity number is $\geqq h$.

The paper is based on the proof of the following theorem.

THEOREM (Tutte [16]). *In a 2-connected planar graph G, let E be an edge lying on the circuits C_0 and C_1, boundaries of faces F_0 and F_1, while E' is another edge of C_1. Then there exists a circuit K passing through E and E' such that none of its bridges have more than three attachments while the bridges having edges in common with C_0 or C_1 have two attachments.*

COROLLARY. *Let G be any 4-connected planar graph having at least two edges. Then G has an Hamiltonian circuit.*

Summary of the proof. Let us call to mind the principal stages of the proof of Tutte's theorem given by Ore [14]. This proof is based upon induction on the number of edges in the graph. It is divided into 4 paragraphs.

1) Construction of a circuit D_0.

2) Construction of a circuit K.
3) Study of bridges of K and D_0.
4) Modification of K.

We can easily convince ourselves that the theorem is true for graphs having few edges.

Let G be a 2-connected planar graph. The number of edges of G is n. We suppose that the theorem holds for all graphs with less than n edges. Let E be an edge lying on the circuits C_0 and C_1, boundaries of faces F_0 and F_1, while E' is another edge on C_1.

1) *Construction of the circuit* D_0. Let L be one of the two paths of C_1 of which terminal edges are E and E'. We call l_0 and l_n the terminal vertices of L ($l_0 \in E$ and $l_n \in E'$). We eliminate from G all edges not belonging to C_1 which have at least one end point in $[L]$ (the set of vertices on L not including the end points is denoted by $[L]$). Then, we replace L by a single edge L_0. We obtain a graph G'. In G', the edge L_0 lies on the boundary of two faces: F_0' and F_1'. F_1' is the face which corresponds to F_1 in G. D_0' and D_1' are the minimal circuits of F_0' and F_1'. D_0' corresponds to a circuit D_0 in G (if we replace L_0 by L in G'). The end points of the path composed by the edges of $D_0 \backslash L$ belonging to C_0 are l_0 and another vertex: d_0 (not necessarily different).

2) *Construction of the circuit* K. We delete from G the inner bridges of D_0 and we replace L by L_0. Let G_0 be the graph obtained. G_0 is a 2-connected planar graph which has fewer edges than G. In G_0, let E_0 be the edge L_0 and E_0' an edge of $D_0 \backslash L$ incident to d_0. As the theorem holds for G_0, there exists a circuit K passing through E_0 and E_0' and satisfying the conditions of Tutte's theorem.

3) *Bridges of* K *and* D_0. The inner bridges of K in G fall into 4 types with respect to D_0:

a) δ_0-bridges not attached on $D_0 \backslash L$ but lying inside D_0,
b) δ_1-bridges with one attachment on $D_0 \backslash L$ and lying inside D_0,
c) γ-bridges with no edges inside D_0,
d) ε-bridges including edges inside D_0 and edges on or outside D_0.

In FIG. 3 we indicate the various types of K-bridges.

The main properties of these bridges are given by the following lemmas.

LEMMA 1. *If the vertex* d_0 *is an attachment of an inner bridge of* K, *then there exists a* δ_1*-bridge attached on* d_0 *which includes all the edges of* C_0 *not on* K. *Otherwise, the bridges of* K *are only of type* δ_0 *and* γ.

LEMMA 2. *Inner bridges of* K *cannot have edges on* $C_1 \backslash L$.

DEFINITION. An inner bridge B of K in G is *exceptional* if it satisfies one of the following conditions:

i) B has more than 3 attachments.
ii) B meets C_1 or C_0 and has more than two attachments.

LEMMA 3. *An exceptional bridge has at least two attachments in* $[L]$.

LEMMA 4. *An* ε*-bridge has only two attachments,* d_1 *and* d_2, *on* $K \backslash L$. *These attachments are both on* K *and* D_0.

4) *Modification of* K. We call the *L-section* of a δ_0, δ_1 or ε-bridge B the shortest path of L (with respect to the number of edges) which contains all the attachments of B on $[L]$. We say that a bridge B_2 is *enclosed* by a bridge B_1 if and only if there exists a circuit N composed of edges of B_1 and L and if B_2 is inside N.

We call *complex* a set X of inner bridges of K which satisfies the 3 following conditions:

i) There exists an exceptional bridge B_0 in X, called the *maximal bridge* of X which encloses all the elements of $X \backslash B_0$.

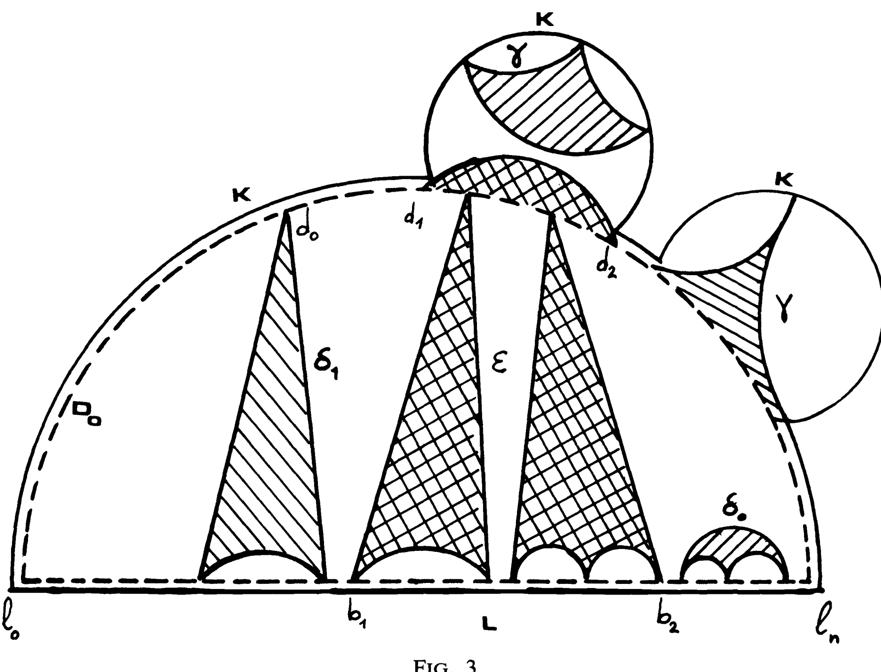


FIG. 3

ii) All the inner bridges of K enclosed by B_0 are in X.

iii) There exists no exceptional bridge of K which encloses B_0.

We call the *L-section of a complex* the L-section of its maximal bridge. We define 3 families of complexes determined by the type of their maximal bridges: the δ_0, δ_1 and ε-complexes. We make the observation that two complexes have no edges in common and so are their L-sections.

For each complex, we construct a graph Γ which has a smaller number of edges than G. We call b_1 and b_2 the end points of the L-section of a complex (b_1 is always different from b_2 by Lemma 3). On L, b_1 is the nearest vertex of l_0 and b_2 is the nearest vertex of l_n.

For a δ_0-complex, Γ is defined by the complex, its L-section and an edge (b_1, b_2).

For a δ_1-complex, Γ is defined by the complex B, its L-section and the two edges (b_1, d_1) and (d_1, b_2) if we call d_1 the single attachment of B on $D_0 \backslash L$.

For an ε-complex, the construction of Γ is more difficult. First, we construct a graph Γ' consisting of the complex B, its L-section and the three edges (b_1, d_1), (d_1, d_2) and (d_2, b_2) if we call d_1 and d_2 the two attachments of B on $D_0 \backslash L$.

Then we eliminate from Γ' all edges which have at least one end point on d_1 or d_2, except (b_1, d_1), (d_1, d_2) and (d_2, b_2).

Now the path $A = (b_1, d_1, d_2, b_2)$ lies on two minimal circuits: K and $C'_0 = A + S(b_2, b_1)$ where S is a path through Γ'. The inner bridges for C'_0 are attached at d_1 and some vertex s_1 on S, or attached at d_2 and some vertex s_2 on S or attached at d_1 and d_2 and a single vertex s_0 on S (see Fig. 4). If there exists no inner-bridge attached at d_1 and d_2 we define s_0 as one of the vertices of S which separate the sets $\{s_1\}$ and $\{s_2\}$. Γ is obtained by deleting from Γ' the inner-bridges of C'_0 and by replacing A by a single edge (b_1, b_2).

In each case, for the graph Γ, we can apply the theorem and construct a circuit J passing through:

- the edge (b_1, b_2) for a δ_0-complex,

- the edges (b_1, d_1) and (d_1, b_2) for a δ_1-complex,
- the edge (b_1, b_2) and the vertex s_0 for an ε-complex.

In the circuit K of graph G we replace the L-section of each complex by the edges of J except:

- (b_1, b_2) for a δ_0-complex,
- (b_1, d_1) and (d_1, b_2) for a δ_1-complex,
- (b_1, b_2) for an ε-complex.

The resulting new circuit of G has no complex, and the proof of Tutte's theorem is completed.

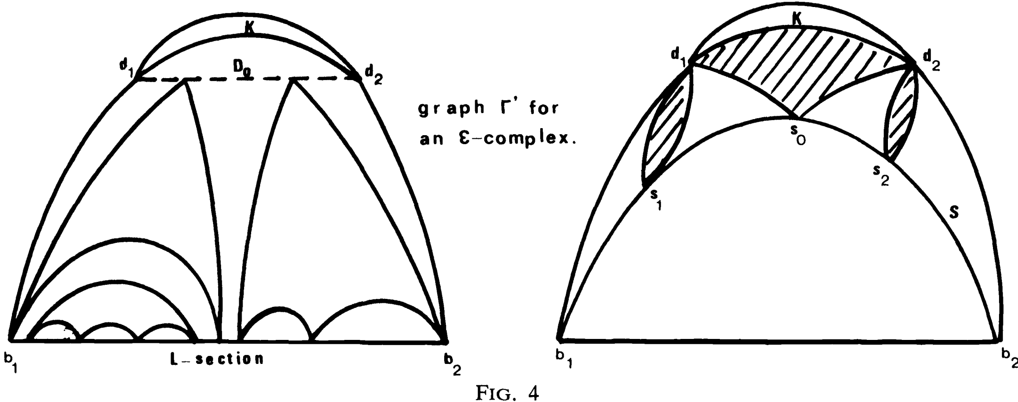


FIG. 4

3. Presentation of the algorithm. The proof of Tutte's theorem can be summarized as follows:

1. The theorem holds for all graphs with a small number of edges, or for the graphs which are reduced to a circuit.

2. Let G be a graph which has n edges. We obtain a graph G_0 by removing in G all the inner-bridges of a circuit D_0. The theorem holds for G_0 and we find a circuit K in G.

3. The circuit K has p complexes $K_1, K_2, \cdots, K_p$. The theorem holds for each complex $K_1, K_2, \cdots, K_p$ and we obtain circuits $J_1, J_2, \cdots, J_p$. Then we modify K including $J_1, J_2, \cdots, J_p$.

From such a proof, we may deduce easily the following recursive algorithm written in "pidgin" Algol:

```
Procedure proc (G, K)
  Begin
    If G is a circuit then K := G else
      Begin
        Construct a circuit D0;
        Construct a graph G0;
        proc (G0, K);
        for each complex Γ of K in G do
          Begin
            proc (Γ, J);
            Modify K including J;
          End;
      End;
  End;
```

The arguments of the procedure proc are the given graph G and the circuit K we are searching for. We define recursively the level of a procedure's call during the performance of the algorithm as follows:

* the first call of procedure proc has level 1.
* the level of a procedure's call is $i+1$ if it is activated by a procedure of which the level is i.

We can make the 3 following remarks:

1) During a performance of the procedure proc, a δ-complex does not have any edge belonging to G_0.

2) During a performance of the procedure proc, two complexes do not have any common edge.

3) During a performance of the procedure proc, only ε-complexes may have a nonempty intersection with G_0.

For the computation of the complexity, we need to prove the 3 following properties about the common edges of graph G_0 and of an ε-complex X, i.e., the edges of the maximal bridge of K which lie on or outside D_0.

Let X be an ε-complex. We call L_x its L-section. We say that an edge of X is *special* if it belongs to X and G_0. Following paragraph 4 of § 2, we can construct two graphs Γ' and Γ. Tutte's theorem give a circuit J in Γ.

LEMMA 5. *If a bridge of J in Γ' contains an edge of L_x then it doesn't contain any special edge.*

Proof. Let B a bridge of J in Γ' which includes an edge b of L_x. By Tutte's theorem, B has two attachments because B meets circuit C_1. L_x is a path connecting b_1 to b_2. But b_1 and b_2 belong to J. Thus there exist two vertices, x_1 and x_2, belonging both to L_x and J which are the only attachments of B (we can have $x_1 = b_1$ and/or $x_2 = b_2$).

Let c be a special edge. Then there exists a path Q composed by edges of the maximal bridge of X which connects c to d_1. All the edges of Q are special and no vertex of Q but d_1 is on J. This condition implies that no vertex of Q is on L_x. But d_1 belongs to J and is necessarily different from x_1 and x_2. Now all the paths connecting an edge of B to d_1 go through x_1 or x_2. Hence c cannot belong to B and Lemma 5 follows.

During the algorithm, the performances of the procedure proc are numbered in the order of their calls.

During the performance of the procedure proc numbered n, all the objects used (vertices, circuits, bridges and edges) are superscripted by n. For a performance of a procedure proc numbered n, we say that an inner-bridge B^n of the modified cycle K^n is *useful* if there exists a number m ($m < n$) such that:

i) the call n results from the call proc (G_0^m, K^m),

ii) all the edges of B^n belong to an ε-bridge of K^m during the call of proc numbered m.

The condition i has a meaning because if the call n results from the call m, some edges of G^n are edges of G^m.

LEMMA 6. *During the performance of a procedure proc numbered n, a necessary condition that an inner-bridge B^n of the modified cycle K^n be useful is that B^n has an edge in common with C_0^n or C_1^n.*

Proof. Suppose that B^n is useful. Then there exists a number m satisfying i and ii, i.e., there exist p numbers $n_1, n_2, \cdots, n_p$ ($p \geqq 2$) such that:

a) $n_1 = m$ and $n_p = n$;

b) n_{i+1} is the number either of a call proc $(G_0^{n_i}, K^{n_i})$ or a call proc (Γ^{n_i}, J^{n_i}) $(i = 2, \cdots, p-1)$;

c) $n_2 = m+1$ is the number of the call proc (G_0^m, K^m);
d) all the edges of B^n belong to an ε-bridge X^m of K^m during the call of proc numbered m.

The algorithm "breaks" bridges and never merges them so the edges of B^n are the only edges of X^m on or outside D_0^m.

Because of the definition of ε-bridges, condition d implies that at least one edge of B^n is on $D_0^m \backslash L^m$. But the edges of $D_0^m \backslash L^m$ are all on $C_1^{n_2}$, so B^n has at least one edge in common with $C_1^{n_2}$. Now suppose that B^n has at least one edge in common with $C_0^{n_i}$ or $C_1^{n_i}$, say edge α. Since α belongs to B^n, α belongs also to $G^{n_{i+1}}$.

If n_{i+1} is the number of a call proc $(G_0^{n_i}, K^{n_i})$ we choose $L_0^{n_i}$ for $E^{n_{i+1}}$ and an edge of $D_0^{n_i} \backslash L^{n_i}$ incident to $d_0^{n_i}$ for $E'^{n_{i+1}}$ and we have $C_0^{n_{i+1}} = C_1^{n_i} \backslash L^{n_i} + L_0^{n_i}$ and $C_1^{n_{i+1}} = D_0^{n_i} \backslash L^{n_i} + L_0^{n_i}$.

Since α belongs to $G^{n_{i+1}}$, a necessary condition that α belongs to $C_0^{n_i}$ or $C_1^{n_i}$ is that α belongs to $C_0^{n_{i+1}}$ or $C_1^{n_{i+1}}$.

If n_{i+1} is the number of a call proc (Γ^{n_i}, J^{n_i}) and if Γ^{n_i} is a δ_1-complex, we choose $(b_1^{n_i}, d_1^{n_i})$ for $E^{n_{i+1}}$ and $(b_2^{n_i}, d_1^{n_i})$ for $E'^{n_{i+1}}$. Then the only edges of $C_0^{n_i}$ and $C_1^{n_i}$ in $G^{n_{i+1}}$ are the L-section of Γ^{n_i} and some edges of $C_0^{n_i}$ if $d_1^{n_i} = d_0^{n_i}$. Thus a necessary condition that α belongs to $C_0^{n_i}$ or $C_1^{n_i}$ is that α belongs to $C_0^{n_{i+1}}$ or $C_1^{n_{i+1}}$.

If n_{i+1} is the number of a call proc (Γ^{n_i}, J^{n_i}) and if Γ^{n_i} is an ε-complex, we choose $(b_1^{n_i}, b_2^{n_i})$ for $E^{n_{i+1}}$ and an edge of S^{n_i} incident to $s_0^{n_i}$ for $E'^{n_{i+1}}$. The only edges of $C_0^{n_i}$ and $C_1^{n_i}$ in $G^{n_{i+1}}$ are the L-section of Γ^{n_i}. Thus a necessary condition that α belongs to $C_0^{n_i}$ or $C_1^{n_i}$ is that α belongs to $C_0^{n_{i+1}}$ or $C_1^{n_{i+1}}$ since α belongs to $G^{n_{i+1}}$.

If n_{i+1} is the number of a call proc (Γ^{n_i}, J^{n_i}) and if Γ^{n_i} a δ_0-complex, we choose for $E^{n_{i+1}}$ the edge $(b_1^{n_i}, b_2^{n_i})$ and for $E'^{n_{i+1}}$ an edge of the L-section of Γ^{n_i}. The only edges of $C_0^{n_i}$ and $C_1^{n_i}$ in $G^{n_{i+1}}$ are the L-section of Γ^{n_i}. Thus a necessary condition that α belongs to $C_0^{n_i}$ or $C_1^{n_i}$ is that α belongs to $C_0^{n_{i+1}}$ or $C_1^{n_{i+1}}$.

So if B^n has at least one edge in common with $C_0^{n_i}$ or $C_1^{n_{i+1}}$, then B^n has at least one edge in common with $C_0^{n_{i+1}}$ or $C_1^{n_{i+1}}$ and the Lemma 6 is proved by induction.

LEMMA 7. *Let α be a special edge during the performance of a procedure proc numbered m. Then, during any call numbered $n (n > m)$ which proceeds from proc (G_0^m, K^m), the edge α cannot be a special edge.*

Proof. Suppose that n is the first number greater than m such that α is a special edge during the performance of the procedure proc numbered n. Suppose also that this procedure proceeds from proc (G_0^m, K^m).

During the performance of the procedure numbered n, α belongs to an ε-complex X^n. Following paragraph 4 of § 2, we can construct two graphs Γ'^n and Γ^n. Tutte's theorem gives a circuit J^n in Γ^n. Let B^n be the bridge of J^n which contains α. B^n is also a bridge of K^n which has been modified. Thus B^n is a useful bridge.

Lemma 6 claims that B^n has an edge in common with C_0^n or C_1^n. The proof of Lemma 6 shows that B^n necessarily contains only any edge with the L-section L_x^n of X^n. But Lemma 5 says that a bridge of J^n in Γ'^n cannot contain at once an edge of L_x^n and a special edge.

This contradiction implies Lemma 7.

4. Complexity of the algorithm. For the definitions of complexity, we refer the reader to Shepherdson and Sturgis [15], Elgot and Robinson [6], Hartmanis [10], Aho, Hopcroft and Ullman [1].

Now, with these definitions we can establish the following property:

PROPERTY 1. *The complexity of the algorithm is polynomial if the complexities of these following instructions are also polynomial*:

i) *Construct a circuit D_0;*

ii) *Construct a graph* G_0;

iii) *Modify* K *including* J.

Proof. To describe an execution of the algorithm, we use a tree A. S is its set of vertices. Each vertex of S represents a call of the recursive procedure proc. We define φ, a mapping from S to $\mathcal{P}(E)$ where E is the set of edges of H, the graph treated by the algorithm, as follows:

For each vertex $s \in S$, $\varphi(s)$ is the set of edges of H belonging to G, the parameter of the procedure corresponding to s.

Every call of the procedure introduces at most two edges not belonging to H:

Proc (G_0, K) introduces L_0 which does not belong to H.

Proc (Γ, J) introduces (b_1, d_1) and (b_2, d_1) for a δ_1-complex, (b_1, b_2) for an ε-complex or a δ_0-complex. But all these edges become edges E_0 or E_1 for the called procedure. So these edges cannot belong to the parameters of the procedures which result from the called procedure and we can ignore them.

Then the 4-tuple (A, S, E, φ) is a φ-tree.

Conditions 2 and 3 for φ show that $G_0, K_1, K_2, \cdots, K_p$ are proper nonempty subgraphs of G.

Condition 4 refers to contents of remarks 1, 2 and 3 of § 3.

Condition 5 summarizes the result of Lemma 7 about ε-complexes.

All executions of the algorithm can be described by such a φ-tree, but some φ-trees do not illustrate any execution.

If the time complexity and space complexity of the three instructions are polynomial, the number of primitive operations or the amount of auxiliary storage is bounded by a polynomial function $P(n)$ where n is the number of edges of G. For each execution of the procedure we can overvalue the number of edges on which the procedures work by the number of edges on which the algorithm works. Thus, the complexity of the algorithm has overvalued the product of $P(n)$ by the number of vertices of A (i.e., by the number of calls of proc). Hence, following Theorem 1, this function is overvalued by $(2n^2 - 2n + 1)\, P(n)$. Then the algorithm is polynomial and Property 1 is proved.

The main result of this paper is included in the following theorem:

THEOREM 2. *Time and space-complexity of the algorithm is overvalued in* $O(n^3)$.

Proof. Property 1 claims that the complexity is in $O(n^2 P(n))$ where $P(n)$ is an over-estimation of the complexity of three instructions. But the complexity of the instruction "construct a circuit D_0" is linear because it amounts to passing through the path L, to removing the edges attached to L and then to passing through the minimal circuit of a face.

The instruction "construct the graph G_0" is also linear because it prescribes to determine and remove the bridges of D_0 in G.

Finally, the set of the p instructions "Modify K including J" is a linear instruction because it does nothing but modify the marks of edges of K and Γ and because the subgraphs Γ have no common edge.

So $P(n) = n$ and Theorem 2 is proved.

In [9], we give an Algol program from this algorithm which uses planar maps according to Jacques [11] and Cori [4].

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1975.

[2] C. BERGE, *Graphes et hypergraphes*, Dunod, Paris, 1970.
[3] S. A. COOK, *The complexity of theorem proving procedures*, Proc. 3rd annual ACM Symposium on Theory of Computing, 1970, pp. 151–158.
[4] R. CORI, *Un code pour les graphes planaires et ses applications*, Thèse de Doctorat d'Etat, Paris, 1973.
[5] D. G. CORNEIL, *The analysis of graph theoretical algorithms*, Technical Report 65, University of Toronto, Toronto, April 1974.
[6] C. C. ELGOT AND A. ROBINSON, *Random access stored program machines*, J. Assoc. Comput. Mach., 11 (1964), pp. 365–399.
[7] M. R. GAREY, D. S. JOHNSON AND L. STOCKMEYER, *Some simplified polynomial complete problems*, Proc. Sixth Annual ACM Symposium on Theory of Computing, Seattle, May 1974.
[8] M. R. GAREY, D. S. JOHNSON AND R. E. TARJAN, *The planar Hamiltonian circuit problem is NP-complete*, this Journal, 5 (1976), pp. 704–714.
[9] D. GOUYOU-BEAUCHAMPS, *Un algorithme polynomial pour la recherche de cycles Hamiltoniens dans les graphes 4-connexes planaires*, Thèse de 3ème Cycle, Bordeaux, 1977.
[10] J. HARTMANIS, *Computational complexity of random access stored program machines*, Math. Syst. Theory, 5 (1971), pp. 232–245.
[11] A. JACQUES, *Constellations et propriétés algébriques des graphes topologiques*, Thèse de 3ème Cycle, Paris, 1969.
[12] R. M. KARP, *Reducibility among combinatorial problems*, in Complexity of Computer Computations, Miller and Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.
[13] D. A. KLARNER, *Correspondence between plane trees and binary sequences*, J. Comb. Theory, 9 (1970), pp. 401–411.
[14] O. ORE, *The Four Color Problem*, Academic Press, New York, 1967.
[15] J. C. SHEPHERDSON AND H. E. STURGIS, *Computability of recursive functions*, J. Assoc. Comput. Mach., 10 (1963), pp. 217–255.
[16] W. T. TUTTE, *A theorem on planar graphs*, Trans. Amer. Math. Soc., 82 (1956), pp. 99–116.
[17] ———, *Bridges and Hamiltonian circuits in planar graphs*, Aequationes Mathematica, 15 (1977), pp. 1–33.

SIAM J. COMPUT.
Vol. 11, No. 3, August 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1103-0012 $01.00/0

ON EDGE COLORING BIPARTITE GRAPHS*

RICHARD COLE† AND JOHN HOPCROFT†

Abstract. The present paper shows how to find a minimal edge coloring of a bipartite graph with E edges and V vertices in time $O(E \log V)$.

Key words. edge coloring, bipartite graph, multigraph, matching, partition

1. Introduction. An algorithm for finding a minimal edge coloring of a bipartite graph in time $O(E \log V)$ is presented. Polynomial time algorithms for this problem have previously been given by Gabow in [2] and by Gabow and Kariv in [3], the best time bounds being $O(E \log^2 V)$ and $O(V^2 \log V)$.

From the work of Gabow [2] and Gabow and Kariv [3] it is known that a fast minimal edge coloring algorithm can be constructed from a fast bipartite graph matching algorithm. The $O(n^{2.5})$ bipartite graph matching algorithm in [4] is not fast enough for our purposes. However, a maximal matching is not required. A matching covering all of the maximum degree vertices is sufficient. Two methods for finding matchings covering all the maximum degree vertices are presented. The first method illustrates the underlying ideas used in the second method which has a running time of $O(\max\{E, V \log V \log^2 D\})$ where D is the maximum degree of any vertex. This running time leads to an $O(E \log V)$ minimal edge coloring algorithm. In fact, the time bound for the coloring algorithm is slightly sharper.

In § 2 the notation and definitions are given. In § 3 the first matching algorithm is given, the heart of it being a partition algorithm. In § 4 the second matching algorithm is given, and in § 5 the coloring algorithm is given.

2. Notation and definitions. Throughout this paper, $G = (V, E)$ denotes a graph with vertex set V and edge set E. $G = (V_1, V_2, E)$ denotes a bipartite graph with disjoint vertex sets V_1 and V_2 and edge set $E \subseteq V_1 \times V_2$. D denotes the maximum degree of any vertex in $V = V_1 \cup V_2$, and M denotes the set of vertices having degree D.

A graph is said to be *regular* if all its vertices have the same degree. A bipartite graph is said to be *semiregular* if all the vertices in V_1 have the same degree D, the maximum degree of any vertex in G; it is said to be *high-low* if there exists an integer k such that $\deg(v) \geqq k$ if $v \in V_1$, and $\deg(v) \leqq k$ if $v \in V_2$.

A *matching*, $N \subseteq E$, is a subset of the edges with the property that no two edges have a common endpoint. A matching is said to *cover* a set of vertices, U, if every vertex in U is an endpoint of an edge in the matching. It need not be the case that both, or either, endpoints of an edge in the matching lie in U. In a semiregular bipartite graph, a matching covering M, the set of maximum degree vertices, is a maximal matching since M must include all of V_1.

An *Euler partition* is a partition of the edges into open and closed paths, so that each vertex of odd degree is at the end of exactly one open path, and each vertex of even degree is at the end of no open path. Euler partitions exist in all graphs, not just bipartite ones.

* Received by the editors December 16, 1980, and in revised form October 21, 1981. This research was supported in part by the Office of Naval Research under grant N00014-76-C-0018.

† Department of Computer Science, Cornell University, Ithaca, New York 14853.

An *Euler split* of a bipartite graph $G=(V_1, V_2, E)$ is a pair of bipartite graphs $G_1=(V_1, V_2, E_1)$ and $G_2=(V_1, V_2, E_2)$ where E_1 and E_2 are formed from an Euler partition of E by placing alternate edges of paths into E_1 and E_2. Any vertex of even degree in G will have the same degree in both G_1 and G_2, while any vertex of odd degree in G will have degrees in G_1 and G_2 differing by one. This implies that if D, the maximum degree of any vertex in G, is even, then all the vertices in M have degree $D/2$ in each of G_1 and G_2, and this is the maximum degree of any vertex in G_1 or G_2. An algorithm for finding an Euler split in time $O(E)$ is given in [2]. Euler splits as defined need exist only in bipartite graphs.

A *partition* of a bipartite graph $G=(V_1, V_2, E)$ is a pair of bipartite graphs $G_1=(V_1, V_2, E_1)$ and $G_2=(V_1, V_2, E_2)$, where E_1 and E_2 are disjoint, and the union of E_1 and E_2 is E. The partition is *M-containing* if the vertices having maximum degree in G_1 include M and the vertices having maximum degree in G_2 include M, where M is the set of vertices having maximum degree in G.

An *edge coloring* of a graph associates a color with each edge in the graph in such a way that no two edges of the same color have a common endpoint. As shown in [1, p. 250] any bipartite graph has a minimal edge coloring using D colors, where D is the maximum degree of any vertex in the graph.

3. First algorithm for finding a matching in a bipartite graph covering the vertices of maximum degree. The algorithm runs in time $O(E \log V)$. Let G be the graph in which we wish to find a matching and again let M be the set of maximum degree vertices. If the maximum degree of any vertex in G is one, then E, the edge set of G, is the required matching. Otherwise an M-containing partition of G into bipartite graphs $G_1=(V_1, V_2, E_1)$ and $G_2=(V_1, V_2, E_2)$ is made where E_1 and E_2 are both nonempty. The algorithm is then applied recursively to that graph, of G_1 and G_2, with the smaller edge set.

To partition G, an Euler split is made, giving graphs H_1 and H_2. If D, the maximum degree of any vertex in G, is even, then the vertices in M have degrees $D/2$ in both H_1 and H_2, and so H_1 and H_2 provide an M-containing partition of G. Otherwise, if D is odd, edges are moved between H_1 and H_2 so that vertices in H_1 have maximum degree D_1, vertices in H_2 have maximum degree D_2, and all the vertices in M have degree D_1 in H_1 and degree D_2 in H_2, for some pair (D_1, D_2), with $D_1+D_2=D$. The method for moving edges is described below.

If D is odd, with $D=4r+e$ and $e=\pm 1$, then each vertex in M has degree $2r$ in one of H_1 or H_2 and degree $2r+e$ in the other. So in one of H_1 or H_2 at least half the vertices in M have degree $2r+e$. Without loss of generality, suppose that in H_1 at least half the vertices in M have degree $2r+e$. Then let M_1 be those vertices of M that have degree $2r+e$ in H_1 (and hence degree $2r$ in H_2), and let M_2 be the remaining vertices of M.

In general, it will be the case that vertices in M_1 have degree $2k$ in H_2 and degree $D-2k$ in H_1, while vertices in M_2 have degree $D-2k-d$ in H_1 and degree $2k+d$ in H_2, for $d=\pm 1$.

An Euler split of H_2 is made giving graphs H_{21} and H_{22}. The vertices from M_1 have degree k in each of H_{21} and H_{22}. Some vertices from M_2 have degree k in H_{21} and degree $k+d$ in H_{22}, while others have degree $k+d$ in H_{21} and degree k in H_{22}. Without loss of generality suppose that in H_{21} at least half the vertices in M_2 have degree $k+d$ (if necessary the labels H_{21} and H_{22} are swapped). Let M_{21} be those vertices of M_2 having degree $k+d$ in H_{21} (and hence having degree k in H_{22}), and let M_{22} be the remaining vertices of M_2. The vertices in M_{22} have degree k in H_{21} and degree $k+d$ in H_{22}.

In the graph $H_1 \cup H_{21}$ the vertices in $M_1 \cup M_{21}$ have degree $D-k$ and the vertices in M_{22} have degree $D-k-d$. In the graph H_{22} the vertices in $M_1 \cup M_{21}$ have degree k and the vertices in M_{22} have degree $k+d$. M_1 is set equal to $M_1 \cup M_{21}$ and M_2 is set equal to M_{22}, which reduces the size of M_2 by a factor of at least two, and $H_1 \cup H_{21}$ and H_{22} become the graphs H_1 and H_2, the correspondence being chosen to maintain the invariant given above. (The desired correspondence depends only on whether k is even or odd.)

The process is repeated until $M_1 = M$ when the vertices in M all have the same odd, maximum degree in H_1, and the same even, maximum degree in H_2. H_1 and H_2 now provide the required partition of G into G_1 and G_2. The partitioning procedure, for the case D is odd, is shown in algol-like form below.

procedure PARTITION
begin
  comment. This procedure partitions the bipartite graph $G=(V_1, V_2, E)$ into bipartite graphs $G_1=(V_1, V_2, E_1)$ and $G_2=(V_1, V_2, E_2)$, in the case that D, the maximum degree of any vertex in $V_1 \cup V_2$, is of the form $4r+e$, $e=\pm 1$.
  M = set of maximum degree vertices in G;
  Let H_1, H_2 be an Euler split of G;
  comment. At least half the vertices in M have degree $2r+e$ in one of H_1 or H_2. Let it be in H_1. Let $k=r$, $d=e$.
  Let $M_1 = \{v \mid v \in M$, and v has degree $2r+e$ in $H_1\}$;
  Let $M_2 = M - M_1$;
  while $(|M_2| \neq 0)$ **do**
  begin
    comment. The vertices in M_1 have degree $D-2k$ in H_1 and degree $2k$ in H_2, while the vertices in M_2 have degree $D-2k-d$ in H_1, and degree $2k+d$ in H_2, for some $d=\pm 1$.
    Let H_{21}, H_{22} be an Euler split of H_2;
    comment. At least half the vertices in M_2 have degree $k+d$ in either H_{21} or H_{22}. Let it be in H_{21}.
    Let $M_{21} = \{v \in M_2$, and v has degree $k+d$ in $H_{21}\}$
    Let $M_{22} = M_2 - M_{21}$;
    if k is even
      then $H_1 := H_1 \cup H_{21}$, $H_2 := H_{22}$;
      else $H_1 := H_{22}$, $H_2 := H_1 \cup H_{21}$;
    $M_1 := M_1 \cup M_{21}$, $M_2 := M_{22}$;
    comment. If k is even then the vertices in M_1 have degree $D-k$ in H_1 and degree k in H_2, while the vertices in M_2 have degree $D-k-d$ in H_1 and degree $k+d$ in H_2. Set $k=k/2$.
      Otherwise k is odd and the vertices in M_1 have degree k in H_1 and degree $D-k$ in H_2, while the vertices in M_2 have degree $k+d$ in H_1 and degree $D-k-d$ in H_2.
      Then set $k=(D-k)/2$ and $d=-d$.
  end
  $G_1 := H_1$, $G_2 := H_2$;
end

Correctness. The correctness of the partition algorithm is shown by the following lemmas.

LEMMA 3.1. *All the vertices in M have the same degree in G_1. Likewise, all the vertices in M have the same degree in G_2.*

Proof. The lemma follows from the inductive hypothesis:

At the start of the ith iteration of the while loop all the vertices in M_1 have the same odd degree $D-2k_i$ in H_1, and the same even degree $2k_i$ in H_2. Likewise, all the vertices in M_2 have the same even degree $D-2k_i-d_i$ in H_1, and the same odd degree $2k_i+d_i$ in H_2, where $d_i=\pm 1$, $i=1,2,\cdots$. d_i and k_i are the values of d and k at the start of the ith iteration of the loop.
It is easy to prove the inductive hypothesis. □

LEMMA 3.2. *For $v \notin M$, $v \in V_1 \cup V_2$, and for $u \in M$,*

$$\deg(v) \text{ in } G_1 \leqq \deg(u) \text{ in } G_1, \quad \text{and}$$

$$\deg(v) \text{ in } G_2 \leqq \deg(u) \text{ in } G_2.$$

Proof. The inductive hypothesis:

For $v \notin M$, $v \in V_1 \cup V_2$, and for $u \in M_1$, $\deg(v)$ *in* $H_1 \leqq \deg(u)$ in H_1, and $\deg(v)$ in $H_2 \leqq \deg(u)$ in H_2,

is shown to be true at the start (and end) of each iteration of the while loop. The lemma then follows. The inductive hypothesis is clearly true at the start of the first iteration of the while loop.

Suppose the program executes an ith iteration of the while loop, and assume the inductive hypothesis is true at the start of the ith iteration of the loop. It is known, by the inductive hypothesis of Lemma 3.1, that all the vertices in M_1 have the same degree in H_2; furthermore, this degree is even. Let it be $2k$. Suppose that a vertex v in M has degree h in H_2. By the inductive hypothesis of this proof, for the start of the ith iteration, it is known that $h \leqq 2k$ and that the degree of v in H_1 is less than or equal to the degree of u in H_1. So v has degree at most k in each of H_{21} and H_{22}, and degree $(v) \leqq$ degree (u) in both $H_1 \cup H_{21}$ and H_{22}. Therefore the inductive hypothesis is true at the end of the ith iteration of the loop (and at the start of the $(i+1)$st iteration, if there is one). □

LEMMA 3.3. *In G_1 the vertices of maximum degree include M; likewise in G_2. That is, G_1 and G_2 form an M-containing partition of G.*

Proof. By Lemma 3.1, all the vertices in M have the same degree in G_1. By Lemma 3.2, it is the maximum degree, and so in G_1 the vertices of maximum degree include M. Likewise in G_2. □

LEMMA 3.4. *G_1 and G_2 both have nonempty edge sets.*

Proof. The following inductive hypothesis is established: the degree of the vertices of M_1 in H_2 is even and nonzero at the start and end of each iteration of the while loop.

By construction, the degree of vertices of M_1 in H_2 is even throughout the algorithm. At the start of the first iteration of the loop the degree of vertices of M_1 in H_2 is nonzero also. Assuming the program executes an ith iteration, suppose inductively that at the start of the ith iteration the degree of vertices of M_1 in H_2 is nonzero, $2k$, say. Then the vertices of M_1 will have the nonzero degree k in both H_{21} and H_{22}. Thus the inductive hypothesis holds at the end of the ith iteration (and at the start of the $(i+1)$st iteration, if there is one), given that it is true at the start of the ith iteration.

On termination of the while loop $M=M_1$, so the vertices in M have nonzero even degree in H_2, and thus in G_2, which makes G_2 nontrivial. Since the vertices in M have odd degree in G and even degree in G_2, they must have odd degree in G_1, implying that G_1 is nontrivial. □

LEMMA 3.5. *G_1 and G_2 form an M-containing partition of G, with both G_1 and G_2 having nonempty edge sets.*

Timing. Each iteration of the while loop reduces the size of M_2 by a factor of at least two. So after at most $O(\log M)$ iterations of the loop $|M_2| = 0$ and the partition procedure terminates. So to partition G takes time $O(E \log M) \leqq O(E \log E/D)$.

Thus there is a constant c such that the running time $T(E)$ of the matching algorithm is bounded by:

$$T(E) \leqq cE \log E/D + T(E/2) = O(E \log E/D) \leqq O(E \log V).$$

4. A second algorithm to find a matching in a bipartite graph, covering M, the set of vertices of maximum degree. The algorithm runs in time $O(E + V \log V (\log D)^2)$. Let $G = (V_1, V_2, E)$ be the graph for which a matching covering M is wanted. By deleting some edges from E and increasing the multiplicity of other edges, a multigraph H is constructed where the degree of each vertex in H is the same as that of the corresponding vertex in G. Further, H will have only $V \log D$ multiedges. Clearly, M is the set of maximum degree vertices in H. Thus there exists a matching in H, that is also a matching in G, covering M. The first algorithm, given above, is used to find this matching. By the advantage of the multiple edges in H, the running time is made faster than on G.

To simplify the timing analysis, vertices of small degree ($\leqq D/2$) are "merged" together, so that, with the exception of at most two vertices, all of the vertices in $V_1 \cup V_2$ have degrees between $\lfloor D/2 \rfloor$ and D in G. Now $E = O(V * D)$. Two vertices u and v are merged by replacing them with a vertex w, where all the edges that had u or v as an endpoint now have w as an endpoint instead. To maintain the bipartite property of G vertices from V_1 are not merged with vertices from V_2.

H is obtained from G by finding cycles among edges of a given multiplicity and replacing alternate edges with edges of double multiplicity and removing the other edges. In turn, all possible cycles are found among the edges of multipliciity one, of multiplicity two, of multiplicity four, and so on, up to at most multiplicity $2^{\lfloor \log D \rfloor}$. Let $r = \lfloor \log D \rfloor$. The graph induced by the edges of multiplicity 2^r is acyclic since each vertex can have at most one multiedge of that multiplicity, and so no cycles can be found among these edges.

To find the cycles among edges of a given multiplicity, a depth first search is carried out. When a cycle is found, the edges are removed from the depth first search tree, alternate ones being given double multiplicity. The depth first search is then continued from the vertex at the root of the cycle. When the search retreats from a leaf because the only edge is the spanning edge into the vertex, the spanning edge and vertex are deleted since they are part of an acyclic graph. Thus at any time the depth first search tree consists of a simple path from the root of the search tree to the vertex currently being examined.

When a spanning edge is deleted from the depth first search tree, it is added to H, with the appropriate multiplicity. Also the acyclic graph with edges of multiplicity 2^r is added to H. So H is a union of at most $\log D$ acyclic multigraphs with edges of multiplicity $1, 2, 4, \cdots, 2^r$, respectively. As an acyclic graph has $O(V)$ edges, H has $O(V \log D)$ multiedges.

From H, a matching covering M is found by using the first matching algorithm. Four copies of each edge are kept, one in each of H_1, H_2, H_{21} and H_{22}. To effectively use the multiedge when making an Euler split, each edge is made to occur as often as possible at that point in the path; the multiplicities of the copies of the edges are changed accordingly. Otherwise, the first algorithm is used unchanged.

Timing. H can be constructed in time $O(E)$. For let $G = G_1, G_2, G_3, \cdots, G_r$ be the sequence of multigraphs with edge multiplicities $1, 2, 4, \cdots, 2^r$, respectively, with G_{i+1} being the graph obtained by performing the depth first search on G_i as described above, for $i = 1, 2, \cdots, r-1$. The size of the edge set of G_{i+1} is at most half the size of the edge set of G_i, and thus is bounded in size by $|E|/2^i$. So there exists a constant c such that the time taken to construct H is bounded by $cE + cE/2 + cE/4 + \cdots + cE/2^r = O(E)$.

The partition algorithm (from algorithm one) takes time $O(V \log D \log E/D)$ to halve the number of edges, counted according to their multiplicity, and thus the first algorithm takes time $O(V \log D \log E/D \log E/V)$ to find a matching covering M. So the overall time bound for the second algorithm is $O(E)$ if $E \geqq O(V \log V (\log \log V)^2)$ and $O(V \log V (\log D)^2)$ if $E < O(V \log V (\log \log V)^2)$, which is always at least as good as the $O(E \log V)$ of the first algorithm.

5. Coloring the edges of a bipartite graph. An $O(E \log V)$ time algorithm for finding a minimal edge coloring is given. Let G be the graph whose edges are to be colored, and let D be the maximum degree of any vertex in G; then D colors are used for the coloring. In the algorithm below, S denotes the collection of sets of edges similarly colored.

procedure COLOR
begin
  If D is odd
    then using the second matching algorithm, find a matching N covering the vertices in M. Color the edges in N with one color and delete N from G. $S := S \cup \{N\}$, $D := D - 1$:
  Make an Euler split of G to give two bipartite graphs, G_1 and G_2, each having $D/2$ as the maximum degree of its vertices;
  WLOG assume G_1 has a smaller edge set than G_2
    (otherwise swap the labels G_1 and G_2);
  Color (G_1);
  Let $2^k < D/2 = 2^{k+1} - r$. Add r sets of colored edges to G_2 and delete them from S;
  Color (G_2);
end

A similar method was used in [3]. That exactly D colors are used can be shown by induction.

Timing. There exists a constant c such that, excluding the time taken to find matchings, the time $T(E, D)$ required by the algorithm is given by

$$T(E, D) = T(E_1, \lfloor D/2 \rfloor) + T(E_2 \cup E_3, \lfloor D/2 \rfloor) + cE,$$

where E_1 is the edge set of G_1, E_2 is the edge set of G_2, and $E_3 \subseteq E_1$ is the union of the r sets of colored edges added to G_2. For D a power of two, $T(E, D) = O(E \log D)$. For D not a power of two, as $|E_3| \leqq |E_1| \leqq |E_2|$, $T(E, D) = O(2E \log (D + 2r)) = O(E \log D)$.

The time required for finding the matchings is bounded by $O(\max \{E, V \log V (\log D)^3\})$, and hence the total time required by the algorithm is bounded by $O(\max \{E \log D, V \log V (\log D)^3\})$, which is at least as good as $O(E \log V)$ for a graph all of whose edges have multiplicity one.

6. Matchings in semiregular and high-low bipartite graphs. By pruning a high-low graph, a semiregular graph with $D = k$ can be obtained, D being the maximum degree of any vertex in the semiregular graph. On applying the second matching algorithm to the semiregular graph, we obtain a matching covering the vertices of maximum degree. As observed in § 2, this is a maximal matching. High-low graphs and semiregular graphs were defined in [3] and the idea of pruning was given there.

Acknowledgments. The authors would like to thank the referees for their many helpful comments.

REFERENCES

[1] C. BERGE, *Graphs and Hypergraphs*, North-Holland, Amsterdam, 1973.
[2] H. GABOW, *Using Euler partitions to edge color bipartite multigraphs*, Internat. J. Comput. Inform. Sci., 5 (1976), pp. 345–355.
[3] H. GABOW AND O. KARIV, *Algorithms for edge coloring bipartite graphs and multigraphs*, this Journal, 11 (1981), pp. 117–129.
[4] J. HOPCROFT AND R. KARP, *An $n^{5/2}$ algorithm for maximal matchings in bipartite graphs*, this Journal, 2 (1975), pp. 225–231.

SIAM J. COMPUT.
Vol. 11, No. 3, August 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1103-0013 $01.00/0

COMPUTATIONAL POWER IN QUERY LANGUAGES*

HENRY W. DAVIS† AND LEON E. WINSLOW†

Abstract. Primitive recursion, recursion and partial recursion are defined for languages which query a relational data base. Necessary and sufficient conditions for a language to satisfy these properties are given. The computational power of several extensions of the relational algebra is considered.

Key words. recursive functions, relational data base query language, relational algebra, computability

1. Introduction and basic definitions. This paper gives necessary and sufficient conditions for a query language on a relational data base to support primitive recursion, recursion or partial recursion. The purpose is to express these concepts in terms of the more familiar ones of recursive function theory. We restrict ourselves to languages which are at least as strong as the relational algebra and ask what additional strength is necessary. The main result is the theorem of § 2. In § 3 we give some examples. A model-theoretic approach to this problem may be found in [5]. (See especially Theorem 3.3.) Their conditions are very different from ours.

Let $J=\{0, 1, \cdots\}$ denote the natural numbers. For $m=1, 2, \cdots$ let ν^m be the set of all finite subsets of J^m. Let $a=(a_1, \cdots, a_k)$ be a tuple of positive integers and let q be a positive integer. A *query of type a and degree q* (*query*, for short) is a partial map from $\nu^{a_1}x \cdots x\nu^{a_k}$ to ν^q. A query is *partial recursive* if and only if it is a partial effectively computable function on its domain. It is *recursive* if and only if it is partial recursive and total. It is *primitive recursive* if and only if it is partial recursive, total and computable without use of unbounded minimization. A *data base* is a tuple of finite relations over J. Thus queries map data bases into finite relations.

A *query language L* is a set of expressions: Expressions are classified analogously to queries: Let $a=(a_1, \cdots, a_k)$ be a tuple of positive integers and take $q \geqq 1$. An *expression of type a and degree q* (*expression*, for short) is a partial map from $\nu^{a_1}x \cdots x\nu^{a_k}$ to ν^q. We require that the expressions in a query language be closed under composition.

Let Q and E be, respectively, a query and an expression of the same type and degree. If Q and E are undefined on exactly the same tuples of their domain and agree where they are defined, then we say the expression *E represents Q*. A query language *L supports partial recursion* if and only if for every partial recursive query Q there is an expression E in L such that E represents Q. We say *L is partial recursive* if and only if it supports partial recursion and every expression in L represents some partial recursive query. Analogous definitions hold for *recursion* and *primitive recursion*.

We describe a simple algebraic query language called BA, for "basic algebra." In it expressions are defined as follows:

⟨expression name⟩INPUT(⟨list of relation names⟩)

⟨degree specification⟩

⟨segment⟩

RETURN(⟨relation name⟩)

* Received by the editors December 19, 1979, and in final revised form September 15, 1981.

† Department of Computer Science, Wright State University, Dayton, Ohio 45435.

The ⟨list of relation names⟩ following INPUT consists of formal arguments representing relations. Upper case Roman letters, sometimes subscripted, are used to represent the names of relations, formal arguments representing relations, variables denoting relations, and expressions. There are an infinite number of such names. All relations named in an expression are of fixed degree. The degree is specified via

$$\langle\text{degree specification}\rangle ::= ((\langle\text{relation name}\rangle, \langle\text{degree}\rangle), \cdots),$$

where ⟨degree⟩ is an integer. A ⟨segment⟩ in BA is a finite sequence of assignment statements. Segments are denoted by an upper bar, e.g. $\bar{S}$. An assignment statement has the form

$$\langle\text{relation name}\rangle \leftarrow \langle\text{infix string}\rangle.$$

⟨infix string⟩ combines relations via the operators union, set difference, Cartesian product, projection and selection. We shall use here the notation of [8], § 4.1. For example $\pi_{1,4}(R)$ is the projection of R onto columns 1,4 and $\sigma_{2>3}(R)$ selects from R tuples whose second component exceeds the third. $\varnothing$ denotes the empty relation.

The left sides of assignment statements in an expression are all local variables. The ⟨relation name⟩ following RETURN is one of these local variables and its value is returned when the expression is executed. The right side of assignment statements may contain local variables, formal arguments occurring in the INPUT list, explicitly defined relations (e.g., {(1,2), (1,3)}), and global variables. Global variables are relations which must be given values outside of the expression before the expression is executed.

For example,

$$\begin{aligned} &E \text{ INPUT } (R)\\ &\quad ((R, 2), (X_0, 2), (X_1, 2))\\ &\quad X_0 \leftarrow \pi_{1,4}(\sigma_{2=3}(R \times R))\\ &\quad X_1 \leftarrow R \cup X_0\\ &\quad \text{RETURN } (X_1) \end{aligned}$$

defines an expression which maps the binary relation R into the union of R and the composition $R \circ R$. To evaluate an expression such as E, above, on a relation S and put the result in T one writes $T \leftarrow E(S)$.

It is not difficult to see that the queries represented by expressions in BA are exactly those represented by Codd's relational algebra [4].

PROPOSITION 1.1. (1) *Define* MAX *to be a unary operator on relations by* $R \to \{(\max\{r_1\}): (r_1, \cdots, r_k) \in R\}$ *if* $R \neq \varnothing$ *and* $R \to \varnothing$ *otherwise. Then* MAX *may be simulated by an infix string in* BA. (2) *Let* S *be a relation and* $\bar{P}$, $\bar{Q}$ *segments in* BA. *Then the following constructs can be simulated by segments in* BA: (a) IF $S = \varnothing$ THEN $\bar{P}$, (b) IF $S \neq \varnothing$ THEN $\bar{Q}$, (c) IF $S = \varnothing$ THEN $\bar{P}$ ELSE $\bar{Q}$, (d) IF $S \neq \varnothing$ THEN $\bar{P}$ ELSE $\bar{Q}$.

Proof. (1) MAX(R) is simulated by $\pi_1(R) - \pi_2(\sigma_{1>2}(\pi_1(R) \times \pi_1(R)))$. (2) Let us show, for example, how to simulate (a). Denote by $X_1, \cdots, X_n$ all the relations which occur on the left in the assignment statements of $\bar{P}$. Denote S by X_0. Asssume $Y_0, \cdots, Y_n$ do not occur anywhere in $\bar{P}$. Let i_j denote the degree of X_j, $1 \leqq j \leqq n$. The required simulation is $Y_0 \leftarrow X_0, \cdots, Y_n \leftarrow X_n, \bar{P}, X_1 \leftarrow (X_1 - \pi_{1 \cdots i_1}(X_1 \times Y_0)) \cup (\pi_{1 \cdots i_1}(Y_1 \times Y_0)), \cdots, X_n \leftarrow (X_n - \pi_{1 \cdots i_n}(X_n \times Y_0)) \cup (\pi_{1 \cdots i_n}(Y_n \times Y_0))$. □

2. The characterization theorem. Our characterization of recursion in query languages uses several definitions. We say that a language L_1 *has at least the retrieval power of* L_2, denoted $L_1 \geqq L_2$, if and only if all queries which may be represented in L_2 may be represented in L_1. A query language *supports segment retrieval* if and only if it contains an expression of type (1) and degree 1 whose value on unary singleton relations is given by $\{(k)\} \rightarrow \{(0), \cdots, (k)\}$. Intuitively, this property implies the existence of a bounded computation by which large relations may be created from small ones. It is used in proving the lemma below.

Let P be any of the properties "partial recursion," "recursion," or "primitive recursion." By a *P-function* is meant a function from tuples of J to J which has property P. We say a query language L *supports P on rows* if and only if the following is true:

Let $f_1, \cdots, f_q$ be P-functions from J^t to J. Let Q be a query of type (t) and degree q defined by $R \rightarrow \{(f_1(r), \cdots, f_q(r)) : r \in R\}$. Then there is an expression in L representing Q.

The support of P on rows is not as powerful as the support of P. For example, it does not assure that the user can request such interrow information as column addition. The next two definitions are designed to isolate a weak form of interrow retrieval power.

Let R be a t-degree relation. By a *lookup function on R* is meant the characteristic function C_R of R; $C_R: J^t \rightarrow \{0, 1\}$ by $C_R(x) = 1$ if and only if $x \in R$. Most interpreted query languages support such functions. If the user may also request basic calculations in C_R, then the user may get interrow information. For example, if R is unary, then $\max\{C_R(x) * x : x \leqq 10\}$ tells the user what the largest entry under 10 in R is. The basic calculations in C_R which we shall use are functions which are primitive recursive in C_R. (A formal definition may be found in [6, § 45].) For our purposes we need only consider unary relations R. We say that a query language L *supports primitive recursion in unary lookup functions* if and only if the following is true:

Let R be a unary relation, C_R its lookup function and F a function which is primitive recursive in C_R. Then there is an expression of type (1) and degree 1 in L whose value on unary singleton relations is given by $\{(x)\} \rightarrow \{(F(x))\}$.

MAIN THEOREM. *Let $L \geqq$ BA be a query language. Let P be any of the properties "primitive recursion," "recursion," or "partial recursion." Then L supports P if and only if* (0) *L supports segment retrieval,* (1) *L supports P on rows and* (2) *L supports primitive recursion in unary lookup functions.*

Proof. The necessity of the conditions is easy to see. For the sufficiency, assume that L satisfies (0), (1) and (2). Let $a = (a_1, \cdots, a_k)$, where $a_i \geqq 1$, and take $q \geqq 1$. We must show that all queries of type a, degree q, and satisfying property P are represented in L. Let $\rho : \nu^{a_1} \cdots \nu^{a_k} \rightarrow J^k$ and $\sigma : J \rightarrow \nu^q$ be bijections (i.e., $1-1$ onto functions) which we specify later. Let $F : J^k \rightarrow J$ be a partial function. The situation is as in Fig. 1. As F ranges through all partial functions $J^k \rightarrow J$, Q ranges through all queries of type a and degree q, and conversely.

Let Q_ρ be the query of type a and degree k given by $(R_1, \cdots, R_k) \rightarrow \{\rho(R_1, \cdots, R_k)\}$. Let Q_F be the query of type (k) and degree 1 given by $R \rightarrow \{(F(r)) : r \in R\}$. Let Q_σ be the query of type (1) and degree q given by $S \rightarrow \{(\sigma(\max\{s\})) : s \in S\}$. We shall define ρ, σ in such a way that their inverses are primitive recursive (i.e., effectively computable without unbounded minimization) and Q_ρ, Q_σ are primitive recursive queries represented in L. Assuming this has been done, let us prove the rest of the theorem.

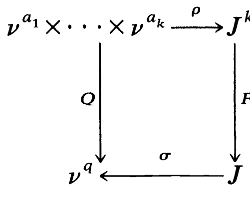


FIG. 1

Let Q be a query of type a, degree q, and satisfying P. Let F correspond to Q as in Fig. 1. Since Fig. 1 commutes and ρ^{-1}, σ^{-1} are primitive recursive, F is a P-function. Since L supports P on rows, Q_F is represented in L. Let E_0, E_1, E_2 represent Q_ρ, Q_F and Q_σ, respectively. Then the composite expression $E_2(E_1(E_0))$ represents Q in L.

It remains to define ρ, σ and prove that they have the desired properties. In order to define ρ we first define a bijection $\psi^m : \nu^m \to J$. Let K^m and L_i^m be Cantor's pairing and projection functions of degree m (see [9, p. 78]): Roughly speaking, K^m maps each distinct tuple of degree m onto a distinct integer and L_i^m maps this integer back into the ith component of the original tuple. Thus, $\{K^m(x) \colon x \in R\}$ is a distinct set of integers for each distinct R. Let $M : \nu^1 \to J$ by M (empty set) $= 0$ and $M(\{j_1, \cdots, j_k\}) = \sum_{i=1}^k 2^{j_i}$. Thus the binary representation of $M(S)$ has a 1 in the $(i+1)$st place from the right if and only if $i \in S$. We now define $\psi^m(R) = M(\{K^m(x) \colon x \in R\})$, for all $R \in \nu^m$. ρ is now defined by

$$\rho(R_1, \cdots, R_k) = (\psi^{a_1}(R_1), \cdots, \psi^{a_k}(R_k)).$$

σ is defined to be the inverse of ψ^q. Since K and M are bijections, so are ψ^m, ρ and σ. The following lemma completes the proof of the theorem.

LEMMA. *Assume* $L \geqq \mathrm{BA}$ *satisfies* (0), (1) *and* (2) *of the theorem. Let* ρ, σ *be as above. Then* ρ^{-1}, σ^{-1} *are primitive recursive and* Q_ρ, Q_σ *are primitive recursive queries represented in* L.

Proof. We consider Q_ρ first. Let R be a relation of degree m. Consider the maps

$$R \xrightarrow{Q_1} \{(2^{K^m(r)}) \colon r \in R\} = T,$$

$$T \xrightarrow{Q_2} \{(\max\{t\}) \colon (t) \in T\} = \{(s)\},$$

$$\{(s)\} \xrightarrow{Q_3} \left\{\left(\sum_{j \leqq s} C_T(j) * j\right)\right\} = \{(\psi^m(R))\}.$$

Q_1 is represented in L by (1). Q_2 is represented in L by Proposition 1.1 (1) and the fact that $L \geqq \mathrm{BA}$. Q_3 is represented in L by (2). As L is closed under composition the query $R \to \{\psi^m(R)\}$ is represented in L. Since BA supports Cartesian products and $L \geqq \mathrm{BA}$, Q_ρ is represented in L. Evidently Q_ρ is primitive recursive.

To see that Q_σ is represented in L let K be a unary relation over J. Define $G : J^2 \to J$ by

$$G(k, j) = \begin{cases} j & \text{if there is a 1 in the } j\text{th place from the right in the binary expansion of } k,\ j \geqq 1, \\ 0 & \text{otherwise.} \end{cases}$$

Consider the maps

$$K \xrightarrow{Q_0} \{(\max\{k\}) : (k) \in K\} = L,$$

$$L \xrightarrow{Q_1} \{(x, j) : (0 \leqq j \leqq x) \wedge (x \in L)\} = M,$$

$$M \xrightarrow{Q_2} \{(G(m)) : m \in M\} = N,$$

$$N \xrightarrow{Q_3} N - \{(0)\} = P,$$

$$P \xrightarrow{Q_4} \{(L_1^q(p-1), \cdots, L_q^q(p-1)) : p \in P\} = Q_\sigma(K).$$

As before, each of the Q_i is represented in L and hence L supports Q_σ. (Q_1 is represented in L because L satisfies (0) and supports Cartesian products.) The formulation above shows that Q_σ is primitive recursive. Finally, it is apparent from their construction that ρ and σ have primitive recursive inverses. □

3. Examples. Practical languages for (relational) data bases not only have at least the retrieval power of BA but also allow arithmetic expressions to occur within query expressions. In this section we indicate, through examples, what additional mechanisms do to the computational power of such languages. To this end we shall addend to BA a basic arithmetic capability. We want one that is simple, compatible with the other operators of BA, and sufficiently weak that it can be simulated in most query languages providing arithmetic capability. Let $\overline{\text{BA}}$ be the same as BA except that we now allow a successor operator, NEXT, to be used in infix strings. NEXT is defined on relations R of arbitrary degree by NEXT $(R) = \{(r_1+1, \cdots, r_k+1) : (r_1, \cdots, r_k) \in R\}$. Let R be a relation with attributes $A1, \cdots, AK$; Figs. 2(a), (b) and (c) show, respectively, how the assignment statement $S \leftarrow$ NEXT (R) is simulated in QUEL [7], SEQUEL [2] and QUERY-BY-EXAMPLE [10]. Thus $\overline{\text{BA}}$ would appear to be a relatively weak arithmetic extension of BA.

```
RANGE OF X IS R
RETRIEVE INTO S(A1 = X.A1 + 1, ···, AK = X.AK + 1)
```

(a)

```
ASSIGN TO S(A1, ···, AK)
  SELECT A1 + 1, ···, AK + 1
  FROM R
```

(b)

R	*A*1	···	*AK*
	<u>*X*1</u>		<u>*XK*</u>

I.S. *I.*	*A*1	···	*AK*
I.	<u>*X*1</u>+1		<u>*XK*</u>+1

(c)

FIG. 2.

We now describe three extensions of $\overline{\text{BA}}$. By an *E-segment* we mean an "extended segment." *E*-segments are defined in the context of particular languages below.

The query language $\text{BL}(\overline{\text{BA}})$ is obtained by closing $\overline{\text{BA}}$ with respect to "bounded looping." More specifically, expressions in $\text{BL}(\overline{\text{BA}})$ are like those of $\overline{\text{BA}}$ except that E-segments rather than segments are used: (i) a segment is an E-segment; (ii) the juxtaposition of two E-segments is an E-segment; and (iii) let $\bar{S}$ be an E-segment and R a relation; then

DO R TIMES

$\bar{S}$

END-DO

is an E-segment. The meaning of (ii) is that the first and then the second E-segments are to be executed. Let MAX be the operator of Proposition 1.1. The meaning of (iii) is that, if $\text{MAX}(R) = \{(r)\}$, then $\bar{S}$ is to be executed exactly r times. If $r = 0$ or $\text{MAX}(R) = \varnothing$, then $\bar{S}$ is not to be executed.

The query language $\text{UL}(\overline{\text{BA}})$ is obtained by closing $\overline{\text{BA}}$ with respect to "unbounded looping." Expressions in $\text{UL}(\overline{\text{BA}})$ are like those in $\text{BL}(\overline{\text{BA}})$ with E-segments now defined as follows: (i)′ and (ii)′ are the same as (i), (ii); (iii)′ let S be an E-segment and R a relation; then

DO WHILE $R = \varnothing$

$\bar{S}$

END-DO

is an E-segment. The meaning of (iii)′ is that if $R \neq \varnothing$ then $\bar{S}$ is not performed. Otherwise it is performed repeatedly until $R \neq \varnothing$. If this never happens, then the E-segment is not defined. In this case the corresponding expression is not defined on the current arguments.

The test for $R = \varnothing$ in (iii)′ may equivalently be replaced by a test for inequality to $\varnothing$: Using Proposition 1.1 (2) the construct of (iii)′ may be simulated by IF $R = \varnothing$ THEN $R_1 \leftarrow \{(1)\}$ ELSE $R_1 \leftarrow \varnothing$, DO WHILE $R_1 \neq \varnothing$, $\bar{S}$, IF $R \neq \varnothing$ THEN $R_1 \leftarrow \varnothing$, END-DO; here we assume R_1 is not in $\bar{S}$. Similarly for the converse. Both forms are convenient. Notice, however, that Proposition 1.1 (2) carries over to $\text{UL}(\overline{\text{BA}})$ only so long as $\bar{P}$, $\bar{Q}$ are segments; they may not be E-segments.

$\text{UL}(\overline{\text{BA}})$ slightly resembles the Chandra–Harel language QL in [3]. However, QL is a much more general language! In it data bases may be over undefined entities as opposed to the natural numbers considered here. In QL the degree of a relation varies dynamically; in $\text{UL}(\overline{\text{BA}})$ relation degrees are permanently fixed by the degree specification. The projection, $\downarrow$, used in QL is different from that of $\text{UL}(\overline{\text{BA}})$. $\downarrow$ fixes the number of columns thrown out letting the number kept vary each time a statement is executed. Such varying does not occur in $\text{UL}(\overline{\text{BA}})$.

Let R be a relation and $\bar{Q}$ an E-segment possibly referencing a local relation variable L. We shall use below the fact that the following construct may be simulated by an E-segment in $\text{BL}(\overline{\text{BA}})$ and in $\text{UL}(\overline{\text{BA}})$: DO ONCE FOR EACH $r \in R$, $L \leftarrow \{r\}$, $\bar{Q}$, END-DO. The proof is an easy exercise and uses the NEXT operator.

THEOREM 3.1. $\text{BL}(\overline{\text{BA}})$ *is primitive recursive and* $\text{UL}(\overline{\text{BA}})$ *is partial recursive.*

Proof. The main part of the argument consists of verifying conditions (0), (1), (2) of the main theorem. (0) is straightforward. It is easy to see that BA supports unary lookup functions so the proof of (2) is essentially the same as that of (1). Consider $\text{UL}(\overline{\text{BA}})$ and, for simplicity, restrict attention to queries of degree 1. Let $R_f = \{(f(r)) \colon (r) \in R\}$. Let $\mathscr{F}$ be the class of functions f for which $R \to R_f$ is represented in

UL($\overline{\text{BA}}$). One must show that $\mathcal{F}$ contains the partial recursive functions. We shall show that $\mathcal{F}$ is closed under unbounded minimization and leave the rest for the reader. Let $g: J^{n+1} \to J$ be in $\mathcal{F}$. Define $f: J^n \to J$ by $f(y) = \mu_x[g(x, y) = 0]$, where y is an n-tuple and μ is the usual minimization function. Let $\bar{G}$ be the E-segment part of a UL($\overline{\text{BA}}$) expression representing $N \to N_g$. Let R have degree n. The following expression represents $R \to R_f$:

INPUT (R)

⟨degree specification for R, R_f, L, X_0, X_1 and the relations of $\bar{G}$⟩

$R_f \leftarrow \varnothing$

DO ONCE FOR EACH $R \in R$

$L \leftarrow \{r\}$

$X_0 \leftarrow \{(0)\}$

$X_1 \leftarrow \{(0)\}$

DO WHILE $X_0 \neq \varnothing$

$N \leftarrow X_1 \times L$

$\bar{G}$

$X_0 \leftarrow N_g - \{(0)\}$

IF $X_0 \neq \varnothing$ THEN $X_1 \leftarrow$ NEXT (X_1)

END-DO

$R_f \leftarrow R_f \cup X_1$

END-DO

RETURN (R_f)

Here we assume R, R_f, L, X_0, and X_1 are names which do not occur in $\bar{G}$. □

Let G be an expression in $\overline{\text{BA}}$ of type (t) and degree t. Let R have degree t. Let $G^*(R)$ denote $R \cup G(R) \cup G(G(R)) \cup \cdots$, provided this is finite. If it is not finite, then $G^*(R)$ is not defined. We define LFP($\overline{\text{BA}}$) to be the same as $\overline{\text{BA}}$ except that, for $\overline{\text{BA}}$ expressions G of type (t) and degree t, G^* is a valid operator to use in infix strings. Whenever G^* is not defined, the expression containing it is undefined on the current arguments. The definition for G^* is suggested by Aho and Ullman [1, § 5, method (3)] in connection with the calculation of least fixpoints over monotone expressions.

THEOREM 3.2. LFP($\overline{\text{BA}}$) *is partial recursive.*

Proof. As LFP($\overline{\text{BA}}$) expressions are obviously effectively computable, it suffices to show that LFP($\overline{\text{BA}}$) $\geqq$ UL($\overline{\text{BA}}$). It is not difficult to show that any E-segment in UL($\overline{\text{BA}}$) which has nested DO WHILE's may be rewritten without the nesting. Thus it suffices to show how LFP($\overline{\text{BA}}$) can simulate the construct

DO WHILE $R = \varnothing$

(†) $\bar{S}$

END-DO

where $\bar{S}$ is a segment, not an E-segment.

The simulation of (†) is via a construct of the form $\bar{P}$, $Y_0 \leftarrow G^*(Y_1)$, $\bar{Q}$. Here Y_1 is a carefully chosen operand, $\bar{P}$ prepares Y_1 and $\bar{Q}$ decodes Y_0 so that the total effect is as though $\bar{S}$ had been executed the required number of times. $\bar{P}$, $\bar{Q}$ are segments, not E-segments. G is an expression in $\overline{\text{BA}}$.

Let $X_1, \cdots, X_k$ be R along with all local relation variables which occur in $\bar{S}$. They will be passed to G coded into the operand Y_1. G decodes, from its operand, the current values of $X_1, \cdots, X_k$. If $R \neq \emptyset$, then G returns its input. If $R = \emptyset$, then it executes $\bar{S}$ and returns a relation into which is coded the new values of $X_1, \cdots, X_k$. In general, $G^n(Y_1)$ returns the status of $\bar{S}$'s local relations and R after $\bar{S}$ has been executed n times. $\bar{Q}$ must obtain from $Y_0 = \bigcup_{n=0}^{\infty} G^n(Y_1)$ the latest such status. This is made possible by having G increment a counter, which is coded into its input, and return the new value as part of its output.

The coding mechanism is as follows. Y_1 is a product of $k+1$ relations: $R_0 \times R_1 \times \cdots \times R_k$. $R_0 = \{(0)\}$. For $i \geqq 1$ R_i has X_i coded into it. We cannot set $R_i = X_i$ because some X_i may be $\emptyset$ and that would make $Y_1 = \emptyset$. Thus we set

$$R_i = \begin{cases} \{(1)\} \times X_1 & \text{if } X_i \neq \emptyset, \\ \{(0)\}^{\text{degree } (X_i)+1} & \text{if } X_i = \emptyset. \end{cases}$$

From Y_1, G uses projections to determine the values of each X_i prior to executing $\bar{S}$. This is possible because the degree of each X_i is fixed in the UL(BA)-expression containing (†). It is also declared in the degree specification of G. The first component of R_i reveals whether or not X_i is empty.

If $\bar{S}$ is executed ($R = \emptyset$), then the new values of X_i are coded into the returned relation. They are coded using $R_1, \cdots, R_k$ as before; but the new R_0 is the old R_0 incremented by one: $R_0 \leftarrow \text{NEXT}(R_0)$. Thus the output from G has the same form as the input, but reflects a more updated version of $X_1, \cdots, X_k$. If (†) is undefined, then R_0 will differ in each $G^n(Y_1)$ so that the simulation will also be undefined. If (†) is defined, then R is eventually $\neq \emptyset$ so $G^i(Y_1) = G^{i+1}(Y_1) = \cdots$ for some i. In this case the simulation is also defined. $\bar{Q}$ decodes the latest status of $X_1, \cdots, X_k$ from Y_0 as follows: Using the MAX operator, the tuples of Y_0 with the largest first coordinate are isolated. The updated X_i are obtained from these via projections. □

Acknowledgment. The authors wish to thank the referee for patiently pointing out many ways to improve the original paper.

REFERENCES

[1] A. V. AHO AND J. D. ULLMAN, *Universality of data retrieval languages*, Proc. 6th ACM Symposium on Principles of Programming Languages, San Antonio, TX, Jan. 1979.

[2] D. D. CHAMBERLIN, M. M. ASTRAHAN, K. P. ESWARAN, P. P. GRISFITHS, R. A. LORIE, J. W. MEHL, P. REISNER, AND B. W. WADE, SEQUEL 2: *a unified approach to data definition, manipulation and control*, IBM J. Res., 20 (1976), pp. 560–575.

[3] A. K. CHANDRA AND D. HAREL, *Computable queries for relational data bases*, J. Comput. System. Sci., 21 (1980), pp. 156–178.

[4] E. F. CODD, *A relational model for large shared data banks*, Comm. ACM, 13 (1970), pp. 377–387.

[5] B. JACOBS AND D. LAVINE, *On completeness of database query languages*, Tech. Rep. 699. University of Maryland Computer Science Center, College Park, Maryland 20742, 1978.

[6] S. C. KLEENE, *Introduction to Metamathematics*, Van Nostrand, New York, 1952.

[7] M. STONEBRAKER, E. WONG, P. KREPS AND G. HELD, *The design and implementation of* INGRES, ACM Trans. Database Systems, 1 (1976), pp. 189–222.

[8] J. D. ULLMAN, *Principles of Database Systems*, Computer Science Press, Potomac, MD, 1980.

[9] A. YASUHARA, *Recursive Function Theory and Logic*, Academic Press, New York, 1971.

[10] M. M. ZLOOF, QUERY-BY-EXAMPLE: *a database language*, IBM Systems J., 16 (1977), pp. 324–343.

SIAM J. COMPUT.
Vol. 11, No. 3, August 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1103-0014 $01.00/0

APPROXIMATION ALGORITHMS FOR THE SET COVERING AND VERTEX COVER PROBLEMS*

DORIT S. HOCHBAUM†

Abstract. We propose a heuristic that delivers in $O(n^3)$ steps a solution for the set covering problem the value of which does not exceed the maximum number of sets covering an element times the optimal value.

Key words. heuristics, complexity, set covering, vertex cover, linear programming, Russian method

The set covering problem seeks a collection of sets that cover all elements of another given set at minimum cost. The problem is formulated as $SC^* = \min c^T \cdot x$ subject to $Ax \geqq e$ for x a 0–1 n-vector, $e = (1, \cdots, 1)$ and A a 0–1 matrix each column of which is the incidence vector of one of the sets. This problem is NP-hard even when $c = e$ and each row of the matrix has only two ones (see Karp [5]). In this case the problem is known as the vertex cover problem.

One heuristic with guaranteed worst case behavior for the set covering problem is the greedy. The heuristic solution value does not exceed $\sum_{j=1}^{d} 1/j$ times the optimum where d is the maximum column sum of A (Chvátal [1]). This bound is tight even for the vertex cover problem. Johnson [4] describes unweighted graphs of maximum vertex degree d in which the greedy vertex cover solution is exactly $\sum_{j=1}^{d} 1/j$ times the optimum. The heuristic proposed yields a bound of 2 on the ratio between the heuristic solution and the optimal vertex cover solution.

Let $X(S)$ be the characteristic n-vector of a set $S \subset \{1, 2, \cdots, n\}$. Let f be the maximum row sum of A, and f_S the maximum row sum in the submatrix of A defined by the columns with subscripts in S. Denote by A_j the jth row of the matrix.

THEOREM. *It takes a polynomial number of steps to find a cover the value of which is at most f times the optimum.*

Proof. Let x^* be the optimal solution of the linear program $\min c^T \cdot x$ subject to $Ax \geqq e$, $x \geqq 0$. Let y^* be the optimal dual solution, i.e., that solves the problem $\max e^T \cdot y$ subject to $y^T A \leqq c$, $y \geqq 0$. It is easy to verify that for the set $J_s = \{j | y^T A_j = c_j\}$ the vector $x(J_s)$ is a cover. Also,

$$\begin{aligned} c^T \cdot x(J_s) &= (y^T \cdot A) \cdot x(J_s) \\ &\leqq \max_{j \in J_s} (A_j \cdot x(J_s)) \cdot (y^T \cdot e) \\ &= f_{J_s} \cdot (c^T \cdot x^*) \leqq f_{J_s} \cdot SC^*. \end{aligned}$$

In order to obtain the set J_s one needs to solve a linear program and that can be done in polynomial time. Q.E.D.

The set J_s is a cover but not necessarily a prime cover, that is, it may be possible to find a proper subset of J_s that is also a feasible cover. For any such subset S of J_s we have as a corollary of the theorem that $c^T \cdot x(S) \leqq f_S \cdot SC^*$. Such subset can be derived by applying a greedy procedure or any other procedure for eliminating redundant sets from J_s.

* Received by the editors July 1, 1980, and in revised form September 21, 1981. This research was supported in part by the National Science Foundation under grant ECS-8204695.

† School of Business Administration, University of California, Berkeley, California 94720.

One special case of a subset of J_s that is a feasible cover is $J_p = \{j | x_j^* > 0\}$. It follows from complementary slackness that $J_p \subseteq J_s$. J_p is obviously a cover though not necessarily a prime cover, so a further elimination of members of this set may still be possible.

Another special case is the set $J_f = \{j | x_j^* \geqq 1/f\}$. $J_f \subseteq J_p$ and is a feasible cover. The advantage of this particular cover is that we do not need to evaluate the exact fractional solution x^*. Instead, the only information necessary is whether $x_j^* \in [0, 1/f)$ or $x_j^* \in [1/f, 1]$. Employing the Russian method as proposed by Padberg and Rao [7] requires $O(n^3 \cdot H)$ steps where H is a parameter used to determine the radius of the initial sphere and the perturbation vector $2^{-H} \cdot e$ to the right-hand side of the constraints. Hence the algorithm requires only $O(n^3 \cdot \max[2, \log_2 2f])$ steps to derive the set J_f. Therefore, the LP-heuristic needs only $O(n^3)$ steps to deliver a solution the value of which is at most f times the optimum. This algorithm is comparable with efficient network flow algorithms that deliver the fractional solution for the vertex cover problem in dense graphs (Galil's [3] algorithm for instance requires $O(n^{5/3}m^{2/3})$ steps for graphs with n vertices and m edges). The above discussion leads to the following corollary of the theorem.

COROLLARY. *It takes $O(n^3)$ steps to obtain a cover the value of which is at most f times the set covering optimal solution value.*

For the case of the vertex cover problem the heuristic value does not exceed twice the optimum. We now propose a heuristic that improves this bound for unweighted graphs. The dual problem in this case is also known as the maximum cardinality matching problem when the condition "y a 0–1 vector" is added. Consider the following procedure derived from the matching solution obtained by Edmonds' algorithm [2]: Select all vertices in each blossom, then select all T-labelled vertices and any $2k-1$ of the $2k$ out-of-tree vertices. Any subgraph spanned by an odd cycle of length can always be covered by any $r-1$ vertices or at least $(r+1)/2$ vertices, and any blossom is constructed of pseudonodes spanned by odd cycles. The number of vertices selected by the above procedure is at most $(r-1)/[\frac{1}{2}(r+1)]$ times the cardinality of the optimal vertex cover, where r is the length of the longest odd cycle in the graph. Since $2(r-1)/(r+1) = 2-[4/(r+1)]$ we obtained a bound strictly less than 2. (Note that for graphs with no odd cycles, i.e. for bipartite graphs, this procedure is optimal.)

REFERENCES

[1] V. CHVÁTAL, *A greedy-heuristic for the set-covering problem*, Math. Oper. Res., 4 (1979) pp. 233–235.

[2] J. EDMONDS, *Paths, trees and flowers*, Canad. J. Math., 17 (1975), pp. 449–467.

[3] Z. GALIL, *A new algorithm for the maximal flow problem*, Proc. 19th Annual Symposium on Foundations of Computer Science, 1978, pp. 231–248.

[4] D. S. JOHNSON, *Approximation algorithms for combinatorial problems*, J. Comput. System Sci. 9 (1974), pp. 256–278.

[5] R. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds. Plenum Press, New York, 1972, pp. 85–103.

[6] G. L. NEMHAUSER AND L. E. TROTTER, *Vertex packings: structural properties and algorithms*, Mathematical Programming 8 (1975), pp. 232–248.

[7] M. W. PADBERG AND M. R. RAO, *The Russian method for linear inequalities II: approximate arithmetic*, W.P. #202, New York University, Graduate School of Business Administration, New York, 1980.

SIAM J. COMPUT.
Vol. 11, No. 3, August 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1103-0015 $01.00/0

TWO SPECTRA OF SELF-ORGANIZING SEQUENTIAL SEARCH ALGORITHMS*

AARON M. TENENBAUM† AND RICHARD M. NEMES†

Abstract. Two sets of algorithms for searching and dynamic reorganization of linear lists are presented. Each set forms a spectrum, with the well-known move-to-front and transposition algorithms at the extrema. A linear ordering on the stationary expected search cost of the algorithms in each of the two spectra is established over a restricted set of probability distributions.

Key words. sequential search, search time, move-to-front algorithm, transposition algorithm, self-organizing list

Introduction. Consider a sequential list of records $R_1, \cdots, R_n$ (where $n > 2$) in which each record R_i is identified by a unique key K_i. A sequential search compares a given argument K with each of the K_i in turn until K is found to equal some K_i or until K is found unequal to them all. In the former case, the search is successful; in the latter, it is unsuccessful. The number of comparisons made in the case when $K = K_i$ is i. If p_i is the probability that the search argument K in a successful search equals K_i, then the expected number of comparisons for a successful search equals $\sum_{i=1}^n i \cdot p_i$. Note that the assumption of success implies that $\sum_{i=1}^n p_i = 1$.

If the probabilities p_i are known, the expected number of comparisons is minimized by arranging the R_i so that $p_1 \geqq p_2 \geqq \cdots \geqq p_n$. However, the p_i are rarely known in advance. If π is a permutation of the integers 1 to n and $\pi(i)$ is the integer in the ith position of π, then we may define cost (π) as the expected number of comparisons in a sequential search, where the n records are arranged in the order $R_{\pi(1)}, R_{\pi(2)}, \cdots, R_{\pi(n)}$. Note that

$$\text{cost}(\pi) = \sum_{i=1}^{n} i \cdot p_{\pi(i)}. \tag{1}$$

We may extend the sequential search algorithm to an algorithm in which the order of the records is altered each time that a sequential search occurs. The most useful type of alteration is where the retrieved record is moved forward one or more positions in the list. In this way more commonly accessed records move towards the front of the list so that fewer comparisons are needed on subsequent retrievals.

Two well-studied "self-organizing" sequential search algorithms are the move-to-front and transposition rules. In the move-to-front rule [1]–[3], [5]–[10], a retrieved record is moved to the front of the list; in the transposition rule [1]–[2], [8]–[10], the record is transposed with its predecessor on the list.

Consider any rule A which moves a retrieved record forward one or more positions (unless the record is in the first position). The $n!$ permutations of the records in the list are states of a system in which a retrieval transforms one state into another. Since it is possible to reach any state from any other by a sequence of one or more transformations, the transformations form an irreducible Markov chain in which each permutation π has its own stationary probability $\text{prob}_A(\pi)$ [4]–[6]. Thus, we may define the expected cost of such a self-organizing rule A, ec (A), by

$$\text{ec}(A) = \sum_{\text{all } \pi} \text{prob}_A(\pi)\,\text{cost}(\pi). \tag{2}$$

* Received by the editors June 4, 1980, and in revised form July 14, 1981. This material is based upon work supported by the National Science Foundation under grants IST79-17568 and IST80-21350.

† Department of Computer and Information Science, Brooklyn College, City University of New York, Brooklyn, New York 11210.

Given two such rules, A and B, $\mathrm{ec}(A) < \mathrm{ec}(B)$ implies that rule A is eventually "better" than rule B; rule A will require fewer comparisons over a large number of retrievals.

Rivest [9] proved that ec (transposition) $\leqq$ ec (move-to-front) and conjectured that the transposition rule has lower asymptotic cost than all other rules, regardless of the probabilities $p_1, \cdots, p_n$. Yao (cited in [1], [9]) showed that a distribution of probabilities can be found for which the transportation rule is as good as the optimal rule, assuming an optimal rule exists. This means that if any rule is best, regardless of probability distribution, it must be the transposition rule.

A recent paper by Kan and Ross [11] has proved that the transposition rule is optimal for any probability distribution in which $p_2 = p_3 = \cdots = p_n$ in the class of self-organizing algorithms for which no later element is moved to an earlier position upon retrieval than an earlier element.

In this paper we examine two spectra of rules in which the transposition rule is at one end and the move-to-front is at the other over the restricted class of probability distributions for which $p_2 = p_3 = \cdots = p_n$. We show that both these spectra are ordered by expected cost, with the transposition rule having the lowest cost, the move-to-front having the highest and the expected costs of the remaining rules in each spectrum linearly ordered between these two extrema.

An expected cost theorem. Let us assume that $p_2 = p_3 = \cdots = p_n$ so that there is a single record R_1 with retrieval probability p_1 and all other records have the same retrieval probability which we reference as p_2. Of course, $p_2 = (1 - p_1)/(n-1)$. Then the $n!$ steady states of the general search algorithm can be reduced to n states, $s_1, \cdots, s_n$, where s_i is the state in which the record R_1 is in position i. Since all permutations with R_1 in position i yield the same cost, the order of the remaining $n-1$ records is irrelevant.

Given a rule A, let $q_i(A)$ be the steady-state probability of s_i under rule A. We may abbreviate $q_i(A)$ to simply q_i when the rule under discussion is known. Applying (1) the cost of searching a list in state s_i is given by

$$\mathrm{cost}(s_i) = i \cdot p_1 + (n(n+1)/2 - i)p_2.$$

We now show that $\mathrm{cost}(s_{i+1}) \geqq \mathrm{cost}(s_i)$ if and only if $p_1 \geqq p_2$. This is true since

$$\begin{aligned}\mathrm{cost}(s_{i+1}) - \mathrm{cost}(s_i) &= (i+1)p_1 + (n(n+1)/2 - i - 1)p_2 - ip_1 - (n(n+1)/2 - i)p_2 \\ &= p_1 - p_2,\end{aligned}$$

which is nonnegative if and only if $p_1 \geqq p_2$.

Applying (2) we may write a formula for the expected cost of a rule A under a distribution in which $p_2 = \cdots = p_n$:

$$\mathrm{ec}(A) = \sum_{i=1}^{n} q_i(A)\, \mathrm{cost}(s_i). \tag{3}$$

It is intuitively clear that given two rules A and B and a distribution $p_1 > p_2 = \cdots = p_n$, rule A has lower expected cost than B if $q_i(A) > q_i(B)$ for low values of i. That is, the rule in which the most probable search argument (R_1) is more likely to be at the front of the list has lower expected cost. Of course, if $q_i(A) > q_i(B)$ for low i, then $q_i(B) > q_i(A)$ for high i since $\sum q_i(A) = \sum q_i(B) = 1$. Similarly, if $p_1 < p_2 = \cdots = p_n$, then rule A has lower expected cost than B if $q_i(A) < q_i(B)$ for low values of i. The following expected cost theorem formalizes this.

THEOREM 1. *Let A and B be two self-organizing sequential search rules. Then if* $p_1 \geqq p_2 = \cdots = p_n$ *and there exists a* $1 \leqq k \leqq n-1$ *such that for all* $i \leqq k$, $q_i(A) \geqq q_i(B)$, *and for all* $i > k$, $q_i(A) \leqq q_i(B)$, *then* $\mathrm{ec}(A) \leqq \mathrm{ec}(B)$. *Similarly, if* $p_1 \leqq p_2 = \cdots = p_n$ *and there exists a* $1 \leqq k \leqq n-1$ *such that for all* $i \leqq k$, $q_i(A) \leqq q_i(B)$, *and for all* $i > k$, $q_i(A) \geqq q(B)$, *then* $\mathrm{ec}(A) \leqq \mathrm{ec}(B)$.

Proof. We prove only the first half, where $p_1 \geqq p_2$, since the proof of the second half is analogous. From (3) we have that

$$\begin{aligned}\mathrm{ec}(A) - \mathrm{ec}(B) &= \sum_{i=1}^{n} q_i(A)\,\mathrm{cost}(s_i) - \sum_{i=1}^{n} q_i(B)\,\mathrm{cost}(s_i)\\ &= \sum_{i=1}^{k} \mathrm{cost}(s_i)(q_i(A)-q_i(B)) - \sum_{i=k+1}^{n} \mathrm{cost}(s_i)(q_i(B)-q_i(A))\\ &\leqq \mathrm{cost}(s_k)\sum_{i=1}^{k}(q_i(A)-q_i(B)) - \mathrm{cost}(s_{k+1})\sum_{i=k+1}^{n}(q_i(B)-q_i(A)).\end{aligned}$$

But

$$\begin{aligned}\sum_{i=1}^{k}(q_i(A)-q_i(B)) &= \sum_{i=1}^{k} q_i(A) - \sum_{i=1}^{k} q_i(B)\\ &= \left(1-\sum_{i=k+1}^{n} q_i(A)\right) - \left(1-\sum_{i=k+1}^{n} q_i(B)\right) = \sum_{i=k+1}^{n}(q_i(B)-q_i(A)).\end{aligned}$$

Thus, $\mathrm{ec}(A) - \mathrm{ec}(B) \leqq (\mathrm{cost}(s_k) - \mathrm{cost}(s_{k+1}))\sum_{i=1}^{k}(q_i(A)-q_i(B))$. But $\mathrm{cost}(s_k) \leqq \mathrm{cost}(s_{k+1})$ and $q_i(A) \geqq q_i(B)$ for $i \leqq k$. Thus $\mathrm{ec}(A) - \mathrm{ec}(B) \leqq 0$, so that $\mathrm{ec}(A) \leqq \mathrm{ec}(B)$. □

This theorem gives us a tool for proving that rule A has lower expected cost than rule B under a distribution in which $p_2 = \cdots = p_n$. We must show that $p_1 > p_2$ implies that $q_i(A) \geqq q_i(B)$ for all i less than some k and $q_i(A) \leqq q_i(B)$ for all $i \geqq k$ and that $p_1 < p_2$ implies that the reverse inequalities are true.

A spectrum of rules. The first spectrum of rules which we examine consists of $n-1$ rules. Let $1 \leqq k \leqq n-1$. The rule POS (k) operates as follows: if an argument is found in a record at a position $j > k$, that record is moved to position k (and all records in positions k through $j-1$ are moved back one position); if an argument is found at a position $j \leqq k$, that record is moved up one position (and the record which was previously in position $j-1$ is now placed in position j). Note that the rule POS (1) is the move-to-front rule, while the rule POS $(n-1)$ is the transposition rule.

If $p_2 = \cdots = p_n$, we use the notation $q_i(k)$ as an abbreviation for $q_i(\mathrm{POS}(k))$, the stationary probability that R_1 is in position i under rule POS (k). When k is fixed, we refer simply to q_i as an abbreviation for $q_i(k)$.

We would like to show that for $1 \leqq k \leqq n-2$, $\mathrm{ec}(\mathrm{POS}(k)) \geqq \mathrm{ec}(\mathrm{POS}(k+1))$ for all probability distributions in which $p_2 = \cdots = p_n$. For the trivial case in which $p_1 = p_2 = \cdots = p_n$, the following theorem yields this result.

THEOREM 2. *Let A and B be any two self-organizing sequential search rules. Then if* $p_1 = p_2 = \cdots = p_n$, $q_j(A) = q_j(B) = 1/n$ *for all j and, hence,* $\mathrm{ec}(A) = \mathrm{ec}(B)$.

Proof. Obvious by symmetry. □

For the case in which $p_1 \neq p_2$, we wish to show that

(i) for $1 \leqq k \leqq n-2$ and $1 \leqq j \leqq k$, $q_j(k+1) > q_j(k)$ if and only if $p_1 > p_2$, and that

(ii) for $1 \leqq k \leqq n-2$ and $k < j \leqq n$, $q_j(k+1) < q_j(k)$ if and only if $p_1 > p_2$.

These relationships are illustrated in Fig. 1 for the case $p_1 > p_2$; the relative positions of the graphs of $q_j(k)$ and $q_j(k+1)$ are reversed for the case $p_1 < p_2$. (These graphs are illustrated as straight lines for clarity; however, in reality they are more complex, as we shall see.) Together with Theorem 1, these inequalities prove that ec (POS (k)) $\geqq$ ec (POS $(k+1)$).

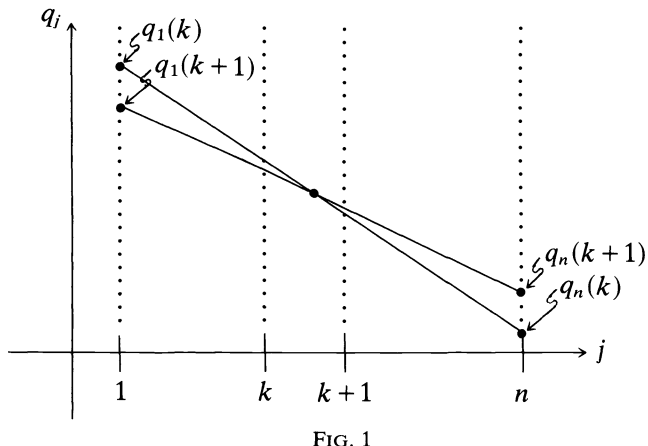


FIG. 1

By the definition of the rules POS (k) and the steady-state probabilities q_i, we may write a series of equations which must hold for rule POS (k) for all $2 \leqq k \leqq n-1$:

$$q_1 = (1-p_2)q_1 + p_1 q_2, \tag{4}$$

$$q_j = p_2 q_{j-1} + (1-p_1-p_2)q_j + p_1 q_{j+1} \quad \text{for all } 1 < j < k, \tag{5}$$

$$q_k = p_2 q_{k-1} + (k-1)p_2 q_k + p_1\left(1 - \sum_{i=1}^{k} q_i\right), \tag{6}$$

$$q_j = (n-j+1)p_2 q_{j-1} + (j-1)p_2 q_j \quad \text{for all } k < j \leqq n. \tag{7}$$

From (4) and (5), we have that

$$\frac{q_j}{q_{j+1}} = \frac{p_1}{p_2} \quad \text{for } 1 \leqq j < k. \tag{8}$$

This yields

$$q_j = \left(\frac{p_1}{p_2}\right)^{i-j} q_i \quad \text{for } 1 \leqq i, j \leqq k. \tag{9}$$

This result, together with (6), yields

$$q_k = \left(\frac{1}{p_1} - (k-1)\frac{p_2}{p_1} + \sum_{i=1}^{k-1} \left(\frac{p_1}{p_2}\right)^{k-i}\right)^{-1}. \tag{10}$$

Although neither (4) nor (6) is valid for $k=1$ (the move-to-front rule), (10) is, nevertheless, valid for $k=1$. This is true since the expression in (10) reduces to the value p_1 in the case $k=1$, and $q_1(1)$ obviously equals p_1 since the probability of R_1 being in the first position under the move-to-front rule equals the probability of the last previous request being for R_1, which equals p_1.

LEMMA 1. *For all* $1 \leqq k \leqq n-2$, $q_k(k+1) > q_k(k)$ *if and only if* $p_1 > p_2$.

Proof. From (9), $q_k(k+1) = (p_1/p_2)q_{k+1}(k+1)$. If we define $r = p_1/p_2$, then (since $p_1 + (n-1)p_2 = 1$) $p_1 = r/(r+n-1)$ and $p_2 = 1/(r+n-1)$. From (10), we have that

$$\frac{1}{q_k(k+1)} - \frac{1}{q_k(k)} = \frac{1}{rq_{k+1}(k+1)} - \frac{1}{q_k(k)}$$
$$= \frac{1}{r}\left(\frac{1}{p_1} - \frac{k}{r} + \sum_{i=1}^{k} r^{k+1-i}\right) - \left(\frac{1}{p_1} - \frac{(k-1)}{r} + \sum_{i=1}^{k-1} r^{k-1}\right)$$
$$= \frac{1}{p_1}\left(\frac{1}{r} - 1\right) + \frac{r(k-1)-k}{r^2} + 1$$
$$= \frac{r^2 + r(k-1-p_2^{-1}) + p_2^{-1} - k}{r^2} = \frac{(r-1)(r-p_2^{-1}+k)}{r^2}$$
$$= (r-1)\left(\frac{p_1 - 1 + kp_2}{p_2}\right)\left(\frac{1}{r}\right)^2.$$

Of the three factors, $(1/r)^2$ is always positive and

$$\frac{p_1 - 1 + kp_2}{p_2} \leqq \frac{p_1 - 1 + (n-2)p_2}{p_2} < \frac{p_1 - 1 + (n-1)p_2}{p_2} = 0$$

so that $(p_1 - 1 + kp_2)/p_2$ is always negative. Therefore, $1/q_k(k+1) - 1/q_k(k) < 0$ if and only if $r - 1 > 0$, which is true if and only if $p_1 > p_2$. □

LEMMA 2. *For* $1 \leqq k \leqq n-2$ *and* $1 \leqq j \leqq k$, $q_j(k+1) > q_j(k)$ *if and only if* $p_1 > p_2$.

Proof. From (9),

$$\frac{q_j(k)}{q_k(k)} = \left(\frac{p_1}{p_2}\right)^{k-j} = \frac{q_j(k+1)}{q_k(k+1)}.$$

Thus, $q_j(k+1) > q_j(k)$ if and only if $q_k(k+1) > q_k(k)$, which by Lemma 1 is true if and only if $p_1 > p_2$. □

LEMMA 3. *For* $1 \leqq k \leqq n-2$, $q_{k+1}(k+1) < q_{k+1}(k)$ *if and only if* $p_1 > p_2$.

Proof. From (7), $q_{k+1}(k) = ((n-k)p_2/(1-kp_2))q_k(k)$. Thus,

$$\frac{q_{k+1}(k)}{q_{k+1}(k+1)} = \frac{(n-k)p_2}{1-kp_2} \cdot \frac{q_k(k)}{q_{k+1}(k+1)}.$$

From (10), and letting $r = p_1/p_2$,

$$\frac{q_{k+1}(k)}{q_{k+1}(k+1)} = \frac{(n-k)p_2(1-kp_2+p_1\sum_{i=1}^{k} r^i)}{(1-kp_2)(1-(k-1)p_2+p_1\sum_{i=1}^{k-1} r^i)}.$$

We show that the numerator is greater than the denominator if and only if $p_1 > p_2$. The difference between the numerator and denominator is

$$(np_2-1)\left(1-kp_2+p_1\sum_{i=1}^{k-1} r^i\right) + p_1p_2(n-k)r^k - (1-kp_2)p_2$$
$$= ((n-1)p_2-1)(1-kp_2) + (np_2-1)p_1\left(\frac{r^k-r}{r-1}\right) + p_1p_2(n-k)r^k$$
$$= (kp_2-1)p_1 + (np_2-1)p_1\left(\frac{r^k-r}{r-1}\right) + p_1p_2(n-k)r^k.$$

Since $p_1 > 0$, dividing by p_1 does not affect the sign, so we obtain

$$(kp_2-1)+(np_2-1)\left(\frac{r^k-r}{r-1}\right)+p_2(n-k)r^k$$

$$=\frac{k}{r+n-1}-1+\left(\frac{n}{r+n-1}-1\right)\left(\frac{r^k-r}{r-1}\right)+\left(\frac{n-k}{r+n-1}\right)r^k$$

$$(11)\qquad =\frac{(r^k-1)(n-k-1)}{r+n-1}.$$

Since $n > 2$ and $k \leqq n-2$, (11) is positive if and only if $r^k - 1 > 0$, which is true if and only if $r > 1$ or $p_1 > p_2$. □

LEMMA 4. *For* $1 \leqq k \leqq n-2$ *and* $k < j \leqq n$, $q_j(k+1) < q_j(k)$ *if and only if* $p_1 > p_2$.

Proof. From (7),

$$\frac{q_j(k+1)}{q_{k+1}(k+1)}=\frac{\prod_{i=j-1}^{k+1}(n-i)p_2}{\prod_{i=j-1}^{k+1}1-ip_2}=\frac{q_j(k)}{q_{k+1}(k)}.$$

Thus, $q_j(k+1) < q_j(k)$ if and only if $q_{k+1}(k+1) < q_{k+1}(k)$ which by Lemma 3 is true if and only if $p_1 > p_2$. □

THEOREM 3. *For* $1 \leqq k \leqq n-2$, ec (POS (k)) $\geqq$ ec (POS $(k+1)$) *for all probability distributions in which* $p_2 = \cdots = p_n$.

Proof. Immediate from Theorem 2, Lemmas 2 and 4 and Theorem 1. □

A second spectrum of rules. The second spectrum of rules which we examine consists of $n-1$ rules. Let $2 \leqq k \leqq n$. Rule SWITCH (k) operates as follows: if an argument is found in a record at a position $j > k$, that record is transposed with its predecessor; if it is found at a position $j \leqq k$, that record is moved to the first position in the list (and all records previously in positions 1 to $j-1$ are moved back one position). The rule SWITCH (k) is a hybrid of the move-to-front and transposition rules. Note that SWITCH (2) is the transposition rule and that SWITCH (n) is the move-to-front rule.

For $p_2 = \cdots = p_n$, we use the notation $q_i(k)$ (or just simply q_i when k is understood) as an abbreviation for q_i(SWITCH (k)), the stationary probability that R_1 is in position i under rule SWITCH (k).

We wish to prove that ec (SWITCH (k)) $\leqq$ ec (SWITCH $(k+1)$) by showing that

(i) if $p_1 > p_2$, there exists $1 \leqq j \leqq n$ such that for all $1 \leqq i \leqq j$, $q_i(k) > q_i(k+1)$, and for all $j < i \leqq n$, $q_i(k) \leqq q_i(k+1)$;

(ii) if $p_1 < p_2$, there exists $1 \leqq j \leqq n$ such that for all $1 \leqq i \leqq j$, $q_i(k) < q_i(k+1)$, and for all $j < i \leqq n$, $q_i(k) \geqq q_i(k+1)$,

and invoking Theorems 1 and 2. Our strategy here is the same as that for the rules POS (k) except that we merely prove the existence of a j where the inequality between $q_j(k)$ and $q_j(k+1)$ reverses rather than explicitly identify the point of reversal (as equal to k) as we did for POS (k).

The equations relating the steady-state probabilities for rule SWITCH (k) are the following:

$$(12)\qquad q_1=\sum_{i=1}^{k}p_1q_i+(n-k)p_2q_1,$$

$$(13)\qquad q_j=(n+j-k-1)p_2q_j+(k-j+1)p_2q_{j-1}\quad \text{for } 1<j<k,$$

(14) $$q_j = (n-2)p_2q_j + p_1q_{j+1} + p_2q_{j-1} \quad \text{for } k \leqq j < n,$$

(15) $$q_n = (n-1)p_2q_n + p_2q_{n-1},$$

(16) $$\sum_{i=1}^{n} q_i = 1.$$

(Equations (12)–(16) hold for all $2 \leqq k \leqq n$, but (14) is vacuous for the move-to-front rule when $k = n$ and (13) is vacuous for the transposition rule when $k = 2$.)

As before, let $r = p_1/p_2$ so that $p_1 = r/(r+n-1)$ and $p_2 = 1/(r+n-1)$.

LEMMA 5. *For* $2 \leqq k \leqq n$,

$$q_1(k) = \frac{\prod_{i=0}^{k-2}(r+i)}{\prod_{i=1}^{k-1}(r+i) + (k-1)!\sum_{j=k+1}^{n} r^{k-j}}.$$

Proof. From (13) we have that for $1 < j < k$

$$q_j = \left(\frac{(k-j+1)p_2}{1-(n-1)p_2+(k-j)p_2}\right)q_{j-1} = \left(\frac{k-j+1}{r+k-j}\right)q_{j-1},$$

which yields for $1 < j < k$

$$q_j = \frac{\prod_{i=1}^{j-1}(k-i)}{\prod_{i=2}^{j}(k+r-i)}q_1$$

and, in particular,

(17) $$q_{k-1} = \frac{(k-1)!}{\prod_{i=1}^{k-2}(r+i)}q_1.$$

(This equation is vacuous for $k = 2$.)

From (14) and (15), by induction we can show that for $k \leqq j \leqq n$

(18) $$q_j = q_{j-1}/r.$$

This yields that for $k \leqq j \leqq n$

$$q_j = \frac{q_{k-1}}{r^{j-k+1}},$$

which together with (17) implies that for $k \leqq j \leqq n$

(19) $$q_j = \frac{(k-1)!}{r^{j-k+1}\prod_{i=1}^{k-2}(r+i)}q_1.$$

(For $k = 2$ the product is vacuous and is defined as 1.)

From (12) and (16),

$$q_1 = p_1\left(1 - \sum_{j=k+1}^{n} q_j\right) + q_1(n-k)p_2,$$

which together with (19) yields

$$q_1 = p_1\left(1 - q_1\frac{(k-1)!}{\prod_{i=1}^{k-2}(r+i)}\sum_{j=k+1}^{n}\frac{1}{r^{j-k+1}}\right) + (n-k)p_2q_1$$

or

$$q_1 = \frac{p_1}{1-(n-k)p_2 + \dfrac{(k-1)!p_1}{\prod_{i=1}^{k-2}(r+i)} \sum_{j=k+1}^{n} r^{k-j-1}}$$

$$= \frac{p_1 \prod_{i=1}^{k-2}(r+i)}{(p_1+(k-1)p_2)\prod_{i=1}^{k-2}(r+i)+(k-1)!p_1\sum_{j=k+1}^{n} r^{k-j-1}}$$

$$= \frac{\prod_{i=1}^{k-2}(r+i)}{\left(1+\dfrac{k-1}{r}\right)\prod_{i=1}^{k-2}(r+i)+(k-1)!\sum_{j=k+1}^{n} r^{k-j-1}}$$

$$= \frac{\prod_{i=0}^{k-2}(r+i)}{\prod_{i=1}^{k-1}(r+i)+(k-1)!\sum_{j=k+1}^{n} r^{k-j}}.$$

(Note that this equation holds for $k=2$ as well and reduces to

$$q_1 = \frac{r}{r+1+\sum_{j=3}^{n} r^{2-j}} = \left(\sum_{j=0}^{n-1} r^{-j}\right)^{-1}$$

in that case.) □

LEMMA 6. *For* $2 \leqq k \leqq n-1$,

(i) *if* $r>1$, $q_1(k) > q_1(k+1)$;

(ii) *if* $r<1$, $q_1(k) < q_1(k+1)$.

Proof. By Lemma 5,

$$q_1(k)^{-1} = \frac{k-1+r}{r} + \frac{(k-1)!\sum_{j=k+1}^{n} r^{k-j}}{\prod_{i=0}^{k-2}(r+i)}$$

and

$$q_1(k+1)^{-1} = \frac{k+r}{r} + \frac{k!\sum_{j=k+2}^{n} r^{k+1-j}}{\prod_{i=0}^{k-1}(r+i)},$$

$$q_1(k)^{-1} - q_1(k+1)^{-1} = -\frac{1}{r} + \frac{(k-1+r)(k-1)!\sum_{j=k+1}^{n} r^{k-j} - k!\sum_{j=k+2}^{n} r^{k-j+1}}{\prod_{i=0}^{k-1}(r+i)}$$

$$\tag{20} = -\frac{1}{r} + \frac{(k-1)!}{\prod_{i=0}^{k-1}(r+i)}\left((k-1+r)\sum_{j=k+1}^{n-1} r^{k-j} + (k-1+r)r^{k-n} - k\sum_{j=k+1}^{n-1} r^{k-j}\right)$$

$$= -\frac{1}{r} + \frac{(k-1)!}{\prod_{i=0}^{k-1}(r+i)}\left((r-1)\sum_{j=k+1}^{n-1} r^{k-j} + (k-1+r)r^{k-n}\right).$$

If $r \neq 1$, $\sum_{j=k+1}^{n-1} r^{k-j} = (r^{n-k}-r)/(r^{n-k}(r-1))$, so from (20),

$$\tag{21} q_1(k)^{-1} - q_1(k+1)^{-1} = -\frac{1}{r} + \frac{(k-1)!}{\prod_{i=0}^{k-1}(r+i)}\left(\frac{r^{n-k}-r}{r^{n-k}} + (k-1+r)r^{k-n}\right).$$

Now

$$\tag{22} \frac{r^{n-k}-r}{r^{n-k}} + (k-1+r)r^{k-n} = \frac{r^{n-k}+k-1}{r^{n-k}}.$$

If $r>1$,

$$\frac{(k-1)!}{\prod_{i=0}^{k-1}(r+i)}<\frac{1}{rk}$$

and so

$$q_1(k)^{-1}-q_1(k+1)^{-1}<-\frac{1}{r}+\frac{1}{rk}\left(\frac{r^{n-k}+k-1}{r^{n-k}}\right)$$

$$=-\frac{1}{r}+\frac{1}{rk}\left(1+\frac{k-1}{r^{n-k}}\right)<-\frac{1}{r}+\frac{1}{rk}(k)=0.$$

Thus, (i) is proven.

If $r<1$,

$$\frac{(k-1)!}{\prod_{i=0}^{k-1}(r+i)}>\frac{1}{rk},$$

and from (21) and (22), we get that

$$q_1(k)^{-1}-q_1(k+1)^{-1}>-\frac{1}{r}+\frac{1}{rk}\left(\frac{r^{n-k}+k-1}{r^{n-k}}\right)$$

$$=-\frac{1}{r}+\frac{1}{rk}\left(1+\frac{k-1}{r^{n-k}}\right)>-\frac{1}{r}+\frac{1}{rk}(k)=0,$$

which proves (ii). □

LEMMA 7. *For* $2\leqq k<n$ *and* $k<j\leqq n$,

(i) $q_j(k)<q_j(k+1)$ *if and only if* $q_{j-1}(k)<q_{j-1}(k+1)$;

(ii) $q_j(k)>q_j(k+1)$ *if and only if* $q_{j-1}(k)>q_{j-1}(k+1)$.

Proof. This follows immediately from (18). □

LEMMA 8. *For all* $2\leqq k\leqq n$ *and* $2\leqq j\leqq n$,

(i) *if* $r>1$ *and* $q_{j-1}(k)\leqq q_{j-1}(k+1)$, *then* $q_j(k)<q_j(k+1)$;

(ii) *if* $r<1$ *and* $q_{j-1}(k)\geqq q_{j-1}(k+1)$, *then* $q_j(k)>q_j(k+1)$.

Proof. Lemma 7 proves the statements for $k<j\leqq n$. From (13) we have that for $1<j<k$

$$q_j(k)=\frac{k-j+1}{r+k-j}\,q_{j-1}(k).$$

We can extend this equation for the case $j=k$ by using (18).

If $r>1$ and $q_{j-1}(k)\leqq q_{j-1}(k+1)$ for some $1<j\leqq k$,

$$q_j(k)=\frac{k-j+1}{r+k-j}\,q_{j-1}(k)<\frac{(k+1)-j+1}{r+(k+1)-j}\,q_{j-1}(k+1)=q_j(k+1).$$

If $r<1$ and $q_{j-1}(k)\geqq q_{j-1}(k+1)$ for some $1<j\leqq k$,

$$q_j(k)=\frac{k-j+1}{r+k-j}\,q_{j-1}(k)>\frac{(k+1)-j+1}{r+(k+1)-j}\,q_{j-1}(k+1)=q_j(k+1).$$

Note that we are using the fact that for $x,\ >0$, $(x/y)>(x+1)/(y+1)$ if and only if $x>y$ and $(x/y)<(x+1)/(y+1)$ if and only if $x<y$. □

LEMMA 9. *Let* $2\leqq k\leqq n$.

(i) *If* $r>1$, *there exists* $1\leqq j\leqq n$ *such that for all* $1\leqq i\leqq j$, $q_i(k)>q_i(k+1)$, *and for all* $j<i\leqq n$, $q_i(k)\leqq q_i(k+1)$.

(ii) *If* $r<1$, *there exists* $1 \leqq j \leqq n$ *such that for all* $1 \leqq i \leqq j$, $q_i(k) < q_i(k+1)$, *and for all* $j < i \leqq n$, $q_i(k) \geqq q_i(k+1)$.

Proof. (i) Let j be the largest integer between 1 and n such that $q_j(k) > q_j(k+1)$. Such an integer must exist by Lemma 6. By Lemma 8 if there were an integer i less than or equal to j such that $q_i(k) \leqq q_i(k+1)$, it would be true that $q_j(k) \leqq q_j(k+1)$, which is not the case. Thus, for all $1 \leqq i \leqq j$, $q_i(k) > q_i(k+1)$. By the choice of j, $q_i(k) \leqq q_i(k+1)$ for all $j < i \leqq n$.

(ii) Let j be the largest integer between 1 and n such that $q_j(k) < q_j(k+1)$. Such an integer must exist by Lemma 6. Then by Lemma 8, for all $1 \leqq i \leqq j$, $q_i(k) < q_i(k+1)$. By the choice of j, $q_i(k) \geqq q_i(k+1)$ for all $j < i \leqq n$. □

THEOREM 4. *For* $2 \leqq k \leqq n-1$, ec (SWITCH (k)) $\leqq$ ec (SWITCH ($k+1$)) *for all probability distributions in which* $p_2 = \cdots = p_n$.

Proof. Immediate from Theorem 2, Lemma 9 and Theorem 1. □

REFERENCES

[1] J. R. BITNER, *Heuristics that dynamically alter data structures to reduce their access time*, Ph.D. thesis, Report UIUCDCS-R-76-818, University of Illinois, Urbana, 1976.

[2] ———, *Heuristics that dynamically organize data structures*, this Journal, 8 (1979), pp. 82–110.

[3] P. J. BURVILLE AND J. F. C. KINGMAN, *On a model for storage and search*, J. Appl. Probability, 10 (1973), pp. 697–701.

[4] W. FELLER, *An Introduction to Probability Theory and Its Applications*, John Wiley, New York, 1968.

[5] W. J. HENDRICKS, *The stationary distribution of an interesting Markov chain*, J. Appl. Probability, 9 (1972), pp. 231–233.

[6] ———, *An extension of a theorem concerning an interesting Markov chain*, Ibid., 10 (1973), pp. 886–890.

[7] D. E. KNUTH, *The Art of Computer Programming, Vol. 3, Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[8] J. MCCABE, *On serial files with relocatable records*, Operations Res., 12 (1965), pp. 609–618.

[9] R. L. RIVEST, *On self-organizing sequential search heuristics*, Comm. ACM, 19 (1976), pp. 63–67.

[10] T. C. TUAN AND R. C. T. LEE, *The stationary probabilities of the transposition heuristic and the moving to the front heuristic for sequential searching*, private communication.

[11] Y. C. KAN AND S. M. ROSS, *Optimal list order under partial memory constraints*, J. Appl. Prob., 17 (1980), pp. 1004–1015.

SIAM J. COMPUT.
Vol. 11, No. 3, August 1982

© 1982 Society for Industrial and Applied Mathematics

0097-5397/82/1103-0016 $01.00/0

FINDING THE CYCLIC INDEX OF AN IRREDUCIBLE, NONNEGATIVE MATRIX*

MIKHAIL J. ATALLAH†

Abstract. The cyclic index δ of an irreducible nonnegative square matrix is the number of eigenvalues of maximum modulus of that matrix. If $\delta = 1$, the matrix is said to be primitive. The notions of primitivity and cyclic index play an important role in the theory of nonnegative matrices. In the context of discrete Markov chains, the words "period" and "aperiodic" are sometimes used in place of "cyclic index" and "primitive", respectively. It is known how to test an irreducible nonnegative square matrix for primitivity, but there is no known practical method for finding the cyclic index δ in the general case. This paper presents a time-optimal algorithm for finding δ.

Key words. matrix theory, graph algorithms, rooted spanning tree, breadth-first search, greatest common divisor (gcd)

1. Introduction. We first review some known results, leading to the most commonly used method for testing primitivity. Throughout, A is an $n \times n$ irreducible nonnegative matrix.

Frobenius [2] has shown that A is primitive if and only if, for some positive integer m, A^m is positive (i.e., has all its elements positive). If A^m is positive, then so is $A^{m'}$ for all $m' > m$ [2]. If A is primitive, let θ be the smallest positive integer such that A^θ is positive. Wielandt [3] gave an example for which $\theta = n^2 - 2n + 2$ and stated without proof that $n^2 - 2n + 2$ is actually an upper bound on θ. Holladay and Varga [4] later gave a proof of this fact. Therefore, by computing $A^2, A^4, A^8, \cdots$ and stopping after, in the worst case, $O(\log_2 n)$ such matrix multiplications, one can determine whether A is primitive or not (i.e., whether $\delta = 1$ or $\delta > 1$). But this approach fails to determine δ in the general case and is computationally inefficient.[1] Our algorithm for finding δ is based on the work of Romanovsky [5], which is briefly presented next.

Let $G = (V, E)$ be the directed graph whose adjacency matrix is the matrix obtained by replacing every nonzero entry of A by 1. Since A is irreducible, G is strongly connected. (In the context of Markov chains, G is known as the "transition graph" of the chain.)

DEFINITION. A partition $V_1, V_2, \cdots, V_\Delta$ of the vertex-set V is said to be *cyclic* if $(x, y) \in E \Rightarrow x \in V_1$ and $y \in V_2$ or $x \in V_2$ and $y \in V_3, \cdots$ or $x \in V_\Delta$ and $y \in V_1$ (Fig. 1). (If $\Delta = 1$, we have the trivial case of $V_1 = V$.)

Observe that if $V_1, \cdots, V_\Delta$ is a cyclic partition of V, then the length of any closed walk in G is a multiple of Δ. Romanovsky [5] has shown the following:

LEMMA 1. *There exists a unique cyclic partition $V_1, \cdots, V_\delta$ of V. Moreover,*

$$\delta = \max \{\Delta \mid \textit{there exists a cyclic partition } V_1, \cdots, V_\Delta \textit{ of } V\}.$$

(For a proof the reader is referred to [5].)

Note that Lemma 1 implies that δ depends on the location of the nonzero entries of A, but not on their magnitude. Therefore the problem is reduced to that of finding the "cylic index" of a strongly connected directed graph G.

---

* Received by the editors May 27, 1981 and in final form November 11, 1981. This work was supported by the National Science Foundation under grant MCS-79-05163.

† Electrical Engineering Department, The Johns Hopkins University, Baltimore, Maryland 21218.

[1] Another commonly used criterion for primitivity is that if some diagonal element of A is nonzero then A is primitive. This, however, is not a necessary condition for primitivity.

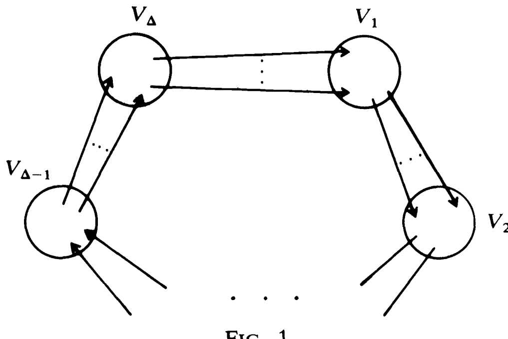


FIG. 1

It should be pointed out that in the context of discrete Markov chains producing the cyclic partition $V_1, \cdots, V_\delta$ is as important as finding δ. Our algorithm finds δ and produces the cyclic partition $V_1, \cdots, V_\delta$ of V.

Now we proceed to describe the algorithm. We assume that the input is the graph G rather than the matrix A.

2. The algorithm.

Input: An adjacency-list representation of a strongly connected directed graph $G=(V, E)$. For every $\nu \in V$,

$$L(\nu)=\{u \in V \mid (\nu, u) \in E\}.$$

Output: The cyclic index δ of G, and the cyclic partition $V_1, V_2, \cdots, V_\delta$ of V.

The following is an informal description of the algorithm. It is followed by a pseudo-Algol program (Fig. 2). Throughout, assume that the gcd (greatest common divisor) subroutine used is the Euclidean algorithm [1].[2] (We insist, however that gcd $(0, x)=x$ for all $x \geqq 0$.)

Step 0. $\Delta \leftarrow 0$.

Step 1. Let T be a spanning tree of G rooted, say, at vertex ν_0.
For every vertex ν, let $d(\nu)$ be the length of the path in T from ν_0 to ν.

Step 2. For each $x \in V$, look at all the elements of $L(x)$: for each $y \in L(x)$, replace the current value of Δ by gcd $(\Delta, |d(x)-d(y)+1|)$.

Step 3. Do the following:
put ν_0 in set V_1.
put all vertices ν having $d(\nu)=1$ in set V_2.
⋮
put all vertices ν having $d(\nu)=\Delta-1$ in set V_Δ.
put all vertices ν having $d(\nu)=\Delta$ in set V_1.
⋮
etc.
⋮
(end of algorithm).

It is easy to see that all of the steps described above are implemented in the program of Fig. 2, where T is a breadth-first spanning tree. Note that Steps 1 and 2 are carried out simultaneously in the while-do loop and that we do not take the

[2] As was pointed out by a referee, even the obvious brute force algorithm for gcd (a, b) can be used—e.g., $d \leftarrow \min(a, b)$; *while* $d \nmid a$ *or* $d \nmid b$ *do* $d \leftarrow d-1$.

absolute value of $d(\nu)-d(\omega)+1$ because, since T is a breadth-first spanning tree, we have

$$\omega \in L(\nu) \Rightarrow d(\nu)-d(\omega)+1 \geqq 0.$$

(The proof of this fact is easy and is omitted.)

```
PROCEDURE CYCLIC.INDEX
Begin
  Δ ← 0
  Pick any vertex ν₀ ∈ V
  d(ν₀) ← 0
  Enqueue ν₀ and mark it as being "old"
  While queue not empty do
    Begin
      ν ← Dequeue
      For each vertex ω on L(ν) do
        If w is not "old" then
          Begin
            d(ω) ← d(ν)+1
            Enqueue ω and mark it as being "old"
          End
        Else do Δ ← gcd (Δ, d(ν)−d(ω)+1)
    End
  Output Δ
  For each vertex ν ∈ V do
    Begin
      j ← d(ν) − ⌊d(ν)/Δ⌋*Δ+1
    Put ν in Set V_j
    End
  Output V_1, ···, V_Δ
End.
```

FIG. 2

We shall shortly prove that, upon termination of the algorithm, we have $\Delta=\delta$. But first we prove the following:

LEMMA 2. *There exists a cyclic partition $V_1, \cdots, V_s$ of V if and only if, for all $(x, y) \in E$, s divides $d(x)-d(y)+1$.*

Proof. only if. Suppose that $V_1, \cdots, V_s$ is a cyclic partition of V, and suppose $(x, y) \in E$. We must show that $d(x)-d(y)+1=ks$, where k is an integer. Since G is strongly connected, there is a path $P_{y\nu_0}$ from y to ν_0, having a length of, say, l. Now, let $P_{\nu_0 y}$ and $P_{\nu_0 x}$ be the paths in T from ν_0 to y and x, respectively.

$P_{\nu_0 y}$ followed by $P_{y\nu_0}$ is a closed walk of length $d(y)+l$.

$P_{\nu_0 x}$ followed by edge (x, y) followed by $P_{y\nu_0}$ is a closed walk of length $d(x)+1+l$.

But the length of any closed walk must be a multiple of s (since $V_1, \cdots, V_s$ is a cyclic partition of V). Therefore,

$$d(y)+l=k_1 s, \tag{1}$$

$$d(x)+1+l=k_2 s, \tag{2}$$

where k_1 and k_2 are integers. Subtracting (1) from (2) gives

$$d(x)+1-d(y)=(k_2-k_1)s=ks.$$

if. Suppose s divides $d(x)-d(y)+1$ for all $(x, y)\in E$. We must show that there exists a cyclic partition $V_1, \cdots, V_s$ of V.

Create sets $V_1, \cdots, V_s$ as described in Step 3 of the algorithm (with s replacing Δ). Now, suppose that $(x, y)\in E$, with $x\in V_i$ and $y\in V_j$. We must show that $i+1\equiv j \bmod s$. Since $(x, y)\in E$, it follows that s divides $d(x)-d(y)+1$, i.e.,

$$d(x)-d(y)+1=ks \qquad (k=\text{integer}).$$

Moreover, from the way Step 3 of the algorithm creates V_i and V_j, we have

$$d(x)\equiv i-1 \bmod s \quad \text{and} \quad d(y)\equiv j-1 \bmod s.$$

Therefore, $(i-1)-(j-1)+1\equiv 0 \bmod s$, $i+1\equiv j \bmod s$. □

From Lemmas 1 and 2, it follows immediately that:

THEOREM 3. *δ is the greatest positive integer which for all $(x, y)\in E$ divides $d(x)-d(y)+1$.*

COROLLARY 4. *When Step 2 of the algorithm terminates, we have $\Delta=\delta$.*

COROLLARY 5. *Step 3 produces the desired cyclic partition $V_1, \cdots, V_\delta$ of V.*

Proof. The proof is identical to the proof of the "if" part of Lemma 2. □

THEOREM 6. *The algorithm we described takes $O(|E|)$ steps, and, hence, is time-optimal.*

Proof. (In what follows, we make use of the fact that since G is strongly connected we have $|V|=O(|E|)$).

It is clear that Step 1 takes $O(|E|)$ steps. If we exclude the calculation of the gcd, Step 2 also takes $O(|E|)$ steps, because $\sum_{x\in V}|L(x)|=|E|$. Since $\Delta\leqq|V|$, the cumulative cost of the gcd calculations in Step 2 is $O(|E|)$ steps. Step 3, obviously, takes $O(|V|)$ steps. Therefore, the running time is $O(\max(|V|, |E|))=O(|E|)$. □

3. Conclusion. We have presented a time-optimal algorithm for finding the cyclic index of an irreducible nonnegative square matrix.

4. Acknowledgments. This problem was brought to my attention by Professor A. Karr [6] in his course on stochastic processes. I also sincerely thank Professor S. R. Kosaraju for many useful suggestions. One of the referees pointed out the existence of reference [7] where Professor Knuth describes a linear-time algorithm for finding the cyclic index of a digraph. Both referees made many useful suggestions.

REFERENCES

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

[2] G. FROBENIUS, *Uber Matrizen aus nicht negativen Elementen*, Sitzungsberichte der Preussichen Akademie der Wissenschaften zu Berlin, 1912, pp. 456–477.

[3] HELMUT WIELANDT, *Unzulegbare, nicht negativen Matrizen*, Math. Zeit., 52 (1950), pp. 642–648.

[4] J. C. HOLLADAY AND R. S. VARGA, *On powers of nonnegative matrices*, Proc. Amer. Math. Soc., 9 (1958), pp. 631–634.

[5] V. ROMANOVSKY, *Recherches sur les chaînes de Markoff*, Acta. Math., 66, pp. 147–251.

[6] A. KARR, personal communication.

[7] D. E. KNUTH, Lecture notes delivered at the Matematisk Institutt of the University of Oslo, Norway, 1972–73.

SIAM J. COMPUT.
Vol. 11, No. 3, August 1982

© 1982 Society for Industiral and Applied Mathematics
0097-5397/82/1103-0017 $01.00/0

RELATIVIZING TIME, SPACE, AND TIME-SPACE*

RONALD V. BOOK,† CHRISTOPHER B. WILSON‡ AND XU MEI-RUI†§

Abstract. Consider the classes of formal languages specified by nondeterministic acceptors that operate simultaneously within time bounds from a set $\mathcal{T}$ and space bounds from a set $\mathcal{S}$. How large must the time bounds be in order to obtain all of the languages specified by nondeterministic acceptors that operate within space bounds from $\mathcal{S}$? How large must the space bounds be in order to obtain all of the languages specified by nondeterministic acceptors that operate with time bounds from $\mathcal{T}$? The first question is shown to be equivalent (with appropriate restrictions on $\mathcal{S}$ and $\mathcal{T}$) to the question of whether it matters if the time bounds apply to all of the steps or only to the steps which query the oracle. The second question is shown to be equivalent to the question of whether it matters if the space bounds apply to all of the configurations or only to the configurations in which the oracle is queried. These results generalize a more specific result [6] comparing NP with PSPACE. Also, it is shown that inclusions between the nondeterministic and deterministic time hierarchies fail to translate downwards in some relativized cases.

Key words. complexity classes, relativizations, oracle machines, time, space, time-space, bounded queries

Introduction. The problem of finding the precise relationship between computation time and space is very important in complexity theory. Because this relationship remains unknown, there is an exponential discrepancy when upper and lower bounds are both expressed in terms of time or space alone. For example, not only is it not known whether a linear upper bound for space implies simultaneous upper bounds of linear space and polynomial time but also it is not known whether a linear upper bound for space implies a polynomial upper bound for time regardless of how much space is used.

One method of approaching questions regarding the relationship between complexity classes is to study relativized complexity classes. One might attempt to prove that NP = PSPACE if and only if for every oracle set A, NP (A) = PSPACE (A); if one could prove this, then one could conclude that NP $\neq$ PSPACE since it is known that there exists a set A such that NP $(A) \neq$ PSPACE (A) [2], [3], [13], and knowing that NP $\neq$ PSPACE would allow one to conclude that there exists a problem solvable in linear space but not solvable in polynomial time [4]. It is known that if one restricts the number of oracle queries a space-bounded oracle machine can make, then it is sometime possible to obtain different results: If for any set A, NPQUERY (A) is the class of languages accepted relative to A by nondeterministic oracle machines with polynomial bounds on work space (and query tape) and polynomial bounds on the number of oracle queries made in any computation, then it is the case that NP = PSPACE if and only if for all B, NP (B) = NPQUERY (B) [6].

The subject of this paper is the question of whether a class of languages specified by Turing machines with work space and running time simultaneously bounded is equal to the class specified by machines with only the work space bounded, or whether that class is equal to the class specified by machines with only the running time bounded. For example, is the class of languages accepted by nondeterministic Turing machines that use linear work space and polynomial running time equal to the class

* Received by the editors February 9, 1981, and in final revised form December 14, 1981. This research was supported in part by the National Science Foundation under grant MCS80-11979. Some of these results were announced at the 22nd IEEE Symposium on Foundations of Computer Science, Nashville, Tennessee, October 1981.

† Department of Mathematics, University of California at Santa Barbara, Santa Barbara, California 93106.

‡ Department of Computer Science, University of Toronto, Toronto, Ontario M5S 1A7, Canada.

§ Harbin University of Science and Technology, Harbin, Heilongjiang, People's Republic of China.

of languages accepted by machines that use linear work space, or is that class equal to the class of languages accepted by machines that operate in polynomial time? To study these questions restricted oracle machines are considered. In the first case the restriction involves bounding the number of oracle calls that can be made in any accepting computation while simultaneously bounding the work space. In the second case the restriction involves bounding the length of the work tapes in those configurations in which oracle calls are made while simultaneously bounding the running time. For example, every language accepted by a nondeterministic machine that uses linear work space is accepted by another machine that simultaneously runs in polynomial time and uses only linear work space if and only if for every oracle set A, every language accepted relative to A by a nondeterministic oracle machine that uses linear work space and can make only a polynomial number of oracle calls in any computation is also accepted relative to A by a nondeterministic oracle machine that simultaneously runs in polynomial time and uses linear work space. Also, every language accepted by a nondeterministic machine that runs in polynomial time is accepted by another machine that simultaneously uses linear work space and runs in polynomial time if and only if for every oracle set A, every language accepted relative to A by a nondeterministic oracle machine that runs in polynomial time and uses only linear work space in those configurations that query the oracle is also accepted relative to A by a nondeterministic oracle machine that simultaneously runs in polynomial time and uses linear work space.

Sections 2 and 3 are devoted to establishing the results described above. In § 4 potential hierarchies are investigated generalizing the results in [7] regarding relativizations of the question of whether the polynomial-time hierarchy is equal to PSPACE.

In § 5 it is shown that there exist oracle sets that do not allow for negative inclusions to translate upwards (or, equivalently, positive inclusions to translate downwards) not only in the case of time-bounded machines but also in the case of "bounded query" machines.

It appears that the types of relativizations studied here and in [6], [7] have not been investigated previously in complexity theory, in computability theory, or in the study of complexity-bounded reducibilities. The results in the present paper as well as those in [6], [7] suggest that there are fundamental issues still to be revealed. For other results showing how varying access to the oracle can yield results differing from the current state of knowledge about nonrelativized complexity classes, see [1], [10].

1. Preliminaries. It is assumed that the reader is familiar with the basic concepts from the theories of automata, computability, and formal languages. Some of the concepts that are most important for this paper are reviewed here and notation is established.

For a string w, $|w|$ denotes the length of w. The empty string is denoted by e, $|e| = 0$.

An *oracle machine* is a multitape Turing machine M with a distinguished work tape, the *query* tape, and three distinguished states QUERY, YES, and NO. At some step of a computation on an input string w, M may transfer into the state QUERY. In state QUERY, M transfers into the state YES if the string currently appearing on the query tape is in some *oracle set* A; otherwise, M transfers into the state NO; in either case the tape is instantly erased. The set of strings *accepted by* M *relative to the oracle set* A is $L(M, A) = \{w \mid \text{there is an accepting computation of } M \text{ on input } w \text{ when the oracle set is } A\}$. If M has no query tape, we write $L(M)$.

Oracle machines may be deterministic or nondeterministic. An oracle machine may operate within some time bound T, where T is a function of the length of the

input string, and the notion of operation within a time bound for an oracle machine is just the same as that notion for an ordinary Turing machine. An oracle machine may operate within some space bound S, where S is a function of the length of the input string, and here we require that the query tape as well as the ordinary work tapes be bounded in length by S.

For any space bound S and any set A, let NSPACE (S, A) (DSPACE (S, A)) be the class of languages accepted relative to A by nondeterministic (respectively, deterministic) oracle machines that operate within space bound $S(n)$. Let NSPACE $(S)=$ NSPACE (S, ϕ) and DSPACE $(S)=$ DSPACE (S, ϕ). For any time bound T and any oracle set A, let NTIME (T, A) (DTIME (T, A)) be the class of languages accepted relative to A by nondeterministic (respectively, deterministic) oracle machines that operate within time bound $T(n)$. Let NTIME $(T)=$ NTIME (T, ϕ) and DTIME $(T)=$ DTIME (T, ϕ).

If $\mathcal{S}$ is a set of space bounds and A is a set, let DSPACE $(\mathcal{S}, A)=\bigcup\{\text{DSPACE}(S, A) \mid S \in \mathcal{S}\}$ and NSPACE $(\mathcal{S}, A)=\bigcup\{\text{NSPACE}(S, A) \mid S \in \mathcal{S}\}$, and let DSPACE $(\mathcal{S})=$ DSPACE $(\mathcal{S}, \phi)$ and NSPACE $(\mathcal{S})=$ NSPACE $(\mathcal{S}, \phi)$. If $\mathcal{T}$ is a set of time bounds and A is a set, let DTIME $(\mathcal{T}, A)=\bigcup\{\text{DTIME}(T, A) \mid T \in \mathcal{T}\}$ and NTIME $(\mathcal{T}, A)=\bigcup\{\text{NTIME}(T, A) \mid T \in \mathcal{T}\}$, and let DTIME $(\mathcal{T})=$ DTIME $(\mathcal{T}, \phi)$ and NTIME $(\mathcal{T})=$ NTIME $(\mathcal{T}, \phi)$.

Let PSPACE $(A)=\bigcup_{k \geqq 1}$ DSPACE (n^k, A). It is known [11], [12] that for every set A and all $k \geqq 1$, NSPACE $(n^k, A) \subseteq$ DSPACE (n^{2k}, A) so that PSPACE $(A)=\bigcup_{k \geqq 1}$ NSPACE (n^k, A). Let PSPACE $=$ PSPACE (ϕ).

For any set A, let NP (A) (P (A)) be the class of languages accepted relative to A by nondeterministic (resp. deterministic) oracle machines that operate within polynomial time. Let P $=$ P (ϕ) and NP $=$ NP (ϕ). For any set A, let DEXT $(A)=\bigcup\{\text{DTIME}(2^{cn}, A) \mid c>0\}$ and let NEXT $(A)=\bigcup\{\text{NTIME}(2^{cn}, A) \mid c>0\}$. Let DEXT $=$ DEXT (ϕ) and NEXT $=$ NEXT (ϕ).

Let poly $=\{n^k \mid k>0$ an integer$\}$ and lin $=\{kn \mid k>0$ an integer$\}$.

2. Time-space vs. space. In this section we define classes of languages specified by oracle machines that are bounded simultaneously in both time and space and by oracle machines that are bounded simultaneously in both space and the number of oracle queries allowed. Then we establish our first result.

Let T and S be functions from the natural numbers to the natural numbers. A *tisp* (T, S) *oracle machine* is an oracle machine such that on every input w, if there is an accepting computation on w, then there is an accepting computation which simultaneously uses at most $T(|w|)$ steps and at most $S(|w|)$ work space. For any oracle set A, let NTISP (T, S, A) be the class of languages accepted relative to A by nondeterministic tisp (T, S) oracle machines. Let NTISP $(T, S)=$ NTISP (T, S, ϕ).

If $\mathcal{T}$ is a set of time bounds and $\mathcal{S}$ is a set of space bounds, then for any oracle set A, let NTISP $(\mathcal{T}, \mathcal{S}, A)=\bigcup\{\text{NTISP}(T, S, A) \mid T \in \mathcal{T}, S \in \mathcal{S}\}$, and let NTISP $(\mathcal{T}, \mathcal{S})=$ NTISP $(\mathcal{T}, \mathcal{S}, \phi)$.

We would like it to be the case that any machine running in time T and space S is a tisp (T, S) machine. The definition requires that the two bounds be achieved in the same accepting computation, so that a deterministic machine running in time T and space S *is* a tisp (T, S) machine. For nondeterministic machines we achieve the same condition by requiring familiar "honesty" conditions.

A function S is *constructible* if there is a deterministic Turing machine M such that on every input string w, M's computation on w halts having used work space $S(|w|)$ and exactly $S(|w|)$ tape squares are marked on some work tape.

A function T is a *running time* if there is a deterministic Turing machine M such that on every input string w, M's computation on w halts in exactly $T(|w|)$ steps.

If T is a running time and S is constructible, then the pair (T, S) is *compatible* if there is a deterministic tisp (T, S) machine that makes no oracle queries and that witnesses the fact that T is a running time and S is constructible.

It is clear that if a pair (T, S) is compatible, then for every oracle set A and every language L in NTISP (T, S, A) there is a nondeterministic tisp (T', S') oracle machine M such that $L(M, A) = L$ and for every input string w, every computation of M on w uses at most $T'(|w|)$ steps and at most $S'(|w|)$ work space, where T' is $O(T)$ and S' is $O(S)$.

If a pair (T, S) is compatible, then it is the case that for some $c > 0$ and all $n \geqq 0$, $n \leqq T(n) \leqq 2^{cS(n)}$. We are interested in situations where $T(n) = o(2^{cS(n)})$. For example, let $T(n) = n^k$ for some integer k and let $S(n) = n$; is NTISP (T, S) equal to NSPACE (S)? More generally, if $\mathcal{T} = \{n^c | c \text{ a positive integer}\}$ and $\mathcal{S} = \{cn | c \text{ a positive integer}\}$, then is NTISP $(\mathcal{T}, \mathcal{S})$ equal to NSPACE $(\mathcal{S})$? We approach these questions by relativizing NSPACE $(\mathcal{S})$ in one particular manner.

Let T and S be functions from the natural numbers to the natural numbers. A *qusp* (T, S) *oracle machine* is an oracle machine such that on every input w, if there is an accepting computation on w, then there is an accepting computation which makes at most $T(|w|)$ oracle queries and uses at most $S(|w|)$ work space. For any set A, let NQUSP (T, S, A) be the class of languages accepted relative to A by nondeterministic qusp (T, S) oracle machines.

If $\mathcal{T}$ is a set of time bounds and $\mathcal{S}$ is a set of space bounds, then for any set A, let NQUSP $(\mathcal{T}, \mathcal{S}, A) = \bigcup\{\text{NQUSP}(T, S, A) | T \in \mathcal{T}, S \in \mathcal{S}\}$.

It is clear that if (T, S) is a compatible pair, then for every set A and every language L in NQUSP (T, S, A) there is a nondeterministic qusp (T', S') oracle machine M such that $L(M, A) = L$ and for every input string w, every computation of M on w makes at most $T'(|w|)$ oracle queries and uses at most $S'(|w|)$ work space, where T' is $O(T)$ and S' is $O(S)$.

For any functions T and S and any set A, NTISP $(T, S, A) \subseteq$ NQUSP $(T, S, A) \subseteq$ NSPACE (S, A), and NQUSP $(T, S, \phi) =$ NSPACE $(S, \phi) =$ NSPACE (S).

For a set $\mathcal{F}$ of functions, we say that a function g is $O(\mathcal{F})$ if there is some $f \in \mathcal{F}$ such that g is $O(f)$.

For the main results we require that the sets $\mathcal{T}$ and $\mathcal{S}$ of bounds satisfy the following conditions.

Condition 2.1. The set $\mathcal{T}$ is a set of running times and the set $\mathcal{S}$ is a set of constructible space bounds such that

(i) for every $T \in \mathcal{T}$ and $S \in \mathcal{S}$, the pair (T, S) is compatible;
(ii) for every $T_1 \in \mathcal{T}$ $(S_1 \in \mathcal{S})$ and $c > 0$, the function $T_2(n) = cT_1(n)$ $(S_2(n) = cS_1(n))$ is bounded above by some function in $\mathcal{T}$ (resp. $\mathcal{S}$);
(iii) if $T_1, T_2 \in \mathcal{T}$, then the function $T(n) = T_1(n)T_2(n)$ is $O(\mathcal{T})$;
(iv) if $S_1, S_2 \in \mathcal{S}$, then the function $S(n) = S_1(S_2(n))$ is $O(\mathcal{S})$;
(v) if $T \in \mathcal{T}$ and $S \in \mathcal{S}$, then the function $T'(n) = T(S(n))$ is $O(\mathcal{T})$.

Now we can establish our first result.

THEOREM 2.2. *Let $\mathcal{T}$ and $\mathcal{S}$ be sets of bounds satisfying Condition* 2.1. *The following are equivalent*:

(a) NTISP $(\mathcal{T}, \mathcal{S}) =$ NSPACE $(\mathcal{S})$.
(b) *For every set* A, NTISP $(\mathcal{T}, \mathcal{S}, A) =$ NQUSP $(\mathcal{T}, \mathcal{S}, A)$.

Proof. That (b) implies (a) is trivial since NTISP $(\mathcal{T}, \mathcal{S}, \phi) =$ NTISP $(\mathcal{T}, \mathcal{S})$ and NQUSP $(\mathcal{T}, \mathcal{S}, \phi) =$ NSPACE $(\mathcal{S})$.

To show that (a) implies (b), choose a set A. We have already observed that NTISP $(\mathcal{T}, \mathcal{S}, A) \subseteq$ NQUSP $(\mathcal{T}, \mathcal{S}, A)$. To prove the other inclusion, we must take an arbitrary $L_1 \in$ NQUSP $(\mathcal{T}, \mathcal{S}, A)$ and show that L_1 is in NTISP $(\mathcal{T}, \mathcal{S}, A)$.

If L_1 is in NQUSP $(\mathcal{T}, \mathcal{S}, A)$, then there is a nondeterministic oracle machine M_1 and functions $T_1 \in \mathcal{T}$ and $S_1 \in \mathcal{S}$ with the properties that for any $n \geqq 0$ in any computation on an input of length n, M_1 makes at most $T_1(n)$ oracle queries, M_1 uses at most $S_1(n)$ work space, and $L(M_1, A) = L_1$. Let $L_2 = \{\mathrm{ID}_\alpha \# \mathrm{ID}_\beta | \mathrm{ID}_\alpha$ is an instantaneous description of M_1 such that M_1 is in the initial state or M_1 is in the YES state or M_1 is in the NO state, ID_β is an instantaneous description of M_1 such that M_1 is in an accepting state or M_1 is in the QUERY state, and there is a computation of M_1 beginning with ID_α and ending with ID_β and in this computation neither the QUERY state nor any accepting state is entered except in instantaneous description $\mathrm{ID}_\beta\}$. The oracle machine M_1 uses work space $S_1 \in \mathcal{S}$ and so from M_1 one can construct a nondeterministic (nonoracle) machine M_2 such that $L(M_2) = L_2$ and M_2 uses work space at most $S_1(|\mathrm{ID}_\alpha \# \mathrm{ID}_\beta|)$. By hypothesis NTISP $(\mathcal{T}, \mathcal{S}) =$ NSPACE $(\mathcal{S})$ so that $L_2 \in$ NSPACE (S_1) and $S_1 \in \mathcal{S}$ imply that for some $T_2 \in \mathcal{T}$ and $S_2 \in \mathcal{S}$, there is a nondeterministic tisp (T_2, S_2) machine M_3 such that $L(M_3) = L_2$. Using M_3 we will show the existence of a nondeterministic tisp (T_3, S_3) oracle machine M_4 such that $L(M_4, A) = L_1$ where T_3 is in $\mathcal{T}$ and S_3 is in $\mathcal{S}$, thus concluding that L_1 is in NTISP $(\mathcal{T}, \mathcal{S}, A)$.

Since M_4 is nondeterministic, it is sufficient to describe its accepting computations. On input string w, M_4 writes the initial instantaneous description of M_1 on input string w, call it ID_0, on one of M_4's tapes. Next, M_4 writes a marker # at the right end of the string ID_0. Then M_4 nondeterministically guesses an instantaneous description, call it ID_1, of M_1 and writes this string to the right of the marker so that this tape of M_4 now contains $\mathrm{ID}_0 \# \mathrm{ID}_1$, with the requirement that the string ID_1 represent an instantaneous description of M_4 in either an accepting state or in the QUERY state. From this point M_4 acts as in (i).

(i) If M_4 has generated $\mathrm{ID}_{2i} \# \mathrm{ID}_{2i+1}$, then M_4 simulates a computation of M_3 on $\mathrm{ID}_{2i} \# \mathrm{ID}_{2i+1}$. If this computation is accepting and ID_{2i+1} represents an accepting instantaneous description of M_1, then M_4 halts in an accepting state. If this computation is accepting and ID_{2i+1} represents an instantaneous description of M_1 in state QUERY, then M_4 acts as in (ii). If this computation is not accepting, then M_4 halts in a nonaccepting state.

(ii) If M_4 has generated $\mathrm{ID}_{2i} \# \mathrm{ID}_{2i+1}$, $\mathrm{ID}_{2i} \# \mathrm{ID}_{2i+1} \in L_2$, and ID_{2i+1} represents an instantaneous description of M_1 in state QUERY, then M_4 queries the oracle and replaces string $\mathrm{ID}_{2i} \# \mathrm{ID}_{2i+1}$ with the proper successor $\mathrm{ID}_{2(i+1)}$ of the instantaneous description represented by ID_{2i+1}. Thus, $\mathrm{ID}_{2(i+1)}$ represents an instantaneous description of M_1 either in state YES or in state NO, depending on whether the string on the query tape represented by ID_{2i+1} is or is not in A. Next, M_4 writes a marker at the right end of the string $\mathrm{ID}_{2(i+1)}$. Then M_4 nondeterministically guesses an instantaneous description, call it $\mathrm{ID}_{2(i+1)+1}$, of M_1 and writes this string to the right of the marker so that this tape of M_4 now contains $\mathrm{ID}_{2(i+1)} \# \mathrm{ID}_{2(i+1)+1}$, with the requirement that $\mathrm{ID}_{2(i+1)+1}$ represent an instantaneous description of M_4 in either an accepting state or in the QUERY state. From this point M_4 acts as in (i).

It is clear from the description of M_4's accepting computations, M_4 accepts input string w relative to oracle set A if and only if there is an accepting computation of M_1 on w relative to oracle set A if and only if $w \in L(M_1, A) = L_1$. Since M_1 uses at most work space $S_1(|w|)$ on any computation on input string w, M_4 can initially mark $S_1(|w|)$ tape squares so that when M_4 writes ID_{2i} or ID_{2i+1} for any $i \geqq 0$, then those

strings have length no greater than $S_1(|w|)$. The machine M_3 uses work space at most S_2. Thus M_4 uses work space at most $S_3(n)$ where for all n, $S_3(n) = S_2(2S_1(n))$. From the conditions on $\mathscr{S}$ including the fact that $\mathscr{S}$ is closed under composition and S_1, $S_2 \in \mathscr{S}$, $S_3 \in \mathscr{S}$. Whenever M_4 writes $\mathrm{ID}_{2i} \# \mathrm{ID}_{2i+1}$, M_4 needs $|\mathrm{ID}_{2i} \# \mathrm{ID}_{2i+1}| \leqq 2S_1(|w|)+1$ steps; whenever M_4 simulates M_3 to determine whether ID_{2i} yields ID_{2i+1}, M_4 needs at most $T_2(S_1(|w|))$ steps; since M_1 makes at most $T_1(|w|)$ oracle queries, each accepting computation of M_4 on input string $w \in L(M_4, A) = L_1$ has at most $T_2(S_1(|w|)) \cdot T_1(|w|)$ steps. By hypothesis, the set of time bounds in $\mathscr{T}$ is closed under composition with functions in $\mathscr{S}$ and is closed under multiplication, so that the function $T_3(n) = T_2(S_1(n)) \cdot T_1(n)$ is in $\mathscr{T}$ and thus M_4 runs in time T_3. Thus, M_4 is a tisp (T_3, S_3) oracle machine and $L_1 = L(M_4, A) \in \mathrm{NTISP}(\mathscr{T}, \mathscr{S}, A)$. □

A special case of qusp machines was studied in [6], [7].

If $\mathscr{T}$ is a set of functions that are both running times and constructible space bounds, then for any oracle set A let $\mathrm{NQUERY}(\mathscr{T}, A) = \mathrm{NQUSP}(\mathscr{T}, \mathscr{T}, A)$. For any set A let $\mathrm{NPQUERY}(A) = \mathrm{NQUERY}(\text{poly}, A)$.

In [6] it is shown that NP = PSPACE if and only if for every A, $\mathrm{NP}(A) = \mathrm{NPQUERY}(A)$. Here we generalize this result.

THEOREM 2.3. *Let $\mathscr{T}$ be a set of functions that are both running times and constructible space bounds, and the pair $\mathscr{T}$, $\mathscr{T}$ satisfies Condition* 2.1. *Then the following statements are equivalent*:

(a) $\mathrm{NTIME}(\mathscr{T}) = \mathrm{NSPACE}(\mathscr{T})$.

(b) *For every set A*, $\mathrm{NTIME}(\mathscr{T}, A) = \mathrm{NQUERY}(\mathscr{T}, A)$.

Theorem 2.3 is a corollary of Theorem 2.2.

3. Time-space vs. time. In this section we define classes of languages specified by oracle machines that are bounded simultaneously in both time and in the size of instantaneous descriptions when the oracle is queried. We compare these classes with those specified by oracle machines that are bounded simultaneously in both time and space.

Let T and S be functions from the natural numbers to the natural numbers. A *tiqs (T, S) oracle machine* is an oracle machine such that on every input w, if there is an accepting computation on w, then there is an accepting computation which simultaneously uses at most $T(|w|)$ steps and every time the oracle is queried every work tape is bounded in length by $S(|w|)$. For any set A, let $\mathrm{NTIQS}(T, S, A)$ be the class of languages accepted relative to A by nondeterministic tiqs (T, S) oracle machines.

If $\mathscr{T}$ is a set of time bounds and $\mathscr{S}$ is a set of space bounds, then for any set A, let $\mathrm{NTIQS}(\mathscr{T}, \mathscr{S}, A) = \bigcup\{\mathrm{NTIQS}(T, S, A) | T \in \mathscr{T}, S \in \mathscr{S}\}$.

For any functions T and S and any set A, $\mathrm{NTISP}(T, S, A) \subseteq \mathrm{NTIQS}(T, S, A) \subseteq \mathrm{NTIME}(T, A)$, and $\mathrm{NTIQS}(T, S, \phi) = \mathrm{NTIME}(T, \phi) = \mathrm{NTIME}(T)$.

THEOREM 3.1. *Let $\mathscr{T}$ and $\mathscr{S}$ be sets of bounds satisfying Condition* 2.1. *The following are equivalent*:

(a) $\mathrm{NTISP}(\mathscr{T}, \mathscr{S}) = \mathrm{NTIME}(\mathscr{T})$.

(b) *For every set A*, $\mathrm{NTISP}(\mathscr{T}, \mathscr{S}, A) = \mathrm{NTIQS}(\mathscr{T}, \mathscr{S}, A)$.

Proof. The proof is similar to that of Theorem 2.2. We point out some of the differences and leave the details to the reader.

For arbitrary A and arbitrary L_1 in $\mathrm{NTIQS}(\mathscr{T}, \mathscr{S}, A)$, we must show that L_1 is in $\mathrm{NTISP}(\mathscr{T}, \mathscr{S}, A)$. If M_1 is a nondeterministic tiqs (T_1, S_1) oracle machine such that $L(M_1, A) = L_1$, $T_1 \in \mathscr{T}$, and $S_1 \in \mathscr{S}$, then we lose no generality by assuming that for every w in $L(M_1, A)$ every accepting instantaneous description of M_1 on w has length at most $S(|w|)$. From M_1 one can construct a nondeterministic (nonoracle) machine

M_2 such that $L(M_2)=L_2$ and M_2 operates in time $T_1(|\mathrm{ID}_\alpha \# \mathrm{ID}_\beta|)$, so that the hypothesis NTISP $(\mathcal{T}, \mathcal{S}) =$ NTIME $(\mathcal{T})$ yields $L_2 \in$ NTISP $(\mathcal{T}, \mathcal{S})$.

The proof now proceeds just like that of Theorem 2.2. □

Examples of pairs of sets $\mathcal{T}$ of time bounds and $\mathcal{S}$ of space bounds which satisfy Condition 2.1 and hence the hypotheses of Theorems 2.2 and 3.1 are the following:

(a) $\mathcal{T}=\{n^k | k>0 \text{ an integer}\}$ and
$\mathcal{S}=\{kn | k>0 \text{ an integer}\}$;
(b) $\mathcal{T}=\{n^k | k>0 \text{ an integer}\}$ and
$\mathcal{S}=\{n(\log n)^k | k>0 \text{ an integer}\}$;
(c) $\mathcal{T}=\{2^{(\log n)^k} | k>0 \text{ an integer}\}$ and
$\mathcal{S}=\{kn | k>0 \text{ an integer}\}$;
(d) $\mathcal{T}=\{2^{(\log n)^k} | k>0 \text{ an integer}\}$ and
$\mathcal{S}=\{n(\log n)^k | k>0 \text{ an integer}\}$;
(e) $\mathcal{T}=\{2^{(\log n)^k} | k>0 \text{ an integer}\}$ and
$\mathcal{S}=\{n^k | k>0 \text{ an integer}\}$.

Neither Theorem 2.2 nor Theorem 3.1 is as strong as one might wish. The main problem is the portion of Condition 2.1 demanding closure under composition for the sets of bounds. If a set $\mathcal{F}$ of time bounds or space bounds contains a function such as $f(n)=2^n$ and is closed under composition, then DTIME $(\mathcal{F}) =$ NTIME $(\mathcal{F}) =$ DSPACE $(\mathcal{F}) =$ NSPACE $(\mathcal{F})$ and there is no need to consider relativizations of any kind. Notice that the proofs of Theorems 2.2 and 3.1 use closure under composition in a crucial way. We do not know how to avoid this problem.

Let us return to the situation of polynomial time and linear space. It is known that NP $\neq$ NSPACE (lin) [4] and so either NTIME (poly, lin) $\neq$ NP or NTIME (poly, lin) $\neq$ NSPACE (lin) (possibly both). Thus, from Theorems 2.3 and 3.1 there exists a set A such that NQUSP (poly, lin A) $\neq$ NTIQS (poly, lin, A).

Certain separation theorems for NTISP classes are known [8]. These theorems have the following form: If (T_1, S_1) and (T_2, S_2) are compatible pairs and if T_2 (S_2) grows sufficiently faster than T_1 (resp. S_1), then NTISP $(T_1, S_1) \subsetneq$ NTISP (T_2, S_2). If one could show this type of theorem for fixed S, then using the technique of [5] one could obtain results such as NTISP $(n^k, n) \subsetneq$ NTISP (n^{k+1}, n) and hence conclude that $\bigcup_k$ NTISP $(n^k, n) \subsetneq$ NSPACE (n).

4. Hierarchies. Observations of Doner [9] regarding diagonalization arguments lead to the following fact, the proof of which is left to the reader.

PROPOSITION 4.1. *There exist sets A and B such that neither* NTISP (poly, lin, A) *nor* NSPACE (lin, B) *is closed under complementation.*

It is not known whether either NTISP (poly, lin) or NSPACE (lin) is closed under complementation so that one might consider possible hierarchies based on NTISP (poly, lin) or NSPACE (lin) just as the polynomial-time hierarchy is based in NP [14], [16]. This type of investigation was carried out in [7] by considering relativized hierarchies with NP(·) and NPQUERY(·) as operators. Here we develop general results similar to those in [7] but beginning with Theorems 2.2 and 3.1.

Let $\mathcal{T}$ and $\mathcal{S}$ be a pair of sets of bounds satisfying Condition 2.1, and let A be a set. Let NTISP$^{(1)}$ $(\mathcal{T}, \mathcal{S}, A) =$ NTISP $(\mathcal{T}, \mathcal{S}, A)$, and for each $i \geqq 1$, let NTISP$^{(i+1)}$ $(\mathcal{T}, \mathcal{S}, A) = \bigcup\{$NTISP $(\mathcal{T}, \mathcal{S}, B) | B \in$ NTISP$^{(i)}$ $(\mathcal{T}, \mathcal{S}, A)\}$ and let NTISP$^{(*)}$ $(\mathcal{T}, \mathcal{S}, A) = \bigcup_{i \geqq 1}$ NTISP$^{(i)}$ $(\mathcal{T}, \mathcal{S}, A)$. For each $i \geqq 1$, define NSPACE$^{(i)}$ $(\mathcal{S}, A)$, NTIME$^{(i)}$ $(\mathcal{T}, A)$, NQUSP$^{(i)}$ $(\mathcal{T}, \mathcal{S}, A)$, and NTIQS$^{(i)}$ $(\mathcal{T}, \mathcal{S}, A)$ similarly, and define NSPACE$^{(*)}$ $(\mathcal{S}, A) = \bigcup_{i \geqq 1}$ NSPACE$^{(i)}$ $(\mathcal{S}, A)$, NTIME$^{(*)}$ $(\mathcal{T}, A) = \bigcup_{i \geqq 1}$ NTIME$^{(i)}$ $(\mathcal{T}, A)$, NQUSP$^{(*)}$ $(\mathcal{T}, \mathcal{S}, A) = \bigcup_{i \geqq 1}$ NQUSP$^{(i)}$ $(\mathcal{T}, \mathcal{S}, A)$, and

$\mathrm{NTIQS}^{(*)}(\mathcal{T}, \mathcal{S}, A) = \bigcup_{i>1} \mathrm{NTIQS}^{(i)}(\mathcal{T}, \mathcal{S}, A)$. Write $\mathrm{NTISP}^{(i)}(\mathcal{T}, \mathcal{S})$, $\mathrm{NSPACE}^{(i)}$ $\mathcal{S})$, $\mathrm{NTIME}^{(i)}(\mathcal{T})$, $\mathrm{NTISP}^{(*)}(\mathcal{T}, \mathcal{S})$, $\mathrm{NSPACE}^{(*)}(\mathcal{S})$, and $\mathrm{NTIME}^{(*)}(\mathcal{T})$ in the case $A = \phi$.

THEOREM 4.2. *Let $\mathcal{T}$ and $\mathcal{S}$ be a pair of sets of bounds satisfying Condition* 2.1. *The following are equivalent*:

(a) $\mathrm{NTISP}^{(*)}(\mathcal{T}, \mathcal{S}) = \mathrm{NSPACE}^{(*)}(\mathcal{S})$.

(b) *For every set A*, $\mathrm{NTISP}^{(*)}(\mathcal{T}, \mathcal{S}, A) = \mathrm{NQUSP}^{(*)}(\mathcal{T}, \mathcal{S}, A)$.

The proof of Theorem 4.2 will follow from the proof of Theorem 2.2 once a preliminary result is established.

LEMMA 4.3. *Let $\mathcal{T}$ and $\mathcal{S}$ be a pair of sets of bounds satisfying Condition* 2.1. *For every set A*, $\mathrm{NQUSP}^{(2)}(\mathcal{T}, \mathcal{S}, A) = \bigcup\{\mathrm{NTISP}(\mathcal{T}, \mathcal{S}, B) \mid B \in \mathrm{NQUSP}(\mathcal{T}, \mathcal{S}, A)\}$.

The proof of Lemma 4.3 is similar to that of Theorem 2.2 and is left to the reader.

A result similar to Lemma 4.3 yields the following fact.

THEOREM 4.4. *Let $\mathcal{T}$ and $\mathcal{S}$ be a pair of sets of bounds satisfying Condition* 2.1. *The following are equivalent*:

(a) $\mathrm{NTISP}^{(*)}(\mathcal{T}, \mathcal{S}) = \mathrm{NTIME}^{(*)}(\mathcal{T})$;

(b) *For every set A*, $\mathrm{NTISP}^{(*)}(\mathcal{T}, \mathcal{S}, A) = \mathrm{NTIQS}^{(*)}(\mathcal{T}, \mathcal{S}, A)$.

Apparently classes of the form $\mathrm{NTISP}^{(*)}(\mathcal{T}, \mathcal{S})$ or $\mathrm{NSPACE}^{(*)}(\mathcal{S})$ have not been studied previously. In one case they may be of independent interest. The class of rudimentary predicates has been studied from the point of view of complexity theory. Let NL denote the class of languages accepted by nondeterministic machines that run in linear time, i.e., NL = NTIME (lin) = NTISP (lin, lin), and for every oracle set A, define $\mathrm{NL}^{(*)}(A)$ as above. Wrathall [17] has shown that the class of rudimentary predicates is precisely $\mathrm{NL}^{(*)} = \mathrm{NL}^{(*)}(\phi)$. Clearly $\mathrm{NL}^{(*)} \subseteq$ DSPACE (lin). Since NL $\subseteq$ NTISP (poly, lin), $\mathrm{NL}^{(*)} \subseteq \mathrm{NTISP}^{(*)}$ (poly, lin) and it is not known whether this inclusion is strict.

5. On downwards translation. Generally, equalities at a low level of some complexity hierarchy imply equality at higher levels. It is not known whether the converse holds. For example, if P = NP, then DEXT = NEXT; but if DEXT = NEXT, then all we know about P =? NP is that every tally language in NP is in P.

When we consider relativized complexity classes, there are examples that show that equality above does not imply equality below: there is an oracle A such that $\mathrm{P}(A) \neq \mathrm{NP}(A)$ but $\mathrm{NP}(A) = \text{co-NP}(A)$ [2]. Here we provide evidence that it will be difficult to prove that equality above implies equality below.

The first result concerns classes specified by time-bounded machines.

THEOREM 5.1. *There exists an oracle set A such that* $\mathrm{P}(A) \neq \mathrm{NP}(A)$ *but* $\mathrm{DEXT}(A) = \mathrm{NEXT}(A)$.

Proof. Let $\mathrm{P}_1, \mathrm{P}_2, \cdots$ be an enumeration of all deterministic polynomial-time-bounded oracle machines and let $\mathrm{NP}_1, \mathrm{NP}_2, \cdots$ be an enumeration of all nondeterministic polynomial-time-bounded oracle machines. Without loss of generality, assume that for each $i = 1, 2, \cdots, p_i$ is a polynomial that bounds the running time of both P_i and NP_i.

Let $\mathrm{E}_1, \mathrm{E}_2, \cdots$ be an enumeration of all deterministic exponential-time-bounded oracle machines and let $\mathrm{NE}_1, \mathrm{NE}_2, \cdots$ be an enumeration of all nondeterministic exponential-time-bounded oracle machines. Without loss of generality, assume that for each $i = 1, 2, \cdots, e_i$ is an exponential function (for some $c_i > 0$, $e_i(n) = 2^{c_i n}$) that bounds the running time of both E_i and NE_i.

For any set B, let $L(B) = \{x \mid \text{there exists a } y \text{ such that } |x| = |y| \text{ and } xy \in B\}$ and let $S(B) = \{\langle i, x, 0^n\rangle \mid \text{the machine } \mathrm{NE}_i \text{ accepts } x \text{ relative to } B \text{ in less than } 2^n \text{ steps}\}$, where $\langle \cdot, \cdot, \cdot \rangle$ is a suitable encoding function. Clearly, $L(B)$ is in $\mathrm{NP}(B)$.

We construct an oracle set A in stages. At stage n the set A_n is constructed from A_{n-1} such that $A_{n-1} \subseteq A_n$. The construction guarantees that (i) $L(A) \notin \mathrm{P}(A)$, and (ii) $y \in S(A)$ if and only if $y0^{2^{|y|}} \in A$. Since $L(A)$ is in NP (A), the first condition shows that P $(A) \neq$ NP (A).

The second condition shows that DEXT $(A) =$ NEXT (A) since $x \in L(\mathrm{NE}_i, A)$ if and only if $\langle i, x, 0^{c_i|x|}\rangle \in S(A)$ if and only if $y0^{2^{|y|}} \in A$ where $y = \langle i, x, 0^{c_i|x|}\rangle$, and for each i there exists a j such that machine E_j can read input x and write $y0^{2^{|y|}}$ on its query tape, with $y = \langle i, x, 0^{c_i|x|}\rangle$, within $e_j(|x|)$ steps.

At some odd stages of the contruction, we will cancel an index j when it can be guaranteed that $L(P_j, A) \neq L(A)$. At even stages condition (ii) is guaranteed.

During the construction of A, strings will be reserved for A or $\bar{A}$. Strings not reserved for either A or $\bar{A}$ will be considered to be in $\bar{A}$ if queried. Once a string is reserved for a set nothing done later affects its status.

Let $g(i)$ be defined as follows: $g(0) = 1$ and $g(i+1) = 2^{2^{g(i)}} + 1$.

Stage $n = 0$. Let $A_0 = \varnothing$.

Stage $n = 2k+1$. If several preconditions are met by at least one index j, then cancel the least such one; otherwise, skip this stage. The preconditions are as follows:

(1) There is an i such that $g(i) = n$;
(2) $p_j(n) < 2^{k+1} + k + 1$;
(3) $2n < 2^{k+1} + k + 1$;
(4) $p_j(n) < 2g(i+1)$.

If index j is cancelled at this stage, find some x such that $|x| = n$ and for no y with $|y| = n$ has xy been reserved for either A or $\bar{A}$. Run machine P_j with oracle set A_{n-1} on input string x and reserve for $\bar{A}$ all unreserved strings queried. If P_j accepts x relative to A_{n-1}, then for all y with $|y| = n$, reserve xy for $\bar{A}$; in this case, it is guaranteed that $x \notin L(A)$. If P_j rejects x relative to A_{n-1}, then for some y such that $|y| = |x|$ and xy is not queried in its computation on x relative to A_{n-1}, place xy into A; in this case, it is guaranteed that $x \in L(A)$. Thus, if index j is cancelled at this stage, then $L(A) \neq L(P_j, A)$.

Stage $n = 2k$. Consider each string y such that $|y| = k$ and y has the form $\langle i, x, 0^m\rangle$. For such a y, run NE_i with oracle set A_{n-1} on input x for at most 2^m steps. If some computation of NE_i with oracle set A_{n-1} accepts input x in at most 2^m steps, place $y0^{2^{|y|}}$ into A and reserve for $\bar{A}$ those unreserved strings queried in one such computation. If no computation of NE_i with oracle set A_{n-1} accepts input x in at most 2^m steps, then determine whether the addition of some unreserved elements to A_{n-1} will cause acceptance. There are two cases:

Case (a): If so, then place those unreserved strings queried on an accepting path appropriately into A and $\bar{A}$. Also, put $y0^{2^{|y|}}$ into A.

Case (b): If not, then do nothing except reserving $y0^{2^{|y|}}$ for $\bar{A}$.

These two cases ensure that the behavior of NE_i on x is immune to any other possible additions to the oracle A.

Consider the construction at stage $n = 2k+1$. Preconditions (2) and (3) ensure that the construction at this stage does not reserve for A or $\bar{A}$ any string $y0^{2^{|y|}}$ which may have to be reserved according to the requirements of the next even stage. Precondition (2) implies precondition (4), and precondition (4) ensures that what is reserved at this stage does not affect the availability of an x (the xys will have length $2g(i+1)$) at the next odd stage.

At any odd-numbered stage, say $n = 2k+1 = g(i)$, we must guarantee that we can find some x ($|x| = n$) such that for no y ($|y| = |x|$) was xy reserved. This entails examining the number of strings reserved during previous odd- and even-numbered

stages. By precondition (4), no previous odd-numbered stage carries out a simulation where any string queried has length $2n$. So only the even-numbered stages cause concern. Now at any stage $2l$, fewer than 2^l strings are reserved by machines represented by any of the 2^l possible encodings, hence fewer than 2^{2l} strings are reserved. So the total number of strings reserved by all previous even-numbered stages is less than $\sum_{l=1}^{k} 2^{2k} < 2^{2k+1} = 2^n$. There are 2^n sets $H(x) = \{xy \mid |x| = |y|\}$ so since fewer than 2^n strings will have been reserved by stage n, there must exist some x for which no element of $H(x)$ is reserved. This means that at any stage numbered $n = 2k+1$ that is performed, one can find some x of length n such that for no y of length n will xy be reserved for either A or $\bar{A}$. Once such an x is found one may still have to find some xy not queried during that stage. But by precondition (2), $p_j(n)$ is less than 2^n which is the size of $H(x)$. Thus not all elements of $H(x)$ can be queried by the computation of P_j on input x. For each j, the preconditions (1)–(4) will be satisfied infinitely often, so every index j will eventually be cancelled. Hence, for every index j, $L(\mathrm{P}_j, A) \neq L(A)$ and so $L(A) \notin \mathrm{P}(A)$.

In an even-numbered stage, strings of the form $y0^{2^{|y|}}$ are placed into A or $\bar{A}$. We must guarantee that these strings were not previously reserved. At an even stage $n = 2|y|$, the running time of any machine encoded by a string of length $|y|$ is less than $2^{|y|}$, so no string $y0^{2^{|y|}}$ could be queried and, hence, reserved. At the odd-numbered stages precondition (2) ensures that queried strings will not be so long and precondition (3) ensures that what is added to A and $\bar{A}$ to produce a string in $L(A)$ and $\overline{L(A)}$ will not be so long. This guarantees that condition (ii) is satisfied. □

Theorem 5.1 appeared earlier in [15].

Baker, Gill and Solovay [2] established the existence of a set B such that $\mathrm{NP}(B) - \mathrm{P}(B)$ contains a tally language, i.e., a language on a one-letter alphabet. Thus, for this particular set B, $\mathrm{DEXT}(B) \neq \mathrm{NEXT}(B)$. In contrast, Theorem 5.1 establishes the existence of a set A such that $\mathrm{P}(A) \neq \mathrm{NP}(A)$ but $\mathrm{DEXT}(A) = \mathrm{NEXT}(A)$, so that $\mathrm{NP}(A) - \mathrm{P}(A)$ contains no tally languages.

It is known [4], [5] that NP ≠ DEXT, PSPACE ≠ DEXT, and PSPACE ≠ NEXT. The questions "PSPACE ⊆? DEXT" and "PSPACE ⊆? NEXT" lead one to study the proof of Theorem 5.1. The time bounds $p_i, \cdots$ for the oracle machines P_i serve only to bound the number of queries made in P_i's computations and to bound the lengths of the strings on the query tape. Thus, we have the following result.

THEOREM 5.2. *There exists an oracle set A such that* $\mathrm{PQUERY}(A) \neq \mathrm{NPQUERY}(A)$ *but* $\mathrm{DEXT}(A) = \mathrm{NEXT}(A)$.

Finally, the techniques of Simon and Gill [13] can be used to establish the following result.

THEOREM 5.3. *For every recursive set A that is not in* PSPACE, *there exists a recursive set B such that* $A \in \mathrm{PSPACE}(B)$ *but* $A \notin \mathrm{NPQUERY}(B)$, *so that* $\mathrm{NPQUERY}(B) \neq \mathrm{PSPACE}(B)$.

REFERENCES

[1] D. ANGLUIN, *On relativizing auxiliary pushdown machines*, Math. Systems Theory, 13 (1980), pp. 283–299.

[2] T. BAKER, J. GILL AND R. SOLOVAY, *Relativizations on the* P =? NP *question*, this Journal, 4 (1975), pp. 431–442.

[3] T. BAKER AND A. SELMAN, *A second step towards the polynomial hierarchy*, Theoret. Comput. Sci., 8 (1979), pp. 177–187.

[4] R. BOOK, *On languages accepted in polynomial time*, this Journal, 1 (1972), pp. 281–287.

[5] ———, *Translational lemmas, polynomial time, and* $(\log n)^j$*-space*, Theoret. Comput. Sci., 1 (1975), pp. 215–226.

[6] ———, *Bounded query machines: on* NP *and* PSPACE, Theoret. Comput. Sci., 15 (1981), pp. 27–39.
[7] R. BOOK AND C. WRATHALL, *Bounded query machines: on* NP( ) *and* NPQUERY( ), Theoret. Comput. Sci., 15 (1981), pp. 41–50.
[8] A. BRUSS AND A. MEYER, *On time-space classes and their relation to the theory of real addition*, Theoret. Comput. Sci. 11 (1980), pp. 59–69.
[9] J. DONER, *Relativized complexity classes*, submitted for publication.
[10] R. LADNER AND N. LYNCH, *Relativization of questions about log space computability*, Math. Systems Theory, 10 (1976), pp. 19–32.
[11] W. SAVITCH, *Relationships between nondeterministic and deterministic space complexities*, J. Comput. Syst. Sci. 4 (1970), pp. 177–192.
[12] I. SIMON, *On some subrecursive reducibilities*, Ph.D. dissertation, Stanford University, Stanford, CA, 1977.
[13] I. SIMON AND J. GILL, *Polynomial reducibilities and upwards diagonalizations*, Proc. 9th ACM Symposium on Theory of Computing, 1977, pp. 186–194.
[14] L. STOCKMEYER, *The polynomial-time hierarchy*, Theoret. Comput. Sci., 3 (1976), pp. 1–22.
[15] C. WILSON, *Relativization, reducibilities, and the exponential hierarchy*, M.S. thesis, University of Toronto, Toronto, 1980.
[16] C. WRATHALL, *Complete sets and the polynomial-time hierarchy*, Theoret. Comput. Sci., 3 (1976), pp. 23–33.
[17] ———, *Rudimentary predicates and relative computation*, this Journal, 7 (1976), pp. 194–209.

SIAM J. COMPUT.
Vol. 11, No. 3, August 1982

© 1982 Society for Industiral and Applied Mathematics
0097-5397/82/1103-0018 $01.00/0

RANKING AND UNRANKING OF 2-3 TREES*

UDAI GUPTA,† D. T. LEE‡ AND C. K. WONG§

Abstract. In this paper we consider the generating, ranking, and unranking of 2-3 trees with n keys. We propose a linear ordering among these trees. The problem of ranking is to determine the rank of a given tree in this ordering, while unranking means constructing the tree of a given rank. The main result is that ranking and unranking can be done in $O(n)$ time after a preprocessing step that takes $O(n^2)$ time and space.

Key words. algorithms, 2-3 trees, permutation, ranking and unranking

1. Introduction. The problem of generating, ranking and unranking binary trees and k-ary trees has received considerable attention in the recent past [2], [8], [9], [10], [13], [14]. Typically, a one-to-one correspondence is established between a class of trees and certain integer sequences. It is then shown how these sequences can be generated in order (usually lexicographic) and how given a sequence its position in this ordering can be determined (ranking) and vice-versa (unranking). These procedures have obvious uses in the generation of random data to test and predict the behavior of algorithms that manipulate these classes of trees.

In [14], Zaks and Richards present algorithms for generating, ranking and unranking all trees with n_i nodes having k_i sons each, $i = 1, 2, \cdots, t$, and $n_0 + 1$ leaves (where $n_0 = \sum_{1 \leqq i \leqq t} (k_i - 1)n_i$). Their algorithm for generation runs $O(n)$ time where $n = \sum_{0 \leqq i \leqq t} n_i$. The algorithms for ranking and unranking take $O(n(t+1))$ time after a preprocessing step that takes $O((t+1)\nu)$ time and $O(\nu)$ space, where $\nu = \prod_{1 \leqq i \leqq t} n_i$. Alternatively, if no preprocessing is done, then ranking and unranking can be done in $O(n^2(t+1)^2)$ time. Although the authors deal only with ordered trees, it is clear that their methods can be applied to binary trees and k-ary trees after augmentation. However, the methods of Zaks and Richard cannot be applied to 2-3 trees.

2. 2-3 trees. A variety of balanced-tree schemes have been proposed for the organization of information so as to guarantee worst-case logarithmic search times. One such scheme, called "2-3 trees" was introduced by J. Hopcroft (see, for instance, [1], [5]). A 2-3 *tree* is a tree in which each internal (nonleaf) node has 2 or 3 sons, and every path from the root to a leaf is of the same length. Internal nodes contain 1 or 2 keys depending upon whether they have 2 or 3 sons respectively.[1] Since we are only interested in the structure of the tree, the key values are immaterial and keys will henceforth simply be shown as dots.

Our problem then is to devise algorithms to generate all 2-3 trees with n keys and to rank and unrank these trees. Figure 1 shows the 4 different 2-3 trees with 7 keys.

2.1. Encoding and ordering. We shall represent a 2-3 tree with n keys by a sequence $a_1 a_2 a_3 \cdots a_k$ that is obtained as follows: Starting with the lowermost level of internal nodes, list the number of sons of each node, level by level ending with the

* Received by the editors July 31, 1980, and in final revised form November 20, 1981.

† Northwestern University, Evanston, Illinois 60201.

‡ Northwestern University, Evanston, Illinois 60201. The research of this author was supported in part by the National Science Foundation under grant MCS-7916847.

§ IBM T. J. Watson Research Center, Yorktown Heights, New York 10598.

[1] Alternatively, the keys may be stored only in the leaves and internal nodes may contain only navigational information [1].

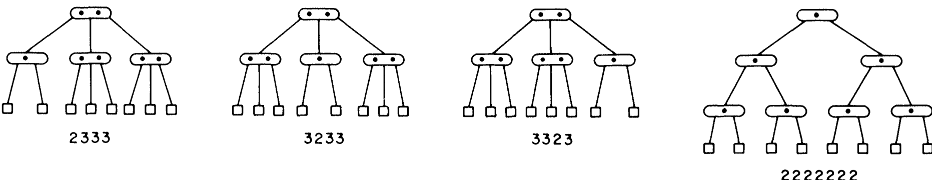


FIG. 1 *All the* 2-3 *trees with* 7 *keys.*

root. Within the same level the listing is done from left to right. The sequences corresponding to the trees in Fig. 1 are listed below each tree.

If the sequence $a_1a_2 \cdots a_k$ represents a 2-3 tree with n keys, then it must have the following properties:

1) $\sum_i (a_i - 1) = n$;
2) $a_i \in \{2, 3\}, i = 1, 2, \cdots, k$;
3) If $k > 1$, then there exist $l_0, l_1, \cdots . l_r$ such that

$$0 = l_0 < l_1 < \cdots < l_{r-1} = k-1, \qquad l_r = k,$$
(1)
$$l_i - l_{i-1} = \sum_{l_i < j \leqq l_{i+1}} a_j \quad \text{for } i = 1, 2, \cdots, r-1.$$

We shall call a sequence that satisfies properties 2) and 3) a *feasible sequence*, and one that satisfies properties 1), 2) and 3) an *n-feasible sequence*. Property 3) says that every feasible sequence should be partitionable into levels such that the number of nodes on a particular level is equal to the sum of the number of sons of nodes at the next higher level, and the highest level has exactly one node. In fact, in the sequence $l_0l_1 \cdots l_r$, l_1 is the number of internal nodes at the lowest level, l_2 is the total number of internal nodes at the lowest two levels and l_3 is the total number of internal nodes at the lowest three levels and so on (Fig. 2). It should be easy to see that there is a one-to-one correspondence between the class of 2-3 trees with n keys and the class of n-feasible sequences. This follows from the fact that for a given feasible sequence $a_1a_2 \cdots a_k$, there is a unique choice of $l_0, l_1, \cdots, l_r$ which we shall henceforth call the *l-sequence* of the given feasible sequence. For a given n-feasible sequence $a_1a_2 \cdots a_k$ we can compute the corresponding l-sequence as follows:

$l_0 = 0$, l_1 is such that $\sum_{0 < i \leqq l_1} a_i = n+1$ and for all j, $2 \leqq j < r$, l_j is such that $\sum_{l_{j-1} < i \leqq l_j} a_i = l_{j-1} - l_{j-2}$, where r is such that $l_{r-1} = k$, $l_r = k$.

Note also that if $a_1a_2 \cdots a_k$ is a feasible sequence with the l-sequence $l_0l_1 \cdots l_r$ then the suffix $a_{1+l_1}a_{2+l_1} \cdots a_k$ is also a feasible sequence and its l-sequence is $l'_0l'_1 \cdots l'_{r-1}$ where $l'_i = l_{i+1} - l_i$ for $i = 0, 1, \cdots, r-1$. The new sequence $a_{1+l_1}a_{2+l_1} \cdots a_k$ represents the 2-3 tree obtained by removing the lowest level of the original 2-3 tree.

Now we need to define an order on the set of n-feasible sequences. We could have chosen to order these sequences lexicographically, but we do not know of efficient algorithms for generation, ranking and unranking, given such an ordering. Instead we define our ordering $<$ recursively as follows:

Given two n-feasible sequences

$$a_1a_2 \cdots a_k \quad \text{and} \quad a'_1a'_2 \cdots a'_{k'}$$

with their corresponding l-sequences

$$l_0l_1 \cdots l_r \quad \text{and} \quad l'_0l'_1 \cdots l'_{r'},$$

$a_1a_2 \cdots a_k < a'_1a'_2 \cdots a'_{k'}$, if and only if

1) $l_1 < l'_1$;

or

2) $l_1 = l'_1$ and $a_1a_2 \cdots a_{l_1} <_L a'_1a'_2 \cdots a'_{l_1}$,
where $<_L$ denotes lexicographic ordering;

or

3) $l_1 = l'_1$ and $a_1a_2 \cdots a_l = a'_1a'_2 \cdots a'_{l_1}$
and $a_{1+l_1} \; a_{2+l_1} \cdots a_k < a'_{1+l_1}a'_{2+l_1} \cdots a'_{k'}$.

Figure 2 shows 3 trees with 17 keys. By the above definition, $T_1 < T_2 < T_3$.

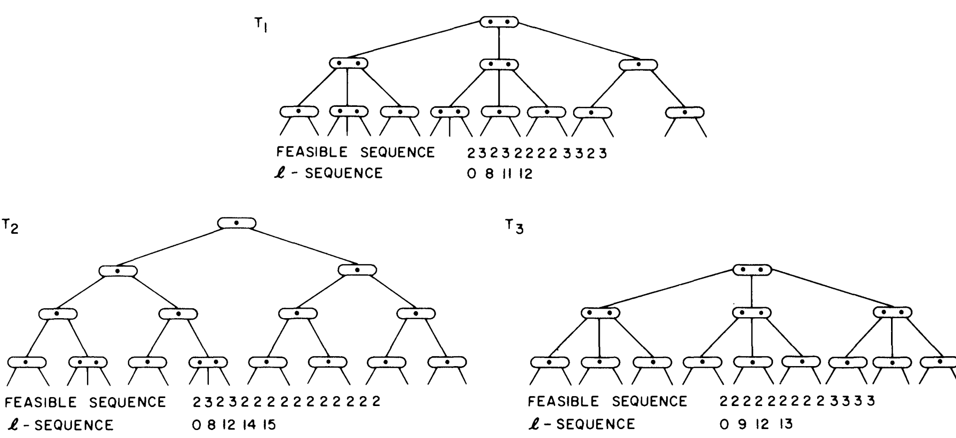


FIG. 2

2.2. The search graph. To rank or unrank 2-3 trees with n keys, we first construct a search graph with the help of which we can then rank and unrank in $O(n)$ time. This idea is often used in the ranking and unranking of combinatorial objects [11], [12], [14]. The search graph stores, among other things, the number of different 2-3 trees with k keys for various "useful" values of $k \leqq n$.

The search graph is a weighted directed acyclic graph with nodes identified by labels: $n+1$ (the number of leaves) and all useful partition sizes (a subset of $\{1, 2, 3, \cdots, \lfloor (n+1)/2 \rfloor\}$). We start with the node labeled $n+1$ and repeat the following process of creating new nodes until label 1 is assigned. Given a node labeled p, we construct nodes labeled $n_1, n_2, \cdots, n_k$ ($k = \lfloor p/2 \rfloor - \lceil p/3 \rceil + 1$) such that there exist partitions of p of sizes $n_1, n_2, \cdots, n_k$. We then repeat the process for other nodes. For two nodes p and q, $p > q$, there is a directed edge from p to q if and only if there is a partition of size q for p. The weight of this edge is the number of distinct permutations of the q-sized partition. The graph is constructed in two passes, the forward pass and the backward pass. At the end of the forward pass we have the graph with all its nodes and edges and weights on the edges.

We illustrate the construction and use of the search graph by an example used through the rest of the paper. Let us suppose we are interested in 2-3 trees that have 19 keys, and therefore 20 leaves (or failure nodes). We consider all the ways in which 20 can be partitioned with just 2's and 3's in the partition. For each partition we compute the number of distinct permutations possible. Table 1 lists the 4 possible partitions and the respective number of permutations.

TABLE 1

Partition	Size of partition	Number of permutations
2, 2, 2, 2, 2, 2, 2, 2, 2, 2	10	$\frac{10!}{10!}=1$
2, 2, 2, 2, 2, 2, 2, 3, 3	9	$\frac{9!}{2!7!}=36$
2, 2, 2, 2, 3, 3, 3, 3	8	$\frac{8!}{4!4!}=70$
2, 3, 3, 3, 3, 3, 3,	7	$\frac{7!}{6!}=7$

The graph obtained at the end of the forward pass is shown in Fig. 3. Note that there is no node corresponding to 6, since this is not a "useful" partition size. The number of nodes in the graph is bounded by $1+\lfloor (n+1)/2 \rfloor$, i.e., $O(n)$, and the number of edges is $O(n^2)$. The outdegree of node p is $\lfloor p/2 \rfloor - \lceil p/3 \rceil + 1$. The edge (p, q) with weight $w(p, q)$ denotes that every 2-3 tree with q leaves and height h can be "extended" to $w(p, q)$ distinct 2-3 trees with p leaves and height $(h+1)$. If we have a table of the factorial functions for arguments $\{1, 2, \cdots, \lfloor (n+1)/2 \rfloor\}$ then clearly the forward pass can be done in $O(n^2)$ time. The backward pass starts at node 1 and successively attaches weights to the nodes of the graph. The weight of node p is the total number of distinct 2-3 trees with p leaves. Let $W(p)$ denote the weight of node p then we have

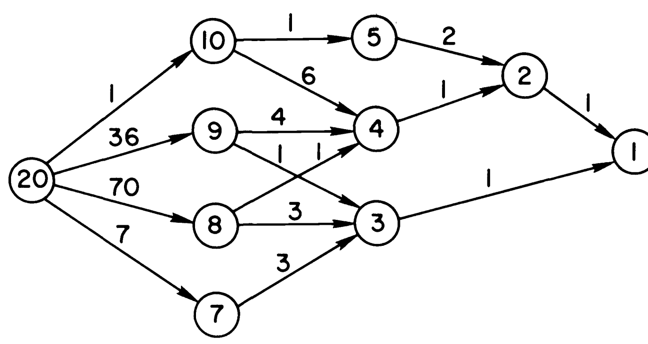


FIG. 3

$$W(1)=1, \qquad W(p)=\sum_{(p,q) \text{ is an edge}} W(q) * w(p, q). \tag{2}$$

The final search graph for our example is shown in Fig. 4. Weights of nodes are shown in parentheses. Again, the backward pass involves looking at each edge once and can be done in $O(n^2)$ time. Therefore, the entire graph can be constructed in $O(n^2)$ time and needs $O(n^2)$ storage space.

We now state the preceding discussion as an algorithm. In order to keep the description simple, we have considered all of $1, 2, 3, \cdots, \lfloor (n+1)/2 \rfloor, n+1$ as nodes of the search graph, even though some of these partition sizes may not be "useful."

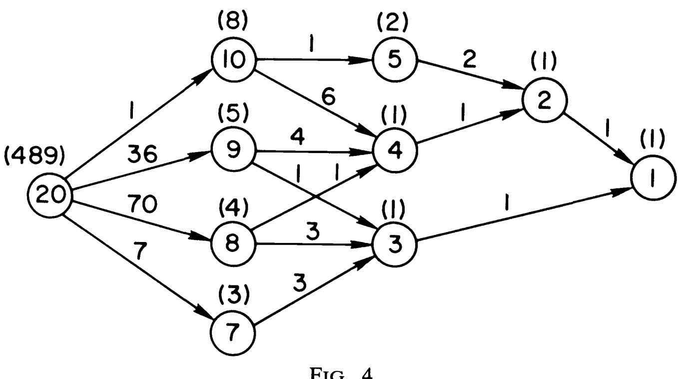


FIG. 4

ALGORITHM SEARCH GRAPH
Input: n, the number of keys.
Output: The search graph with $W(p)$, the weight of node p and $w(p, q)$, the weight of edge (p, q).
Method: The nodes of the graph are $1, 2, 3, \cdots, \lfloor(n+1)/2\rfloor$, and $n+1$.

begin
  (**Comment:** Forward Pass.)
  for $p = n+1, \lfloor(n+1)/2\rfloor, \lfloor(n+1)/2\rfloor - 1, \cdots, 3, 2$ **do**
    begin
      $s \leftarrow p - 2 * \lfloor p/2\rfloor; r \leftarrow \lfloor p/2\rfloor - s;$
      while $r \geqq 1$ **do**
        begin
          $q \leftarrow r + s;$
          $w(p, q) \leftarrow q!/(r! * s!);$
          $r \leftarrow r - 3; s \leftarrow s + 2$
        end
    end
  (**Comment:** Backward Pass.)
  $W(1) \leftarrow 1;$
  for $p = 2, 3, \cdots, \lfloor(n+1)/2\rfloor, n+1$ **do**
    begin
      sum $\leftarrow 0;$
      for all q such that (p, q) is an edge, i.e.,
        $w(p, q)$ was set in the forward pass **do**
          sum $\leftarrow$ sum $+ w(p, q) * W(q);$
      $W(p) \leftarrow$ sum
    end
end

2.3. Ranking, unranking and generation. Once the search graph has been constructed, ranking and unranking can be done in $O(n)$ time. Let us look at our running example of 2-3 trees with 19 keys. Given an n-feasible sequence, say 323322323233, we wish to find its position in the previously defined ordering of all 2-3 trees with 19 keys. The first step is to find the corresponding l-sequence. This is easily done and for our sequence we have the l-sequence 0, 8, 11, 12. This means that the lowest level of internal nodes has 8 nodes in it. Our search tells us that there are $w(20, 7) * W(7)$, i.e. $7 \times 3 = 21$ trees that have 7 nodes in the lowest level of internal

nodes and these precede our given tree. Next we determine the rank (in a lexicographic ordering) of the permutation 32332232 among all permutations of 4 2's and 4 3's. This turns out to be 53, which means that there are another $52 * W(8)$, i.e., 208 trees that precede our given tree. We have thus determined, looking at the lowest internal level alone, that there are $21+208=229$ trees that precede our tree. The process is then continued for each successive higher level until the root is reached. Finally we have,

Number of trees that precede our tree equals

$$(21+208)+(0+1)=230.$$

Thus the rank of the tree is 231.

ALGORITHM RANK

(**Comment:** This algorithm computes the rank of a given 2-3 tree represented as an n-feasible sequence. It computes the rank in a level-by-level manner. Initially, we have $p=n+1$ leaves and $q=l_1-l_0$ internal nodes at the lowest level. In the next iteration we have $p=q$ leaves and $q=l_2-l_1$ internal nodes. The process repeats as if we were removing the leaves and treating the lowest level of internal nodes of the previous tree as leaves of a new tree. In each iteration we have a 2-3 tree with p leaves and q internal nodes. The number of trees preceding the tree in the defined ordering is $\sum_{q'=\lceil p/3 \rceil}^{q-1} w(p, q') * W(q')$ plus the number of the same kind of trees (i.e., trees with p leaves and q internal nodes at the lowest level) whose permutation of the numbers of sons of the q internal nodes precede the corresponding permutation of the given tree. (See steps (3) and (4) below.) The variable sum will hold the value of the rank when the algorithm terminates.)

Input: An n-feasible sequence $a_1 a_2 \cdots a_k$ and the corresponding l-sequence $l_0, l_1, \cdots, l_s$.

Output; The rank of this sequence in the defined ordering of all n-feasible sequences.

Method:

```
  begin
(1)     p ← n+1; sum ← 1;
        for u ← 1 until s−1 do
          begin
(2)         q ← l_u − l_{u−1};
            for q' ← ⌈p/3⌉ until q−1 do
(3)           sum ← sum + w(p, q') * W(q');
(4)         sum ← sum + (RANKPERM(a_{1+l_{u−1}} a_{2+l_{u−1}} ··· a_{l_u}) − 1) * W(q);
(5)         p ← q
          end;
        return (sum)
  end
```

The algorithm RANKPERM takes a permutation $a_1 a_2 \cdots a_k$ that has r 2's and $(k-r)$ 3's and outputs in $O(k)$ time the rank of the given permutation in a lexicographical ordering of all permutations of r 2's and $(k-r)$ 3's, e.g., $(2)^r(3)^{k-r}$ has rank 1 and $(3)^{k-r}(2)^r$ has rank $\binom{k}{r}$.[2] The problem is similar to the lexicographical ranking of r-sized subsets of k distinct objects (see, for example, [6], [7]) and the algorithm is omitted here. See [3] for details.

[2] $(a)^r$ denotes a sequence of r a's.

Let us now look at the complexity of the algorithm RANK. Given an n-feasible sequence $a_1a_2 \cdots a_k$ whose l-sequence is $l_0l_1l_2 \cdots l_r$, clearly k is $O(n)$ and r is $O(\log n)$. Statement (3) is executed at most $(n+1)/6$ times for the first execution of the outer **for** loop, at most $(n+1)/12$ times for the second execution of the outer **for** loop and so on, i.e., it is executed a maximum of $(n+1)/6+(n+1)/12+\cdots$ times, i.e., $O(n)$ times. Statement (4) takes $O(n)$ time. We can see then that the entire ranking algorithm runs in linear time.

The unranking process is essentially the reverse of the ranking process. Let us suppose we wish to determine the 231st tree with 19 keys. We shall first determine the number of nodes at the lowest internal level. Since,

$$231 > w(20,7) * W(7) = 7 \times 3 = 21$$

and

$$231 \leqq w(20,7) * W(7) + w(20,8) * W(8) = 21 + 280 = 301$$

that number must be 8. Also, since,

$$\left\lfloor \frac{231-21}{w(20,8)} \right\rfloor = \left\lfloor \frac{210}{4} \right\rfloor = 52,$$

we know that this level corresponds to the 53rd permutation of 4 2's and 4 3's. This happens to be 32332232. We now have a prefix of the 19-feasible sequence that we are looking for, and we continue the search by working for the 231–21-208 = 2nd 7-feasible sequence.

We now present the algorithm UNRANK.

ALGORITHM UNRANK

(**Comment**: This algorithm computes an n-feasible sequence given the rank of the n-feasible sequence. It computes the number q of internal nodes at the lowest level and then the permutation of the number of sons of these q internal nodes. The process repeats in a level-by-level manner. Initially we have $p = n+1$ leaves. The number q of internal nodes is obtained by $\sum_{q'=\lceil p/3 \rceil}^{q} w(p,q') * W(q') \leqq \text{rank}$. After q has been determined it computes the number r of 2's and the rank u of the permutation of the number of sons of these q internal nodes. With the values q, r and u it invokes the function UNRANKPERM (q, r, u) to obtain the permutation of r 2's and $(q-r)$ 3's whose rank is u. It then updates the rank (whose value is stored in variable sum) and the number p of leaves which is set to be equal to q. The process repeats until the root of tree is reached; i.e., $p = 1$.)

Input: (n, rank) where rank is the given rank of an n-feasible sequence.

Output: The corresponding n-feasible sequence $a_1a_2 \cdots a_k$.

Method:

```
    begin
(1)       sum ← rank; p ← n + 1; s ← 1;
          while p > 1 do
            begin
(2)           q ← ⌈p/3⌉; t ← w(p, q) * W(q);
              while sum > t do
                begin
(3)               sum ← sum − t; q ← q + 1;
(4)               t ← w(p, q) * W(q)
                end;
              r ← 3 * q − p; u ← ⌈sum/W(q)⌉;
```

```
(6)         (a_s, a_{s+1}, ···, a_{s+q-1}) ← UNRANKPERM(q, r, u);
(7)         sum ← sum − (u − 1) * W(q);
(8)         p ← q; s ← s + q
          end
  end
```

UNRANKPERM(q, r, u) runs in $O(q)$ time. For details the reader is referred to [3]. The outer **while** loop in UNRANK is executed $O(\log n)$ times and by a reasoning similar to the one used for algorithm RANK, statements (3) and (4) in the inner **while** loop are executed at most $O(n)$ times. Also, statement (6) which is executed $O(\log n)$ times takes a total of $O(n)$ time. Thus the time complexity of algorithm UNRANK is $O(n)$.

Since the unranking algorithm takes time that is proportional to the length of the generated n-feasible sequence, it can be used to generate all n-feasible sequences in time that is proportional to the size of the output.

```
begin
  for i ← 1 until W(n + 1) do
    print (UNRANK(n, i))
end
```

3. Concluding remarks. We have shown that for 2-3 trees with n keys, ranking and unranking can be done in $O(n)$ time after a preprocessing step that takes $O(n^2)$ time and space. Also, all n-feasible sequences can be generated in time proportional to the size of the output.

It may be worthwhile to point out that in our ranking and unranking algorithms, the search graph is never explicitly used. All that we need are the quantities $w(p, q)$ and $W(p)$, and these could have been obtained recursively from (2). But the search graph helps give clarity to the presentation. Also, it could be used to generate all the n-feasible sequences more effectively than the simple generation algorithm given in § 2. Traversal of the search graph allows us to generate the next n-feasible sequence with an expected time of less than $O(n)$ per generated sequence exclusive of the output time. It remains to be seen whether the next n-feasible sequence can be generated in constant expected time as in the case of some other combinatorial objects [2], [8], [9], [12], [13]. Similar ranking and unranking algorithms for B-trees are discussed in [4].

Acknowledgment. The authors are grateful to the referees for their valuable suggestions on improving the presentation of this paper.

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison Wesley, Reading, MA, 1976.

[2] T. BEYER AND S. M. HEDETNIEMI, *Constant time generation of rooted trees*, this Journal, 9 (1980), pp. 706–712.

[3] U. GUPTA, D. T. LEE AND C. K. WONG, *On generating* 2-3 *trees*, Tech. Rep. #80-06-DBM-03, Dept. of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL, 1980.

[4] ———, *Ranking and unranking of B-trees*, Tech. Rep. #81-02-DBM-01, Dept. of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL, 1981.

[5] D. E. KNUTH, *The Art of Computer Programming, Vol.* 3, *Sorting and Searching*, Addison Wesley, Reading, MA, 1973.

[6] D. H. LEHMER, *The machine tools of combinatorics*, in Applied Combinatorial Mathematics, E. F. Bechenbach, ed., John Wiley, New York, 1964, pp. 5–31.

[7] E. M. REINGOLD, J. NIEVERGELT AND N. DEO, *Combinatorial Algorithms—Theory and Practice*, Prentice-Hall, Englewood Cliffs, NJ, 1977.

[8] F. RUSKEY, *Generating t-ary trees lexicographically*, this Journal, 7 (1978), pp. 424–439.

[9] F. RUSKEY AND T. C. HU, *Generating binary trees lexicographically*, this Journal, 6 (1977), pp. 745–758.

[10] A. E. TROJANOWSKI, *Ranking and listing algorithms for k-ary trees*, this Journal, 7 (1978), pp. 492–509.

[11] H. S. WILF, *A unified setting for sequencing, ranking and selection algorithms for combinatorial objects*, Advances in Math., 24 (1977), pp. 281–291.

[12] S. G. WILLIAMSON, *On the ordering, ranking, and random generation of basic combinatorial sets*, in Lecture Notes in Mathematics, 579, Springer-Verlag, Berlin, 1976.

[13] S. ZAKS, *Lexicographic generation of ordered trees*, Theoretical Computer Science, 10 (1980), pp. 63–82.

[14] S. ZAKS AND D. RICHARDS, *Generating trees and other combinatorial objects lexicographically*, this Journal, 8 (1979), pp. 73–81.

SIAM J. COMPUT.
Vol. 11, No. 3, August 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1103-0019 $01.00/0

ON SOME DETERMINISTIC SPACE COMPLEXITY PROBLEMS*

JIA-WEI HONG†

Abstract. In this paper we give a complete problem in DSPACE(n). The problem is whether there exists a cycle in the connected component containing $(0, 0, \cdots, 0)$ in the graph G_p of the zeros of a polynomial P over $GF(2)$ under a suitable natural coding. Hence the deterministic space complexity of this problem is $O(n)$ but not $o(n)$. We give as well several problems for which we can obtain very close upper and lower deterministic space bounds. For example, the deterministic space complexity to determine whether there exists a cycle in the graph of the set of assignments satisfying a Boolean formula is $O(n/\log n)$ but not $o(n/\log^2 n)$.

Key words. space complexity, deterministic space complete problem, lower bounds, cycle-free problem, set of assignments, Boolean expression, polynomial over $GF(2)$

1. A typical theorem. The purpose of this paper is to find some "natural problems" which are complete in DSPACE (n), or at least for which we can obtain very close upper and lower bounds of deterministic space complexity. It is well known that there exists a hardest language in NSPACE (n) [1], but we do not know any complete language in DSPACE (n), except the universal one. In this paper, we will give a "natural problem" which is complete in DSPACE (n). People have obtained several results about lower space bounds, but the bounds apply not only to deterministic Turing machines but also to nondeterministic Turing machines (see [2]–[4]). So the upper bounds are the squares of the lower bounds. In order to obtain close upper and lower space complexity bounds, we have to use some methods that can only be used in a deterministic situation.

Consider the following problem: Let F be a Boolean formula constructed from m variables $x_1, x_2, \cdots, x_m$. The distance of two assignments $Y = (y_1, \cdots, y_m)$ and $Z = (z_1, \cdots, z_m)$ (two m-tuples of zeros and ones) is defined as

$$d(Y, Z) = \sum_{i=1}^{m} |y_i - z_i|,$$

that is, the number of indices at which they differ. Set

$$V_F = \{(x_1, \cdots, x_m) | F(x_1, \cdots, x_m) = 1\},$$

$$E_F = \{(Y, Z) \mid Y, Z \in V_F, d(Y, Z) = 1\}.$$

Given a Boolean formula F, there is a graph $G_F = \{V_F, E_F\}$. We want to determine whether there exists a cycle in G_F, and to determine the space complexity of solving this problem. A typical result is the following.

THEOREM 1. *The deterministic space complexity to determine whether there exists a cycle in graph G_F is $O(n/\log n)$ but not $o(n/\log^2 n)$, where n is the length of the binary expression of F.*

2. Proof of the theorem. The theorem depends on the following

LEMMA. *For every Turing machine M, there exists a Turing machine R, whose input is a binary string $W = w_1 w_2 \cdots w_l$, whose output is a binary coding F^* of a*

* Received by the editors April 30, 1980, and in final form July 15, 1981. Significant portions of this paper are reprinted with permission from "On Some Deterministic Space Complexity Problems" by Hong Jia-wei published in Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing, Copyright 1980, Association for Computing Machinery, Inc.

† Peking Municipal Computing Centre, Peking, China, and Department of Computer Science, University of Toronto, Toronto, Ontario, Canada.

Boolean formula F in t variables, such that

1) *The length of the work tape of R is $o(l)$.*
2) *M accepts W in space l iff there is a cycle in G_F.*
3) *The length of F^* is $O(l \log^2 l)$.*

Using this lemma, we can prove the theorem as follows. Suppose that g is a space-constructable function such that $g(n \log^2 n) = o(n)$. Suppose that there were a Turing machine T which can determine whether there exists a cycle in graph G_F in space $g(n)$. Let M be an arbitrary space linear bounded Turing machine. We construct a Turing machine S as follows:

1) The input of S is $W = w_1 w_2 \cdots w_l$, placed on a read-only tape.
2) Using W as input, simulate R on the work tape; calculating the coding F^* of the formula F, the work space is $o(l)$.
3) Simulate T, using F^* as input, determining whether there exists a cycle in graph G_F.

The work space is $g(n)$, where n is the length of F^*. Therefore, $g(n) \leqq g(cl \log^2 l) \leqq g(cl \log^2 (cl)) = o(cl) = o(l)$ for some constant c. Although the length of F^* is $cl \log^2 l$, we do not store F^* in step 2), but calculate every digit of F^* from the very beginning. Hence the work space is $o(l)$.

Now, S would accept in space $o(l)$ a language that is accepted by M in space l. This is impossible because M is an arbitrary space linear bounded Turing machine (see [5]).

If we take $g(n) = o(n/\log^2 n)$, then $g(n) \log^2 n/n \to 0$ $(n \to \infty)$. Hence, substituting $n \log^2 n$ for n, we obtain

$$\frac{g(n \log^2 n) \log^2 (n \log^2 n)}{n \log^2 n} \to 0 \qquad (n \to \infty).$$

Because of $\log^2 (n \log^2 n) \geqq \log^2 n$, we must have $g(n \log^2 n)/n \to 0$ $(n \to \infty)$; that is, $g(n \log^2 n) = o(n)$. This completes the proof of the lower bound.

Suppose G is a graph and the number of vertices of G is not more than 2^r, so every vertex in G has a coding whose length is not more than r. If vertex Y has k neighbors, we can define which one is its first neighbor, which is the second and which is the last, according to the coding's order. We define a Cycle-Search-Procedure as follows: When we come to a vertex Y, if the vertex Y_1 we just come from is the ith neighbor of Y, then we go to the $(i+1)$th neighbor of Y; if Y_1 is Y's last neighbor, then we go to the first neighbor of Y. Using two pebbles, we can do a Cycle-Search on an undirected graph. We start from vertex Y, and go to its first neighbor Y_1. Then, if Y is Y_1's ith neighbor, we go to Y_1's $(i+1)$th neighbor, and so forth. Now, if the connected area containing Y is a tree, then 1) we must eventually come back to Y from its last neighbor and begin another period, and 2) in one period, from every vertex we visit, we go to its every neighbor exactly once according to a cyclic order. And, it is not difficult to prove that these two conditions are sufficient to guarantee that the connected area is a tree.

The algorithm is from a Chinese story. There are three Chinese, the father, the son and the grandfather, in a large maze. They have only a clock, but want to determine whether there is a cycle in the maze. Standing at a point, the father lets his son do the Cycle-Search. If his son always comes back to the point from the direction the son goes, the father knows that there is no problem with this point. Then he will go one step ahead according to the Cycle-Search and let his son do the whole Cycle-Search again. Therefore the son is very busy. The grandfather is too old to move. He just

watches the clock. If his son and grandson have spent too much time, he knows there must be a cycle. To simulate these three Chinese, logarithmic space is enough.

In the following program, P_1 and P_2 are pairs of pebbles; CSP(P) means to put the pair P of pebbles a step ahead according to Cycle-Search-Procedure. This program can determine whether there exists a cycle in graph G.

```
for every X in G do {every time change X according to the order of its coding}
  begin put P1 at X; {use P1 to check condition 2) at different places}
    while P1 has not come back to X from its last neighbor do
    begin put P2 at X;
      0 -> j;
    while P2 has not come back to X from its last neighbor do
        begin j+1 -> j;
          if j > 2^r then go to Cyc; {there must be a cycle}
        if P2 coincides with P1, then check condition 2) if is not satisfied then
            go to Cyc; {use another pebble to do so}
        CSP(P2)
        end
        CSP(P1)
    end
  end stop; (no cycle)
Cyc: stop (cycle);
```

For the first line, one pebble is enough. In order to check condition 2), another pebble will do. Therefore, we need 6 pebbles altogether, and each pebble uses space $\log 2^r = r$. For counting, we need space r. Hence the total work space is $O(r)$. As to our problem, the length of the binary expression of the formula is n, so we have $r \log r \leqq n$, $r \log n/n = (r \log r/n)(\log n/\log r) = O(1)$. Therefore $r = O(n/\log n)$; this is the upper space bound.

3. The construction of the admissable set. Without loss of generality, we can suppose that the input of the linear bounded Turing machine M is $W = w_1 \cdots w_l$, that the head will never move to the outside, that the head moves either right or left in every step and that there is only one acceptable instantaneous description I_t. Suppose δ is the transition function of M and $I = a_1 a_2 \cdots q a_i \cdots a_l$ is an instaneous description (I.D.) saying that machine M is in the state q, scanning the ith square. If $\delta(q, a_i) = (q', a'_i, -1)$, we should substitute $q' a_{i-1} a'_i$ for $a_{i-1} q a_i$ in I; if $\delta(q, a_i) = (q', a'_i, 1)$, we should substitute $a_{i-1} a'_i q'$ for $a_{i-1} q a_i$.

In the following, if the number of 1's in a coding is even, we say the coding is even; otherwise the coding is odd. Suppose that the number of different states and the number of different letters in the alphabet are all less than or equal to 2^{s-4}, so we can use an s-digit binary number to express the states and letters such that the last three digits of the letter's coding are 000 and the last three digits of the state's coding are 111, and every coding is even. These three last digits are called character codings, and the first $s-3$ digits are called information codings. To obtain the coding of the I.D. I, for every letter and state in I, we substitute its coding, and then add d zeros at the end (called complement coding), where d is the number of different substitutions in M. The length of the coding I^* of I is $(l+1)s + d$.

For example, suppose there is a substitution $aqb \to q'ab'$, and the codings of a, b, b', q', q are 011000, 000000, 110000, 100111, 010111 respectively. Then the

substitution becomes

$$\begin{vmatrix} 011000 & 010111 & 000000 \\ 100111 & 011000 & 110000 \end{vmatrix}.$$

In G_F, this substitution is realized by the "path" in Table 1.

TABLE 1

				S_1	S_2	S_3
L_{ij1}	011000	010111	000000	0	0	0
L_{ij2}	011000	010111	000000	1	0	0
L_{ij3}	111000	010111	000000	1	0	0
L_{ij4}	101000	010111	000000	1	0	0
L_{ij5}	100000	010111	000000	1	0	0
L_{ij6}	100000	011111	000000	1	0	0
L_{ij7}	100000	011111	100000	1	0	0
L_{ij8}	100000	011111	110000	1	0	0
L_{ij9}	100001	011111	110000	1	0	0
L_{ij10}	100011	011111	110000	1	0	0
L_{ij11}	100111	011111	110000	1	0	0
L_{ij12}	100111	011110	110000	1	0	0
L_{ij13}	100111	011110	110000	0	0	0
L_{ij14}	100111	011100	110000	0	0	0

In this table, two successive lines are different from each other only in one digit. It realizes the substitution by the following steps:

1) There are d digits $(s_1, s_2, \cdots, s_d)$ corresponding to the substitutions in machine M. Suppose this is the jth substitution; then we change the value of s_j from 0 to 1, so that we can separate this path from the paths corresponding to other substitutions. In this example we have supposed that $d = 3$ and $j = 1$.

2) From left to right, change the digits one by one. Leave the character codings unchanged. In this example it takes 6 lines to accomplish this procedure.

3) Change the character codings of q' from 000 to 001 to 011 to 111, and then change the original character coding of q from 111 to 110.

4) Abolish the 1 at the position s_j.

5) Change the character coding of q from 110 to 100, so the last line can be connected with the first line of the next substitution. Notice the position changed is x_{is-1}.

According to this procedure, the string is changed from an even coding to an odd coding, then to an even coding again, $\cdots$ and so forth. Generally speaking, there are k lines in the table, k is even and is a function of j. To simplify the discussion below, we suppose k is a constant.

There are $(l+1)s+d$ digits in the coding of an instantaneous description. We use variables $x_1, x_2, \cdots x_{(l+1)s}$ and $s_1, \cdots, s_d$ to express their values. Suppose the head is scanning the ith square. Then the columns of the table correspond to $x_{(i-2)s+1}, \cdots, x_{(i+1)s}, s_1, \cdots, s_d$. Hence for every position i and every substitution j there is a table. For every line L_{iju}, we construct a conjunction C'_{iju} as follows: it is a conjunction of $3s+d$ variables $x_{(i-2)s+1}, \cdots, x_{(i+1)s}, s_1, \cdots, s_d$ (later we call them the group i); but if the value of the variable in that line L_{iju} is 0, we should put a negation sign on it. In this example, we have $C'_{311} = \bar{x}_7 x_8 x_9 \bar{x}_{10} \bar{x}_{11} \bar{x}_{12} \bar{x}_{13} x_{14} \bar{x}_{15} x_{16} x_{17} x_{18} \bar{x}_{19} \bar{x}_{20} \bar{x}_{21} \bar{x}_{22} \bar{x}_{23} \bar{x}_{24} \bar{S}_1 \bar{S}_2 \bar{S}_3$. We define $C_{iju} = \bar{x}_{(i-2)s-2} \bar{x}_{(i-2)s-1} \bar{x}_{(i-2)s} C'_{iju} \bar{x}_{(i+2)s-2} \bar{x}_{(i+2)s-1} \bar{x}_{(i+2)s}$. These 6 variables correspond to the $(i-2)$ and $(i+2)$th

character codings. Set

$$D_u = \begin{cases} x_1 \oplus \cdots \oplus x_{(l+1)s} \oplus s_1 \oplus \cdots \oplus s_d & \text{if } u \text{ is odd,} \\ x_1 \oplus \cdots \oplus x_{(l+1)s} \oplus s_1 \oplus \cdots \oplus s_d \oplus 1 & \text{if } u \text{ is even,} \end{cases}$$

where $\oplus$ means exclusive or. $D_u = 1$ means the coding is even, iff u is even. Set

$$D_{iju} = C_{iju} D_u,$$

$$D = \bigcup_{iju} D_{iju} = \bigcup_u D_u\Big(\bigcup_{ij} C_{iju}\Big) = \Big(D_1\Big(\bigcup_{\substack{ij \\ 2\nmid u}} C_{iju}\Big)\Big) \cup \Big(D_2\Big(\bigcup_{\substack{ij \\ 2|u}} C_{iju}\Big)\Big).$$

The binary length of formula D is $O(l \log l)$.

Remark. Here we use three operations $\vee$, $\wedge$, $\oplus$ to construct the formula. We cannot express D_u in length $O(l \log l)$ if we only use the usual operations $\neg$, $\vee$, $\wedge$. Anyway, it is easy to see that in this procedure at most one coding of a group is odd and all the others are even. Hence when $\bigoplus_{i=1}^{(l+1)s} x_i$ is even, we can use

$$E = \bigcap_{i=1}^{l} \Big(\neg \bigoplus_{h=1-s}^{2s} x_{is+h}\Big)$$

instead of it. The length of E is $O(l \log l)$. When $\bigoplus_{i=1}^{(l+1)s} x_i$ is odd, we can use

$$E' = \bigcup_{g=1}^{l} \Big(\bigcap_{\substack{i=1 \\ i \neq g}}^{l} \Big(\neg \bigoplus_{h=1-s}^{2s} x_{is+h}\Big) \cap \Big(\bigoplus_{h=1-s}^{2s} x_{gs+h}\Big)\Big)$$

instead. Its binary length is $O(l^2 \log l)$. Here s is a constant. Using "divide and conquer" (see [4, Chapt. 2]), after collecting common factors, its length can be reduced to $O(l \log^2 l)$.

In order to get rid of meaningless binary strings, we define an admissible set A. A binary string Y belongs to A iff

1) There is at most one complement coding digit which equals 1, and

2) $Y \in D_{iju}$ for some $u = 1, 2, \cdots, k-6$ and there is only one character coding which equals 111 (all the others equal 000); or $Y \in D_{ijk-5}$ ($Y \in D_{ijk-4}$, $Y \in D_{ijk-3}$, $Y \in D_{ijk-2} \cup D_{ijk-1}$, $Y \in D_{ijk}$ respectively) and there is only one pair of successive character codings which equal 001 and 111, (011 and 111, 111 and 111, 111 and 110, 111 and 100, respectively), and the others are equal to 000.

We can express set A with a Boolean formula whose binary length is $O(l \log^2 l)$. For example, the sentence, "There is only one character coding which equals 111, all the others equal 000", can be expressed with a formula of binary length $O(l^2 \log l)$. Using "divide and conquer", after collecting common factors, its length can be reduced to $O(l \log^2 l)$.

4. The proof of the lemma. Because M is deterministic, for every admissible I.D. (i.e., whose coding is in A) I_1, we can use at most one substitution of M such that $I_1 \to I_2$, and this defines a directed path in set A. All these paths make set A a directed graph $\vec{A}$.

More precisely, we say $\alpha \to \beta$ iff 1) there exists D_{iju} such that $\alpha \in D_{iju}$, $\beta \in D_{iju+1}$, α and β are adjacent in A; or 2) there exists D_{ijk} such that $\alpha \in D_{ijk} \cap A$, and β is obtained by changing the value of the position x_{is-2} of α from 1 to 0.

This directed graph $\vec{A}$ satisfies

1) For every point in $\vec{A}$, the fan-out number is at most one. This is because M is deterministic and all the paths realizing these substitutions are separated by the design of the set A.

We are going to prove this statement. (We suggest the reader ignore this paragraph at the first reading.) Suppose $\alpha \to \beta_1$, $\alpha \to \beta_2$, then there exist D_{iju} and D_{abc} satisfying the above condition. If $u \in \{2, 3, \cdots, k-2\}$, then $j = b$. Hence $i = a$ and $u = c$. In this case, D_{iju+1} is different from D_{iju} in one position, and β_1, β_2 are different from α at one position, so β_1 and β_2 are different from α at the same position; therefore $\beta_1 = \beta_2$. If $u = 1$, then $i = a$. In this case, because the Turing machine is deterministic, there is only one substitution we can use. Hence $\beta_1 = \beta_2$. If $u = k-1$, then $i = a$. β_1 and β_2 are different from α at position x_{is-1}, therefore $\beta_1 = \beta_2$. If $u = k$, then $i = a$, $c = u = k$. There exists D_{ijk} such that $\alpha \in A \cap D_{ijk}$ and β_1, β_2 are obtained by changing the value of the position x_{is-2} from 1 to 0; hence $\beta_1 = \beta_2$.

2) An I.D. I_1 leads to an I.D. I_2 iff there is a path from the coding of I_1 to the coding of I_2 in $\vec{A}$.

Now, if $\alpha, \beta \in \vec{A}$ and there is an arrow from α to β, then we write $\alpha \to \beta$ or $\beta \leftarrow \alpha$. In the same time, A is a set of codings. Two codings are adjacent if they are different in only one digit. Hence A is a undirected graph. What is the relation between A and $\vec{A}$? We have

PROPOSITION 1. *α and β are adjacent in A iff $\alpha \to \beta$ or $\beta \to \alpha$ in $\vec{A}$.*

Proof. In fact, we are going to prove the following stronger proposition: If $\alpha \in A$, $\beta \in D$, $d(\alpha, \beta) = 1$, then $\beta \in A$ and $\alpha \to \beta$ or $\beta \to \alpha$ in $\vec{A}$. Hence A is an isolated part of graph D. (We suggest the reader ignore this proof at the first reading.)

Suppose $\alpha \in A$, $\beta \in D$, $d(\alpha, \beta) = 1$, $\alpha \in D_{iju}$, $\beta \in D_{abc}$. If $|i-a| \geqq 2$, then $d(\alpha, \beta) \geqq 2$. Therefore, we need only consider the following two cases.

Case 1. $i = a$. Because one of α and β is even and the other is odd, $u \neq c$. Therefore, they are different at just one position in the group i.

If $j = b$, then $d(\alpha, \beta) \geqq \min\{|u-c|, 2\}$ by the structure of the table. Hence $|u-c| = 1$ and $\alpha \to \beta$ or $\beta \to \alpha$, $\beta \in A$.

Now suppose $j \neq b$. If u and c both belong to $\{2, 3, \cdots, k-2\}$, then α and β are different at position s_j and s_b. This is impossible, so we can assume $u = 1, k-1, k$. (If $c = 1, k-1, k$, we can treat it the same way.)

If $c \in \{2, 3, \cdots, k-2\}$, α and β are different at s_h ($h = 1, 2, \cdots, d$), so they are the same at other positions. Hence $u \neq k$. Therefore $u = k-1$ or 1. If $u = k-1$, we must have $c = k-2$ and $\beta \to \alpha$, $\beta \in A$. If $u = 1$, then because α and β are the same at all other positions, $\beta \in D_{ib2}$. Because of $\alpha \in D_{ij1} = D_{ib1}$, $\alpha \to \beta$ and $\beta \in A$.

Now suppose both u and c belong to $\{1, k-1, k\}$. In this case, we must have $u = k$, $c = k-1$, or $c = k$, $u = k-1$. Hence $\alpha \to \beta$ or $\beta \to \alpha$. $\beta \in A$.

Case 2. $|i-a| = 1$. Suppose $i = a+1$.

Subcase 1. $u, c \in \{2, 3, \cdots, k-2\}$. We must have $b = j$. That means the corresponding substitutions are the same; especially, the heads move in the same direction (say left). Then at the positions of the ith character coding, α has 2 1's at least, but β has none. This is impossible.

Subcase 2. $u \in \{1, k-1, k\}$, $c \in \{2, 3, \cdots, k-2\}$. Then α and β should be the same at the position x's. Compare the pairs of two successive character codings (the ith and the ath) of α and β. They cannot be the same.

Subcase 3. $u \in \{2, 3, \cdots, k-2\}$, $c \in \{1, k-1, k\}$. This subcase is similar to subcase 2.

Subcase 4. $u, c \in \{1, k-1, k\}$. Compare the pairs of two successive character codings (the ith and ath) of α and β. They have at most one position different. We must have $u, c \neq k-1$. Hence $u = k$, $c = 1$, $\alpha \to \beta$ or $c = k$, $u = 1$, $\beta \to \alpha$, $\beta \in A$.

PROPOSITION 2. *Suppose G is an undirected graph. If we can define a direction on every edge of G such that the fan-out number of any point of the directed graph $\vec{G}$*

is at most one, and if β is a terminal point of $\vec{G}$, $\alpha \in \vec{G}$, then α is connected with β in G iff there is a directed path from α to β in $\vec{G}$. Furthermore, there is a cycle in G iff there is a cycle in $\vec{G}$.

Proof. If α is connected with β, then there is an undirected chain from α to β; $\alpha = \alpha_1, \alpha_2, \cdots, \alpha_n = \beta$. Because β is a terminal point, we must have $\alpha_{n-1} \to \alpha_n$. If there were a α_i such that $\alpha_{i-1} \leftarrow \alpha_i$, then we can suppose i is the maximum number such that $\alpha_{i-1} \leftarrow \alpha_i$. Now we would have both $\alpha_{i-1} \leftarrow \alpha_i$ and $\alpha_i \to \alpha_{i+1}$. The fan-out number of α_i is 2 at least. The second part can be proved in the same way.

For every linear bounded Turing machine M, we can construct another linear bounded Turing machine M_1, which simulates M on the one hand and counts the number of steps simultaneously on the other hand. If the number of steps exceeds the number of states that M and its tape can express, then M_1 refuses the input and stops. With a little skill, the reader can construct M_1 such that no matter which admissible instantaneous description it starts from (this I.D. may never be reached, if M_1 starts from $I_0 = q_0 w_1 \cdots w_l$), M_1 will stop eventually. Therefore, we can assume that M itself has this property.

PROPOSITION 3. *There is no cycle in graph A.*

Proof. If there is a cycle in A, then there is a cycle in the directed graph $\vec{A}$ by Proposition 2. Hence there is a cycle in the computation of M, which is impossible.

Without loss of generality, we can suppose there is a unique accepting I.D. I_t, and $I_0 = q_0 w_1 \cdots w_l$ is the initial I.D. Let I_0^* and I_t^* be the codings of I_0 and I_t. From the discussion above, we obtain

PROPOSITION 4. *The linear bounded Turing machine M accepts input $W = w_1 \cdots w_l$ iff I_0^* is connected with I_t^* in the graph A.*

Now, we should construct a path connecting I_0^* and I_t^*. Instead, we construct a path connecting I_0^* with $(1, 1, \cdots, 1)$ and another path connecting I_t^* with $(1, 1, \cdots, 1)$. We use $C_1 = x_1 x_2 \cdots x_{(l+1)s} s_1 \cdots s_d$ to express point $(1, 1, \cdots, 1)$. In the same way, suppose the conjunctions to express I_0^* and I_t^* are $C(I_0^*)$ and $C(I_t^*)$, respectively.

First, we design a path G_1 connecting I_0^* and $(1, 1, \cdots, 1)$ satisfying that there is no cycle in G_1 and that the number of total variables in G_1 is linear in l.

We use the following logical symbol $\geq$ to express "not less than": $x \geq y$ means "not x is false and y is true".

$$x \geq y \equiv 1 \oplus (1 \oplus x) y.$$

Assume $y_1, y_2, \cdots, y_e$ are the variables in $\{x_1, \cdots, x_{(l+1)s}, s_1, \cdots, s_d\}$ which have a negation sign in $C(I_0^*)$; $z_1, z_2, \cdots, z_f$ are those without a negation sign in $C(I_0^*)$. Then

$$G_1 = z_1 z_2 \cdots z_f \bigcap_{i=1}^{e-1} (y_i \geq y_{i+1}).$$

In the set G_1, every point has the property that $y_1 \geq y_2 \geq y_3 \geq \cdots \geq y_e$. So there is no cycle in G_1. The binary length of G_1 is $O(l \log l)$. Notice that the y_i's are x_i's whose value is 0 in I_0^*, and nearly every character coding in I_0^* is 000. Hence the distance between the positions of y_i and y_{i+1} in the list $\{x_1, x_2, \cdots, x_{(l+1)s}\}$ is not more than a constant. We will use this property later.

Using the same technique we can design another path G_2 connecting I_t^* with $(1, 1, \cdots, 1)$. Set

$$F = A \bar{t}_1 \bar{t}_2 \bar{t}_3 \bar{t}_4 \cup C(I_0^*) t_1 \bar{t}_2 \bar{t}_3 \bar{t}_4 \cup G_1 t_1 t_2 \bar{t}_3 \bar{t}_4$$
$$\cup C_1 (t_1 t_2 t_3 \bar{t}_4 \cup t_1 t_2 t_3 t_4 \cup \bar{t}_1 t_2 t_3 t_4) \cup G_2 \bar{t}_1 \bar{t}_2 t_3 t_4 \cup C(I_t^*) \bar{t}_1 \bar{t}_2 \bar{t}_3 t_4.$$

We have

PROPOSITION 5. *M accepts $W = w_1 \cdots w_l$, iff there is a cycle in the graph G_F.*

It is easy to construct the coding F^* of F in space $o(l)$. This completes the proof of the lemma.

5. The main results. In this section, we want to improve the result. The key is to reduce the length of F^*, which is $O(l \log^2 l)$ in the lemma. The square on the log l comes from two places.

One of them is that if we use only the logical operations $\vee$, $\wedge$, $\neg$, we cannot express $x_1 \oplus x_2 \oplus \cdots \oplus x_l$ in a short form. Therefore we have to use the technique mentioned in the remark. The length becomes $O(l \log^2 l)$. But, if we use the formulae

$$\bar{x} = 1 \oplus x,$$

$$x_1 \vee x_2 \vee \cdots \vee x_n = (x_1 \oplus 1)(x_2 \oplus 1) \cdots (x_n \oplus 1) \oplus 1,$$

we can express F as a polynomial over $GF(2)$. Then the length of D is only $O(l \log l)$.

Another trouble comes from the expression of A. But in Proposition 1 we proved that A is an isolated part of D. Therefore there is a cycle in the graph F iff there is a cycle in the connected component containing $(1, 1, \cdots, 1)$ (notice that there are many cycles in $D \backslash A$) in the graph

$$F_1 = D\bar{t}_1\bar{t}_2\bar{t}_3\bar{t}_4 \cup C(I_0^*)t_1\bar{t}_2\bar{t}_3\bar{t}_4 \cup G_1 t_1 t_2 \bar{t}_3 \bar{t}_4$$
$$\cup C_1(t_1t_2t_3\bar{t}_4 \cup t_1t_2t_3t_4 \cup \bar{t}_1t_2t_3t_4) \cup G_2\bar{t}_1\bar{t}_2t_3t_4 \cup C(I_t^*)\bar{t}_1\bar{t}_2\bar{t}_3t_4,$$

whose binary length is $O(l \log l)$ as a polynomial over $GF(2)$. Therefore, we can remove the square of the $\log l$ from the lower bound. As to the upper bound, we need only to test whether there is a cycle in the connected component containing $(1, 1, \cdots, 1)$, so it is even easier. We exchange 0 and 1 in the coding to get the following

THEOREM 2. *The deterministic space complexity to determine whether there is a cycle in the connected component containing $(0, 0, \cdots, 0)$ in the graph G_p of the zeros of a polynomial P over $GF(2)$ is $O(n/\log n)$ but not $o(n/\log n)$, where n is the binary length of the polynomial P.*

Because of the definition of D, we have

$$D = D_1\Big(\bigcup_i \Big(\bigcup_{\substack{j \\ 2\nmid u}} C_{iju}\Big)\Big) \cup D_2\Big(\bigcup_i \Big(\bigcup_{\substack{j \\ 2\mid u}} C_{iju}\Big)\Big).$$

So the formula F_1 consists of nine "main" parts, that is,

$$D_1, D_2, \bigcup_i \Big(\bigcup_{\substack{j \\ 2\nmid u}} C_{iju}\Big), \quad \bigcup_i \Big(\bigcup_{\substack{j \\ 2\mid u}} C_{iju}\Big), \quad C(I_0^*), \quad C(I_t^*), \quad C_1, \quad G_1, \quad G_2.$$

Only $C(I_0^*)$ and G_1 depend on the context of the input $W = w_1 \cdots w_l$; the other parts only depend on the length l. To output F_1, machine R needs only 9 reversals of its input head. At every reversal, R outputs one part. Machine R need only remember the subscript of the variable output at that time. Therefore, the work space is $O(\log l)$.

Although Theorem 2 is better, we still cannot get a complete problem in DSPACE(n), because the length of the binary expression of P is $O(l \log l)$. We want to reduce the length to linear. Notice that the polynomial has to reflect every bit of the input, so there are l variables in P at least, and the length of the coding of every variable is $\log l$ at least. What can we do? We have to invent a new kind of coding.

Imagine that there is a list of infinite many variables: $L=\{x_1, x_2, \cdots\}$. We use the following three kinds of words to express a string of variables in L.

Δ: the first variable x_1 in L. After the use of this symbol, every variable in L becomes a "new" variable again.

$\Delta 0$: a "new" variable; after the use of it, this variable becomes a "used" one.

Δn: the same variable as its nth left neighbor.

For example, the coding

$$\Delta(\Delta 0 \oplus \Delta 2)(\Delta 2 \oplus \Delta 2)(\Delta \oplus \Delta 0) \oplus \Delta 0 \oplus \Delta 0 \oplus \Delta 0 \oplus \Delta 2$$

has the meaning

$$x_1(x_2 \oplus x_1)(x_2 \oplus x_1)(x_1 \oplus x_2) \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_4.$$

After assuming $L=\{s_1, \cdots, s_d, x_1, \cdots, x_{(l+1)s}, t_1, t_2, t_3, t_4\}$ and rearranging the variables in F_1 suitably, it is easy to see that, in every part of the expression F_1,

1) the leftmost variable is s_1,

2) the distance between two adjacent occurrences of the same variable is not more than a fixed constant.

Therefore, we have a linear length expression of F_1 in this Δ coding system (use 9 Δ's). Furthermore, when we use machine R to output the coding of F_1, we need not remember the subscript used. We only need a finite memory. Now the machine R can be a finite automaton, its input head does only nine reversals and the whole time is linear in l. We obtain

THEOREM 3. *Under the Δ coding system, the problem whether there exists a cycle in the connected component containing $(0, 0, \cdots, 0)$ in the graph G_p of the zeros of a polynomial over $GF(2)$ is complete in* DSPACE(n) *in the sense that*

1) *This problem is in* DSPACE(n).
2) *Every problem in* DSPACE(n) *can be reduced to this problem using an oblivious Turing machine within a constant space, linear time and nine reversals of the input head.*

Therefore, the deterministic space complexity is $O(n)$ but not $o(n)$.

The Δ-coding system seems a little bit strange, because we are not used to it. We have seen that if we use ordinary coding in the above problem, we have to use $\log n$ space and $n \log n$ time, and we cannot get a complete problem. That means that the Δ coding system is even more "natural" than the ordinary one.

6. Further discussion. If we discuss the connectivity of two points in a graph, we can get the following lower bounds, but we fail to obtain a close upper bound. The best known upper bounds are the squares of the lower bounds [7].

THEOREM 4. *The deterministic space complexity to determine whether two points $(0, 0, \cdots, 0)$ and $(1, 1, \cdots, 1)$ are connected in a graph G_p of the zeros of a polynomial P over $GF(2)$ is not $o(n/\log n)$ (in the ordinary coding) and is not $o(n)$ (in the Δ coding system).*

In the situation of DSPACE$(\log n)$, we can use an adjacency matrix to express an undirected graph. Using the same algorithm, we can show that the cycle-free problem (CFP) for undirected graphs belongs to DSPACE$(\log n)$. The lower bound is not very interesting, because $\log n$ space is needed to determine whether a coding is legitimate. Regardless, using the same method, under Cook's definition and formation [8], we can prove that the CFP is log depth complete for DSPACE$(\log n)$. We know Depth$(\log n) \subseteq$ DSPACE$(\log n)$ by a theorem of Borodin [9], but we do not know whether Depth$(\log n)$ = DSPACE$(\log n)$. Now it holds iff CFP $\in$ Depth$(\log n)$.

At the end of this paper, we discuss some higher space complexity problems briefly. Let $f(n) \geqq n$ be a space constructable function. We insert several pairs of [,] and $\langle , \rangle$ into a Δ coding string. These pairs cannot be nested inside each other, every pair of [,] can only contain one variable $\Delta 0$ and there exists at least one variable symbol Δ between every two adjacent pairs of [,]. In this coding system, we only use the following 11 symbols: Δ, $\bar{\Delta}$, $\oplus$, 0, 1, (,), $\langle , \rangle$, [,]. Four binary bits are enough to encode them. Suppose the total number of characters inside all the pairs $\langle , \rangle$ is l; the meaning of [,] is that the context inside each [,] should be repeated $f(l)$ times. For example, if $f(1)=4$, we have

$$\Delta[\oplus\bar{\Delta}0]\oplus\Delta[\Delta 0]\oplus\langle\Delta\rangle = \Delta\oplus\bar{\Delta}0\oplus\bar{\Delta}0\oplus\bar{\Delta}0\oplus\bar{\Delta}0\oplus\Delta\Delta 0\Delta 0\Delta 0\Delta 0\oplus\Delta$$

$$= x_1\oplus\bar{x}_2\oplus\bar{x}_3\oplus\bar{x}_4\oplus\bar{x}_5\oplus x_1x_2x_3x_4x_5\oplus x_1.$$

Because the length of Δ is 1, $f(1)=4$, the context inside [,] should be repeated 4 times. We call this the $\Delta(f)$ coding system.

Suppose

$$\alpha_i \in \{\Delta, \bar{\Delta}, 0, 1, (\,,), [\,,], \oplus\}^{\dagger}, \qquad \beta_i \in \{\Delta, \bar{\Delta}, 0, 1, (\,,), \oplus\}^{\dagger}.$$

We call the word $\alpha_1\langle\beta_1\rangle\alpha_2\langle\beta_2\rangle\cdots\alpha_n\langle\beta_n\rangle$ the $\alpha-\beta$ problem: whether there exists a cycle in the connected component containing $(0, 0, \cdots, 0)$ in the graph G_p of the zeros of P, whose $\Delta(f)$ coding is $\alpha_1\langle\beta_1\rangle\alpha_2\langle\beta_2\rangle\cdots\alpha_n\langle\beta_n\rangle$.

Notice that if we write any "main" part of F_1 in the Δ coding system, the same segment will repeat again and again. Therefore, if we use the $\Delta(f)$ coding system, except for $C(I_0^*)$ and G_1, all the main parts will become fixed words. If $f(n)$ is the space used, no matter how big $f(n)$ is, or how long the polynomial P is, we can always get a short coding of P linear in n. With this in mind, we can prove

PROPOSITION 6. *Suppose $f(n) \geqq n$ is a space constructable function, M is a Turing machine, $W = w_1w_2\cdots w_n$ is the input. Then there exist words α_1, α_2, α_3, β_1, β_2, β_3 such that*

1) *$\alpha_i \in \{\Delta, \bar{\Delta}, 0, 1, (\,,), [\,,], \oplus\}^*$, α_i only depend on M.*
2) *$\beta_i \in \{\Delta, \bar{\Delta}, 0, 1, (\,,), \oplus\}^*$, β_i only depend on W, and can be obtained by an automaton with input W, $|\beta_i| \leqq c|W|$, c is a constant.*
3) *$\alpha_1\langle\beta_1\rangle\alpha_2\langle\beta_2\rangle\alpha_3\langle\beta_3\rangle$ is a $\Delta(f)$ coding of a polynomial P.*
4) *M accepts W in DSPACE$(f(n))$ iff there exists a cycle in the connected component containing $(0, 0, \cdots, 0)$ in the graph G_p of the zeros of P.*

THEOREM 5. *Suppose $f(n)$ is space constructable such that $f(m+n) \geqq f(m)+f(n)$. Then under the $\Delta(f)$ coding system, the α–β problem is complete in the following sense:*

1) *The α–β problem belongs to DSPACE$(f(n))$.*
2) *Every problem in DSPACE$(f(n))$ can be reduced to an α–β problem of linear length by a finite automaton within 3 reversals.*

Proof. 1) Suppose the binary length of $\alpha_1\langle\beta_1\rangle\alpha_2\langle\beta_2\rangle\cdots\alpha_i\langle\beta_i\rangle$ is n. $l = |\beta_1|+|\beta_2|+\cdots+|\beta_i| \leqq n$. There are at most $n+f(l) \leqq n+f(n) \leqq 2f(n)$ different variables in the polynomial P whose $\Delta(f)$ coding is $\alpha_1\langle\beta_1\rangle\cdots\alpha_i\langle\beta_i\rangle$. We can construct $f(n)$ and calculate the value of P in space $O(f(n))$. Using the Cycle-Search-Procedure, we can determine the α–β problem in space $O(f(n))$.

2) Suppose machine M accepts a language in DSPACE$(f(n))$. According to Proposition 6, there exist α_1, α_2, α_3, β_1, β_2, β_3, satisfying the condition, $|\alpha_1\langle\beta_1\rangle\alpha_2\langle\beta_2\rangle\alpha_3\langle\beta_3\rangle| \leqq 3c|W|+c_1$. We can realize this reduction by a finite automaton within 3 reversals of the input head.

COROLLARY. *Under the $\Delta(n^t)$ coding system, the deterministic space complexity of the α–β problem is $O(n^t)$ but not $o(n^t)$.*

Acknowledgments. The author thanks Professor Hao Wang for his kind help, and Professor S. A. Cook for his kind help, his valuable suggestions and careful examination. He also thanks Professor C. Rackoff for his references and discussion.

REFERENCES

[1] A. R. MEYER, AND L. J. STOCKMEYER, *The equivalence problem for regular expression with squaring requires exponential space*, Proc. 13th IEEE Symposium on Switching and Automata Theory, pp. 125–129.

[2] H. B. HUNT, III, *The equivalence problem for regular expressions with intersection is not polynomial in tape*, TR73-156, Dept. Computer Science, Cornell University, Ithaca, NY, 1973.

[3] L. J. STOCKMEYER, *The complexity of decision problems in automata theory and logic*, Massachusetts Institute of Technology Project MAC, Cambridge, MA, 1974.

[4] J. FERRANTE AND C. W. RACKOFF, *The computational complexity of logical theories*, Lecture Notes in Mathematics 718, Springer-Verlag, New York, 1979.

[5] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1975.

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

[7] S. A. COOK AND C. W. RACKOFF, *Space lower bounds for maze threadability on restricted machines*, this Journal, 9 (1980), pp. 636–652.

[8] S. A. COOK, *Towards a complexity theory of synchronous parallel computation*, presented at Internationales Symposium über Logik und Algorithmik zu Ehren von Professor Ernst Specker, Zurich, Switzerland, 1980.

[9] A. B. BORODIN, *On relating time and space to size and depth*, this Journal, 6 (1977), pp. 733–744.

SIAM J. COMPUT.
Vol. 11, No. 3, August 1982
© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1103-0020 $01.00/0

FINDING AUGMENTED-SET BASES*

VIRGIL GLIGOR† AND DAVID MAIER‡

Abstract. The problem of finding a minimum-cost, augmented-set basis is NP-complete. In this paper we show that this problem is not approximable. That is, if $\mathrm{P} \neq \mathrm{NP}$, then no constants c and d exist so that $A \leqq c\ \mathrm{ASB} + d$, where A is the cost provided by a polynomial-time approximation algorithm and ASB is the optimal cost. We also provide a brief characterization of the cost functions for which this result remains valid. The proof technique used in the augmented-set basis problem is applied directly to other NP-complete problems, such as several graph augmentation and deletion problems, to show that they are also not approximable.

Key words. augmentation problems, set basis, deletion problems, computational complexity, polynomial-time approximation algorithms, NP-complete problems

1. Introduction. Augmentation problems arise naturally in graph theory. Using a framework similar to that proposed in [2], [8], [10] for graph augmentation and deletion, we suggest the following set-augmentation problem:

Given a family of sets $V = \{V1, \cdots, Vp\}$, $Vi \subseteq S$, find an augmented family $\bar{V} = \{\bar{V}1, \cdots, \bar{V}p\}$, where $\bar{V}i = Vi \cup Ai$, $Ai \subseteq S$, such that:

(i) the family of sets $\bar{V}i$ have a set basis of size k, denoted by ASB (V, k),

(ii) the cost of the augmentation $\sum_i (|\bar{V}i| - |Vi|)$ is minimum.

The important property that gives rise to this set-augmentation problem, i.e., "there exists a set basis of size k," is defined as follows: The family of sets $B1, \cdots, Bk$ of S is a *set basis of size k* for the family $V1, \cdots, Vp$ of S if each Vi can be expressed as a union of some Bj's. Some families of sets do not have set bases of certain sizes and, when nontrivial set bases exist, they are not necessarily unique.

The need to find a set basis of a given size (or, equivalently, of the minimum possible size) appears in several important applications of feature extraction, of compiler design, and of data compression [1], [3], [7]. For example, consider the transmission of high-resolution images over a narrow-bandwidth channel. Significant time savings can be achieved if, instead of sending information about each point in the digitized image which would require wide-transmission bandwidth, we transmit only sets of features (i.e., subsets of each image) and their membership functions for various images. A smaller transmission bandwidth would thus be possible. The high-resolution images are reconstituted at the receiving end by combining the features according to their membership function. The membership function can be a 0–1 string of bits that specifies whether a given feature belongs to a given image. The transmission of digitized speech is another example of an application that requires the compact representation of a large number of related sets of data.

However, since not all families of sets have set bases of certain sizes (e.g., the size of the minimum set basis may not be small enough), it may be beneficial to augment various sets of a family so that the family has a sufficiently small set basis. For example, the family $V1 = \{a\}$, $V2 = \{b\}$ and $V3 = \{a, b, c\}$ has a trivial minimum set basis of size three. If we augment set $V2$ by adding element c to it, the augmented sets $\bar{V}1 = V1$, $\bar{V}2 = V2 \cup \{c\}$ and $\bar{V}3 = V3$ have a minimum set basis whose size is two, i.e., $B1 = \bar{V}1$ and $B2 = \bar{V}2$. The cost of augmenting various sets must be minimized in

* Received by the editors March 21, 1980, and in final form December 15, 1981.

† Center for Information Sciences Research, University of Maryland, College Park, Maryland 20742.

‡ Department of Computer Science, State University of New York, Stony Brook, New York 11794.
The work of this author was supported by the National Science Foundation under grant IST 79-18264.

the above sense, otherwise the cost of reconstruction of the original family may become too high. For example, if augmented-set bases are used for data transmission through any channel, they must be accompanied by a membership function that also determines how to reconstruct the original family sets from the augmented-set basis. However, this would be impractical if a bandwidth-limited channel were used, because the representation of the membership function can become quite complex whenever the size of each set and of the entire family is large.

The minimum set basis problem is known to be NP-complete[1] [9]. It is easy to see that the augmented-set basis problem is also NP-complete and that, as with set bases, augmented-set bases are not necessarily unique. In § 2 of this paper we show that the problem of finding a minimum-cost, augmented-set basis of size k for a family of sets is *not approximable.* That is, if $\mathrm{P} \neq \mathrm{NP}$, no constants c and d exist so that $A \leqq c\ \mathrm{ASB} + d$, where A is the cost provided by a polynomial-time approximation algorithm and ASB is the optimal cost. In § 3 we show that this result remains valid when alternate cost functions are considered. In § 4, the proof technique used for the augmented-set basis problem is applied directly to several NP-complete augmentation and deletion problems to show that they are not approximable.

In spite of this somewhat surprising result, we expect that in practice it would be possible to design not only near-optimal approximation algorithms, but also optimal algorithms for particular classes of augmentation and deletion problems.

2. The complexity of the augmented-set basis problem. In this section we define the notions of the minimum disjoint set basis and of the k-segmentation of a family of sets. Then we present an important common property shared by the minimum set basis (denoted by MSB (V) henceforth), by the k-segmentation, and by the augmented-set basis of a family of sets. Using that property, we then prove the main result of this paper.

DEFINITION 1. The minimum disjoint set basis of a family of sets $V = \{V1, \cdots, Vp\}$, $Vi \subseteq S$, denoted by MDSB (V), is the coarsest partition $B1, \cdots, Bk$ of S such that each Vi is the exact union of some Bj's, i.e., $Vi = \{\bigcup Bj \mid Vi \cap Bj \neq \varnothing, i \in [1, p], j \in [1, k]\}$.

It is easy to see that, unlike the MSB (V), the MDSB (V) is unique. Also, the MDSB (V) can be computed in $O(|W|p)$, where $W = \bigcup Vi$, $i = 1, \cdots, p$. (For example, assume that the elements of W are the integers $1, 2, \cdots, |W|$. Then construct a table with the elements of W as row labels and those of V as column labels. Place a "1" in the (i, j) position of the table if element $i \in W$ is in Vj; otherwise place a "0" at (i, j). Radix sort the rows of the table in $O(|W|p)$ time. The sets of the MDSB (V) are those groups of elements of W that label equal rows. Also, note that the worst case requires $|W|n$ time to read the input of the problem. The correctness proof for this algorithm is left to the reader.)

DEFINITION 2. The k-segmentation of a family of sets $V = \{V1, \cdots, Vp\}$, $Vi \subseteq S$, denoted by KSEG (V, k), is an augmented-set basis consisting of k *disjoint* elements called segments.

The k-segmentation problem is introduced in [4] in the context of data base organization. It is also shown in [4] that k-segmentations are not unique and that the problem of finding a k-segmentation of a family of sets is NP-complete. No bounds have been found on how nearly optimal polynomial-time approximation algorithms for this problem can be. However, we have found that typical polynomial-time

[1] For an introduction to the theory of NP-completeness, the reader is referred to [6].

approximation algorithms, such as "greedy" and "local improvement," have approximate-to-optimal cost ratios that cannot be bounded by a constant.

In the following three lemmas we show that the minimum set basis, the augmented-set basis, and the k-segmentation have some optimal solutions MSB (V), ASB (V, k), and KSEG (V, k) respectively, that are refined by the MDSB (V). Hence, both optimal and approximation algorithms need only combine elements of the MDSB (V) to find solutions to each of the above problems.

LEMMA 1. *For any family of sets* $V=\{V1, \cdots, Vp\}$, $Vi \subseteq S$, *there exists a minimum set basis*, MSB (V), *such that* MDSB (V) = MDSB [MSB (V)].

Proof. First we must prove that for any disjoint set basis of $V1, \cdots, Vp$, DSB (V), there exists a MSB (V) such that DSB (V) = DSB [MSB (V)]. We know that any disjoint set basis of V refines the minimum disjoint set basis of V. Let us choose DSB $(V)=\{D1, D2, \cdots, Dp\}$ and let MSB $(V)=\{B1, B2, \cdots, Bk\}$. We can show that any MSB (V) can be modified so that DSB (V) = DSB [MSB (V)]. All we really need to show is that if $Di \cap Bj \neq \emptyset$, then $Di \subseteq Bj$. (It is then quite easy to show that every Bj is the union of some Di's.) Assume, on the contrary, that $Di \cap Bj \neq \emptyset$, but $Di \nsubseteq Bj$. Let $B'j = Bj \cup Di$ and MSB' (V) = MSB $(V) - \{Bj\} \cup \{B'j\}$.

CLAIM 1. MSB' (V) *is actually a minimum set basis.*

Proof. We need to show only that MSB' (V) is still a set basis. If MSB' (V) is not a set basis, then there must exist a Vq which is not covered exactly by unions of MSB' (V) elements. This happens because $B'j$ cannot be used in the cover of Vq even though Bj was in the Vq cover originally. Therefore, $B'j \nsubseteq Vq$. However, we know that $Bj \subseteq Vq$ and $Bj \cap Di \neq \emptyset$; thus $Du \cap Vq \neq \emptyset$. Since $Di \in$ DSB (V), $Di \subseteq Vq$. Therefore, $B'j = Bj \cup Di \subseteq Vq$, which is a contradiction. Hence MSB' (V) is a set basis and a minimum set basis, which gives us Claim 1. □

Similarly, we can show that all $Di \subseteq Bj$ and that every Bi is a union of some Di.

Secondly, we must show that MDSB (V) = MDSB [MSB (V)]. From the first part of the proof, it follows that MDSB (V) is a disjoint set basis of MSB (V). Suppose that the MDSB (V) is not a MDSB [MSB (V)], but some other disjoint set basis of MSB (V), namely DSB [MSB (V)], is minimum and |DSB [MSB (V)]| < |MDSB (V)|. Since MSB (V) is a set of basis of V, its disjoint set basis DSB [MSB (V)] must be a disjoint set basis for V of size smaller than that of MDSB (V). This is impossible, and hence MDSB (V) = MDSB [MSB (V)]. This completes the proof of Lemma 1. □

LEMMA 2. *For any family of sets* $V=\{V1, \cdots, Vp\}$, $Vi \subseteq S$, *there exists an optimal k-segmentation* KSEG (V, k) *such that the* MDSB (V) *is a refinement of the* KSEG (V, k).

Proof. Start out with an arbitrary segmentation, and show that the segmentation cost is not increased by rearranging the elements of the segments in a way that makes the MDSB (V) a refinement of that segmentation [5]. □

LEMMA 3. *For any family of sets* $V=\{V1, \cdots, Vp\}$, $Vi \subseteq S$, *there exists an optimal augmented-set basis* ABS (V, k) *such that the* MDSB (V) *is a refinement of the* ASB (V, k).

Proof. Let C be the cost of an optimal augmented-set basis of family V, and let $\bar{V}=\{\bar{V}1, \cdots, \bar{V}p\}$ be the augmented family. An optimal augmented-set basis of family V is also a minimum set basis of the augmented family $\bar{V}$, and vice versa. To complete the proof, use Lemmas 1 and 2 and the fact that the MDSB $(\bar{V})$ is a |MDSB $(\bar{V})$|-segmentation with the same cost C as that of the optimal augmented-set basis of family V [5]. □

The following lemma and theorem state the main result of this paper.

LEMMA 4. *If there exists a polynomial-time approximation algorithm for the augmented-set basis problem with cost $C \leqq c$ ASB $+ d$, where* ASB *is the optimal cost and c, d are constants, then there exists a polynomial-time approximation algorithm with cost $C' \leqq c$ ASB.*

Proof. Given the original augmented-set basis problem, let us construct a new augmented-set basis problem as follows:

(i) Construct a new set $S' = \{e'1, \cdots, e'_{2n}\}$ with twice as many elements as in the original set S.

(ii) For each $Bj \in$ MDSB (V) construct a $B'j \subseteq S' = \{e'1, \cdots, e'_{2n}\}$ such that $B'j = \{(e'i \cup e'_{n+i}) \mid ei \in Bj, i \in [1, n]\}$; i.e., for each element ei of Bj there will be two elements $e'i$ and e'_{n+i} in $B'j$. (Please see Fig. 1 for an example.)

(iii) For each Vj construct a $V'j$ such that $(Bi \in Vj) \Leftrightarrow (B'i \in V'j)$.

(An example of this construction is shown in Fig. 1. In that example, $S = \{e1, e2, e3\}$, $n = 3$, $V = \{V1, V2, V3\}$ where $V1 = \{e1\}$, $V2 = \{e2\}$, $V3 = \{e1, e2, e3\}$.)

By construction $B'i \in$ MDSB (V') and $|B'i| = 2|Bi|$. The new problem is isomorphic to the original one up to the size of each element of the input family (and of the minimum disjoint set basis). Using Lemma 3 and the fact that the MDSB (V) is unique, it is easy to show that ASB$' = 2$ASB, where ASB$'$ is the optimal cost of the new problem.

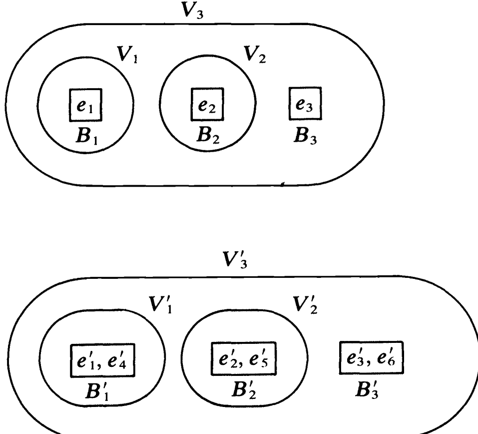


FIG. 1. *An example of how to construct family $V' \subseteq S'$ from family $V \subseteq S$.*

Now let us construct a polynomial-time approximation algorithm for the original problem as follows. First, we construct the new family $V'1, \cdots, V'p$ from the original family $V1, \cdots, Vp$ as described above. Then we run the algorithm that has the cost $C \leqq c$ ASB$' + d$ as stated in the hypothesis. Lastly, we divide the cost of the algorithm to obtain the cost $\underset{\sim}{C}$ of the original problem. Thus,

$$\underset{\sim}{C} = \lfloor C: 2 \rfloor \leqq \lfloor (c \text{ ASB}' + d): 2 \rfloor = \lfloor (2c \text{ ASB} + d): 2 \rfloor = c \text{ ASB} + \lfloor d: 2 \rfloor.$$

We repeat the construction suggested above $\lfloor \log(d) \rfloor + 1$ times. Thus, we create a new family of sets $\underset{\sim}{V}1, \cdots, \underset{\sim}{V}p$ whose $\underset{\sim}{B}i \in$ MDSB $(\underset{\sim}{V})$ have the size $|\underset{\sim}{B}i| = 2d|Bi|$. Note that this construction is performed in polynomial time. Then we apply the algorithm suggested in the hypothesis and divide the cost $|\log(d)| + 1$ times to get a final cost for the original problem $C' \leqq c$ ASB. $\square$

THEOREM 1. *If* $P \neq NP$, *there exists no polynomial-time approximation algorithm for the augmented-set basis problem with cost* $C \leqq c$ ASB, *where* ASB *is the optimal cost and c is a constant.*

Proof. Suppose that such an algorithm does exist and that we use it to test whether ASB = 0. Then c ASB $= 0 \Leftrightarrow C = 0$, which means that we have discovered a *polynomial-time* algorithm which finds whether the family of sets $V1, \cdots, Vp$ has a *set basis of size* k. But from reference [9] we know that, if $P \neq NP$, this is impossible. □

The key element for obtaining the above result is provided in the proof of Lemma 4, i.e., the observation that starting with the original problem we can find a new problem whose optimal cost is known to be double that of the original problem. (Also, from the new problem we can always reconstruct the original problem.) In its turn, this observation is facilitated by Lemma 3.

The proof technique used above can also be used to show that several NP-complete augmentation and deletion problems are not approximable in the sense of the above problem. (Several such problems are presented in § 4 below.) Two conditions must be met, however. First, the zero-cost version of the augmentation (deletion) problem must be NP-complete. Second, the cost function of the augmentation (deletion) problem must allow relationships between the optimal costs of different instances of the same problem to be established; these relationships, such as the doubling of the optimal cost, help eliminate the constant d from the bound of the approximate cost. Note that the above proof technique cannot be applied directly to the k-segmentation problem because the first condition mentioned above is not met; i.e., the zero-cost segmentation of any family V is the MDSB (V).

3. Other cost functions for augmented-set bases. As an example of a cost function for the augmented-set basis problem for which the main result of this paper remains valid, consider:

$$C = \sum_i |\{j \mid Vi \cap Vj = \emptyset,\ \bar{V}i \cap \bar{V}j \neq \emptyset\}|.$$

This cost function merely (double-)counts the disjoint sets that, due to augmentation, end up with a nonempty intersection after reconstruction. Although this cost function is of no immediate practical interest,[2] it differs sufficiently from the previous cost function to illustrate the strength of the above result. In the following theorem we prove that the augmented set basis with the new cost function is still NP-complete. As part of the proof of this theorem, we show that the zero-cost version of the new problem is NP-complete by reducing the k-colorability problem to the new version of our problem.

THEOREM 2. *The augmented-set basis problem with the cost defined by* $\sum_i |\{j \mid Vi \cap Vj = \emptyset,\ \bar{V}i \cap \bar{V}j \neq \emptyset\}|$ *is* NP-*complete.*

Proof. Let us transform an instance of the coloring problem $[(N, E), k]$ as follows [6]: The elements of set S are the "nonedges" of the graph, i.e., $a, b \in N$ and $(a, b) \notin E$. Let each set of the family V correspond to a node in N. For each node a, define Va as the set of all nonedges (a, b) for some $b \in N$. Assume that $(a, a) \in Va$, so $Va \neq \emptyset$.

CLAIM 2. *The family* $V1, \cdots, V_{|N|}$ *has an augmented-set basis of size k and cost zero if an only if the graph* (N, E) *is k-colorable.*

[2] However, in the context of the k-segmentation problem, this cost function provides a very useful measure of the "locking conflicts" that may appear in data bases due to lack of a sufficient number of small segments [4].

To prove this claim let us associate each color with an element of the augmented-set basis, and note that if two nodes of the graph are note adjacent, then their sets already overlap. Thus there is no added cost if the augmentations of these sets overlap.

Proof of "if." Suppose that there exists a k-coloring. Then we generate each element of the augmented-set basis from nonedges of nodes with the same color. We note that, although some elements of the augmented-set basis overlap, all nonedges are in the same element and each augmented set is represented by a single element of the basis. Thus the cost of the augmentation is zero.

Proof of "only if." Suppose that there exists a zero-cost augmented-set basis. Assume that each element of the basis is labeled with a distinct color. For each set Va, assign to node a the color of one of the basis elements that represents $\bar{V}a$. We claim that adjacent nodes have different colors. If adjacent nodes a and b have the same color, the corresponding augmented sets $\bar{V}a$ and $\bar{V}b$ must have a common element Bi of the basis in their representation. Since a and b are adjacent nodes, $Va \cap Vb = \emptyset$, but $\bar{V}a \cap \bar{V}b \neq \emptyset$ because their intersection contains Bi. Hence, the cost of the augmentation cannot be zero. □

Now it is easy to see that the new augmented-set basis problem is NP-complete. □

THEOREM 3. *If* P $\neq$ NP, *there exists no polynomial-time approximation algorithm with cost* $C \leqq c_1$ ASB $+ d_1$, *where* ASB *is the optimal cost computed by* $\sum_i |\{j \mid Vi \cap Vj \neq \emptyset, \bar{V}i \cap \bar{V}j = \emptyset\}|$ *and* c_1, d_1 *are constants.*

Proof. All we need to show is that, given the original problem instance, we can construct in polynomial time a new problem instance such that $\underline{\mathrm{ASB}} = 2$ ASB, where ASB is the optimal cost of the original problem and $\underline{\mathrm{ASB}}$ is the optimal cost of the new problem instance. The rest of the proof will then proceed as in the proofs of Lemma 4 and Theorem 1.

Let $V1, \cdots, Vp$ be a family of sets of $S = \{e1, \cdots, en\}$. Then let us construct the sets $S' = \{e'1, \cdots, e'n\}$ and $\underline{S} = S \cup S' \cup \{eij \mid 1 \leqq i \leqq j \leqq p\}$. Now let us define $\mathbf{V}'i = Vi \cup \{eij \mid 1 \leqq j \leqq p\}$, $\mathbf{V}i = \{e' \mid \in Vi\} \cup \{eji \mid 1 \leqq j \leqq p\}$, and let $\underline{V} = \{\mathbf{V}1, \cdots, \mathbf{V}p, \mathbf{V}'1, \cdots, \mathbf{V}'p\}$. Thus we have made two copies of the family $V1, \cdots, Vp$ with each $\mathbf{V}'i$ having a common element eij with each $\mathbf{V}j$.

CLAIM 3. $\underline{\mathrm{ASB}} = 2$ ASB, *where* $\underline{\mathrm{ASB}}$ *is the optimal cost of augmenting* $\underline{V}$.

Proof. Let $B1, \cdots, Bk$ be a cost C augmented-set basis for the original problem and let $\mathbf{B}i = Bi \cup \{e' \mid e \in Bi\}$. Now let us form $\mathbf{B}'1, \cdots, \mathbf{B}'k$ from $\mathbf{B}1, \cdots, \mathbf{B}k$ by adding each eij to a Bq such that Bq contains an $e \in \mathbf{V}'i$ or an $e' \in \mathbf{V}j$. Thus we have started with the basis $B1, \cdots, Bk$ and we have put an e' with every e, and eij's where they add no cost. Therefore,

(i) no $\mathbf{V}'i$ and $\mathbf{V}j$ can add any cost, since $\mathbf{V}'i \cap \mathbf{V}j = \{eij\}$, i.e., $\mathbf{V}'i$ and $\mathbf{V}j$ already overlap.

(ii) if $V1$ and $V2$ of the original family do not add any cost when they are represented with $B1, \cdots, Bk$, then neither do $\mathbf{V}'1$ and $\mathbf{V}'2$ nor $\mathbf{V}1$ and $\mathbf{V}2$ when they are represented with $\mathbf{B}'1, \cdots, \mathbf{B}'k$.

Therefore, the representations of $\mathbf{V}'1$, $\mathbf{V}'2$ and of $\mathbf{V}1$, $\mathbf{V}2$ increase cost only when the representations of $V1$ and $V2$ increase their cost. Thus, $\mathbf{B}'1, \cdots, \mathbf{B}'k$ is an augmented-set basis of $\underline{V}$ with cost $2C$. Hence, $\underline{\mathrm{ASB}} \leqq 2$ ASB.

On the other hand, suppose that there exists an augmented-set basis $\mathbf{B}'1, \cdots, \mathbf{B}'k$ for $\underline{V}$ with cost C. Let $Bi = \mathbf{B}'i \cap S$, i.e., Bi is $\mathbf{B}'i$ with all e' and eij's removed, and $\mathbf{B}i = \mathbf{B}'i \cap S'$, i.e., $\mathbf{B}i$ is $\mathbf{B}'i$ with all e and eij's removed.

Both $B1, \cdots, Bk$ and $\mathbf{B}1, \cdots, \mathbf{B}k$ are augmented-set bases for the original family (i.e., we have removed the primes from the elements e' in Bi's). Furthermore, if $V1$ and $V2$ increase cost when they are represented with $B1, \cdots, Bk$, then (1)

$\mathbf{V}'1$ and $\mathbf{V}'2$ increase cost when they are represented with $\mathbf{B}'1, \cdots, \mathbf{B}'k$, and (2) $\mathbf{V}1$ and $\mathbf{V}2$ increase cost when they are represented with $\mathbf{B}'1, \cdots, \mathbf{B}'k$. Therefore the sum of the costs of $\{B1, \cdots, Bk\}$ and of $\{\mathbf{B}1, \cdots, \mathbf{B}k\}$ is no greater than C. Thus the cost of one of the two bases is no greater than $\lfloor C:2 \rfloor$. Therefore $\underline{\text{ASB}} \geqq 2\,\text{ASB}$, and thus $\underline{\text{ASB}} = 2\,\text{ASB}$. □

4. Augmentation and deletion problems. In this section we define some graph-augmentation and deletion problems that are NP-complete. Using the proof technique of the previous section, we show that these problems are not approximable. Undoubtedly, similar proofs can be used to demonstrate that other NP-complete augmentation and deletion problems are not approximable.

4.1. Graph augmentation. Let $G = (N, E)$ be a graph and k, m be integers. The problem of finding a *clique cover of a minimally-augmented graph* is to find a new graph $AG = (N, E \cup E\text{co})$, where $E\text{co}$ is a subset of the "nonedges" of G, such that:

(i) the graph AG has clique cover of size k, and

(ii) the number of nonedges added to G, $|E\text{co}|$, is a minimum.

Similarly, we define the problem of finding a *clique of a minimally-augmented graph* (1) by rephrasing the property (i) above as "the graph AG has a clique size m," and (2) by denoting the subset of nonedges added to G as $E\text{cq}$. Both of these problems are NP-complete and their optimal solutions are not unique.

THEOREM 4. *If* $\text{P} \neq \text{NP}$, *there are no polynomial-time approximation algorithms for finding*:

(1) *A clique cover by adding* $C\text{co} \leqq c_2|E\text{co}| + d_2$ *nonedges to* G, *and*

(2) *A clique by adding* $C\text{cq} \leqq c_3|E\text{cq}| + d_3$ *nonedges to G*,

where $|E\text{co}|$, $|E\text{cq}|$ *are the optimal numbers of added nonedges and* c_2, c_3, d_2, d_3 *are constants.*

Proof. The zero-cost versions of these problems are NP-complete [6]. All we need to show is that, given an original problem instance, we can construct in polynomial time a new problem instance with twice the cost of the original problem. Given $G = (N, E)$, let $G' = (N', E')$ and $G'' = (N'', E'')$ be two copies of the graph G.

For the proof of part (1), the new problem instance is to find a minimum-cost augmentation so that the graph $(N' \cup N'', E' \cup E'')$ is $2k$-colorable. For the proof of part (2), the new problem instance is to find a minimum-cost augmentation so that the graph $(N' \cup N'', E' \cup E'' \cup \{(n'i, n''j) \mid n'i \in N', n''j \in N'', i, j \in [1, |N|]\})$ has a clique of size $2m$. For each of the new problem instances, it is easy to see that the optimal cost of the original problem is $|E\text{o}|$ if and only if the optimal cost of the new problem is $2|E\text{o}|$. The rest of the proof proceeds in the same way as the proofs of Lemma 4 and Theorem 1. □

4.2. Deletion problems. By analogy with the *node-deletion* problems [8], [10], we can define a class of *edge-deletion* problems as follows: Given a graph G, find the minimum number of edges that need to be deleted so that the remaining subgraph has a specified property. Let us consider the following examples.

Let $k1$ and $m1$ be integers. The problem of *coloring a graph by edge deletion* is to find a subgraph $DG = (N, E - E\text{dc})$, where $E\text{dc}$ is a subset of E, so that:

(i) The subgraph DG has a $k1$-coloring, and

(ii) The number of edges deleted from the graph G, $|\text{E}dc|$, is a minimum. Similarly, we define the problem of finding a *node cover of a graph G by edge deletion* (1) by rephrasing property (i) above as "the subgraph DG has a node cover of size $m1$,"

and (2) by denoting the subset of edges deleted from G as Edn. Both of these problems are NP-complete and their optimal solutions are not unique.

THEOREM 5. *If* $P \neq NP$, *there are no polynomial-time approximation algorithms for finding*:

(1) *A coloring of* G *by deleting* $C\mathrm{dc} \leqq c_4|E\mathrm{dc}| + d_4$ *edges, and*

(2) *A node cover of* G *by deleting* $C\mathrm{dn} \leqq c_5|E\mathrm{dn}| + d_5$ *edges*,

where $|E\mathrm{dc}|$, $|E\mathrm{dn}|$ *are the optimal numbers of deleted edges and* c_4, c_5, d_4, d_5 *are constants.*

Proof. In the proof, we use the transformations suggested in [6] for the reductions between the zero-cost versions of these problems (i.e., clique to node cover and colorability to clique cover). Then we show the equivalence between the approximation algorithms for the graph-augmentation problems defined above and the corresponding ones for the particular instances of the edge-deletion problems provided by the transformations. (Alternatively, the same proof technique as that used for the previous problems could also be used.) □

The same results can be easily obtained for other NP-complete problems. Among these problems we include (1) other edge-deletion problems (e.g., finding feedback arc (node) sets by edge deletion), (2) node-deletion problems (e.g., node-deleted maximal clique [8], [10]), and (3) appropriately defined element-deletion problems for families of sets.

5. Conclusion. A large number of combinatorial problems that appear in practice have been shown to be NP-complete in the past. However, few of these problems have been known to be unapproximable in the sense defined in this paper. The proof technique that we have used to show that alternate versions of the augmented-set basis problem are unapproximable can also be used to obtain similar results for other NP-complete augmentation problems. Efficient, practical algorithms may still be found for particular instances of the augmented-set basis problem and for the other non-approximable problems discussed above.

REFERENCES

[1] H. D. BLOCK, N. J. NILSSON AND R. O. DUDA, *Determination and detection of features in patterns*, in Computer and Information Sciences, J. T. Tou and R. H. Wilcox, eds., Spartan Books, Washington DC, 1964, pp. 75–100.

[2] K. P. ESWARAN AND R. E. TARJAN, *Augmentation problems*, this Journal, 5 (1976), pp. 653–664.

[3] J. F. GIMPEL, *The minimization of spatially-multiplexed character sets*, Comm. ACM, 17 (1974), pp. 315–318.

[4] V. D. GLIGOR AND D. MAIER, *Representing data bases in segmented name spaces*, in Databases: Improving Usability and Responsiveness, B. Shneiderman, ed., Academic Press, New York, 1978.

[5] ———, *Finding augmented-set bases*, Technical Report TR-882, Univ. of Maryland, College Park, February 1980.

[6] R. KARP, *Reducibility among combinatorial problems*, in Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, pp. 85–104.

[7] L. T. KOU AND C. K. WONG, *A note on the set basis problem related to the compaction of character sets*, Comm. ACM, 18 (1975), pp. 656–657.

[8] M. S. KRISHNAMOORTHY AND N. DEO, *Node NP-complete problems*, this Journal, 8 (1979), pp. 619–625.

[9] L. J. STOCKMEYER, *The minimal set basis problem is NP-complete*, IBM Research Rep. RC5431, May 1975.

[10] M. YANNAKAKIS, *The effect of a connectivity requirement on the complexity of maximum subgraph problems*, J. ACM., 26 (1979), pp. 619–625.

SIAM J. COMPUT.
Vol. 11, No. 4, November 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1104-0001 $01.00/0

PREDICTING THE NUMBER OF DISTINCT ELEMENTS IN A MULTISET*

KOHEI NOSHITA†

Abstract. The bounds of the number of comparisons are obtained for the problem of determining whether the number of distinct elements in a given multiset is no more than a given threshold value. Those bounds are asymptotically optimal within constant factors. Two models of the algorithm are considered to derive those bounds.

Key words. multiset, comparison, decision tree, predicting problem, oracle argument, complexity

1. Introduction. The problem of computing the set of distinct elements in a multiset appears in some practical applications. For instance, in the relational data base system [2], the projection operation will be implemented by means of the algorithm for this problem. Stockmeyer and Wong [6] have dealt with some related problems.

In this paper we shall consider a predicting problem on multisets. More specifically, the problem is to determine, for a given multiset S of n elements and a given threshold t, whether the number of distinct elements in S is less than or equal to t. This problem has been formulated from a rather practical motivation. In a relational data base system we are often required to know in advance the size of the result of the projection operation, because it may crucially affect the computation time for data movements or the amount of the output. Therefore, it is important to devise an efficient algorithm for predicting whether the size of the result is acceptable to the predetermined threshold, which runs within less computation time than that needed for actually obtaining the set of the distinct elements.

We shall show the asymptotic complexity, in terms of the number of comparisons, for the predicting problem on two different models, i.e., on the equal-unequal model and on the linear-order model. Both models have been discussed in Stockmeyer and Wong [6]. Munro and Spira [5] and Dobkin and Munro [3] have also introduced the linear-order model for certain problems on multisets. Although those models may be regarded as too mathematically simplified, the bounds of complexity for the predicting problem will be expected to give some insight into the more practical situations.

2. Basic definitions. Let $S=\{v_1, v_2, \cdots, v_n\}$ be an arbitrarily given multiset of n elements ($n \geqq 1$). The universal set from which each element in S is extracted will be implicitly defined in the description of the algorithms below.

DEFINITION 1. Let $k(S)$ denote the set of distinct elements in a multiset S.

The problem is to decide, for any multiset S and integer t, whether $|k(S)| \leqq t$ or not. We shall call this problem the "predicting" problem.

Any algorithm to solve the predicting problem will be represented as a decision tree, for which we shall consider two different types of branching. The first one is based on the "equal-unequal" model, in which each comparison between any two elements x and y in S will give the information, either "$x=y$" or "$x \neq y$". On the other hand, in the second model, called the "linear-order" model, the corresponding

* Received by the editors November 18, 1980. This research was supported by the Scientific Research Council of the United Kingdom under grant GR/A/9418.8.

† Department of Computer Science, University of Warwick, Coventry CV4 7AL, United Kingdom. Now at Department of Computer Science, Denkitusin University, Chofu, Tokyo 182, Japan.

information is to be either "$x<y$" or "$x=y$" or "$x>y$"; namely, the universal set is linearly ordered.

The reader will be assumed to be familiar with the notion of the decision trees of those types, and with the worst case complexity defined on the decision trees [3], [5], [6]. For the general terminology on multisets as well as decision trees, see also Knuth's book [4].

DEFINITION 2. Let $T_E(n, t)$ and $T_L(n, t)$ denote the number of comparisons required to solve the predicting problem in the worst case on the equal-unequal model and on the linear-order model, respectively.

In the following sections, when the model being considered is clearly understood from the context, the abbreviated notation $T(n, t)$ will also be used in place of $T_E(n, t)$ or $T_L(n, t)$.

Clearly, $T_E(n, n)=T_L(n, n)=0$. Furthermore, it is easy to see that $T_E(n, 1)= T_L(n, 1)=n-1$. Hence, from now on we shall assume that $1<t<n$ and $n \geqq 3$. The purpose of this paper is to investigate the asymptotic behavior of $T_E(n, t)$ and $T_L(n, t)$ as n becomes sufficiently large.

3. Bounds on the equal-unequal model. In this section we shall show the bounds of the number of comparisons to solve the predicting problem on the equal-unequal model.

PROPOSITION 1.

$$T_E(n, t) \leqq tn - \binom{t+1}{2}.$$

Proof. The following straightforward algorithm gives an upper bound for $T(n, t)$.

```
begin
  V ← {v_1};
  for i ← 2 step 1 until n do
    begin for each v_j in V do
            if v_i = v_j then goto L;
      V ← V ∪ {v_i};
      if |V| ≧ t+1 then return "|k(S)| > t";
  L: end;
  return "|k(S)| ≦ t"
end.
```

It is easy to see that the worst case occurs in the following situation. The first t elements $V_t=\{v_1, v_2, \cdots, v_t\}$ that are picked up in the outer loop are all distinct, and each of the remaining $(n-t)$ elements turns out to be equal to the tth element of V_t after $(t-1)$ comparisons of inequality, except the final comparison of the last nth element. The final comparison decides whether $|k(S)|=t$ or $|k(S)|=t+1$. Hence the upper bound for $T(n, t)$ is derived as follows:

$$T(n, t) \leqq \binom{t}{2} + t(n-t) = tn - \binom{t+1}{2} \quad \text{for any } n \geqq 3 \text{ and } 1<t<n. \qquad \square$$

THEOREM 2.

$$T_E(n, t) \geqq \lceil tn/2 \rceil - (t+1) \text{ if } t < \lceil n/2 \rceil, \text{ and}$$

$$T_E(n, t) \geqq n^2/2 - (n-t)^2 - (t - n/2) \text{ if } t \geqq \lceil n/2 \rceil.$$

Proof. The lower bound for $T(n, t)$ will be derived by means of the oracle (adversary) argument [4]. First we shall describe the construction of the oracle B, which replies to each question of any algorithm A in such a way that B forces A to perform as many comparisons as possible. Next we shall count the number of comparisons which must be performed by any algorithm A.

During the execution of an algorithm A, the oracle B represents the result of each comparison as a system $\Sigma = (G, h)$, where G is an undirected graph (V, E) and h is a mapping from V to $\{0, 1, 2, \cdots, t-1\}$. The system Σ always satisfies the following conditions. The set of nodes V of G is a subset of the "set" S in the sense that each element v_i in the "multiset" S is regarded as the element of the set S distinguished by its own index i. Thus, for example, the notation $X \cup Y$ will be used to denote the union of two "sets" X and Y in this sense. Furthermore, it always holds that, for any (x, y) in E, $h(x) \neq h(y)$, i.e., h is a t'-coloring of V for some $t' \leqq t$ [1]. The existence of an edge (x, y) in E implies that, in the multiset S, two elements x and y have been determined to be distinct by A. In general, G consists of m mutually disjoint connected components, $G_1, G_2, \cdots, G_m$, where $G_i = (V_i, E_i)$ is a connected component ($V_i \neq \varnothing$, $1 \leqq i \leqq m$, $m \geqq 1$).

The basic idea for constructing B is that Σ always satisfies the condition that G has t-coloring as long as $|V| \geqq t+1$. While this condition is satisfied, A cannot decide whether $|k(S)| \leqq t$ or not. However, there seems to be no useful characterization of graphs having t-coloring ($t \geqq 3$) [1], which facilitates calculating the number of edges in those graphs. Therefore, we shall content ourselves with a relatively simple procedure which preserves the t-coloring property of G.

Before A starts, B will initialize Σ as follows:

begin $V \leftarrow S; E \leftarrow \varnothing;$
    $h(x) \leftarrow 0$ **for** each x in V
end.

The initialized system Σ consists of n connected components, each of which is a singleton node.

For each comparison of A between x and y (x, y in V, $x \neq y$), the answer of B and the corresponding new Σ will be determined by the following rules, Cases I and II, up until the number of nodes in V becomes equal to $t+1$. The period during which these rules are applied will be called the "intermediate" stage.

Case I. If $\deg(x) < t-1$ or $\deg(y) < t-1$[1]. The answer of B is "$x \neq y$". Σ is modified as follows. Without loss of generality, assume that $\deg(x) < t-1$.

Case IA1. If both x and y are in V_i for some i ($1 \leqq i \leqq m$), and $h(x) = h(y)$, then change the color of x from $h(x)$ to another color h' ($0 \leqq h' \leqq t-1$) such that $h' \neq h(y)$ and $h' \neq h(z)$ for all (z, x) in E_i, and assign $E_i \leftarrow E_i \cup \{(x, y)\}$. Note that this change of color is always possible, because at least one color has not yet been used in $\{z \mid (z, x) \text{ in } E_i\} \cup \{y\}$.

Case IA2. If both x and y are in V_i for some i ($1 \leqq i \leqq m$), and $h(x) \neq h(y)$, then assign $E_i \leftarrow E_i \cup \{(x, y)\}$.

Case IB1. If x is in V_i and y is in V_j for $i \neq j$ ($1 \leqq i, j \leqq m$), and $h(x) = h(y)$, then change the color of z from $h(z)$ to $h(z)+1 \pmod t$ for each z in V_i.

Case IB2. Consider the case in which x is in V_i and y is in V_j for $i \neq j$ ($1 \leqq i, j \leqq m$), and $h(x) \neq h(y)$.

In both Cases IB1 and IB2, assign

$$V_i \leftarrow V_i \cup V_j \quad \text{and} \quad E_i \leftarrow E_i \cup E_j \cup \{(x, y)\}.$$

[1] As usual, $\deg(x)$ denotes the number of edges incident to node x.

Thus, the number of connected components is reduced by one ($m \leftarrow m-1$). Renumber all the connected components from 1 to m.

Case II. If $\deg(x) \geqq t-1$ and $\deg(y) \geqq t-1$.

Case IIA1. If both x and y are in V_i for some i ($1 \leqq i \leqq m$), and $h(x)=h(y)$, then the answer is $x=y$. Contract two nodes x and y into one new node. Let

$$X = \{(z, y) \text{ in } E \mid (z, x) \text{ is not in } E\}, \quad \text{and}$$

$$Y = \{(z, y) \text{ in } E \mid (z, x) \text{ is in } E\}.$$

The set of edges incident to the new node becomes

$$\{(z, x) \text{ in } E\} \cup X.$$

The new node is given the name x again. The contracted node y will be called an "absorbed" node. In this sense the old node x is an "absorbing" node. Furthermore, any edge in Y will be called an "absorbed" edge. For convenience, any comparison which induces the absorption will be called a "contracted" edge, which never actually exists in G.

The following three cases will be dealt with in just the same way as Cases IA2 and IB1 and 2, respectively. In any case, the answer is $x \neq y$.

Case IIA2. If both x and y are in V_i for some i ($1 \leqq i \leqq m$), and $h(x) \neq h(y)$, and

Case IIB1 and 2. If x is in V_i and y is in V_j for $i \neq j$ ($1 \leqq i, j \leqq m$).

At the time when the number of elements in V is reduced to exactly $t+1$, B changes the rules described above. The period from this point of time to the termination of the execution of A will be called the "final" stage. It should be noticed that, until the beginning of the final stage, as long as B maintains Σ by those rules described above, A cannot decide whether $|k(S)| \leqq t$ or not.

The rule of B during the final stage is very simple. For any comparison between nodes x and y, the answer of B is $x \neq y$ and edge (x, y) is added to G in Σ, changing the color of either of the nodes whenever necessary, except when the addition of edge (x, y) makes G to be a complete graph of $(t+1)$ elements. This rule may be always applied, because there exists t-coloring of G of $(t+1)$ nodes unless G is a complete graph. By the last exceptional comparison between x and y, A can finally decide whether $|k(S)| \leqq t$ or not.

We shall introduce several notations to be used in the counting argument. For any fixed S and t, let C, C_I and C_F denote the number of comparisons performed by A, respectively throughout the execution, during the intermediate stage and during the final stage. Obviously,

$$C = C_I + C_F.$$

Let $\Sigma_F = (G_F, h_F)$ denote the representing system of B at the time when the final stage begins. The notation $G_F = (V_F, E_F)$ will be naturally understood. Without loss of generality, assume that $V_F = \{v_1, v_2, \cdots, v_{t+1}\}$. Let $V_F = V_1 \cup V_2$, where

$$V_1 = \{x \mid x \text{ has absorbed at least one node during the intermediate stage}\}$$

and $V_2 = V_F - V_1$. For brevity, let $s = |V_1|$. Furthermore, let $E_F = E_1 \cup E_2 \cup E_3$, where

$$E_1 = \{e \mid e = (x, y) \text{ and } x, y \text{ in } V_1\},$$

$$E_2 = \{e \mid e = (x, y) \text{ and } x, y \text{ in } V_2\}, \quad \text{and}$$

$$E_3 = \{e \mid e = (x, y) \text{ and } x \text{ in } V_1 \text{ and } y \text{ in } V_2\}.$$

Two different counting arguments will be employed to derive the lower bounds for C. The first argument is based on the fact that any node v in $(S-V_F)\cup V_1$ must have at least $(t-1)$ incident edges when v is absorbed by or absorbs some other node for the first time in Case IIA1. Hence, we have the following inequality.

$$2C_I \geqq (t-1)(n-t-1+s)+(n-t-1)+2|E_2|+|E_3|.$$

On the right-hand side, the second term represents the necessary number of contracted edges. On the other hand, we have

$$C_F \geqq \binom{t+1}{2} - |E_F| = \binom{t+1}{2} - |E_1| - |E_2| - |E_3|.$$

Combining those two inequalities, we find that the following inequality holds.

$$2(C_I+C_F) \geqq t(n-t-1)+(t-1)s+t(t+1)-2|E_1|-|E_3|.$$

Obviously, $|E_1| \leqq s(s-1)/2$, and $|E_3| \leqq s(t+1-s)$. Therefore we have $C = C_I + C_F \geqq tn/2 - s \geqq tn/2-(t+1)$, because $s \leqq t+1$. Hence we have the first lower bound for $T(n,t)$, because the right-hand side of the above inequality represents the necessary number of comparisons for any A; namely,

$$T(n,t) \geqq \ulcorner tn/2 \urcorner - (t+1). \tag{1}$$

The second counting argument is based on the fact that C_I must be greater than or equal to the number of absorbed edges, plus the number of contracted edges and remaining edges in G_F. Hence we have

$$C_I \geqq D + (n-t-1) + |E_F|,$$

where D denotes the number of edges which have been absorbed. Assume that

$$v_{t+2}, v_{t+3}, \cdots, v_n$$

is the sequence of absorbed nodes in the reverse order; namely, v_n is absorbed first, and v_{n-1} is absorbed second, etc. When v_{t+2} is absorbed to some v_i in V_F, both $\deg(v_{t+2})$ and $\deg(v_i)$ are at least $(t-1)$. Hence, at least

$$2(t-1)-t$$

edges must be absorbed, because at this point of time there remain only t other nodes than v_{t+2} and v_i in V. Similarly, when v_{t+3} is absorbed, at least

$$2(t-1)-(t+1)$$

edges must be absorbed.

In general, if $2t<n$, at least

$$\sum_{i=2}^{t-2} i = \frac{(t-1)(t-2)}{2} - 1$$

edges must be absorbed throughout the intermediate stage. On the other hand, if $2t \geqq n$, the corresponding number of absorbed edges is at least

$$\sum_{i=2}^{n-t} (t-i) = \frac{(n-t-1)(3t-n-2)}{2}.$$

Hence we have another representation for the lower bound for $T(n, t)$.

$$T(n,t) \geqq \frac{(t-1)(t-2)}{2} - 1 + (n-t-1) + \binom{t+1}{2}$$

$$= n + t(t-2) - 1 \tag{2}$$

if $2t < n$, and

$$T(n,t) \geqq \frac{(n-t-1)(3t-n-2)}{2} + (n-t-1) + \binom{t+1}{2}$$

$$= \binom{n}{2} - (n-t)(n-t-1) \tag{3}$$

if $2t \geqq n$.

Comparing those lower bounds (1), (2) and (3) derived above completes the proof of Theorem 2 □

COROLLARY 3. *$T_E(n, t)$ is of order nt within a constant factor.*

Note that, if we take the lower term of our lower bounds into consideration, the lower bound (2) may be marginally superior to the other one (1) when t is very near $n/2$ in case of $t < n/2$. For certain specific values of t, we can obtain the exact number of comparisons; namely,

$$T_E(n, n-1) = n(n-1)/2,$$

$$T_E(n, n-2) = (n+1)(n-2)/2, \quad \text{and}$$

$$T_E(n, 2) = 2n-3.$$

The proof is left to the interested reader.

4. Bounds on the linear-order model. The following theorem is the final result of this section. We shall assume that n is sufficiently large.

THEOREM 4. *$T_L(n, t)$ is of order n* log *t within a constant factor.*

Proof. The upper bound for $T(n, t)$ may be easily achieved by modifying the algorithm in the previous section. In the algorithm, V will be maintained as a linearly ordered set. And, for each element v, the binary search algorithm will be applied for checking whether v is in V or not.

The worst case analysis of this modified algorithm is also similar to the case of the equal-unequal model. In the worst case, the first t elements are mutually distinct and each of the remaining $(n-t)$ elements is absorbed after $\lceil \log (t+1) \rceil$ comparisons in binary searching. Thus, the total number of comparisons performed in the worst case is derived as follows.

$$T(n,t) \leqq t\lceil \log (t+1) \rceil + (n-t)\lceil \log (t+1) \rceil \leqq n \lceil \log (t+1) \rceil.$$

In the linear-order model, two proof techniques to derive lower bounds have been frequently used in the literature. One is based on the information–theoretic counting principle. The other employs the oracle argument. In our proof we shall use the latter argument in order to show not only the final result but also the novel technique, which may be applicable to certain related problems. The delicate difficulty in proving the result is that any algorithm solving the predicting problem may correctly terminate even when the exact number of distinct elements of the given multiset has not yet been known, in the same sense as on the equal-unequal model. The main point of our argument is to construct the oracle B which prevents any algorithm A

from correctly guessing whether $|k(s)|=t$ or $t+1$ until A classifies almost all elements in S into t classes with equal values.

We shall describe the construction of the oracle B, which gives the answer to each comparison of the algorithm A. For simplicity, assume that $t=2^p$ for some integer $p \geqq 1$.

Consider the complete binary tree T with t leaves. Here, a complete binary tree means, as usual, a binary tree of the special shape, which is used to represent a heap in the Heapsort algorithm [4]. Each leaf of T will be dealt with as containing the elements with the equal value.

For each subtree (more formally for each subcomplete binary tree) T' of T, assign the capacity $c(T')$ as follows. If T' is a subtree whose root is at level m for $0 \leqq m \leqq p$, then let $c(T)=n/2^m$. In particular, $c(T)=n$, and $C(L)=n/t$ for any leaf L. The notation $c(M)$ will be used to denote $c(T')$, where M is the root of T'.

Before A starts, let all n elements in S be contained in the root of T. During the execution of A, each element in S will visit some nodes in T as it goes down from the root to some leaf. The oracle B always maintains the distribution of n elements in the nodes of T, such that the number of elements contained in any subtree T' must be less than or equal to its capacity $c(T')$.

For convenience, the following notations will be used. For two distinct nodes M and N in T, let $M \rightarrow N$ denote that N is a proper descendant of M. For two distinct nodes M and N in T, let $M<N$ denote the relation satisfying the following condition: There exists some node K such that M is a descendent of the leftson of K and N is a descendant of the rightson of K. For any node M in T, let @ M denote the number of elements which are contained in the subtree whose root is M.

We shall describe the rules of B to answer each question $x<y$, $x=y$, $x>y$. When the number of elements contained in the nonleaf nodes is more than two, apply rule I, II or III. Otherwise, if both x and y are either in some leaf or in some two leaves, apply III or I, respectively. In all other cases, apply IV.

I. If x is in M and y is in N such that $M<N$, then the answer is "$x<y$".

II. If x is in M and y is in N such that $M \rightarrow N$, determine the node M' which x will visit by the following procedure. Let $M_1, M_2, \cdots, M_k$ be the sequence of nodes on the path from M to N, where $M=M_1$ and $N=M_k$ ($k \geqq 2$).

```
begin
  for i ← 1 step 1 until k − 1 do
    begin Let M' be the son of M_i which is not M_{i+1};
      if @ M' < c(M')
        then begin Let x visit M';
                   The answer is either
                   "x < y" or "x > y" depending
                   on either M' < N or M' > N,
                   respectively;
                   return
                end
    end;
  Let x visit N, and apply III
end
```

III. Let M be the node at which x and y are staying. If M is a leaf, then the answer is "$x=y$". Otherwise, let M_L and M_R be the leftson and rightson of M,

respectively. If @ $M_L = c(M_L)$, let x and y visit M_R, and apply III again. If @ $M_R = c(M_R)$, let x and y visit M_L, and apply III again. Now @ $M_L < c(M_L)$ and @ $M_R < c(M_R)$. Let x visit M_L and y visit M_R, and the answer is "$x < y$".

IV. Without loss of generality, assume that x is in a nonleaf node M. If y is in a nonleaf node N, let y visit the leaf N' such that $N \to N'$ and @ $N' < c(N')$. If the prediction of A is "$|k(S)| \leqq t$", then assign the new $(t+1)$th value to x, and place x in the entirely new position between two adjacent leaves which is consistent with the linear order so far given to A; otherwise, let x visit the leaf M' such that $M \to M'$ and @ $M' < c(M')$. The answer is naturally decided depending on their relative positions. Note that at the time when this rule IV is applied, B still has the freedom to choose one of the two cases $|k(S)| = t$ and $|k(S)| = t+1$. It is easy to see that, except for the last one or two elements, all the elements in the root of T will eventually be divided into t classes, each of which is contained in some leaf as having equal value.

We shall count the number of comparisons necessary to lead to the stage in which rule IV is applied. Although one comparison may cause some number of (intermediate) visits, only the last "visit" will be taken into consideration henceforth.

LEMMA. *Let X be an arbitrary nonleaf node at level m in T. The sons of X will be denoted by Y and Z. Then, before rule* IV *is applied, when $n/2^m$ elements are going down from X or its ancestors to the substrees M and N whose roots are Y and Z, respectively, at least $n/2^{m+1}$ elements must visit either Y or Z or both.*

Proof. In the execution of A, all the comparisons that have been performed among the elements in M and N may be ignored without affecting the lemma. If none of Y and Z has been visited by the number of elements equal to those capacities, the going-down element must visit either Y or Z. Hence, at least one of the nodes Y and Z will be visited by exactly $n/2^{m+1}$ elements. □

In the following, the word "nearly" will be used to take the last one or two elements into consideration. In any case, the last one or two elements have no significant influence on the asymptotic behavior of the number of comparisons. Because of the property stated in the lemma, n elements in S initially contained in the root of T will be distributed as follows.

Nearly $n/2$ elements visit one of the two nodes at level 1. The other nearly $n/2$ elements go down to the subtree whose root is the other node at level 1 considered above. By the lemma, nearly half of those elements, namely nearly $n/4$ elements, visit one node at level 2. On the other hand, the former nearly $n/2$ elements go down to two subtrees whose roots are at level 2. One of the root nodes is visited by nearly $n/4$ elements. Repeating this argument shows that at any level $m (0 < m < p)$, at least $n/2$ elements visit the nodes at this level m. Hence, the total number of visits at nodes in T is at least $(n/2)p$.

For each comparison, at most two elements leave from the corresponding nodes, at which those elements are now staying. Therefore, the total number of comparisons is at least $np/4$, which gives the desired lower bound.

This proof may be immediately generalized to the case of $2^{p-1} < t \leqq 2^p$ for some integer p. □

Note that the number of comparisons performed after rule IV is applied will be negligible and that the number of comparisons performed in leaf nodes will also be of lower order, unless t is extremely small.

This proof may be easily modified to derive the lower bound for the problem of partitioning n given elements into t classes of equal size. In particular, the $n \log n$ lower bound for sorting n elements is derived within a constant factor.

5. Concluding remarks. In this paper we have introduced the predicting problem on multisets, and applied the oracle argument to derive the complexity of the problem in terms of the number of comparisons on two different models. Those bounds obtained are asymptotically optimal within constant factors with respect to n and t.

The open problem naturally posed is to determine the exact number of comparisons required to solve the predicting problem on each model.

Acknowledgments. The author expresses his thanks to Bill McColl for his discussion and also to the referees for their useful suggestions to refine the paper.

REFERENCES

[1] C. BERGE, *Graphs and Hypergraphs*, North-Holland, Amsterdam, 1973.
[2] E. F. CODD, *A relational model of data for large shared data banks*, Comm. ACM, 13 (1970), pp. 377–387.
[3] D. DOBKIN AND J. I. MUNRO, *Determining the mode*, Theoret. Comput. Sci. 12 (1980), pp. 255–263.
[4] D. E. KNUTH, *Sorting and Searching, The Art of Computer Programming, vol.* 3, Addison-Wesley, Reading, MA, 1973.
[5] J. I. MUNRO AND P. M. SPIRA, *Sorting and searching in multisets*, this Journal, 5 (1976), pp. 1–8.
[6] L. J. STOCKMEYER AND C. K. WONG, *On the number of comparisons to find the intersection of two relations*, this Journal, 8 (1979), pp. 388–404.

SIAM J. COMPUT.
Vol. 11, No. 4, November 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1104-0002 $01.00/0

ON LINEAR CHARACTERIZATIONS OF COMBINATORIAL OPTIMIZATION PROBLEMS*

RICHARD M. KARP† AND CHRISTOS H. PAPADIMITRIOU‡

Abstract. We show that there can be no computationally tractable description by linear inequalities of the polyhedron associated with any NP-complete combinatorial optimization problem unless NP = co-NP—a very unlikely event. We also apply the ellipsoid method for linear programming to show that a combinatorial optimization problem is solvable in polynomial time if and only if it admits a small generator of violated inequalities.

Key words. combinatorial optimization problem, polyhedral combinatorics, facial description, facets, NP, co-NP, ellipsoid method, separating

1. Introduction. It is well known that many important combinatorial problems can be formulated as the maximization of a linear functional over a polytope with integer vertices. Examples include matching, the knapsack problem [Ba], the traveling-salesman problem [DFJ], vertex packing and set packing [NT], [Pa], the three-dimensional matching problem and many others [PS].

There has been a very large body of literature aimed at the characterization of such convex polytopes by linear inequalities. The motivation apparently has been that such a characterization would bring a combinatorial optimization problem within the scope of linear programming methods, and thus might yield an efficient algorithm for its solution. This approach has worked in some cases, most notably the matching problem of Edmonds [Ed1].

Unfortunately, many combinatorial optimization problems are NP-complete. Despite the evidence that such problems are intractable, research on the description by linear inequalities of the convex polytopes associated with these problems has continued—besides the above references we mention [Ch], [Gr], [GP1], [GP2], [Ma], and [PR]. The main motivation has been the development of reasonably efficient algorithms by application of the simplex method to a heuristically generated subset of the inequalities describing the polytope. In order for such an algorithm to be guaranteed to terminate at the optimum, a complete description of the polytope by inequalities must be available. If only a partial description is used, then certain objective functions will force the simplex algorithm to terminate at an infeasible point. Unfortunately, so far, despite much intensive research effort, there has been no satisfactory description by linear inequalities of any convex polytope corresponding to an NP-complete combinatorial optimization problem. Note that, since an exponential number of inequalities might be required, such a description would not directly imply $P = \text{NP}$ via the recently discovered polynomial-time algorithm for linear programming [Kh].

It is natural to ask whether, in principle, there can be a satisfactory description by linear inequalities of the set of polytopes associated with an NP-complete combinatorial optimization problem. To address this question we must, of course, define what we mean by the terms "combinatorial optimization problem" and "satisfactory

* Received by the editors March 3, 1980, and in final form January 14, 1982.

† Computer Science Division, Department of Electrical Engineering and Computer Sciences, and Electronics Research Laboratory, University of California, Berkeley, California 94720. The research of this author was supported by the National Science Foundation under grant MCS 77-09906.

‡ Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139, and National Technical University of Athens, Athens, Greece. The research of this author was supported by the National Science Foundation under grant MCS 77-08965.

description". Loosely, a combinatorial optimization problem is a set of polytopes with integer vertices. This set of polytopes is indexed by strings in $\{0, 1\}^*$. The indexing is computationally efficient in the sense that, given a string z and an integer vector y, it is decidable in polynomial time whether y is a vertex of the polytope associated with z. An instance of the problem is specified by a string z, a vector c of profit coefficients and a threshold k; the instance is accepted if there is a point x in the polytope determined by z such that $c \cdot x \geqq k$.

A characterization of this set of polytopes by linear inequalities is a rule associating with each string z a set of linear inequalities determining the associated polytope. For a characterization to be satisfactory we only require that it be in NP. In other words, if a string z and an inequality are presented, there should be a short proof of the fact that the inequality is part of the description of the polytope associated with z. Most such descriptions discussed in the literature are in NP; in fact, all but the comb inequalities of [Ch] and their generalizations given in [GP2] are in P.

Our first main result (Theorem 1) is that no satisfactory description by linear inequalities of the polytopes corresponding to an NP-complete combinatorial optimization problem is possible, unless NP = co-NP. The class co-NP consists of those sets whose complements are in NP. The hypothesis that NP = co-NP is weaker than P = NP, but is generally considered almost as improbable. For example, NP = co-NP would imply that there is a "good" characterization of non-Hamiltonian graphs, and that there is a short proof of every contradiction in the propositional calculus.

The second main concern of this paper is a set of complexity questions motivated by the now-famous ellipsoid method for solving the linear programming problem. Our specific motivation is to apply the method to combinatorial optimization problems for which the defining linear inequalities are not explicitly given, but are determined by some structural property. Application of the method in such cases requires the algorithmic generation of violated inequalities. Define a *generator* for a combinatorial optimization problem as an algorithm which, given a string z and a point y in R^n, either determines that y lies in the convex polytope associated with z, or else produces a linear inequality violated by y but satisfied by all points in the polytope. A generator is called *small* if it always generates an inequality whose coefficients are bounded in length by a polynomial in the length of the description of the problem instance. Our second main theorem is that a combinatorial optimization problem is solvable in polynomial time if and only if it admits a small generator of violated inequalities.

Similar theorems have been proven independently by Grotschel, Lovasz and Schrijver [GLS]. Their theorems, however, apply only to combinatorial optimization problems whose associated polyhedra are full-dimensional. We get around this limitation by introducing the *repeated projection procedure*, a variant of the ellipsoid method which applies to all combinatorial optimization problems, whether or not they are full-dimensional. By analyzing this method we establish, under very general conditions, that the complexity of a combinatorial optimization problem is polynomially related to the problem of generating violated inequalities.

2. Combinatorial optimization problems and facial descriptions. A common type of combinatorial optimization problem is the following:

$$\text{(1)} \qquad \begin{aligned} &\text{maximize} \quad c \cdot x \\ &\text{subject to} \quad x \in S \end{aligned}$$

where[1] $S \subseteq Z^n$ is the set of feasible solutions and $c \in Z^n$. It is well known that an

[1] We denote the integers by Z, the nonnegative integers by Z^+, and the rationals by R.

equivalent formulation of (1) is the following:

$$\text{(2)} \qquad \begin{array}{ll} \text{maximize} & c \cdot x \\ \text{subject to} & x \in \mathrm{CH}\,(S) \end{array}$$

where CH (V) denotes the convex hull of the point set V.

Taking a view toward algorithmic issues, we shall carefully distinguish between *problems* and *instances*. A problem will generally have an infinite number of instances, each of which is similar in form to (1).

DEFINITION 1. A combinatorial optimization problem (or, briefly, c.o.p.) C is specified by:

(i) a set $L \subseteq (0, 1\}^*$;

(ii) a function n from L into Z^+;

(iii) for each $z \in L$, a set $S(z) \subseteq (Z^+)^{n(z)}$ such that each of the following three languages is recognizable in polynomial time:

$$L, \quad \{\langle z, y\rangle \| y| = n(z)\} \quad \text{and} \quad \{\langle z, x\rangle | x \in S(z)\}.$$

DEFINITION 2. An *instance* of C is a pair $\langle z, c\rangle$ where $z \in L$ and $c \in Z^{n(z)}$. The instance $\langle z, c\rangle$ corresponds to:

$$\max \sum_{j=1}^{n(z)} c_j x_j$$

subject to $x = (x_1, x_2, \cdots, x_{n(z)}) \in S(z)$.

Weighted matching, set covering, integer programming, the traveling-salesman problem and a plethora of other problems can be expressed as combinatorial optimization problems. In one formulation of the undirected traveling-salesman problem, for example, z is the binary representation of a positive integer n, $n(z)$ is $\binom{n}{2}$, the number of edges in K_n, the complete graph on n vertices, and $S(z)$ is the set of characteristic vectors of the Hamiltonian circuits of K_n.

Given a c.o.p. C, define $D(C)$, the *decision problem for* C, as $D(C) = \{\langle z, c, k\rangle | \langle z, c\rangle$ is an instance of C, $k \in Z$ and $\exists x \in S(z)$ such that $c \cdot x \geqq k\}$.

If $D(C)$ is NP-complete, then C is an NP-complete combinatorial optimization problem.

A *facial description* of a c.o.p. C is a set $F(C)$ such that:

(i) each element of $F(C)$ is of the form $\langle z, f, g\rangle$ where $z \in L$, $f \in Z^{n(z)}$ and $g \in Z$;

(ii) for each $z \in L$, and for all $x \in R^{n(z)}$, the following are equivalent:

(a) $x \in \mathrm{CH}\,(S(z))$,

(b) for each triple $\langle z, f, g\rangle \in F(C)$, $f \cdot x \leqq g$.

Thus $F(C)$ gives a description by linear inequalities of CH $(S(z))$, for each $z \in L$.

The facial description $F(C)$ is called a *small facial description* if there is a polynomial $p(\cdot)$ such that, for every $\langle x, f, g\rangle \in F(C)$, each component of f, g has absolute value $\leqq 2^{p(|z|+n(z))}$. The existence of a small facial description implies, in particular, that, for every $z \in L$, CH $(S(z))$ is a convex polyhedron (i.e., it is the intersection of a finite number of half-spaces).

The c.o.p.s that occur in practice invariably have small facial descriptions. We describe two especially common classes of such problems.

Class 1. *Zero-one problems*. This is the case where every vector in $S(z)$ is a 0–1 vector.

Class 2. *Problems of integer programming type.* In this case the input z is the binary encoding of an integer $m \times n$ matrix A and an integer n-vector b, and

$$S(z) = \{x \mid Ax \leqq b, x \geqq 0, x \text{ integer}\}.$$

LEMMA 1. *Every zero–one problem or problem of integer programming type has a small facial description.*

Proof. Any convex polyhedron Q in R^n can be expressed in terms of a finite set V of *vertices* and a finite set W of *extreme rays*; Q is just the set of vectors of the form $x_1 + x_2$, where x_1 is a convex combination of vertices and x_2 is a positive combination of extreme rays. Let S be the unique minimum-dimensional affine subspace of R^n containing Q, and let the dimension of S be d. Then, if $w_1 \in V$, $S - w_1 = \{x - w_1 \mid x \in S\}$ is a linear subspace of dimension d. Let B be a set of $n - d$ unit vectors, none of which lie in $S - w_1$. Then Q can be described by a finite number of linear inequalities, each of the form $f \cdot x \leqq g$, where $f \cdot x = g$ is the equation of a supporting hyperplane of Q. It follows that f and g are determined by a selection process of the following type: Select $l+1$ vertices $v_0, v_1, \cdots, v_l$, where $0 \leqq l \leqq n-1$, and $n-1-l$ vectors $h_1, h_2, \cdots, h_{n-1-l}$ from $W \cup B$, such that $\{v_1 - v_0, v_2 - v_0, \cdots, v_l - v_0, h_1, \cdots, h_{n-1-l}\}$ is linearly independent. Then f and g are determined, up to a constant multiple, by:

$$\begin{aligned} &g = f \cdot v_0, \\ &f \cdot (v_i - v_0) = 0, \qquad i = 1, 2, \cdots, l, \\ &f \cdot h_i = 0, \qquad j = 1, 2, \cdots, n-1-l. \end{aligned}$$

We first show that, in the two cases of interest, the vertices and extreme rays of CH $(S(z))$ are integer vectors whose coefficients are small. In the zero–one case this is especially simple: the vertices are zero–one vectors, and there are no extreme rays. In the case where z is the binary encoding of an integer matrix A and a vector b, and $S(z) = \{x \mid Ax \leqq b, x \text{ integer}\}$, then the extreme rays of $S(z)$ coincide with those of the corresponding linear program (unless, of course, $S(z) = \varnothing$). Therefore each extreme ray of $S(z)$ is a row of A; hence each of its coefficients is of absolute value $\leqq 2^{|z|}$. As for the vertices, we can rely on a result which was independently discovered recently by several authors: [BT], [Co], [GS], [KM], [Pap2]. There is a polynomial $q(\cdot)$ such that every component of every vertex of $\{x \mid Ax \leqq b, x \text{ integer}\}$ is of absolute value $\leqq 2^{q(s)}$, where $s = \Sigma_s |\log(1 + |t|)]$. Here t ranges over all entries of A and b; thus, $s \sim |z|$.

Now we are ready to show that all coefficients of f and g are suitably small. Recall that f satisfies $H \cdot f = 0$, where H is an $(n-1) \times n$ matrix of rank $n-1$; each row of H is either of the form $v_1 - v_0$ or of the form h_j. Without loss of generality, assume that the first $n-1$ columns of H are linearly independent, and write

$H = C \vdots d$, where C is a nonsingular $(n-1) \times (n-1)$ matrix, and d is a column vector. Then f is determined by

$$\begin{matrix} f_1 \\ \vdots \\ f_n - 1 \end{matrix} + C^{-1} d f_n = 0.$$

By Cramer's rule,

$$(C^{-1})_{ij} = \frac{(-1)^{i+j} \Delta_{ji}}{|C|}$$

where Δ_{ji} is the $j-i$ minor of C. Hence, we can take f to be the following integer vector (or its negative).

$$f_i = \sum_{j=1}^{n-1} (-1)^{i+j} \Delta_{ji} d_j, \qquad i = 1, 2, \cdots, n-1, \quad f_n = |C|.$$

It follows that each component of f or g has absolute value $\leqq (2nu)^n$, where u is the largest absolute value of an entry in a vertex or extreme ray. And the result that, for a suitable polynomial p, each coefficient $\leqq 2^{p(|z|+n(z))}$ now follows from the bounds derived earlier on the coefficients of vertices and extreme rays. □

3. The computational complexity of small facial descriptions. The following theorem is our first main result.

THEOREM 1. *Let C be a c.o.p. If C has a small facial description $F(C) \in$ NP then $D(C) \in$ co-NP.*

Proof. (*Only if*). Assuming as given a nondeterministic polynomial-time algorithm for recognizing the triples $\langle z, f, g\rangle \in F(C)$, we give a nondeterministic polynomial-time algorithm for recognizing the complement of $D(C)$. □

ALGORITHM A.

Step i. If the input is not of the form $\langle z, c, k\rangle$, where $z \in L$, $c \in Z^{n(z)}$ and $k \in Z$, then accept the input and halt;

Step ii. Generate nondeterministically an $n \times n$ matrix $F = (f_{ij})$ of integers having absolute value $\leqq 2^{p(|z|+n(z))}$ and an n-vector g of integers having absolute value $\leqq 2^{p(|z|+n(z))}$;

Step iii. Apply the nondeterministic polynomial-time recognition algorithm for $F(C)$ to verify that each triple $\langle z, (f_{i1}, f_{i2}, \cdots, f_{in}), g_i\rangle$ is in $F(C)$;

Step iv. Verify that F is nonsingular and (in polynomial time) solve the system, $y^T F = c$;

Step v. Verify that $y \geqq 0$ and $y^T g < k$.

Algorithm A clearly runs in polynomial time. To prove that it accepts the complement of $D(C)$, we note the equivalence of the following statements:

(i) $\langle z, c, k\rangle \notin D(C)$;

(ii) the program

$$\begin{array}{ll} \max & c \cdot x \\ \text{subject to} & x \in CH(S(z)) \end{array}$$

has optimal value $<k$;

(iii) the program

$$\text{(I)} \qquad \begin{array}{ll} \max & c \cdot x \\ \text{subject to} & f \cdot x \leqq g, \langle z, f, g\rangle \in F(C) \end{array}$$

has optimal value $<k$;

(iv) the dual of program (I) has optimal value $<k$;

(v) the dual of program (I) has a basic feasible solution of value $<k$;

(vi) there exists an $n(z) \times n(z)$ matrix F and an $n(z)$-vector g such that

$$\langle z, (f_{i1}, f_{i2}, \cdots, f_{in(z)}), g_i\rangle \in F(C), \qquad i = 1, 2, \cdots, n(z)$$

and the system

$$y^T F = c$$

has a unique nonnegative solution y such that

$$y^T g < k. \qquad \square$$

Following a suggestion of Andy Yao, we observe that the *converse* of Theorem 1 is true in the case of 0–1 c.o.p.s (or c.o.p.s with provably small, in terms of z, vertices). It does not necessarily hold in the general case.

The following corollary constitutes our evidence that computationally tractable facial descriptions for NP-complete combinatorial optimization problems are unlikely to exist.

COROLLARY 1. *If $F(C)$ is a small facial description of an* NP-*complete c.o.p. and* $F(C) \in$ NP, *then* NP = co-NP.

Proof. The NP-complete language $D(C)$ is in co-NP. Since the complement of every language in NP is reducible to the complement of $D(C)$, it follows that co-NP $\subseteq$ NP and NP = co-(co-NP) $\subseteq$ co-NP. $\square$

Our approach can also prove a slightly different kind of result. Let $F(C)$ be a collection of *valid inequalities* for C, i.e., each element of $F(C)$ is a triple $\langle z, f, g\rangle$, such that $f \cdot x \leqq g$ holds for every $x \in S(z)$. Suppose that $F(C) \in$ NP. Call an instance $\langle z, c\rangle$ of C *bad* for $F(C)$ if the optimum solution for the instance is *not* optimum for

$$\begin{aligned} &\max c \cdot x \\ &\text{subject to} \quad f \cdot x \leqq g, \langle z, f, g\rangle \in F(C). \end{aligned}$$

Let I be a subset of the set of instances of C such that $I \in P$ but $\{\langle z, c, k\rangle | \langle z, c\rangle \in I$ and $\langle z, c, k\rangle \in D(C)\}$ is NP-complete.

COROLLARY 2. *Let I and $F(C)$ be as above. If* NP $\neq$ co-NP *then I contains infinitely many instances that are bad for $F(C)$.*

Example. Let F(TSP) be the ingenious partial characterization of the facets of the traveling-salesman polytope given in [GP2]. Call an instance of the traveling-salesman problem *Euclidean* if the cost vector can be realized as the L_2 distances of a finite set of points in the plane. Then, since the Euclidean restriction of the traveling-salesman problem (TSP) is NP-complete [Pap1] we conclude that, unless NP = co-NP, there exist infinitely many bad Euclidean instances of the TSP. Similarly, since the Hamiltonian circuit problem is NP-complete, we can claim that, unless NP = co-NP, there exist infinitely many bad instances in which each component of c is 0 or 1.

4. The complexity of generators. The famous ellipsoid method [Kh] for finding a point within a full-dimensional convex body K in R^n computes a sequence

$$\{(x_k, E_k), k = 0, 1, \cdots\},$$

where, for all k,

(i) E_k is an ellipsoid in R^n and x_k is the center of E_k;

(ii) $E_{k+1} \cap K \supseteq E_k \cap K$;

(iii) $\text{vol}(E_{k+1}) \leqq 2^{-1/(2n+1)} \text{vol}(E_k)$.

At each iteration, the ellipsoid method calls on a *separating hyperplane subroutine* for K. Such a subroutine, when given a point $y \in R^n$, either determines that $y \in K$, or else produces a vector f such that

$$f \cdot y > \max_{x \in K} f \cdot x.$$

A version of the ellipsoid method, using finite-precision arithmetic and incorporating a perturbation technique to deal with polyhedra of less than full dimension, solves the general linear programming problem in polynomial time. In this application the separating hyperplane subroutine is trivial, since K is defined by an explicit list of linear inequalities.

It is also of interest to apply the ellipsoid method in cases where K is not defined by such an explicit list. Such cases arise in nonlinear programming, where K is a nonpolyhedral convex set, and in certain combinatorial applications, where K is a polyhedron with a huge number of facets. In such cases the separating hyperplane subroutine cannot be based on an explicit list of inequalities; it must be based on an algorithm.

Given a c.o.p. C, a *generator of violated inequalities* is an algorithm $G(C)$ which accepts as input pairs of the form $\langle z, p\rangle$, where $z \in L$ and $p \in R^{n(z)}$. The output of $G(C)$ is as follows:

if $p \in CH(S(z))$ **then** "O.K."
else a pair (f, g) such that $f \in Z^{n(z)}$
$g \in Z, f \cdot p > g$ and, for all $x \in S(z), f \cdot x \leqq g$.

Associated naturally with any generator $G(C)$ is the following facial description $F_G(C)$:

$$F_G(C) = \{\langle z, f, g\rangle | \text{ for some } p, G(C) \text{ has input } (z, p) \text{ and output } (f, g)\}.$$

The generator $G(C)$ is called a *small generator* if $F_G(C)$ is a small facial description.

THEOREM 2. *Let C be a c.o.p. and let $G(C)$ be a small generator of violated inequalities for C. If $G(C)$ runs in polynomial time, then $D(C) \in P$.*

The remainder of this section is devoted to the proof of Theorem 2.

COROLLARY 3. *If C is* NP*-complete, then it has no small polynomial-time generator unless P=*NP.

The chief technical difficulties in the proof of Theorem 2 arise because the polyhedra CH $(S(z))$ associated with a c.o.p. C are not necessarily full-dimensional; i.e., CH $(S(z))$ may lie in a $(n(z)-1)$-dimensional flat of $R^{n(z)}$.

To prove Theorem 2 we show that, using a small polynomial-time generator $G(C)$, one can test in polynomial time whether $\langle z, c, k\rangle \in D(C)$. This is equivalent to testing whether the system

$$c \cdot x \leqq k, \qquad f \cdot x \leqq g, \langle z, f, g\rangle \in F_G(C) \tag{4}$$

is feasible.

Since the solution set of (4) may be nonempty but of less than full dimension, we apply a perturbation which converts (4) to a system of strict inequalities. The solution set of this system will, of course, be full-dimensional whenever it is not empty. Let $p(\cdot)$ be a polynomial such that, for every $\langle z, f, g\rangle \in F_G(C)$, each component of f, g has absolute value $\leqq 2^{p(|z|+n(z))}$. For a fixed z, let $t = (n(z)+1)^2 \cdot (p|z|+n(z)+1)+\Sigma_j(\lceil \log_2 c_j\rceil+1)+(\lceil \log_2 k\rceil+1))$; t is an upper bound on the number of binary digits needed to write down any affinely independent subsystem of the system (4).

LEMMA 2. *The system* (4) *is feasible if and only if the systems* (5) *and* (5.1) *are both feasible, where*

$$(5)\qquad \begin{aligned} &c\cdot x > k-2^{-t},\\ &f\cdot x > g+\varepsilon\|f\|, \quad \langle z,f,g\rangle\in F_G(C),\\ &-2^t\leqq x_j\leqq 2^t,\qquad j=1,2,\cdots,n(z),\end{aligned}$$

and (5.1) *is* (5) *with the first inequality replaced by* $c\cdot x > k-\frac{1}{2}2^{-t}$.

Here by $\|f\|$ *we denote the* L_2*-norm of the vector* f, *and* $\varepsilon = 2^{-(2n(z)+2)t}$. *Notice that* $fx < g+\varepsilon\|f\|$ *is satisfied by those points* x *that have Euclidean distance less than* ε *from some point* x' *satisfying* $fx'\leqq g$.

LEMMA 3. *If the system* (5) *is feasible, then the set* T *of feasible points has volume at least* $\varepsilon^{n(z)}$.

These lemmas are similar to results given in [AS] and [Kh]; we omit the proofs.

Next we present a procedure which, when presented with a point p, terminates, after a polynomial-bounded number of calls on the generator $G(C)$, with one of the following two outcomes:

(a) a point q is found which satisfies (5), or

(b) a hyperplane H is found which separates p from the feasible set for (5.1).

The procedure is designed to circumvent the difficulty that, when an input (y,z) is presented to $G(C)$, the result may be a pair (f,g) such that $g+\varepsilon\|f\| > f\cdot y > g$; i.e., a hyperplane which separates y from the feasible set for (4), but not from the (larger) feasible set for (5.1).

REPEATED PROJECTION PROCEDURE

if $c\cdot p\leqq k-\frac{1}{2}2^{-t}$ **then** $H := \{x \mid c\cdot x = k-\frac{1}{2}2^{-t}\}$
else
  $p_0 := p;\ \ j := 0;$
  while neither q nor H has been determined **do** present (p_j, z) to $G(C)$;
    if $G(C)$ returns "O.K." **then** $q_j := p_j$;
    if $G(C)$ returns (f_j, g_j) **then**
      if $f_j\cdot p_j \geqq g_i+\varepsilon\|f\|$ **then** $H := \{x \mid f_j\cdot x = g_j+\varepsilon\|f_j\|\}$;
      else
      $H_j := \{x \mid f_j\cdot x = g_j\}$;
      $p_{j+1} :=$ the closest point to $p_j \in \bigcap_{i=0}^{j} H_i$;
      $j := j+1$.

LEMMA 4. *The Repeated Projection Procedure has the following properties*:

(i) *it terminates after at most* $n(z)+1$ *calls on the generator* $G(C)$;

(ii) *if it determines a point* q, *then* q *satisfies* (5);

(iii) *if it determines a hyperplane* H, *then* H *separates* p *from the feasible set for* (5.1).

For the proof of this result we require two preliminary lemmata.

LEMMA 5. *Let* r *be a rational point with denominators bounded by* 2^t *and let* H *be a small hyperplane such that* $r\notin H$. *Then the distance from* r *to* H *is at least* $2^{-(n(z)+1)t}$.

Proof. This distance is $(fr-g)/\|f\|$, where $H=\{x: fx=g\}$. The numerator is a positive integer, whereas the denominator is at most $\|f\|2^{n(z)t}\leqq\sqrt{n}\cdot 2^{p(|z|+n(z))}\cdot 2^{n(z)}\leqq 2^{(n(z)+1)t}$. □

Recall that a set of hyperplanes is called *affinely independent* if the associated set of normal vectors is linearly independent. A set of k affinely independent hyperplanes has a nonempty intersection, which has dimension $n-k$.

COROLLARY 4. *Let r be a point in the intersection of affinely independent small hyperplanes $\{H_j\}_{j=1}^m$ and let H be a small hyperplane such that $H \cap \bigcap_{j\leqq m} H_j = \emptyset$. Then the distance from r to H is at least $2^{-(n(z)+1)t}$.*

Proof. The flat $\bigcap_{j\leqq m} H_j$ has at least one rational point r' with denominators at most 2^t, and the distances from r and r' to H are equal. Apply Lemma 5. □

LEMMA 6. *Let $H_0, \cdots, H_{j+1}$ be affinely independent small hyperplanes, let $r \in \bigcap_{i\leqq j} H_i$, and let the distance from r to H_{j+1} be δ. Then the distance from r to $\bigcap_{i\leqq j+1} H_i$ is at most $(2^t-1)\delta$.*

Proof. Without loss of generality $H_{j+1} = \{x : x_1 = 0\}$, and $r = (\delta, r_2, \cdots, r_{n(z)})$. Also without loss of generality, suppose that the columns $2, 3, \cdots, j+1$ of the matrix whose rows are $f_1, \cdots, f_j$ (where $H_i = \{x : f_i x = g_i, i = 1, \cdots, j\}$) are independent. The point r is thus obtained by setting $x_k = r_k$ for $k = j+2, \cdots, n(z)$, $x_1 = \delta$ and solving for the remaining x's. Similarly, one point r' in $\bigcap_{i\leqq j+1} H_j$ is obtained by setting $x_n = r_k$ for $k = j+2, \cdots, n(z)$, $x_1 = 0$ and solving for the remaining x's. It is easy to see that $\|r - r'\| \leqq \delta 2^{p(|z|+n(z))} \cdot n^2(z) \cdot b$, where b is the largest element in absolute value of the inverse of the nonsingular matrix of the columns 2 through $j+1$ of the matrix whose rows are $f_1, \cdots, f_j$. Thus $\|r - r'\| \leqq (2^t - 1)\delta$. □

Proof of Lemma 4. Suppose the repeated projection procedure determines a sequence of points $p_0, p_1, \cdots, p_{m+1}$ and a sequence of hyperplanes $H_0, H_1, \cdots, H_m$. Let $d(p, H)$ denote the distance from point p to hyperplane H. We shall prove that

(a) $d(p_j, H_j) \leqq 2^{jt}\varepsilon, j = 0, 1, \cdots, m$;

(b) the set $\{H_j, j = 0, 1, \cdots, m\}$ is affinely independent;

(c) $\|p - p_j\| \leqq \varepsilon(2^{jt} - 1), j = 0, 1, \cdots, m+1$.

Let S_j be the statement

$$d(p_j, H_j) \leqq 2^{jt}\varepsilon$$

and

$$\{H_0, H_1, \cdots, H_j\} \text{ is affinely independent}$$

and

$$d(p, p_{j+1}) \leqq \varepsilon(2^{(j+1)t} - 1).$$

It will suffice to prove by induction that S_j holds for $j = 0, 1, \cdots, m$. This is certainly true for $j = 0$. For the induction step, we show that S_{j+1} follows from S_j:

(a) $d(p_j, H_j) \leqq d(p, p_j) + d(p, H_j) \leqq \varepsilon(2^{jt} - 1) + \varepsilon = 2^{jt}\varepsilon$.

(b) Suppose $\{H_0, H_1, \cdots, H_j\}$ were not affinely independent. We know from Corollary 4 that $d(p_j, H_j) \geqq 2^{-(n(z)+1)t}$; this contradicts (a), since $2^{-(n(z)+1)t} > 2^{jt}\varepsilon$.

(c) Since $d(p, p_{j+1}) \leqq d(p, p_j) + d(p_j, p_{j+1})$ and (by Lemma 5) $d(p_j, p_{j+1}) \leqq (2^t - 1) d(p_j, H_j)$ we have

$$d(p, p_{j+1}) \leqq \varepsilon(2^{jt} - 1) + (2^t - 1)2^{jt}\varepsilon = (2^{(j+1)t} - 1)\varepsilon.$$

Now we are ready to verify the three properties that comprise Lemma 4.

Proof of (i). It is not possible to have more than $n(z)$ affinely independent hyperplanes in $R^{n(z)}$.

Proof of (ii). We must show that q satisfies (5). Since $G(C)$ on input (q, z) returns "O.K.", the only thing to be proven is that $c \cdot q > k - 2^{-t}$. We know that $q = p_j$ for some j. Thus $\|q - p\| \leqq (2^{jt} - 1)\varepsilon$ and $c \cdot p > k - \frac{1}{2} \cdot 2^{-t}$. Hence $c \cdot q > k - 2^{-t}$.

Proof of (iii). This is immediate from inspection of the Repeated Projection Procedure. □

Thus, given the center p of the ellipsoid $E(p, A)$, we can in polynomial time either determine a feasible point, or isolate a violated inequality of (5). Then using this violated inequality a new pair p', A' is computed. The ellipsoid $E(p', A')$ has smaller volume than $E(p, A)$, but does include those points in $E(p, A)$ which satisfy the inequality violated at p. Following the proof in [AS], the following convergence result is obtained.

LEMMA 7. *There are a constant c and a polynomial π such that, if* (5) *is feasible, then a feasible solution will be found within $\pi(n, t)$ iterations. This is true even if intermediate results are kept to only cnt bits of precision.*

Our variant of the ellipsoid method tests feasibility of (5), and hence membership of $\langle z, c, k\rangle$ in $D(C)$, in polynomial time. This completes the proof of Theorem 2. □

5. Optimization versus separation. When the ellipsoid method is applied to the linear programming problem, the task of finding a hyperplane separating an infeasible point y from the set of feasible solutions can be carried out easily by examining the given list of linear inequalities, and identifying one which is violated by y. The central idea in the previous section was that the ellipsoid method can also be applied when an efficient generator of violated inequalities, instead of an explicit list, is provided. The same observation was independently made by Grotschel, Lovasz and Schrijver [GLS] and Padberg and Rao [PR].

The former paper is concerned with the *optimization version* of c.o.p. C, which, in our formalism, can be defined as follows: "Given an instance $\langle z, c\rangle$ of C, compute the element $x \in S(z)$ which maximizes $c \cdot x$." The fact that in our formalism the feasible solutions are integer vectors is not too important here. Rationals with exponentially bounded denominators would also support our analysis. Translated into our terminology, the result of [GLS] states that, under certain assumptions, there is a polynomial-time algorithm for the optimization version of C if and only if there is a polynomial-time small generator of violated inequalities for C. Several beautiful algorithmic results are derived using this approach.

All the theorems in [GLS] require the hypothesis that the convex hull of $S(z)$ be full-dimensional, i.e., not contained in any $(n(z)-1)$-dimensional flat. However, the repeated projection subroutine of the previous section can be combined with the method of [GLS] to prove that for all c.o.p.s C (full-dimensional or not) the optimization version is polynomial-time equivalent with the problem of generating small violated inequalities.

THEOREM 3. *Let C be a c.o.p. that has a small facial description. Then the following are equivalent*:

(a) *The optimization problem for C is solvable in polynomial time.*

(*b*) *C has a small polynomial-time generator.*

Proof. To prove that (b)→(a), we use a variant of the method described in [GLS] for solving the optimization problem associated with a full-dimensional c.o.p. Given the input $\langle z, c\rangle$, we first perturb c slightly so that the optimum solution is now unique. We construct a sequence $E_0 = (p_0, A_0)$, $E_1 = (p_1, A_1), \cdots$ of ellipsoids, starting with the sphere centered at the origin and having radius t defined in the previous section. In the kth step there will be an ellipsoid E_k, which includes all points s of the convex hull of $S(z)$ for which $c \cdot x$ is at least as large as the best feasible value found so far; that is, $E_k \supseteq \{x \in \mathrm{CH}(S(z)): c'x \geqq c'p_j,\ j \leqq k,\ p_j \in \mathrm{CH}(S(z))\}$. At each iteration of the algorithm, we present z, p_k to $G(C)$. If the center p_k of E_k is infeasible, we obtain a small violated inequality $\langle f, g\rangle$, and we let $\frac{1}{2}E_k = \{x \in E_k : f \cdot x \leqq f \cdot p_k\}$. On the other hand, if $G(C)$ returns "O.K.", signaling that p_k is feasible, then we let $\frac{1}{2}E_k =$

$\{x \in E_k : c \cdot x \geqq c \cdot p_k\}$. This has the effect of cutting away those points in E_k where the objective function has a smaller value than at the feasible point p. Next we define E_{k+1} in such a way that $E_{k+1} \supseteq \frac{1}{2}E_k$, and $\mathrm{vol}(E_{k+1}) \leqq 2^{-1/(2n+1)} \mathrm{vol}(E_k)$. If CH $(S(z))$ is full-dimensional, then it is not hard to argue that, after a polynomial-bounded number of such iterations, the center of the ellipsoid is bound to be very close to the optimal solution, which can then be recovered by "rounding". For details see [GLS].

If we do not assume that CH $(S(z))$ is full-dimensional, then the last part of the argument breaks down, and the algorithm does not converge to the optimum. We can, however, apply the algorithm to the polytope $\mathrm{CH}_\varepsilon(S(z)) = \{x \in R^{n(z)} : fx \leqq g + \varepsilon\|f\| \text{ for all } \langle f, g\rangle \in F(G)\}$, for some appropriately small $\varepsilon > 0$. Notice that $\mathrm{CH}_\varepsilon(S(z))$ is guaranteed to be full-dimensional for nonempty $(S(z))$. This, however, creates a familiar problem: $F(G)$ may now return, on input $\langle z, p_k\rangle$, an inequality $\langle f, g\rangle$ for which

$$g < f \cdot p_k \leqq g + \varepsilon\|f\|.$$

It is at this point that our repeated projection procedure of the previous section is needed. By applying this procedure either we find a hyperplane separating p_k from $\mathrm{CH}_\varepsilon(S(z))$—in which case we continue the ellipsoid algorithm—or we locate a feasible point p'_k such that $\mathrm{dist}(p_k, p'_k) \leqq 2\varepsilon^{nt}$. In the latter case, we replace the ellipsoid (p_k, E_k) by another ellipsoid (p'_k, E'_k) centered at the feasible point p'_k and having the following properties.

$$E_k \subseteq E'_k, \qquad \mathrm{vol}(E'_k) \leqq \mathrm{vol}(E_k) 2^{1/(2n+1)}.$$

From then on we continue the ellipsoid method by defining $\frac{1}{2}E'_k = \{x \in E'_k : c'x \geqq c'p'_k\}$, and constructing E_{k+1} so that it includes $\frac{1}{2}E'_k$, and also satisfies $\mathrm{vol}(E_{k+1}) \leqq 2^{-1/2(2n+1)} \mathrm{vol}(E'_k) \leqq 2^{-1/(2n+1)} \mathrm{vol}(E_k)$. Hence, we still have a geometric decrease of the volumes of the E_ks.

It remains to describe E'_k. The positive-definite matrix of E'_k is that of E_k, only with all coefficients multiplied by $1+\delta$, where δ is also an appropriately small positive real, to be fixed later. Now the smallest axis of E_k is shown in [GLS, (proof of Lemma 2.1)] to be at least $4^{t-(k+1)}$. Hence, if we choose δ and ε so that

$$4^{t-(k+1)}\delta \geqq 2^{nt}\varepsilon \geqq \mathrm{dist}(p_k, p'_k),$$

we can guarantee that $E'_k \supseteq E_k$. Also, if we choose δ so that

$$(1+\delta)^n \geqq 2^{1/2(2n+1)},$$

we have that

$$\mathrm{vol}(E'_k) = (1+\delta)^n \mathrm{vol}(E_k) \leqq \mathrm{vol}(E_k) 2^{1/2(2n+1)}.$$

It is quite easy to see that

$$\delta = 2^{1/2n(2n+1)} - 1$$

and

$$\varepsilon = \delta \cdot 2^{-nt-I},$$

where I is the number of iterations required, are appropriate choices. It follows then from the argument of [GLS] and that of the previous section that the algorithm sketched above solves the optimization version of C in polynomial time, assuming that $G(C)$ operates within polynomial time.

To conclude the proof we shall prove that (a)→(b). To do so, we show how to construct a small polynomial-time generator for C, given a polynomial-time algorithm to solve the optimization problem for C.

Given as input a point $y \in R^{n(z)}$, such a generator must find an $n(z)$-vector h such that $h'x < h'y$ for all $x \in \mathrm{CH}(S(z))$, or else determine that no such h exists. Let VERT $(S(z))$ denote the set of vertices of CH $(S(z))$. The desired vector h exists if and only if the following linear program has either an optimal value >1 or an unbounded objective function.

$$\text{(6)} \qquad \begin{array}{l} \max h'y \\ \text{subject to} \quad h'v \leqq 1, \qquad v \in \mathrm{VERT}(S(z)). \end{array}$$

To convert (6) to a bounded linear program, we simply intersect its feasible region with a suitable hypercube. The resulting program is of the form

$$\text{(7)} \qquad \begin{array}{ll} \max h'y & \\ \text{subject to} & h'v \leqq 1, \quad v \in \mathrm{VERT}(S(z)) \\ & -A \leqq h_i \leqq A, \quad i = 1, 2, \cdots, n(z). \end{array}$$

With an appropriate choice of A, it will follow that a separating hyperplane exists if and only if the optimal value of (7) is greater than 1.

It remains to choose A. Since the c.o.p. C has a small facial description, Cramer's rule shows that there is a polynomial such that each component of each $v \in$ VERT $(S(z))$ is an integer of absolute value $\leqq 2^{q(|z|+n(z))}$. By a second application of Cramer's rule, there is a polynomial q' such that each component of each extreme point of (6) has absolute value $\leqq 2^{q'(|z|+n(z))}$. Then an appropriate choice of A is $2 \cdot 2^{q'(|z|+n(z))}$.

By the method described in the first half of the present proof, we can solve (7) in polynomial time, provided a small generator of violated inequalities exists for (7). Such a generator is easily constructed. Given a trial solution h, check whether the explicit bounds on the components of h are satisfied. If so, use the optimization algorithm for the c.o.p. C to find an optimal extreme point w for the problem

$$\max h'z, \qquad z \in \mathrm{CH}(S(Z)).$$

If $h'w \leqq 1$ then h is feasible for (7). Otherwise, $h'w \leqq 1$ is the required violated inequality. □

REFERENCES

[AS] B. ASPVALL AND R. STONE, *Khaciyan's linear programming algorithm*, Computer Science Department Report STAN-CS-776, Stanford Univ., Stanford, CA, November, 1979.

[Ba] E. BALAS, *Facets of the knapsack polytope*, Mathematical Programming 8 (1975), pp. 146–164.

[BT] I. BOROSH AND L. B. TREYBIG, *Bounds on positive solutions of linear diophantine equations*, Proc. Amer. Math. Soc., 55 (1976), p. 299.

[Ch] V. CHVATAL, *Edmonds polytopes and weakly Hamiltonian graphs*, Mathematical Programming, 5 (1973), pp. 29–40.

[Co) S. A. COOK, *A short proof that the linear diophantine problem is in* NP, unpublished, 1976.

[DFJ] G. B. DANTZIG, D. R. FULKERSON AND S. M. JOHNSON, *Solutions of a large-scale traveling-salesman problem*, Oper. Res., 2 (1954), pp. 393–410.

[Ed1] J. EDMONDS, *Maximum matching and a polyhedron with* 0–1 *vertices*, J. Res. National Bureau of Standards, B 69 (1965), pp. 125–130.

[Ed2] ———, *Paths, trees and flowers*, Canad. J. Math., 17 (1965), pp. 449–467.

[GS] J. GATHEN AND M. SIEVEKING, *Linear integer inequalities are* NP-*complete*, Manuscript, 1978.

[Gr] M. GROTSCHEL, *Polyedrische Characterisierunge Kombinatorischer Optimierungsprobleme*, Verlag Anton Hain, Berlin, 1977.

[GP1] M. GROTSCHEL AND M. W. PADBERG, *On the symmetric traveling-salesman problem I: Inequalities*, Math. Programming, 16 (1979), pp. 265–280.

[GP2] ———, *On the symmetric traveling-salesman problem II: Lifting theorems and facets*, Math. Programming, 16 (1979), pp. 281–302.

[GLS] M. L. GROTSCHEL, L. LOVASZ AND S. SCHRIJVER, *The ellipsoid method and its consequences in combinatorial optimization*, Combinatorica, 1 (1981), pp. 169–198.

[HP] S. HONG AND M. PADBERG, *On the solution of traveling salesman problems*, 9th International Symposium on Mathematical Programming, Budapest, 1976.

[KM] R. KANNAN AND C. L. MONMA, *On the computational complexity of integer programming problems*, Lecture Notes in Economics and Mathematical Systems, Springer-Verlag, New York, 1978.

[Kh] L. G. KHACIYAN, *A polynomial algorithm for linear programming*, Dokl. Akad. Nauk SSSR, 244:5 (1979), pp. 1093–1096 (in Russian).

[Ma] J. F. MAURRAS, *Some results on the convex hull of the Hamiltonian cycles of symmetric complete graphs* in Combinatorial Programming: Methods and Applications, B. Roy, ed., Reidel, Boston, 1975.

[NT] G. L. NEMHAUSER AND L. E. TROTTER, *Properties of vertex packing and independence systems polyhedra*, Math. Programming, 6 (1974), pp. 48–61.

[Pa] M. W. PADBERG, *On the facial structure of set packing polyhedra*, Math. Programming, 5 (1973), pp. 199–215.

[PR] M. W. PADBERG AND M. R. RAO, *Minimum cut-sets and B-matchings*, to appear.

[Pap1] C. H. PAPADIMITRIOU, *The Euclidean traveling salesman problem is* NP-*complete*, Theor. Comput. Sci., 4 (1977), pp. 237–244.

[Pap2] ———, *On the complexity of integer programming*, J. Assoc. Comput. Math., to appear.

[PS] C. H. PAPADIMITRIOU AND K. STEIGLITZ, *Combinatorial Optimization Algorithms and Complexity*, Prentice Hall, Englewood Cliffs, NJ, 1981.

[Wo] L. A. WOLSEY, *Facets of a linear inequality in* 0–1 *variables*, Math. Programming, 8 (1975), pp. 165–178.

SIAM J. COMPUT.
Vol. 11, No. 4, November 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1104-0003 $01.00/0

EVALUATION OF ARITHMETIC EXPRESSIONS WITH ALGEBRAIC IDENTITIES*

TEOFILO GONZALEZ† AND JOSEPH JA'JA'‡

Abstract. We consider the problem of evaluating arithmetic expressions under a set of algebraic laws including the distributive law. An arithmetic expression can be represented by a dag and our problem is to find an equivalent dag with the fewest number of interior nodes. We attack the case when it is possible to eliminate common subexpressions and transform the dag into a tree; efficient algorithms to handle different cases of this problem are developed. These algorithms are based on the following strategy: we first transform the dag into a tree, assuming that such a transformation is possible, and we later check to see whether the tree and the given dag are indeed equivalent.

Key words. evaluation of arithmetic expressions, code generation, algorithms, dags

1. Introduction. Several interesting results concerning the problem of code generation for arithmetic expressions have been established by several authors. Extending the work of Anderson [A], Nakata [N], and Redziejowski [R], Sethi and Ullman [SU] have presented an efficient algorithm to generate minimal length codes for a special type of arithmetic expressions, namely those expressions with no common subexpressions. Aho and Johnson [AJ] have found a more general algorithm which allows general addressing features such as indirect addressing, but again restricting themselves to the same type of expressions. The case of arbitrary expressions has been proven to be difficult in a precise sense, i.e., it is NP-complete, even for the class of one-register machines with no algebraic identities allowed (Bruno and Sethi [BSe]). Aho et al. [AJU] have shown that the problem remains NP-complete for dags whose shared nodes are leaves or nodes at level one and have developed heuristic algorithms to generate good codes.

The effect of algebraic laws on code generation has received little attention in the literature. Sethi and Ullman [SU] have discussed the case where some of the operators of an expression tree are associative and commutative, and Breuer [B] used the distributive law to factor polynomials in a manner similar to that of Horner's algorithm. When certain algebraic transformations apply for an arithmetic expression A, we are not required to generate codes for A, but we may generate codes for any equivalent expression A' obtained by successive applications of the algebraic laws. Since the number of arithmetic operations may then vary, the optimality criterion of generated codes should depend on the number of arithmetic operations as well as on the code length. In this paper, we assume that the distributive law holds and consider the problem of minimizing the number of arithmetic operations for single arithmetic expressions which involve only addition and multiplication. We also assume that addition is commutative and associative and that multiplication is associative. This problem has been shown to be NP-hard [GJ1], [GJ2] even for expressions of degree 2 (degree of the arithmetic expression when viewed as a polynomial) and whose

* Received by the editors November 15, 1978, and in final revised form December 15, 1981. Preliminary versions of this paper appear as Technical Reports CS-80-4 and CS-78-13 at Pennsylvania State University.

† Programs in Mathematical Sciences, The University of Texas, Dallas, Texas 75080. The work of this author was supported in part by the National Science Foundation under grant MCS 77-21092.

‡ Computer Science Department, Pennsylvania State University, University Park, Pennsylvania 16802. The work of this author was supported in part by the National Science Foundation under grants MCS 78-06118 and MCS 78-27600.

corresponding graphs are leaf dags. However, in the case when common subexpressions can be eliminated, and the dag can thus be transformed into a tree, we develop efficient algorithms for the following types of dags:

a) the dag is a leaf dag (Theorem 5.15),

b) no term in the expression is repeated (Theorem 5.10),

c) the degree of the expression is bounded by some fixed constant (Theorem 5.16),

d) the level of sharing in the dag is bounded by some fixed constant (comment after Theorem 5.16).

2. Basic definitions. An arithmetic expression can be conveniently represented by a *directed acyclic graph*, referred to as a *dag*, in the same way basic blocks in code optimization are represented (Aho and Ullman [AU]). A dag has an interior node for each operation whose operands are the children of the node; the leaves of the dag represent initial values (variable names). For example, the arithmetic expression $A=(a*b+a*c)+(d*b+d*c)$ can be represented by the dag:

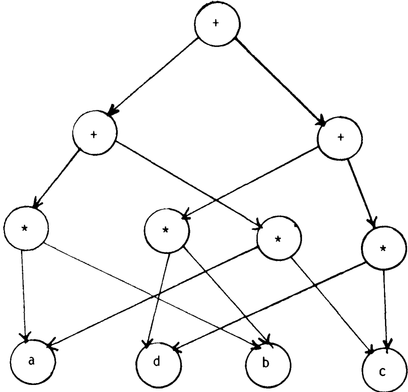


FIG. 2.1

The order of the children of an interior node is important; the leftmost child represents the first operand and the rightmost child represents the last operand.

We will restrict our attention to the following class of objects. Let Σ be a countable set of variable names and let $\theta=\{+, *\}$ be the set of binary operators on Σ such that the following laws hold:

(i) + and $*$ are associative, i.e.,

$$(a+b)+c=a+(b+c),$$

$$(a*b)*c=a*(b*c), \quad \text{for all } a, b, c \in \Sigma;$$

(†) (ii) + is commutative, i.e.,

$$a+b=b+a, \quad \text{for all } a, b \in \Sigma;$$

(iii) $*$ is distributive with respect to +, i.e.,

$$a*(b+c)=a*b+a*c,$$

$$(b+c)*a=b*a+c*a, \quad \text{for all } a, b, c \in \Sigma.$$

Strictly speaking, the above laws do not necessarily hold for actual expressions because of round-off errors in finite precision arithmetic. However, there is a general feeling [JMMW] that, for a given computation, the fewer the arithmetic operations, the less the worst-case round-off errors are, in spite of the fact that there are special situations where the opposite is true.

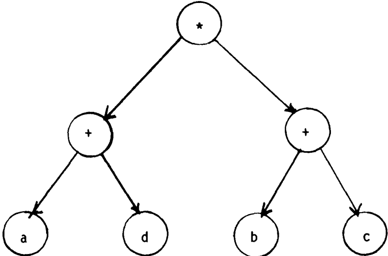
Another remark is concerned with the fact that we have not assumed that $*$ is commutative; the main reason is that the same techniques can be applied to a matrix expression to reduce the number of arithmetic operations. As an example, the expression $AB+AC+DB+DC$, where A, B, C and D are $n \times n$ matrices, is equivalent to $(A+D) \cdot (B+C)$ whose computation requires considerably fewer arithmetic operations than the original expression. This might also have some applications to code generation for parallel computers in which most of the operations are written in matrix form.

As a final remark, we note that identities such as $x+x=2x$, $x * x = x^2$ or $xy+x = x(y+1)$ do not exist.

We now define precisely the class of dags we are interested in. A *σ-dag* is a dag with a single root (i.e., a node without parents), whose interior nodes are either $+$ or $*$ from θ and whose leaves are from Σ. Note that no two leaves will represent the same element in Σ. If D is such a dag and v is a node from D, then the *expression corresponding to* v, denoted by $\exp(v)$, is defined as follows:

1) if v is a leaf, then $\exp(v) = v$,
2) else v corresponds to $\alpha \in \theta$; let v_1 and v_2 be the left and right children respectively, then $\exp(v) = \exp(v_1)\alpha \exp(v_2)$.

The expression corresponding to the dag D is just $\exp(r)$, where r is the root of D. Define two σ-dags D_1 and D_2 to be *equivalent* ($D_1 \equiv D_2$) if there exists a sequence of transformations from (†) which will transform D_1 into D_2. Given a σ-dag D, let $C[D]$ be the class of σ-dags equivalent to D; we are going to investigate the problem of finding a σ-dag $D' \in C[D]$ such that D' has the smallest possible number of interior nodes. If we define a *tree* to be a σ-dag such that none of its nodes has more than one parent, then we will attempt to design an algorithm which finds a tree $T \in C[D]$, whenever such a tree exists. In this case, we call the expression corresponding to T an *expression tree*, and we say that D is *tree-transformable*. If we consider, once again, the dag of Fig. 2.1, then it is easy to see that this dag is equivalent to the following tree:



whose evaluation requires 2 additions and one multiplication compared to 3 additions and 4 multiplications necessary to compute the dag of Fig. 2.1.

It is clear that if a tree T belongs to $C[D]$, for a σ-dag D, then T has the minimal number of interior nodes. Moreover, we can now use the algorithms already available in literature to generate corresponding minimal length codes.

Before closing this section, we make two more definitions and a comment. A *shared node* in a dag is a node with more than one parent; a *leaf dag* is a dag in which every shared node is a leaf (Aho et al. [AJU]). Let D be an arbitrary σ-dag with n total nodes, e edges, n_i interior nodes and v leaves (from Σ). It is easy to check that we always have the following relations:

$$n \geqq 2v-1, \qquad n_i \geqq v-1,$$
$$e = 2n_i = 2(n-v), \qquad \text{i.e.,} \quad e = O(n).$$

3. Motivation of the algorithm. We study, in this section, several properties of expression trees and examine some problems which are encountered in trying to develop an algorithm to transform a dag into a tree, whenever this is possible. We also develop some terminology which will be used in the subsequent sections.

Let A be an arithmetic expression with corresponding dag D; $L(A)$ or $L(D)$ will denote the set of leaves of D. $N(D)$ and $E(D)$ will represent the sets of nodes and edges in D respectively. Note that A can be written as $A = P_1 + \cdots + P_k$, where each P_i is a leaf or can be expressed as a product of arithmetic expressions. We call the P_is, the *product terms* of A or D. An arithmetic expression is in *normal form* (NF) if it is not possible to expand it using the distributive law. The expression $A_1 = (a*b+a*c)+(d*b+d*c)$ is in normal form and has $a*b$ as a product term, while $A_2 = a*(b+c)+d*(b+c)$ is not in normal form and has $a*(b+c)$ as a product term. The product terms of an arithmetic expression in normal form are called *normal terms*. $N_t(D)$ will denote the set of normal terms in the σ-dag D. Transforming a σ-dag into a normal form might correspond to an exponential growth in the number of nodes of the dag. For example, the expression $A = (x_1+x_2)*(x_3+x_4)*\cdots*(x_{2n-1}+x_{2n})$ (up to a fixed order) has a normal form whose dag has more than 2^n edges. We now define two more important terms.

DEFINITION 3.1. Given a σ-dag D, the *left factors* of D consist of the leaves which are the leftmost children of the normal terms of D.

DEFINITION 3.2. Given a σ-dag D with left factors $\{x_i\}_{i=1}^k$, a set of *right products* of D consists of a set of dags $\{P_i\}_{i=1}^k$ such that D is equivalent to the dag

$$D' = x_1 * P_1 + x_2 * P_2 + \cdots + x_k * P_k.$$

The expression $A_1 = (a*b+a*c)+(d*b+d*c)$ has the left factors $\{a, d\}$ with coresponding right products $\{(b+c), (b+c)\}$.

One could solve our problem by using the following divide-and-conquer approach:

(1) Find all the left factors and the corresponding right products of the given dag D.

(2) Recursively transform each right product into a tree.

(3) Combine the common right factors.

It may now seem that once we have efficient algorithms to implement steps (1) and (3), then we have an efficient algorithm to solve our problem. This is not the case since several of the right products could share several subexpressions, each of which will be processed several times. It is not hard to exhibit an example where the above strategy takes exponential time, given that each of steps (1) and (3) could be done in linear time.

It follows that we should avoid processing any subexpression more than once. What makes the problem harder is that two right products might be precisely the same and appear at different stages of the algorithm. On the other hand, it is not

posssible to transform a shared subexpression into a tree because there are subexpressions which are not tree-transformable and yet the expression to which they belong is tree-transformable. However, if a dag is tree-transformable, then we have the following characterization.

THEOREM 3.1. *Let* $\{x_i\}_{i=1}^{l}$ *be the set of left factors of a* σ*-dag* D *and let* $\{P_i\}_{i=1}^{l}$ *be the corresponding right products. If* D *is tree-transformable, then, for each* $i_1 \neq i_2$, *either*

(1) $L(P_{i_1}) \cap L(P_{i_2}) = \varnothing$ *or*

(2) *there exist* D_{i_1}, D_{i_2} *and* R *such that*

$$P_{i_1} \equiv D_{i_1} * R, \quad P_{i_2} \equiv D_{i_2} * R, \quad L(D_{i_1}) \cap L(D_{i_2}) = \varnothing.$$

Note that we can distinguish between (1) and (2) above quite easily by checking whether $L(P_{i_1}) \cap L(P_{i_2}) = \varnothing$ or not.

Let $\{P_{i_k}\}_{k=1}^{l}$ be a set of product terms which, we know, should overlap each other. The above characterization suggests that we solve the problem for just one P_{i_k}, say P_{i_1}, and use its right subtree to eliminate the overlap with all the other P_is. For example in the case where we only have two right products P_{i_1} and P_{i_2}, we transform (recursively) P_{i_1} into a tree and write it as a product, say

$$P_{i_1} \equiv (\cdots (Q_m * Q_{m-1}) * \cdots * Q_1), \qquad m \geqq 1.$$

Now if $L(Q_1) \cap L(P_{i_2}) \neq \varnothing$ (which should be true in this case), we somehow factor Q_1, and write P_{i_2} as $P_{i_2} \equiv R_{i_2} * Q_1$. We continue in this way until we have no more overlap; then we apply the procedure recursively to the nonoverlapping parts. We would like to emphasize one more point about this procedure: We assume that Q_1 will be a factor, and find R_{i_2}. This assumption is justfied if the dag is tree-transformable. Otherwise the procedure will construct an inequivalent tree; we will use the equivalent algorithm in § 5 to check whether a given σ-dag and a given tree are equivalent.

Let us summarize the general strategy of the algorithm. It consists of two parts:

(i) *The transformation algorithm* which proceeds and transforms the given σ-dag into a tree assuming the dag is tree-transformable;

(ii) *The equivalence algorithm* which, for a given σ-dag and a given tree, checks if they are indeed equivalent.

4. The transformation algorithm. The different parts of the transformation algorithm will be described in this section, together with the proofs of its correctness and its complexity. We start by discussing the algorithms to find the left factors and the corresponding right products of a general σ-dag D. $|N(D)|$ will denote the number of nodes in D.

The procedure to find the left factors is fairly straightforward. It is just a bottom-up labeling of the dag based on the following observation. Let LF(y) denote the set of left factors of the subdag rooted at y. Then

$$\mathrm{LF}(y) = \begin{cases} \{y\} & \text{if } y \text{ is a leaf,} \\ \mathrm{LF}(s_1) \cup \mathrm{LF}(s_2) & \text{if } y \text{ is a + node with children } s_1 \text{ and } s_2, \\ \mathrm{LF}(s_1) & \text{if } y \text{ is a * node and } s_1 \text{ is its left child.} \end{cases}$$

In order to have a linear time algorithm, we must avoid visiting nodes more than once. In order to guarantee this, we mark the nodes visited. We use a function tc$(\cdot)$ for this purpose; this function will serve another purpose in the next procedure when initialized properly by the procedure to find the left factors.

Let r be the root of a dag D. Let $\mathrm{tc}(z)=0$, for every $z \in N(D)$, and let $L=\varnothing$. When LEFT-FACTORS (r) terminates, L will denote the set of left factors of D and $\mathrm{tc}(z)$ will be equal to the size of {w| there is a call to LEFT-FACTORS (w) and (w is a + node with z as one of its children or w is a $*$ node with z as its left child)}. LC (r) and RC (r) denote respectively the left and right children of r. This notation will be used consistently throughout the paper.

```
    procedure LEFT-FACTORS (r)
    begin
    global (tc [·], L)
 1.     If tc (r) ≠ 0 then [tc (r) ← tc (r) + 1;
                           return];
 3.     case
 4.     :r is a leaf: [L ← L ∪ {r}];
 5.     :r is a * node: [call LEFT-FACTORS (LC (r))];
 6.     :r is a + node: [call LEFT-FACTORS (LC (r));
 7.                      call LEFT-FACTORS (RC (r))];
 8.     endcase
 9.     tc (r) ← 1;
10.     return
11. end LEFT-FACTOR
```

Let r be the root of a σ-dag D. The set of left factors of the σ-dag with root $y \in N(D)$ is denoted by L_y. Consider now any call to LEFT-FACTORS (y); let $L'=L$ just before the call and let $L''=L$ after the procedure LEFT-FACTORS (y) terminates, where L is the set of left factors computed by the algorithm.

LEMMA 4.1. *Let r, y, L', L'' and L_y be as defined above.*

a) *If* $\mathrm{tc}(y) \geqq 1$ *before the call to* LEFT-FACTORS (y), *then* $L_y \subseteq L'$.

b) *If* $\mathrm{tc}(y)=0$ *before the call to* LEFT-FACTORS (y), *then when the procedure terminates* $\mathrm{tc}(y)=1$ *and*

$$L''=L' \cup (L_y - L').$$

Proof. The proof for part a) follows from part b) and the one for part b) is by induction on the height of the σ-dag with root y. □

LEMMA 4.2. LEFT-FACTORS (r) *correctly finds the left factors of D, i.e.,* $L=L_r$.

Proof. The proof follows from Lemma 4.1 together with the initial condition $L=\varnothing$ and $\mathrm{tc}(y)=0$, for every $y \in N(D)$. □

The algorithm to find a set of right products of a given σ-dag D is discussed now. It is easy to design a bottom-up algorithm to do the job, but for efficiency reasons, our algorithm will process the dag top-down. Before presenting the algorithm, we introduce some notation. $\mathrm{pt}(y)$ will denote a pointer from a node y of D to a dag written as a right-hand side of an assignment statement; e.g., $\mathrm{pt}(y) \leftarrow \varnothing$ should be interpreted as a pointer from y to the empty dag. Another convention is that the assignment

$$\mathrm{pt}(N_0) \leftarrow \mathrm{pt}(N_1)\ \theta\ \mathrm{pt}(N_2),$$

where $\theta=\{+, *\}$, is understood to mean the assignment of Fig. 4.1. If one of $\mathrm{pt}(N_1)$ or $\mathrm{pt}(N_2)$ happened to be $\varnothing$, then the above statement is interpreted as $\mathrm{pt}(N_0) \leftarrow \mathrm{pt}(N_2)$ or $\mathrm{pt}(N_0) \leftarrow \mathrm{pt}(N_1)$, respectively.

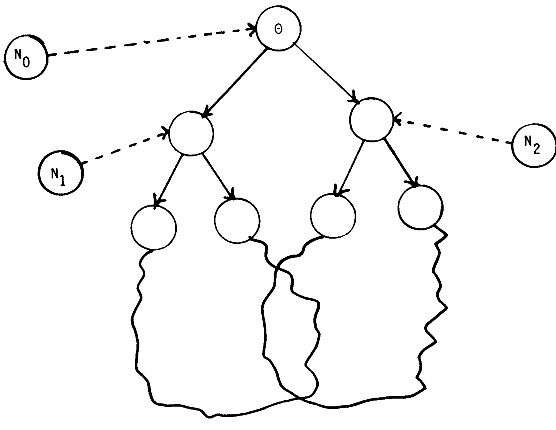


FIG. 4.1

We are now ready for the procedure. The main idea of the algorithm is a top-down traversal of the dag which makes nodes point to subdags in such a way that the left factor nodes will point to the corresponding right products. If we are visiting node y with s_1 and s_2 as its left and right children, then the following changes are made:

a) y is a + node,

$$\text{pt}(s_1) \leftarrow \text{pt}(y) \boxplus \text{pt}(s_1)$$

$$\text{pt}(s_2) \leftarrow \text{pt}(y) \boxplus \text{pt}(s_2).$$

b) y is a $*$ node,

$$\text{pt}(s_1) \leftarrow \text{pt}(s_1) \boxplus (s_2 \circledast \text{pt}(y)).$$

We cannot proceed and visit any of the children unless all of its parents have been visited. That is why we use the function tc( ) in our procedure below. In order to make inductive assertions about the algorithm, we use a function $w: N(D) \leftarrow \{0, 1\}$, which is initialized by $w(x) = 0$, for all $x \in N(D)$ unless x is the root r in which case $w(r) = 1$. This function serves no other purpose.

```
     procedure RIGHT-PRDS (r)
     begin
     global (tc[·], pt[·], w[·]);
 1.     tc (r) ← tc (r) − 1
 2.     If tc (r) ≠ 0 then [return];
 3.     case
 4.        :r is a leaf: [return];
 5.        :r is a ⊕ node:
 6.              [w(r) ← 0; w(LC (r)) ← w(RC (r)) ← 1;
 7.              pt (LC (r)) ← pt (LC (r)) ⊞ pt (r);
 8.              pt (RC (r)) ← pt (RC (r)) ⊞ pt (r);
 9.              call RIGHT-PRDS (LC (r));
10.              call RIGHT-PRDS (RC (r));
11.              return];
12.        :r is a ⊛ node:
13.              [w(r) ← 0; w(LC (r)) ← 1;
14.              pt (LC (r)) ← pt (LC (r)) ⊞ (RC (r) ⊛ pt (r));
```

15. **call** RIGHT-PRDS (LC (r));
16. **return**];
17. **endcase**
18. **end of procedure** RIGHT-PRDS (r)

To clearly illustrate the usefulness of tc [·], we introduce an example.

Example 4.1. Consider the dag of Fig. 4.2. At the end of LEFT-FACTORS (N_0), we have

$$\text{tc}(N_0) = \text{tc}(N_1) = \text{tc}(N_2) = \text{tc}(N_4) = \text{tc}(a) = 1,$$

$$\text{tc}(N_3) = 2 \quad \text{and} \quad L = \{a\}.$$

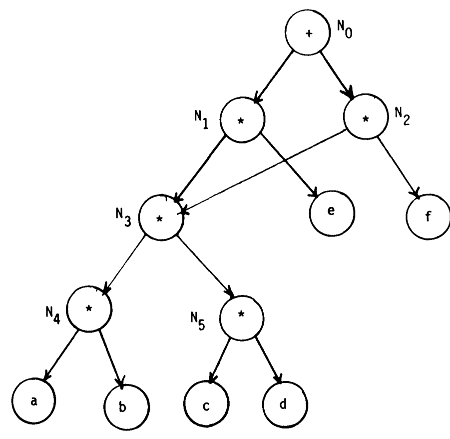


FIG. 4.2

Let us now apply RP (N_0).

Recursive call	Result
RP (N_0)	RP (N_1), RP (N_2)
RP (N_1)	pt (N_3) ← e, RP (N_3)
RP (N_3)	tc (N_3) ← 1
RP (N_2)	pt (N_3) ← $e+f$, RP (N_3)
RP (N_3)	pt (N_4) ← $N_5 * (e+f)$, RP (N_4)
RP (N_4)	pt (a) ← $b * (N_5 * (e+f))$, RP (a)
RP (a)	–

Note that the original expression $(a*b)*(c*d)*e+(a*b)*(c*d)*f$ is indeed equivalent to $a * \text{pt}(a) = a*(b*((c*d)*(e+f)))$. We now proceed to prove the correctness of the above algorithm.

Let D be a tree-transformable σ-dag with root r. Just before a call to RIGHT-PRDS (y), let $R = \{z \mid z \in N(D) \text{ and } w(z) = 1\}$ and $D = \sum_{z \in R} z * \text{pt}(z)$. After the procedure RIGHT-PRDS (y) terminates, let $R' = \{z \mid z \in N(D) \text{ and } w(z) = 1\}$ and $D' = \sum_{z \in R'} z * \text{pt}(z)$.

LEMMA 4.3. *Let R, R', D and D' be as defined above. Assume $y \in R$. If there is a call to* RIGHT-PRDS (y), *then after the procedure terminates, $D \equiv D'$.*

Proof. There are two cases depending on the value of tc (y) at the time of the call.

Case 1. tc $(y) \neq 1$. It is simple to verify that in the procedure none of the values of $w(z)$ or pt (z), for $z \in N(D)$, is modified. Hence $R' = R$ and $D' \equiv D$.

Case 2. $\mathrm{tc}(y)=1$. The proof is by induction on the height $h \geqq 0$ of y. If $h=0$, then y must be a leaf. It is simple to verify that in the procedure none of the values $w(z)$ or $\mathrm{pt}(z)$, for $z \in N(D)$, is modified. Hence $R'=R$ and $D' \equiv D$. Suppose now that the height of y is $h+1>1$. As $h+1>1$, it must be that y is a ⊛ or a ⊕ node. By assumption $\mathrm{tc}(y)=1$, so step 3 is executed and as $y \in R$ then

$$D = \sum_{z \in R-\{y\}} z * \mathrm{pt}(z) + y * \mathrm{pt}(y).$$

There are two subcases.

Subcase a. *r is a ⊛ node.*
a.1. $w(\mathrm{LC}(y))=1$ *before line* 3. At this point D can be written as

$$D = \sum_{z \in R-\{y,\mathrm{LC}(y)\}} z * \mathrm{pt}(z) + y * \mathrm{pt}(y) + \mathrm{LC}(y) * \mathrm{pt}(\mathrm{LC}(y)).$$

By definition, $y = \mathrm{LC}(y) * \mathrm{RC}(y)$. Applying the distributive law, we obtain

$$D = \sum_{z \in R-\{y,\mathrm{LC}(y)\}} z * \mathrm{pt}(z) + \mathrm{LC}(y) * (\mathrm{pt}(\mathrm{LC}(y)) + \mathrm{RC}(y) * \mathrm{pt}(y)).$$

After line 14, $D = \sum_{z \in R-\{y\}} z * \mathrm{pt}(z)$. In line 13, $w(y)$ is set to zero. Let $R'' = \{z \mid z \in N(D) \text{ and } w(z)=1\}$ at this point. Clearly $R'' = R - \{y\}$ so $D'' \equiv D$, where D'' is defined using R''. Since the height of the σ-dag with root $\mathrm{LC}(r)$ is $\leqq h$ and $\mathrm{LC}(r) \in R''$, it then follows by induction that after line 15,

$$D \equiv D'' \equiv D' = \sum_{z \in R'} z * \mathrm{pt}(z).$$

The procedure terminates (line 16) with

$$D \equiv D' = \sum_{z \in R'} z * \mathrm{pt}(z).$$

a.2. $w(\mathrm{LC}(y))=0$ *before line* 3. The proof is similar to that for a.1.

Subcase b. *r is a ⊕ node.* The proof is similar to that for subcase a and will be omitted. □

We collect all the above facts about the procedures LEFT-FACTORS and RIGHT-PRDS in the following theorems.

THEOREM 4.4. *Let D be a tree-transformable σ-dag with root r. Then* LEFT-FACTORS (r) *and* RIGHT-PRDS (r) *will generate an equivalent σ-dag D'' of the form*

$$D \equiv D'' = \sum_{x \in L_r} x * \mathrm{pt}(x),$$

where L_r is the set of left factors of D.

Proof. Using the initial conditions $w(y)=0$ for every $y \in N(D)$, $w(r)=1$ and $D=r$ together with Lemma 4.3, it then follows that after procedure RIGHT-PRDS (r) terminates, $D' = \sum_{z \in R'} z * \mathrm{pt}(z)$. Furthermore, $D' \equiv D = r$. So, $D' \equiv D''$. To complete the proof it is required to show that $R' = L_r$. Both LEFT-FACTORS (r) and RIGHT-PRDS (r) will make the same recursive calls. So, it must be that for each $y \in N(D)$, if there were l calls to RIGHT-PRDS (y), then there must have been l calls to LEFT-FACTORS (y). As $\mathrm{tc}(y)$ is the total number of calls to LEFT-FACTORS (y) after LEFT-FACTORS (r) terminates, then $\mathrm{tc}(y)=0$ for all $y \in N(D)$ after RIGHT-PRDS (r) terminates. So, after RIGHT-PRDS (r) terminates, all internal nodes y of D will have $w(y)=0$ (see lines 6 and 13) and all leaves visited will have $w(\cdot)=1$. Hence, $y \in R'$ if and only if y is a leaf visited from RIGHT-PRDS (r). From Lemma

4.2 we have that $L=L_r$ when LEFT-FACTORS (r) terminates. By inspection of procedure LEFT-FACTORS, $y \in L$ if and only if y is a leaf visited from LEFT-FACTORS (r). As the same leaves are visited by both procedures, then $R'=L_r$.

This completes the proof of the theorem. □

THEOREM 4.5. *The time complexity of* LEFT-FACTORS (r) *and* RIGHT-PRDS (r) *is* $O(|N(D)|)$.

Proof. The proof follows from the observations that no edge in D is traversed more than once. □

We will now establish a relationship between the number of nodes in the dag constructed by LEFT-FACTORS and RIGHT-PRDS and the number of nodes in the original graph (Theorem 4.7). This result is used in the proof of Theorem 4.9. Beforehand, we need the following lemma.

LEMMA 4.6. *Let* $p(y)$ *be the number of parents of a node* y, $y \in N(D)$. *If, after the execution of* LEFT-FACTORS (r), *there exists* $z \in N(D)$ *such that* $0<\mathrm{tc}(z)<p(z)$, *then* D *is not tree-transformable.*

Proof. Note that since $\mathrm{tc}(z)<p(z)$, z must be a right descendant of a ⊛ node. On the other hand, $\mathrm{tc}(z)>0$ implies that the left factors of z are also left factors of the dag D. These two observations imply that D is not tree transformable. □

Our procedure does not compute $p(z)$; however, it is trivial to design a procedure which computes $p(z)$. Another procedure could also be constructed to verify that, for each node $z \in N(D)$, either $\mathrm{tc}(z)=p(z)$ or $\mathrm{tc}(z)=0$. Both procedures would run in time $O(|N(D)|)$. In what follows, we assume that these two procedures are actually executed in between LEFT-FACTORS (r) and RIGHT-PRDS (r) and therefore the situation of Lemma 4.6 cannot arise.

Let D be a σ-dag with root r and let n_1 be the number of nodes in the g product terms of D. Clearly, $|N(D)|=n_1+g-1$. LEFT-FACTORS (r) and RIGHT-PRDS (r) generate an equivalent σ-dag of the form $D'=\sum_{i=1}^{k} x_i * \mathrm{pt}(x_i)$, where the summation is in some fixed order.

THEOREM 4.7. *Let* D, n_1, g, D' *and* k *be as defined above. Then*

$$|N(D')| \leqq |N(D)|+k-g.$$

Proof. Let m' and p' be the numbers of ⊛ and ⊕ nodes respectively whose $\mathrm{tc}(\cdot) \neq 0$ after procedure LEFT-FACTORS (r) terminates. Clearly k is the number of leaves with $\mathrm{tc}(\cdot) \neq 0$ (i.e., the left factors). We try now to account for all the new nodes we create in D'. Each ⊕ node with $\mathrm{tc}(\cdot) \neq 0$ may generate two plus nodes in lines 7 and 8 of the procedure RIGHT-PRDS (r) except in the case where the pointer of the node is $\varnothing$. It is easy to see that the ⊛ nodes could generate at most $2(p'-(g-1))$ new nodes. On the other hand, each ⊛ node with $\mathrm{tc}(\cdot) \neq 0$ creates at most one ⊕ node and one ⊛ node in line 14 of RIGHT-PRDS (r). However, as before, whenever the pointer of the ⊛ node is $\varnothing$, no new ⊛ node will be generated; therefore, the m' ⊛ nodes can generate at most $2m'-g$ new nodes. Since $\mathrm{pt}(y)$ is initialized to $\varnothing$, for each $y \in N(D)$, the first time lines 7, 8 and 14 are executed, the corresponding ⊕ nodes are not introduced. Therefore, the total number of new nodes in D' is

$$\begin{aligned} 2(p'-(g-1))+2m'-g-((p'-(g-1))+(m'-g)+k) \\ =p'+m'-g-k+1. \end{aligned}$$

Constructing the expression $x_1 * \mathrm{pt}(x_1)+\cdots+x_k * \mathrm{pt}(x_k)$ requires the introduction of at most $2k-1$ new nodes. It follows that the total number of new nodes is at most $(p'+m'-g-k+1)+(2k-1)=p'+m'+k-g$.

On the other hand, all interior nodes with $\mathrm{tc}(\cdot)\neq 0$, after procedure LEFT-FACTORS (r) terminates, will not appear in D'. Therefore

$$|N(D')|\leqq|N(D)|-(m'+p')+(p'+m'+k-g),\quad \text{i.e.,}$$
$$|(D')|\leqq|N(D)+k-g. \qquad \square$$

COROLLARY. *Let D, n_1, D' and k be as defined above. Then* $|N(D')|\leqq n_1+k-1$.

Before giving the precise overall transformation algorithm, we outline its general strategy. Let D be a given σ-dag.

(1) Identify the left factors and the set of right products $\{P_i\}_{i=1}^{l}$ of D using the procedures described above.
(2) Split the right products into nonoverlapping sets of dags $\{S_i\}_{i=1}^{k}$.
(3) Suppose the set S_i consists of $\{i_1, i_2, \cdots, i_t\}$. Recursively, transform P_{i_1} into a tree, say T_i; T_i can be written as $T_i=((\cdots(Q_{i,m}*Q_{i,m-1})*\cdots)*Q_{i,1})$, $m\geqq 1$, where each $Q_{i,j}$ is a leaf or a tree with a $\oplus$ root. Figure out the overlap between T_i and $x_{i_l}*P_{i_l}$, $l>1$, and the missing factor in $x_{i_l}*P_{i_l}$. Transform the missing factor in $x_{i_l}*P_{i_l}$ into a tree.
(4) Combine the subtrees.

Step (2) is very easy to do: just find the connected components of the graph induced by the set of right products $\{P_i\}_{i=1}^{l}$. Step (4) is also quite straightforward. Step (3) is a bit harder and can be done as follows. For each $1\leqq s\leqq m$, obtain a normal term f_s of $Q_{i,s}$. For each z, let r_z be the maximum integer such that $Q_{i,1}, Q_{i,2}, \cdots, Q_{i,r_z}$ appear in $x_{i_z}*P_{i_z}$, i.e.,

$$x_{i_z}*P_{i_z}=R_{i_z}*((\cdots((Q_{i,r_z}*Q_{i,r_z-1})*\cdots)*Q_{i,2})*Q_{i,1}),$$

where $R_{i_z}\cap Q_{i,j}=\varnothing$, for all $j>r_z$ (such an $r_z\geqq 1$ exists by virtue of Theorem 3.1). Let $N_t(Q_{i,j})$ be the set of normal terms of $Q_{i,j}$. Then

$$Q_{i,j}=\sum_{q\in N_t(Q_{i,j})} q=f_j+\sum_{\substack{q\in N_t(Q_{i,j})\\ q\neq f_j}} q.$$

Therefore,

$$x_{i_z}*P_{i_z}=R_{i_z}*f_{r_z}*\cdots*f_2*f_1+R_{i_z}*\Gamma,$$

where

$$N_t(\Gamma)=\sum_{\substack{q\in N_t(Q_{i,r_z}*\cdots*Q_{i,1}),\\ q\neq f_{r_z}*\cdots*f_2*f_1}} q.$$

Assume without loss of generality that y_0 and $y_1\notin\Sigma$, i.e., no leaf in the σ-dag contains the symbols y_0 or y_1. Now, let us substitute in the σ-dag for $x_{i_z}*P_{i_z}$, the symbol y_1 for all the leaves in $L(f_{r_z}*\cdots*f_2*f_1)$ and the symbol y_0 for all the leaves in $L(Q_{i,r_z}*\cdots*Q_{i,2}*Q_{i,1})-L(f_{r_z}*\cdots*f_1)$.

The reader should associate the symbol y_0 with the value zero and y_1 with the value one. The σ-dag obtained by *partially evaluating* a σ-dag with root y, pe(y); is defined in terms of the normal terms it contains:

$$N_t(\mathrm{pe}\,(y))=\begin{cases} y_1 & \text{if } \exists c\in N_t(y) \text{ such that } L(c)\subseteq L(y_1) \text{ and } \forall c\in N_t(y) \\ & \text{either } L(c)\subseteq L(y_1) \text{ or } y_0\in L(c),\\ y_0 & \forall c\in N_t(y),\ y_0\in L(c),\\ w & \exists c\in N_t(y) \text{ such that } y_0\notin L(c) \text{ and } L(c)\cap\Sigma\neq\varnothing,\end{cases}$$

where $w = \{h | c \in N_t(y), y_0 \notin L(c), L(c) \cap \Sigma \neq \emptyset$ and h is c after deleting all the $y_1 s\}$. From the definition of pe (y) it should be clear that pe $(x_{1_z} \circledast P_{i_z})$ is R_{i_z}.

The computation of pe (y) for all $y \in N(x_{i_z} \circledast P_{i_z})$ is carried out bottom-up in the obvious way after initializing the leaves in $Q_{i,r_z}, \cdots, Q_{i,1}$, as mentioned above. We actually compute all the pe (y) for all the $x_{i_z} \circledast P_{i_z}$ simultaneously by first constructing the dag for $rr = x_{i_2} \circledast P_{i_2} \boxplus x_{i_3} \circledast P_{i_3} \boxplus \cdots \boxplus x_{i_z} \circledast P_{i_z}$; then initializing the leaves in $Q_{i,m}, \cdots, Q_{i,1}$ and $z_q, \cdots, z_1$, where $z_q \circledast \cdots \circledast z_1$ is one normal term in $Q_{i,m} * \cdots * Q_{i,1}$; then computing pe (y) for all $y \in N(rr)$ as outlined above and finally extracting pe $(y) = R_{i_z}$, where y is the root of $x_{i_z} \circledast P_{i_z}$, for all z.

The formal algorithm is given below. If r is the root of D, TRANSFORM (r) will generate the tree. $D(y)$ will denote the subdag rooted at y.

procedure TRANSFORM (r)
begin

1. Let $x_1 * P_1, x_2 * P_2, \cdots, x_h * P_h$ be the set of left factors and right products of $D(r)$;
2. Let $A = \{i | P_i = \emptyset\}$ and $B = \{i | P_i \neq \emptyset\}$;
   //t will be the root of the tree constructed//
3. $t \leftarrow \emptyset$;
   //the sum of the left factors with empty right products is constructed//
4. **for** each $i \in A$ **do** $t \leftarrow t \boxplus x_i$ **endfor**;
   //Partition the set B in such a way that α and β belong to the same set iff $L(P_\alpha) \cap L(P_\beta) \neq \emptyset$.//
5. Let R be the equivalence relation defined over the elements of set B in such a way that $\alpha R \beta$ iff $L(P_\alpha) \cap L(P_\beta) \neq \emptyset$. Partition B into the sets of equivalence classes $S_1, S_2, \cdots, S_k$ under R.
   //Clearly $\sum_{i \in A} x_i + \sum_{i \in B} x_i * P_i \equiv t + \sum_{j=1}^{k} \sum_{i \in S_j} x_i * P_i$//
   //for each S_i, find an equivalent tree (t')//
6. **for** $i = 1$ **to** k **do**
7. Let $S_i = \{j_1, j_2, \cdots, j_m\}$;
8. $n_0 \leftarrow$ TRANSFORM (P_{j_1}); //construct a tree equivalent to P_{j_1}//
   case
9. $:m = 1$: $[t' \leftarrow x_{j_1} \circledast n_0$; //$|S_i| = 1$//];
10. $:m > 1$: $[rr \leftarrow x_{j_2} \circledast P_{j_2} \boxplus x_{j_3} \circledast P_{j_3} \boxplus \cdots \boxplus x_{j_m} \circledast P_{j_m}$;
11. Let $Q_w, Q_{w-1}, \cdots, Q_1$ be the factors in n_0, i.e., $n_0 = ((\cdots(Q_w \circledast Q_{w-1}) \circledast \cdots) \circledast Q_2) \circledast Q_1$, where each Q_p is a leaf or a subtree with a $\oplus$ root.
12. Compute label (y) for each $y \in N(rr)$;
    //if the root of $x_{i_a} * P_{i_a}$ is labeled s then $x_{i_a} * P_{i_a}$ can be written as $R_{i_a} * Q_s * \cdots * Q_1$ and R_{i_a} does not overlap with $Q_w, \cdots, Q_{s+1}$//
    //take a normal term from $Q_w \circledast Q_{w-1} \circledast \cdots \circledast Q_1$//
13. $(z_q \circledast z_{q-1} \circledast \cdots \circledast z_1) \leftarrow$ NORMAL-TERM $(Q_w \circledast Q_{w-1} \circledast \cdots \circledast Q_1)$;
    //Transform each $x_{i_a} * P_{i_a}$ into $R_{i_a} * Q_s * \cdots * Q_1$ where s is the label of the root of $x_{i_a} * P_{i_a}$. This operation is performed by partially evaluating $x_{i_a} \circledast P_{i_a}$ (actually we partially evaluate all the $x_{i_a} \circledast P_{i_a}$s at the same time by partially evaluating rr)//
14. **for** each $y \in N(rr)$ **do**
    compute pe (y) after initializing the leaves in
    $L(z_q \circledast \cdots \circledast z_1)$ to y_1 and the leaves in $L(Q_w \circledast \cdots \circledast Q_1) - L(z_q \circledast \cdots \circledast z_1)$ to y_0.

```
         endfor
15.      RS_l ← ∅ for l = 1, 2, ···, w;
16.      for l = 2 to m do
17.         Let y be the root of x_{i_l} ⊛ P_{i_l};
18.         RS_label(y) ← RS_label(y) ⊞ pe (y);
19.      endfor
20.      t' ← ((··· ((TRANSFORM (RS_w) ⊞ x_{j_1}) ⊛ Q_w ⊞
              TRANSFORM (RS_{w-1})) ⊛ Q_{w-1} ⊞ ··· ⊞ TRANSFORM (RS_2)) ⊛ Q_2 ⊞
              TRANSFORM (RS_1)) ⊛ Q_1];
      endcase
21.      t ← t ⊞ t';
22. endfor
23. return (t);
    end of procedure TRANSFORM
```

We now describe the procedure NORMAL-TERM $(\cdot)$ which was used in the body of the above procedure.

```
procedure NORMAL-TERM (n_0)
begin
     case
        :n_0 is a leaf: [return (n_0)];
        :n_0 is a ⊞: [return (NORMAL-TERM (RC (n_0))];
        :else: [return (NORMAL-TERM (LC (n_0))
              ⊛ NORMAL-TERM (RC (n_0)))];
     endcase
end of procedure NORMAL-TERM
```

Note that procedure NORMAL-TERM (n_0) does not mark the nodes of n_0. However, when n_0 is the root of a tree, the procedure takes linear time with respect to the number of nodes in the tree. If the dag we wish to transform is tree-transformable, then all calls made from procedure TRANSFORM to procedure NORMAL-TERM will involve trees.

We now consider an example. Let $A = ((a+b*c)*f+b*(c*e+d*f))+(a+b*d)*e$. In this case, it is easy to see that the left factors are given by $x_1 = a$ and $x_2 = b$ with corresponding right products $\text{pt}(a) = (f+e) = P_1$ and $\text{pt}(b) = [c*(f+e)+d*(f+e)] = P_2$. Thus P_1 and P_2 are in the same connected component. At step 8, we will have $n_0 \leftarrow (f+e)$ which has one factor $Q_1 = (f+e)$. At step 12, the root of $b*P_2$ will be labeled 1 at step 13 NORMAL-TERM (Q_1) will return e. The partial evaluation of rr will produce the expression $b*(c+d)$. In lines 15–19 RS_1 is set to be $b*(c+d)$. Finally, at step 20, we get (TRANSFORM $(b*(c+d))+a)*(f+e)$, i.e., $(b*(c+d)+a)*(f+e)$, which is indeed an equivalent tree.

We are now ready to prove the main theorem of this section.

THEOREM 4.8. *Let D be a tree-transformable σ-dag with root r. Then* TRANSFORM (r) *generates an equivalent tree T with root t.*

Before giving the proof, we make the following definition.

DEFINITION 4.3. The *degree d* of a dag D (or of the corresponding expression) is the maximum number of variables in any normal term of D.

Proof of Theorem 4.8. The proof is by induction on the degree d of D.

Assume $d = 1$. In line 1 the set of left factors and corresponding right products of D is obtained using LEFT-FACTORS (r) and RIGHT-PRDS (r). From Theorem

4.4, it follows that $D \equiv \sum_{i=1}^{h} x_i * P_i$. Since $d=1$, D consists of simple variables, i.e., $P_i = \emptyset$, for $1 \leqq i \leqq h$. Therefore $A = \{1, 2, \cdots, h\}$ and $B = \emptyset$ in line 2; executing line 4 produces a tree $t \equiv D$. All the other parts of the algorithm will be skipped because $B = \emptyset$ and $k = 0$.

Suppose now $d > 1$. As before, the execution of line 1 produces an equivalent dag $D \equiv \sum_{i=1}^{h} x_i * P_i$. In line 2, the set $\{1, 2, \cdots, h\}$ is partitioned into two sets A and B such that $D \equiv \sum_{i \in A} x_i + \sum_{i \in B} x_i * P_i$. Line 4 sets t to $t = \sum_{i \in A} x_i$. Line 5 partitions B into disjoint sets in such a way that if i and j belong to the same set, then $L(P_i) \cap L(P_j) \neq \emptyset$. It follows that $D \equiv t + \sum_{i=1}^{k} U_i$, where $U_i = \sum_{j \in S_i} x_j * P_j$, $1 \leqq i \leqq k$. This partition is justified by Theorem 3.1.

Loop 6–22 transforms each U_i into a tree t' which is added to t in line 21. To complete the proof, it is only required to show that, after the execution of lines 7–20, $t' \equiv U_i$.

CLAIM. *For each $1 \leqq i \leqq k$, the tree t' generated by lines 7–20 is equivalent to U_i.*

Proof of the claim. Since S_i consists of the elements $\{j_1, j_2, \cdots, j_m\}$, U_i is the dag $x_{j_1} * P_{j_1} + \cdots + x_{j_m} * P_{j_m}$ (for some fixed order). It is clear that since the degree of P_{j_1} is $\leqq d-1$, it follows by the induction hypothesis that n_0 is a tree equivalent to P_{j_1}. Thus $U_i \equiv x_{j_1} * n_0 + \sum_{z=2}^{m} x_{j_z} * P_{j_z}$, after line 8. Note that if $m=1$, then we are done. Therefore, let's assume that $m > 1$. After lines 10 and 11, we obtain $U_i \equiv x_{j_1} * ((\cdots (Q_w * Q_{w-1}) * \cdots) * Q_2) * Q_1 + rr$, where each Q_i is a leaf or a tree with a $\oplus$ root.

Label(y) is computed for each $y \in N(rr)$ (line 12) in such a way that if the root of $x_{i_a} * P_{i_a}$ is labeled b then $x_{i_a} * P_{i_a}$ can be written as $R_{i_a} * Q_b * \cdots * Q_1$ and $L(R_{i_a}) \cap L(Q_w * \cdots * Q_{b+1}) = \emptyset$. In line 13 we extract a normal term from $Q_w * Q_{w-1} * \cdots * Q_1$. pe ($y$) is computed in line 14 in such a way that if y is the root of $x_{i_a} * P_{i_a}$ then pe (y) $= R_{i_a}$ as defined above. After the execution of lines 15–19 we have:

$$U_i \equiv x_{j_1} * P_{j_1} + \sum_{a=2}^{w} x_{j_a} * P_{j_a}$$

$$\equiv ((\cdots((RS_w \boxplus x_{j_1}) \circledast Q_w \boxplus RS_{w-1}) \circledast Q_{w-1} \boxplus \cdots \boxplus RS_2) \circledast Q_2 \boxplus RS_1) \circledast Q_1.$$

Procedure TRANSFORM is used in step 20 to obtain equivalent trees for $RS_w, \cdots, RS_1$. Since the degree of each RS_l is $<d$, it then follows by induction that t', as constructed by line 20, is equivalent to U_i. This completes the proof of the claim and the theorem. $\square$

Let D be a tree-transformable σ-dag with root r and of degree d. Let n be the number of nodes in the g product terms and let v be the number of leaves in D. Let $T(n, d)$ be the time complexity of procedure TRANSFORM (r).

THEOREM 4.9. *Let d, n, v, r, g and D be as defined above. Then $T(n, d) \leqq C_1 dn$, where C_1 is some fixed constant.*

Proof. First of all, let us determine the time complexity of steps 1–5. Line 1 takes $\leqq C_2(n+g)$ time since D has exactly $n+g-1$ nodes and procedures LEFT-FACTORS and RIGHT-PRDS take time $O(|N(D)|)$ (see Theorem 4.5). Lines 2–4 take time $\leqq C_2 h$. Line 5 can be implemented by finding the connected components of a graph for P_j, $j \in B$, (of course, we ignore the direction of the edges) which can be easily carried out in $\leqq C_2 m$ steps, where m is the number of nodes in the graph for P_j, $j \in B$. Since the number of nodes in $\sum_{i=1}^{h} x_i * P_i$ is $\leqq n+h-1$ (see Theorem 4.7), then line 5 takes time $\leqq C_2(n+h)$. Clearly $g \leqq n$ and $h \leqq n$. Hence, lines 1–5 take time $\leqq 5C_2 n$.

In what follows, we prove by induction on the degree $d \geqq 1$ of the σ-dag with root r, that $T(n, d) \leqq C_1 dn$, where $C_1 = 12C_2$.

For $d = 1$, we know that B must be empty. Therefore, loop 6–22 is not executed. By assumption $C_1 > 5C_2$. Hence, $T(n, 1) \leqq C_1 n \leqq C_1 dn$.

Suppose now the degree of D is $d > 1$. In this case loop 6–22 has to be considered. Let U_i be the σ-dag corresponding to S_i (see Theorem 4.8). Let g_i be the total number of product terms in U_i and let n_i be the number of nodes in the descendants of the product terms of U_i. Let v_i be the number of leaves in U_i and let d_i be the degree of U_i. Clearly, $d_i \leqq d$. Since $N(U_i) \cap N(U_j) = \emptyset\ \forall i \neq j$ (see Theorem 4.8) and since the number of nodes in $\sum_{i=1}^{h} x_i P_i$ is $\leqq n + h - 1$ (see Theorem 4.7), it follows that $\sum n_i \leqq n$. Let $T''(n_i, d_i)$ be the time required by loop 6–22 when processing U_i. Then the overall time complexity for TRANSFORM (r) is $\leqq 5C_2 n + \sum_i T''(n_i, d_i)$. We now claim the following:

CLAIM.

$$5C_2 n_i + T''(n_i, d_i) \leqq C_1(d_i) n_i.$$

Once we prove this claim it will follow that $T(n, d) \leqq C_1 dn$, which will complete the proof of the theorem.

Proof of claim. We treat the following two cases separately.

Case 1. $m = 1$. Lines 7 and 9 take constant time, and by the induction hypothesis, line 8 takes time $\leqq C_1(d_i - 1) n_i$. Hence, $T''(n_i, d_i) \leqq C_1(d_i - 1) n_i + 2C_2$. In this case, the proof of the claim follows from the assumption that $7C_2 < C_1$.

Case 2. $m > 1$. Line 7 takes $\leqq C_2 g_i$ time and by induction, line 8 takes time $\leqq C_1(d'')(n_i'')$, where d'' is the degree of P_{j_1}; n_i'', v_i'' are the number of nodes and leaves in P_{j_1} respectively. The execution time of lines 10 and 11 can be easily shown to be $C_2 g_i$ and $C_2 v_i''$, respectively. Hence, the time taken by steps 7–11 is

$$\leqq C_1(d'') n_i'' + 2C_2(g_i + v_i'').$$

Since $g_i \leqq n_i$ and $v_i'' \leqq n_i'' \leqq n_i$, we have that the above inequality is

$$\leqq C_1(d'') n_i'' + 4C_2 n_i \leqq C_1(d_i - 1) n_i'' + 4C_2 n_i.$$

Lines 12–19 can be easily shown to take time $\leqq C_2 n_i$. Let g_l' be the number of product terms in RS_l with n_l' descendants and v_l' leaves. Step 20 can be easily shown to take time $\leqq C_2 n_i + \sum_{l=1}^{w} C_1(d_i - 1) n_l'$, since the degree of each RS_l is $< d_i$. Line 21 takes time $\leqq C_2$. Collecting all the above facts we obtain:

$$\begin{aligned} T''(n_i, d_i) \leqq\ & C_1(d_i - 1) n_i'' + 4C_2 n_i && \text{(lines 7–11)} \\ & + c_2 n_i && \text{(lines 12–19)} \\ & + C_2 n_i - \sum_{l=1}^{w} C_1(d_i - 1) n_l' && \text{(line 20)} \\ & + C_2 && \text{(line 21).} \end{aligned}$$

Since $C_1 = 12C_2$, we have that

$$T''(n_i, d_i) + 5C_2 n_i \leqq C_1(d_i - 1) n_i'' + C_1 n_i + \sum_{l=1}^{w} C_1(d_i - 1) n_l'.$$

A straightforward implementation of line 14 can be used to show that $n_i'' + \sum_{l=1}^{w} n_l' \leqq n_i$. Hence, $T''(n_i, d_i) + 5C_2 n_i \leqq C_1(d_i - 1) n_i + C_1 n_i \leqq C_1 d_i n_i$.

This completes the proof of the claim and the theorem. □

Let us finally remark that the above transformation algorithm could output a tree which is not equivalent to the input dag, as the following example shows.

Example 4.2. Suppose we are given the expression

$$E = (((x * b) * c + (a * b) * c) + x * (e + d)) + a * (b * c + d),$$

whose dag is drawn in Fig. 4.3. In this case, the left factors are the variables x and a with corresponding right products $P_1 = b * c + (e + d)$ and $P_2 = b * c + (b * c + d)$. Therefore we have one connected component consisting of $\{P_1, P_2\}$. At step 8 of the procedure TRANSFORM, we have $n_0 \leftarrow b * c + (e + d)$ in which case n_0 has one factor Q_1. Now NORMAL-TERM $(Q_1) = d$ and $rr = a * (b * c + (b * c + d))$. The label assigned in line 12 to $a * P_2$ is 1 and the partial evaluation of rr produces $rr = a$. Therefore, the output will be $(a + x) * (b * c + (e + d))$, which is not equivalent to E.

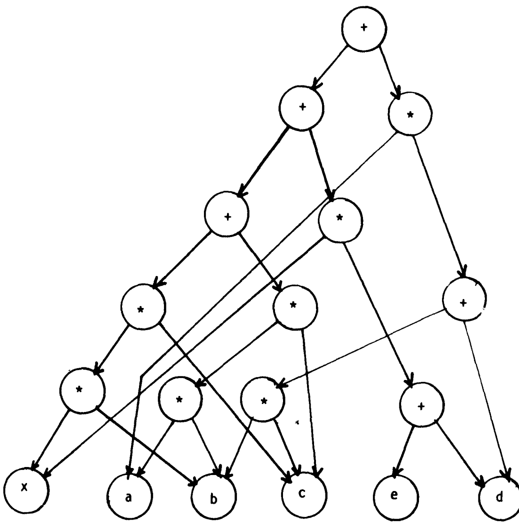


FIG. 4.3

5. The equivalence algorithm. As we have seen in the previous section, the transformation algorithm might generate a tree from a σ-dag which is not tree-transformable. Algorithms to check whether a given tree T and a given σ-dag D are equivalent are developed for several cases in this section. One way to solve this problem would be to find the normal terms of T and D and compare them; this is not efficient since the number of normal terms could be an exponential function of the number of nodes in D. The approach we take here is based upon the following characterization.

THEOREM 5.1. *A σ-dag D is equivalent to a tree T if and only if the following conditions are satisfied*:

(i) *Every normal term of D is a normal term of T.*

(ii) *The number of normal terms of D is equal to the number of normal terms of T.*

(iii) *No two normal terms of D are equal.*

In the rest of this section, we will examine the problem of designing efficient algorithms to check each of the above properties separately.

To handle (i), we will associate a graph with T, denoted by $G(T)$ in which a normal term induces a path and every path corresponds to a normal term. Before

doing so, we label the nodes of the tree by the following procedure. Initially, n_o represents the root of T, mark $(n_0) \leftarrow (0, 1)$, left $\leftarrow 0$, right $\leftarrow 1$ and next $\leftarrow 2$.

```
procedure LABEL-TREE (n0, left, right)
begin
global (next, mark [·]);
  case
    :n0 is a leaf [return];
    :n0 is a ⊛: [m ← next;
                 next ← next + 1;
                 mark (LC (n0)) ← (left, m);
                 call LABEL-TREE (LC (n0), left, m);
                 mark (RC (n0)) ← (m, right);
                 call LABEL-TREE (RC (n0), m, right);
                 return];
    :n0 is a ⊕: [mark (LC (n0)) ← mark (RC (n0)) ← (left, right);
                 call LABEL-TREE (LC (n0), left, right);
                 call LABEL-TREE (RC (n0), left, right);
                 return];
  endcase
end of procedure LABEL-TREE
```

To illustrate the ideas, we consider the expression $A = ((a+b*c)*f+b*(c*e+d*f))+(a+b*d)*e$.

The tree generated by algorithm TRANSFORM is given in Fig. 5.1. The labels are as assigned by the above algorithm. With the labels given in Fig. 5.1, we can associate the following graph:

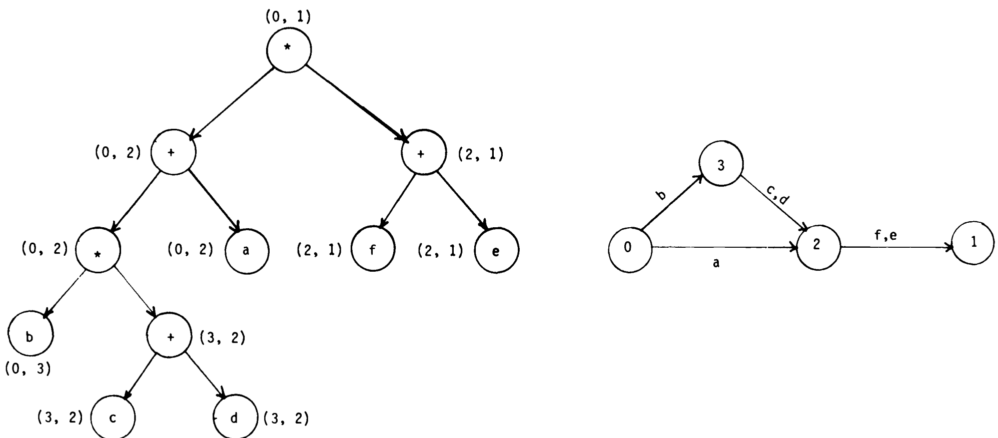


FIG. 5.1

Note that each normal term of the tree can be viewed as representing a path from 0 to 1 and vice versa.

In general, we can construct the graph $G(T)$ associated with a tree T as follows: suppose all of the leaves of T have been marked by the procedure LABEL-TREE. Create two nodes 0 and 1. If a leaf v is labeled (α, β), create nodes α and β (if

necessary) and draw a directed edge from α to β with label v; if such an edge already exists, simply attach the label v to it.[1]

The above graph could be thought of as the transition graph of a finite automaton with 0 as the initial state and 1 as the accepting state; the alphabet consists of the set of leaves of T. A word is accepted by this automaton if and only if it represents a normal term of the tree T.

We are ready to prove the following lemma.

LEMMA 5.2. *Let y be a node of the tree T whose descendants form a subtree T_1 and such that* mark $(y) = (\alpha, \beta)$. *Then, every normal term of T_1 is represented by a path from α to β in $G(T_1)$ and, conversely, every path form α to β in $G(T_1)$ represents a normal term of T_1.*

Proof. By induction on the height h of T_1.

If $h = 0$, y is a leaf and the result is obvious from the definition of $G(T_1)$.

Suppose now $h \geqq 1$. Two cases might arise:

(1) y is a $\oplus$ node with children generating subtrees C_1 and C_2, as shown in Fig. 5.2. Note that every normal term of T_1 is a normal term of either C_1 or C_2. Moreover, it is easy to see (from the definition of $G(T_1)$) that each edge in $G(T_1)$ is an edge in either $G(C_1)$ or $G(C_2)$, and conversely. Furthermore, no two edges of $G(C_1)$ and $G(C_2)$ are the same; therefore every path[2] from α to β in $G(T_1)$ is a path in either $G(C_1)$ or $G(C_2)$, and conversely. The proof follows now by induction for this case.

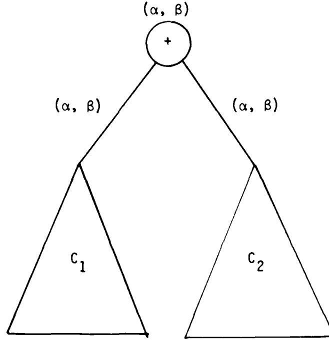


FIG. 5.2

(2) y is a $\circledast$ node with subtrees C_1 and C_2 (Fig 5.3(a)). Note that $G(T_1) = G(C_1) \cup G(C_2)$ as shown in Fig. 5.3(b) and where the edges of $G(C_1)$ are distinct from those of $G(C_2)$. Let $N_t(T)$ designate the set of normal terms of a tree T. Assume

$$N_t(C_1) = \{p_i | i = 1, \cdots, k_1\} \quad \text{and}$$

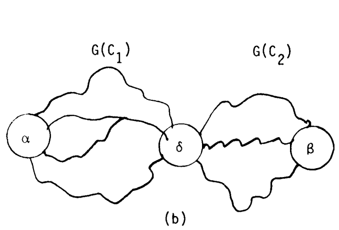
$$N_t(C_2) = \{q_j | j = 1, \cdots, k_2\}.$$

Then

$$N_t(T_1) = \{p_i * q_j | 1 \leqq i \leqq k_1, 1 \leqq j \leqq k_2\}.$$

[1] Note that we are actually constructing a series-parallel graph $G(T)$ from T. Every $\oplus$ causes a parallel connection and every $\circledast$ causes a series connection.

[2] As a sequence of edges.



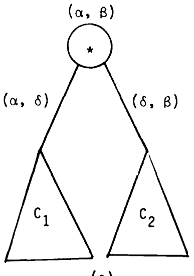


FIG. 5.3

Therefore each normal term of T_1 is of the form $p_i * q_j$; by induction, p_i is represented by a path from α to δ in $G(C_1)$ and q_j is represented by a path from δ to β in $G(C_2)$. It follows that $p_i * q_j$ can be represented by a path from α to β in $G(T_1)$.

Conversely, every path from α to β in $G(T_1)$ consists of two paths P_1 and P_2, where P_1 is a path from α to δ in $G(C_1)$ and P_2 is a path from δ to β in $G(C_2)$. The proof follows now by induction. □

COROLLARY 5.2.1. *Let $G(T)$ be the graph associated with a tree T. Every normal term of T is represented by a path from 0 to 1 in $G(T)$, and conversely, every such path represents a normal term of T.*

Suppose now we use the label of the leaves of T and assign them to the corresponding leaves of the dag D. If a node y of D has two children labeled, say, (α, β) and (α', β'), this means that the induced paths in $G(T)$ should match so that each normal term of y will be a consistent part of a normal term in T. It follows that if y is a $\oplus$ node, we must have $\alpha = \alpha'$ and $\beta = \beta'$; else, we must have $\beta = \alpha'$. Moreover, if every normal term of D is a normal term of T, then the root of the dag should get the label (0, 1). Formal proofs of these facts will be given after we present the procedure which implements the above policy.

```
procedure LABEL-DAG (n0)
begin
global mark ([·]);
   If n0 has been labeled then [return];
   If n0 is a leaf then [stop];
   call LABEL-DAG (LC (n0));
   call LABEL-DAG (RC (n0));
   (LX, LY) ← mark (n0));
   (RX, RY) ← mark (RC (n0));
   case
     :n0 is a ⊕ node: [If RX = LX and LY = RY
                              then mark (n0) ← (LX, LY);
                              else stop;
                      return];
     :n0 is a ⊛ node; [If RX = LY then mark (n0) ← (LX, RY);
                              else stop;
                      return];
   endcase
end of procedure LABEL-DAG
```

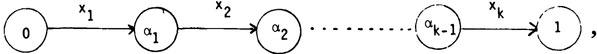
LEMMA 5.3. *Let $G(T)$ be the graph of a tree T. Suppose the leaves of a dag D are initialized with the same labels as those of T. If the dag could be labeled consistently up to a node p, with* mark $(p)=(\alpha, \beta)$, *then each normal term of p corresponds to a path from α to β in $G(T)$.*

Proof. By induction on the height h of the subdag induced by p. The case $h=1$ is trivial. Suppose $h>1$. We consider again two cases depending on whether p is a $\oplus$ node or a $\circledast$ node. The proof follows the same line as that of Lemma 5.2. □

COROLLARY 5.3.1. *If D could be labeled such that* mark $(r)=(0,1)$, *where r is the root of D, then each normal term of D is a normal term of T.*

LEMMA 5.4. *Suppose that each normal term of D is a normal term of T, then* LABEL-DAG (r) *will terminate with* mark $(r)=(0,1)$, *where r is the root of D.*

Proof. We will only prove that each product term of D will be labeled (0, 1). Let P be a product term in D. Suppose that $w=x_1 * x_2 * \cdots * x_k$ (up to a fixed order) is a normal term in P, $x_i \in \Sigma$, $1 \leqq i \leqq k$. It follows that w is a normal term in T and hence there exists a path from 0 to 1 in $G(T)$ which represents w. Suppose this path is given by



i.e., mark $(x_i)=(\alpha_{i-1}, \alpha_i)$, $1<i<k$, and mark $(x_1)=(0, \alpha_1)$, mark $(x_k)=(\alpha_{k-1}, 1)$.

Since w is a normal term in P, P must be of the following form: $P=(x_1+D_1)*(x_2+D_2)*\cdots*(x_k+D_k)$ up to a fixed order, where the D_is could be arbitrary subdags of D. We assume that the order of multiplications is as shown in Fig. 5.4; the same argument will hold for any other ordering. Now since x_1 is labeled

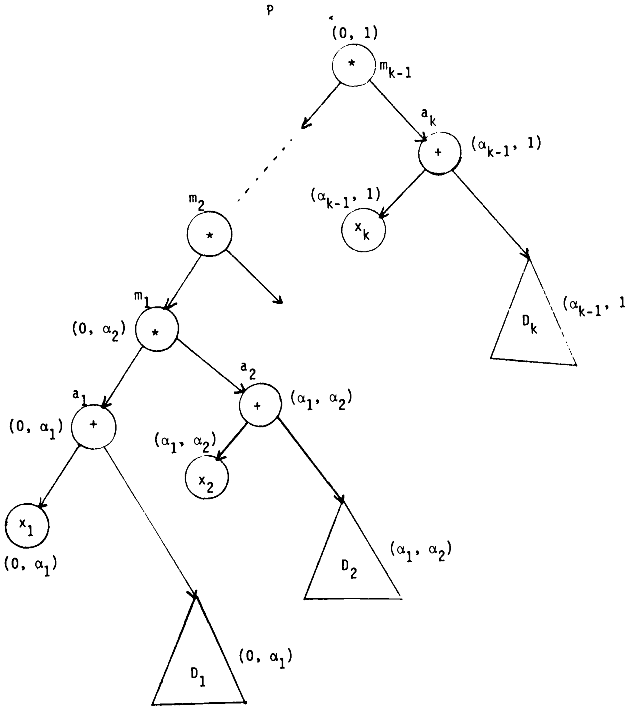


FIG. 5.4

$(0, \alpha_1)$, the root of D_1 must have the same label $(0, \alpha_1)$ and thus a_1 will have the label $(0, \alpha_1)$. Similarly, a_i will have the label (α_{i-1}, α_i), $1<i<k$, where a_i is the root of the dag $x_i + D_i$; a_k gets the label $(\alpha_{k-1}, 1)$. Now m_1 is a multiplication node with children a_1 and a_2; thus it gets the label $(0, \alpha_2)$.
Using the argument $k-1$ times, we get that m_{k-1} has the label $(0, 1)$ and thus P has the label $(0, 1)$. This completes the proof of the lemma. □

We now collect the above facts in the following theorem.

THEOREM 5.5. *Given a tree T and a σ-dag D, it is possible to check whether each normal term of D is a normal term of T in $O(n)$ time, where n is the number of nodes in D.*

We now consider property (ii) of Theorem 5.1, namely checking whether the number of normal terms of T is the same as that of D. This is fairly easy(?) and the counting of normal terms in a dag D could be done by the procedure COUNT (r), where r is the root of D. Initially, all the nodes are not marked.

```
procedure COUNT (n0)
begin
global (C [·]);
     If n0 is marked then [return (C(n0))];
                    else mark n0;
     case
       :n0 is a leaf: [C(n0) ← 1];
       :n0 is a ⊛: [C(n0) ← COUNT (LC (n0)) * COUNT (RC (n0))];
       :n0 is a ⊕: [C(n0) ← COUNT (LC (n0)) + COUNT (RC (n0))];
     end case
     return (C(n0));
end of procedure COUNT
```

It is easy to prove the following lemma.

LEMMA 5.6. *Let D be any σ-dag with root r. Then* COUNT (r) *correctly computes the number of normal terms in D.*

As for the complexity, we have $O(n)$ steps, where n is the number of nodes in D. However, some steps might involve the multiplication of two large numbers, each of which might take considerably more than one "unit time." Before finding the number of bit operations required by the above algorithm, we establish an upper bound on the magnitude of the numbers used in COUNT.

LEMMA 5.7. *Let T be a tree with root r and such that*[3] $|L(T)| = v$. *Then* $C(x) \leqq 2^{v/2}$, *for all nodes x in T.*

Proof. By induction on the height h of T. □

COROLLARY 5.7.1. *Each $C(x)$ requires at most $v/2$ bits.*

Note that if we are considering the dag D and if at any one point we need more than $v/2$ bits to store any number, then we halt and declare that the numbers of normal terms are not equal. Therefore, the above upper bound holds true for D.

We are now ready to establish the complexity of the procedure COUNT.

THEOREM 5.8. *Let D be a σ-dag with root r. Then* COUNT (r) *takes* $O(nv \log v \log\log v)$ *bit operations, where n and v are respectively the numbers of nodes and leaves in D.*

---

[3] Recall that $L(T)$ represent the set of leaves in T.

Proof. The largest number in COUNT (r) require at most $v/2$ bits; adding such numbers could be done in $O(v)$ bit operations. Multiplying two such numbers could be done in $O(v \log v \log\log v)$ bit operations by using the Schönhage–Strassen integer-multiplication algorithm [SS]. Therefore COUNT (r) requires at most $O(nv \log v \log \log v)$ bit operations. □

Strangely enough, the above (rough) bound cannot be improved (under the assumption that multiplying two $k \times k$ bit numbers takes $k \log k \log\log k$ bit operations), i.e., there exist tree-transformable dags which will require $\Omega(nv \log v \log\log v)$ bit operations, as the following example shows.

Let

$$P_1 = (Z_1 + Z_2) * (Z_3 + Z_4) * \cdots * (Z_{2p-1} + Z_{2p}),$$

$$P_2 = (Y_1 + Y_2) * (Y_3 + Y_4) * \cdots * (Y_{2p-1} + Y_{2p}), \qquad Z_i, Y_i \in \Sigma.$$

Construct now the following dag D (Fig. 5.5).

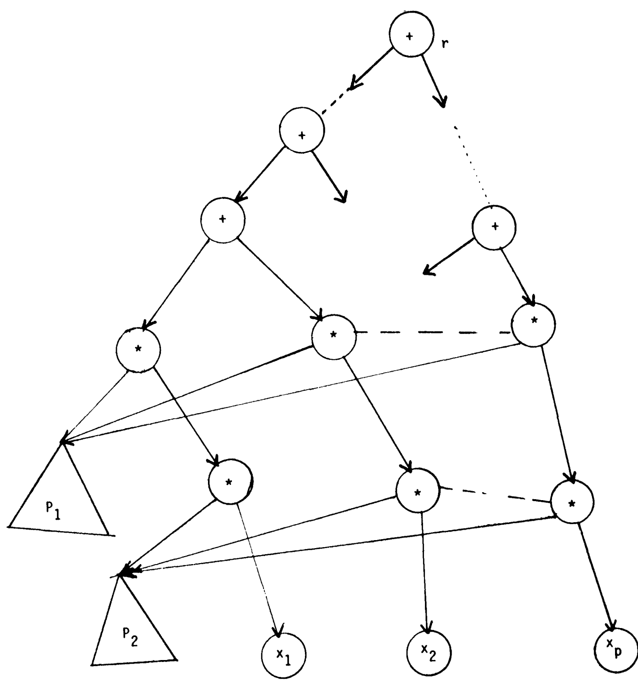


FIG. 5.5

In this case $|N(D)| = O(p)$, $|L(D)| = 5p$. However, COUNT (r) will have p multiplications, each of which occurs between two p-bit numbers. Therefore COUNT (r) requires at least $\Omega(nv \log v \log\log v)$ bit operations in this case.

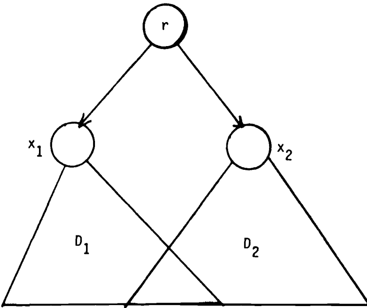
One might be tempted to say that it is possible to design another algorithm which does not multiply the same pair of numbers more than once and which will solve our problem in $O(n^2)$ time; however we can give another example where the multiplications involved are all between different numbers and yet the algorithm requires $\Omega(nv \log v \log\log v)$ bit operations. Let's remark that the execution time of this procedure dominates all the other parts of the equivalence algorithm.

We now consider the performance of COUNT on a special class of dags which will be considered later in more detail, namely that of leaf dags. COUNT is faster for this class even if we use the naive integer-multiplication algorithm as the following theorem shows.

THEOREM 5.9. *Let D be a leaf dag with root r. Then the execution time of* COUNT (r) *is of $O(n^2)$, where n is the number of nodes in D.*

Proof. Let e be the number of edges in D. Since $e = O(n)$, it follows that it is enough to prove that COUNT (r) takes $O(e^2)$ time. The proof is by induction on e, being trivial for $e = 1$.

Suppose $e > 1$. Let x_1 and x_2 be the left and right children, respectively, of r. x_1 and x_2 generate two dags D_1 and D_2 whose edges don't overlap. Let e_1 and e_2 be the numbers of edges in D_1 and D_2, respectively. Then $e = e_1 + e_2 + 2$. Thus the execution time $T(e)$ of the algorithm satisfies



$$T(e) \leqq T(e_1) + T(e_2) + O(e_1 e_2),$$

if we use the naive algorithm to multiply the number of normal times in x_1 by the ones in x_2. It follows that $T(e) = O(e^2)$. □

To terminate the equivalence algorithm, the problem of whether a given σ-dag has two identical normal terms will be investigated now. This is the hardest part of the equivalence algorithm and its complexity seems to depend crucially on two parameters: the degree of the expression and the type of sharing in the dag. If we restrict either one of these parameters, the problem becomes relatively easy and corresponding efficient algorithms can be developed. However, in the general case, the problem looks difficult and we feel that the general problem might be NP-complete. Therefore, we will attack this problem for two special cases: (i) the degree of the corresponding expression is bounded by a constant d and (ii) the given σ-dag is a leaf dag.

Before proceeding, we state the main result which has been obtained so far.

THEOREM 5.10. *Let D be an arbitrary σ-dag with no identical normal terms. Then checking whether D is tree transformable and obtaining an equivalent tree, whenever possible, could be done in $O(nv \log v \log \log v)$ time, where n and v are respectively the number of nodes and leaves in D.*

Proof. Immediate from Theorems 4.8, 4.9, 5.5 and 5.8. □

We now discuss the problem of identifying identical normal terms for leaf dags. We first transform the dag D into an equivalent dag D' which is *left-justified*, i.e., every ⊛ node of D' has a leaf as its left child. Figure 5.6 shows an example of a dag D with an equivalent dag D' which is left-justified.

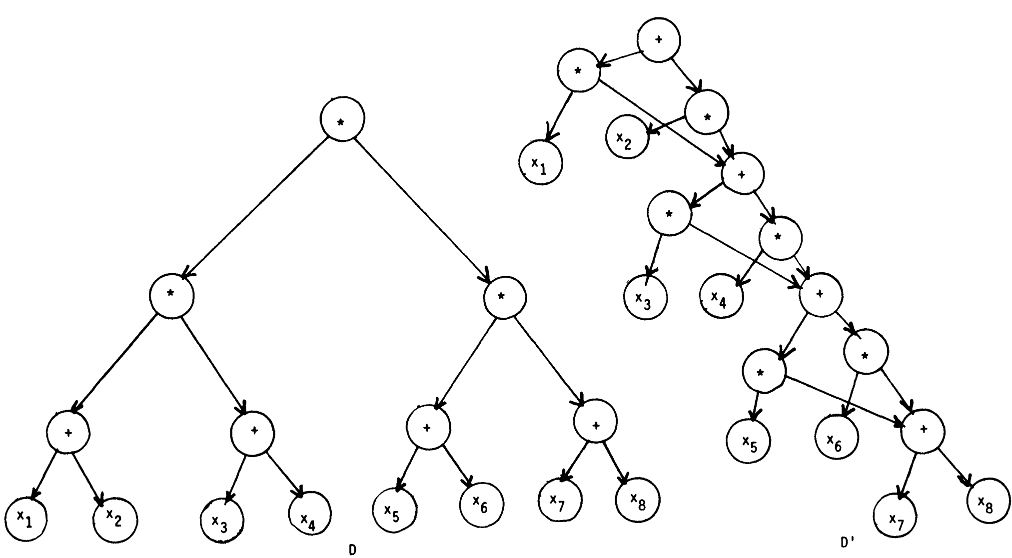


FIG. 5.6

We remark that in the above transformation no ⊕ node will be modified unless it is a left child of a ⊛ node. That is why the above transformation will increase the number of edges by, at most, a factor of 2, as we will later prove. The procedure to implement the above transformation is given below; pt ( ) has the same meaning as in § 4 and it is initialized to the empty dag, for all nodes of D. OP $(\cdot)$ denotes the operator of a node. The assignment $n_0 \leftarrow$ GETNODE means that a new node n_0 is created and PUTNODE (n_0) means that the node n_0 has been destroyed.

procedure LEFT-JUST (r)
begin //Left justify the leaf dag with root r which is multiplied by the left justified dag pointed at by pt (r)//
  If $r =$ leaf **then** [**return**]
 loop
  $n_2 \leftarrow$ LC (r);
  $n_3 \leftarrow$ RC (r);
 case
  :r is a ⊕: [**If** $n_2 =$ leaf **then** [**If** pt $(r) \neq \varnothing$ **then** [$n_0 \leftarrow$ GETNODE;
   OP $(n_0) \leftarrow$ '*';
   LC $(n_0) \leftarrow n_2$;
   RC $(n_0) \leftarrow$ pt (r);
   LC $(r) \leftarrow n_0$]]
  else [pt $(n_2) \leftarrow$ pt (r);
   call LEFT-JUST (n_2)]
  If $n_3 =$ leaf **then** [**If** pt $(r) \neq \varnothing$ **then** [$n_0 \leftarrow$ GETNODE;
   OP $(n_0) \leftarrow$ '*';
   LC $(n_0) \leftarrow n_3$;
   RC $(n_0) \leftarrow$ pt (r);
   RC $(r) \leftarrow n_0$;
   return]
  else [**return**]]

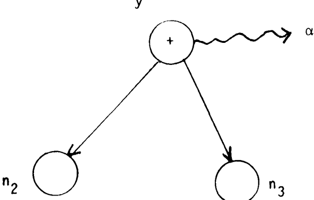
```
                    else [pt (n3) ← pt (r);
                          call LEFT-JUST (n3);
                          return]
                  ];
:r is a ⊛: [If (n2 and n3 are leaves) then
             [If pt (r) = ∅ then [return];
             n0 ← GETNODE;
             OP (n0) ← '*';
             LC (n0) ← n3;
             RC (n0) ← pt (r);
             RC (r) ← n0;
             return]
           If (n2 is a leaf) then
                 [pt (n3) ← pt (r);
                 call LEFT-JUST (n3);
                 return]
           If (n3 is a leaf) then
                [OP (r) ← OP (n2);
                LC (r) ← LC (n2);
                RC (r) ← RC (n2);
                If pt (r) = ∅ then
                            [PUTNODE (n2);
                            pt (r) ← n3]
                          else
                            [OP (n2) ← '*';
                            LC (n2) ← n3;
                            RC (n2) ← pt (r);
                            RC (n2) ← pt (r);
                            pt (r) ← n2]
                        ]

           else
             [pt (n3) ← pt (r);
             call LEFT-JUST (n3);
             OP (r) ← OP (n2);
             LC (r) ← LC (n2);
             RC (r) ← RC (n2);
             pt (r) ← n3;
             PUTNODE (n2)]];

     endcase
  forever
end of procedure LEFT-JUST
```

The next lemma essentially establishes the correctness of the above procedure.

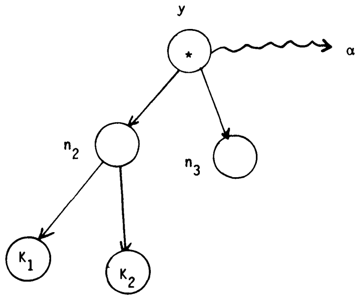
LEMMA 5.11. *Let D be a leaf dag and let y be any node of D such that* pt $(y) = \alpha$ *is left-justified. Then* LEFT-JUST (y) *will return a left-justified dag equivalent to the dag whose root is a* ⊛ *node with y and* α *as the left and right children, respectively.*

Proof. By induction on the depth of the recursive call. The proof involves eight different cases; we will only consider two typical cases, since all the others are similar to one or the other of the two.

1) y is a $\oplus$ node such that none of its children is a leaf. Two subcases have to be discussed separately.
   1.a) $\alpha = \emptyset$, then LEFT-JUST (y) returns by calling LEFT-JUST (n_2) and LEFT-JUST (n_3). By induction, the dags induced by n_2 and n_3 will be transformed into equivalent left-justified dags. It is easy to see that the dag with root y will then be left-justified.
   1.b) $\alpha \neq \emptyset$, then LEFT-JUST (y) makes the assignments pt $(n_2) \leftarrow$ pt $(n_3) \leftarrow \alpha$ and calls LEFT-JUST (n_2) and LEFT-JUST (n_3). By the induction hypothesis, we will have two left-justified dags equivalent to $(n_2 \circledast \alpha)$ and $(n_3 \circledast \alpha)$. Using the distributive law, it is easy to see that LEFT-JUST (y) will return a left-justified dag equivalent to $(y \circledast \alpha)$.



2) y is a $\circledast$ node such that none of its children is a leaf. Again, we will discuss two subcases separately. Note that the dag which is being transformed into an equivalent left-justified dag is given by $(n_2 \circledast n_3) \circledast \alpha$.
   2.a) $\alpha = \emptyset$, similar to 1.a) and 2.b).
   2.b) $\alpha \neq \emptyset$, LEFT-JUST (y) will first make the assignment pt $(n_3) \leftarrow \alpha$ and call LEFT-JUST (n_3). By the induction hypothesis, LEFT-JUST (n_3) returns a left-justified dag which is equivalent to $(n_3 \circledast \alpha)$. LEFT-JUST (y) will then make n_2 as the new node y with pt $(y) \leftarrow n_3$ and the infinite loop behaves as if it were really a recursive call to n_2 with pt $(n_2) = n_3$ (the "new" n_3 after it was modified by LEFT-JUST (n_3)). By the induction hypothesis, this should return a left-justified dag equivalent to $n_2 \circledast n_3$. □



COROLLARY. *Let D be a leaf dag with root r such that* pt (r) *is initialized to the empty dag. Then* LEFT-JUST (r) *returns a left-justified dag D′ which is equivalent to D.*

The following theorem establishes the possible growth in the number of edges as well as the complexity of the procedure LEFT-JUST.

THEOREM 5.12. *Let D be a leaf dag with root r such that* pt (r) *is initialized to the empty dag. Then* LEFT-JUST (r) *returns a left-justified dag* D' *such that* $|E(D')| \leqq 2|E(D)|$. *Moreover, the execution time of this procedure is linear in the number of nodes in D.*

Proof. The proof is straightforward and will be left to the reader. □

Once we transform the given dag into an equivalent left-justified dag, the problem of checking the existence of two identical normal terms becomes easy. The main procedure, REPEATED-TERM, which, at each ⊕ node, calls another procedure EQUAL to check whether this node has two identical normal terms, is given below.

```
procedure REPEATED-TERM (n₀)
begin
global (rmark [·])
     case
        :n₀ has been rmarked: [return (rmark (n₀))];
        :n₀ is a leaf: [rmark (n₀) ← false];
        :else: [rmark (n₀) ← REPEATED-TERM (LC (n₀))
                          or REPEATED-TERM (RC (n₀));
              If rmark (n₀) = false and n₀ is a ⊕
                     then rmark (n₀) ← EQUAL (LC (n₀), RC (n₀))];
     endcase
     return (rmark (n₀));
end of procedure REPEATED-TERM
```

This procedure is fairly straightforward; it rmarks false all the leaves and proceeds from the bottom up, marking all nodes false until it meets a ⊕ node. It then calls the procedure EQUAL to check whether this ⊕ node has two identical normal terms in which case rmark (n_0) will be set true and this truth assignment will propagate to the root of the dag. Otherwise, everything will be set false and REPEATED-TERM returns false. We describe precisely the procedure EQUAL below.

```
procedure EQUAL (x, y)
begin
global (equal [·, ·]);
   If x = y then [return (true)]
   If (x, y)has been previously checked
            then [return (equal (x, y))]
   case
      :x and y are leaves: [equal (x, y) ← false];
      :x is a ⊕ or y is a ⊕:
         [Let w be one of the ⊕ nodes and let z the other node;
         equal (x, y) ← EQUAL (LC (w), z) or EQUAL (RC (w), z)];
      :x and y are ⊛:
         [If LC (x) ≠ LC (y) then equal (x, y) ← false;
                else equal (x, y) ← EQUAL (RC (x), RC (y))];
      :else: [equal (x, y) ← false];
   endcase
   return (equal (x, y));
end of procedure EQUAL
```

Before proving the correctness of the above procedure, we discuss the example mentioned at the end of § 4. The original expression $E = (((x * b) * c +$

$(a*b)*c)+x*(e+d))+a*(b*c+d)$ has been transformed into the tree $T=(a+x)*(b*c+(e+d))$. Moreover, E could be transformed into an equivalent dag E' which is left-justified (Fig. 5.7). Note that n_0 and n'_0 are the nodes created by the procedure LEFT-JUST.

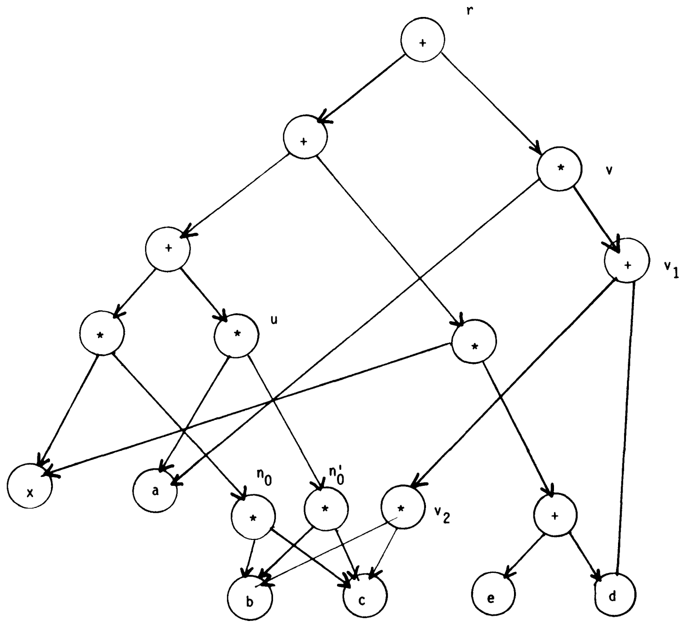


FIG. 5.7

Let us apply the algorithm EQUAL to the nodes u and v indicated in Fig. 5.7. Note that EQUAL (u, v) will be called at some point when REPEATED-TERM is applied to the above dag.

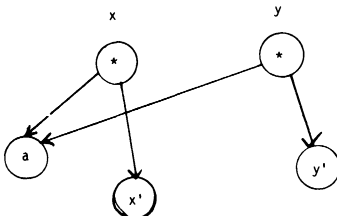
Recursive call	Result
EQUAL (u, v)	EQUAL (n'_0, v_1)
EQUAL (n'_0, v_1)	EQUAL (n'_0, v_2) OR EQUAL (n'_0, d)
EQUAL (n'_0, v_2)	EQUAL (c, c)
EQUAL (c, c)	**true**
–	equal $(n'_0, v_1) \leftarrow$ **true**
–	equal $(u, v) \leftarrow$ **true**

Therefore, EQUAL (u, v) returns true and REPEATED-TERM will also return true. Indeed, E does contain two duplicates of the normal term $a*b*c$.

The correctness of the above procedures is established in the next theorem.

THEOREM 5.13. *Let r be the root of the left-justified σ-dag D. Then* REPEATED-TERM (r) *returns true if, and only if, D has two identical normal terms.*

Proof. Suppose REPEATED-TERM (r) returns true, then it is easy to see by inspection that there exist two ⊛ nodes x and y such that EQUAL (X, Y) is true and



mark $(X) = \text{mark}(Y)$.[4] The proof is now by induction on $h_x + h_y$, where h_x and h_y are the heights of x and y, respectively.

If $h_x + h_y = 2$, the proof follows easily.

Suppose $h_x + h_y > 2$. Then x and y must have the same left child, say a, and, moreover, EQUAL (x', y') must be true, where x' and y' are the right children of x and y, respectively. It is easy to check (since EQUAL (x, y) is true) that either the descendants[5] of both x' and y' contain a $\circledast$ node or none of the descendants of x' or y' is a $\circledast$ node. In the latter case, it is easy to check that the algorithm is correct. Thus, suppose the descendants of x' and y' contain $\circledast$ nodes, there exist two $\circledast$ nodes u and v such that EQUAL (u, v) is true and h_u and h_v are maximal among the $\circledast$ nodes which are descendants of x' and y', respectively. The rest of the proof follows by the induction hypothesis.

Suppose now that D has two identical normal terms. The proof is similar to that of Lemma 5.4, taking into consideration the fact that D is left-justified. □

THEOREM 5.14. *If D is a left-justified dag with root r such that n is the total number of nodes in D, then the execution time of* REPEATED-TERM (r) *is of* $O(n^2)$.

Proof. Note that if x and y are two nodes of D such that x has n_1 descendants and y has n_2 descendants, then EQUAL (x, y) takes at most $O(n_1 n_2)$ time. Moreover, for each pair of nodes x and y, EQUAL (x, y) is called at most once. The proof of the theorem follows from these observations. □

We collect all the facts we have established about leaf dags in the following theorem.

THEOREM 5.15. *Let D be a leaf dag with n nodes. Then it is possible to check whether D is tree-transformable or not, and to find an equivalent tree, whenever possible, in $O(n^2)$ time.*

This settles the case of leaf dags. Consider the case where the degree of the dag is bounded by a constant d. Then there are at most v^d normal term, where $v = |L(D)|$. Thus checking whether D has two identical normal terms could be done in $O(v^d)$ time. Note that, in this case, all the previous procedures run in linear time. Therefore, we have the following.

THEOREM 5.16. *Let D be an arbitrary σ-dag whose degree is bounded by a constant d. Then transforming D into an equivalent tree, whenever possible, could be done in $O(\max(|N(D)|, |V(D)|^d))$ time.*

Let us remark that if the level of sharing is a fixed constant then it is possible to transform the dag into a leaf dag in polynomial time. Therefore the corresponding problem can be solved efficiently in this case too.

Acknowledgment. We would like to thank the referees for their careful reading of the manuscript and for their constructive comments.

[4] REPEATED-TERM (r) returns also true if a $\oplus$ node has a leaf v as its left and right child. This case will be ruled out by the preceding algorithms.

[5] Including x' and y'.

REFERENCES

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

[AJ] A. V. AHO AND S. C. JOHNSON, *Optimal code generation of expression trees*, J. Assoc. Comput. Mach., 23 (1976), pp. 488–501.

[AJU] A. V. AHO, S. C. JOHNSON AND J. D. ULLMAN, *Code generation for expressions with common subexpressions*, J. Asssoc. Comput. Mach., 24 (1977), pp. 146–160.

[AU] A. V. AHO AND J. D. ULLMAN, *The Theory of Parsing, Translation and Compiling. Vol. II: Compiling*, Prentice-Hall, Englewood Cliffs, NJ, 1973.

[A] J. P. ANDERSON, *A note on some compiling algorithms*, Comm. ACM, 7 (1964), pp. 149–150.

[B] M. A. BREUER, *Generation of optimal codes for expression via factorization*, Comm. ACM, 12 (1969), pp. 333–340.

[BSe] J. L. BRUNO AND R. SETHI, *Code generation for a one-register machine*, J. Assoc. Comput. Mach., 23 (1976), pp. 502–510.

[DS] P. J. DOWNEY AND R. SETHI, *Variations on the common subexpression problem*, unpublished manuscript, 1977.

[GJ1] T. GONZALEZ AND J. JA'JA', *On the complexity of computing bilinear forms with {0, 1} constants*, J. Comput. Systems Sci., 20 (1980), pp. 77–95.

[GJ2] ———, *Computing arithmetic expressions with algebraic identities is hard*, in Proc. 1979 Conference on Information Sciences and Systems, March 1979, pp. 167–173.

[JMMW] D. B. JOHNSON, W. MILLER, B. MINNIHAN AND C. WRATHALL, *Reducibility among floating-point graphs*, J. Assoc. Comput. Mach., 26 (1979), pp. 739–760.

[K] R. M. KARP, *Reducibility among combinatorial problems*, in Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.

[N] I. NAKATA, *On compiling algorithms for arithmetic expressions*, Comm. ACM, 10 (1967), 492–494.

[R] R. R. REDZIEJOWSKI, *On arithmetic expressions and press*, Comm. ACM, 12 (1969), 81–84.

[SS] A. SCHÖNHAGE AND V. STRASSEN, *Schnelle Multiplikation grosser Tählen*, Computing, 7 (1971), pp. 281–292.

[SU] T. SETHI AND J. D. ULLMAN, *The generation of optimal code for arithmetic expressions*, J. Assoc. Comput. Mach., 17 (1970), pp. 715–728.

SIAM J. COMPUT.
Vol. 11, No. 4, November 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1004-0004 $01.00/0

AN APPROXIMATION ALGORITHM FOR THE MAXIMUM INDEPENDENT SET PROBLEM ON PLANAR GRAPHS*

NORISHIGE CHIBA,† TAKAO NISHIZEKI† AND NOBUJI SAITO†

Abstract. In this paper we consider the maximum independent set problem in which one would like to find a maximum set of independent (i.e., pairwise nonadjacent) vertices in a given graph. The problem is NP-complete, and still remains so even if we restrict ourselves to the class of planar graphs. It has been conjectured that there exist no polynomial-time exact algorithms for any NP-complete problems. We present a polynomial-time approximation algorithm for the maximum independent set problem on planar graphs. For a given planar graph having any number n of vertices, our algorithm finds, in $O(n \log n)$ time, an independent set that is necessarily larger in size than half a maximum independent set. Thus the absolute worst case ratio of our algorithm is greater than $\frac{1}{2}$.

Key words. approximation algorithm, maximum independent set problem, bounded worst case ratio, time-complexity, planar graph, vertex-identification

1. Introduction. We consider the maximum independent set problem in which one would like to find a maximum set of independent vertices in a given graph. It is an NP-complete problem, and still remains NP-complete even if we restrict ourselves to the class of planar graphs. There is no good hope for being able to design polynomial-time algorithms for exactly solving any NP-complete problem. Therefore we have to look for a polynomial-time approximation algorithm for the problem, which does not always find a maximum independent set, but does find a large independent set. An approximation algorithm is often evaluated by the worst case ratio: the smallest ratio of the size of an approximation solution to the size of a maximum solution, where the ratio is taken over all problem instances. It is known that if there would exist a polynomial-time algorithm with a constant worst case ratio >0 for the maximum independent set problem on general graphs, then one could design a polynomial-time algorithm with any constant worst case ratio <1 [6]. This fact does not imply that there exist no polynomial-time approximation algorithms with the constant worst case ratio >0 for the problem on a special class of graphs, such as planar graphs of our interest. In fact, Lipton and Tarjan [7] have given an $O(n \log n)$ time approximation algorithm with worst case ratio $1-O(1/\sqrt{\log \log n})$, asymptotically tending to 1 as $n \to \infty$, for the problem on a planar graph with n vertices. Such a ratio is called an "asymptotic worst case ratio." On the other hand, some approximation algorithms have an "absolute worst case ratio," which does not depend on the size n of a graph. For example, one may design a polynomial-time approximation algorithm with absolute worst case ratio $\frac{1}{4}$ with the aid of the four color theorem of Appel and Haken [2]: color a given planar graph with four colors; and simply find the largest color class, that is, the largest set of vertices of the same color. Meanwhile, the five-color algorithm of ours [3] or Matula, Siloach and Tarjan [8] provides a linear time approximation algorithm with absolute worst case ratio $\frac{1}{5}$ for our problem.

In this paper we present an $O(n \log n)$ time approximation algorithm with absolute worst case ratio $\frac{1}{2}$ for the maximum independent set problem on planar graphs. That is, for a given planar graph of any number n of vertices, our algorithm finds, in

* Received by the editors March 27, 1981. This work was partly supported by the Grant in Aid for Scientific Research of the Ministry of Education, Science and Culture of Japan under grants: Cooperative Research (A) 435013 (1979) and YSE (A) 475235 (1979).

† Department of Electrical Communications, Faculty of Engineering, Tohoku University, Sendai, Japan 980.

$O(n \log n)$ time, an independent vertex set that is necessarily larger in size than half a maximum independent set. The algorithm is based on the following ideas:

(i) We can reduce the problem on a general planar graph to one on a planar graph having no vertices of degree 4 or less; such a graph cannot have a large independent set (see Lemma 2); and

(ii) We can design an $O(n \log n)$ time "on-line" algorithm to execute any sequence of vertex-identifications and edge-deletions of a given planar graph. (See [1] for the definition of an on-line algorithm.)

It should be noted that, although the algorithm of Lipton and Tarjan can also guarantee the worst case ratio $\frac{1}{2}$, the number n of vertices must be quite huge, say $2^{2^{400}}$, so the algorithm is not practical.

2. The approximation algorithm. In this section, we present our approximation algorithm. We will show in § 3 that the algorithm correctly finds an independent set with worst case ratio $>\frac{1}{2}$, and show in § 4 that the time complexity of the algorithm is $O(n \log n)$.

We begin by defining some terms. Let $G=(V, E)$ be a graph with vertex set V and edge set E. We consider only a *simple graph* G, that is, a graph with no multiple edges or loops. A graph G is *planar* if it is embeddable in the plane. The *neighborhood* $N(v)$ of a vertex v is the set of all vertices which are adjacent to v. The *degree* of a vertex v of G is denoted by $d(G, v)$ or simply $d(v)$. Let $\delta(G)$ denote the *minimum degree* of vertices in G. A set $I(G)$ of vertices in G is *independent* if no two of them are adjacent. A *maximum independent set* $I^*(G)$ of G is an independent set of maximum cardinality. For a subset S of V, $G-S$ denotes a graph obtained from G by deleting all vertices in S and all edges adjacent to a vertex in S, that is, $G-S$ is a subgraph of G induced by vertex set $V-S$.

We denote by H_4, H_4' and H_4'' the graphs shown in Fig. 1(a), (b) and (c), respectively, which have labelled vertices v, v_1, v_2, v_3 and v_4. These graphs will be referred in the algorithm. We have the following lemma about them.

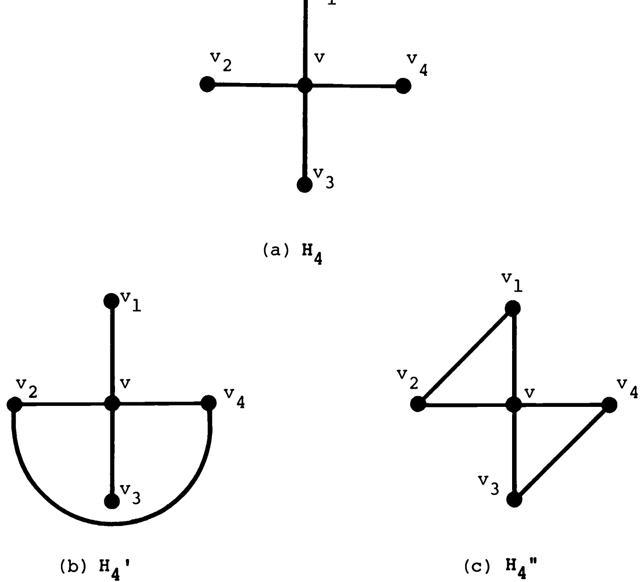


FIG. 1. *Graphs H_4, H_4' and H_4''.*

LEMMA 1. *Let a planar graph* $G=(V,E)$ *contain a vertex* v *of degree* 4 *with* $N(v)=\{v_1, v_2, v_3, v_4\}$, *and let* H *be a subgraph of* G *induced by* $\{v\}\cup N(v)$. *Then, renaming vertices in* $N(v)$ *if necessary, we can assume that either*

(i) H *is isomorphic with* H_4; *or*

(ii) H *contains* H_4' *or* H_4'' *as a subgraph, and moreover* $(v_1, v_3)\notin E$.

Proof. Suppose that H is not isomorphic with H_4. Then there exists at least one edge of both ends in $N(v)$, say (v_2, v_4). If $(v_1, v_3)\notin E$, then (ii) holds with respect to H_4'. If $(v_1, v_3)\in E$, then, renaming vertices in $N(v)$, we may assume that H contains H_4'' as a subgraph. Since G is planar, $(v_1, v_3)\notin E$ or $(v_2, v_4)\notin E$. Thus, renaming vertices if necessary, we have (ii) with respect to H_4''. Q.E.D.

Our algorithm uses a simple recursion. The outline is as follows: from an original planar graph G we construct a planar graph G' smaller than G by deleting a vertex v of minimum degree together with some of its neighbors and adding some edges; find an independent set $I(G')$ of G' larger than half a maximum independent set $I^*(G')$ of G' by recursively applying the algorithm; and form an independent set $I(G)$ of G by adding one or two of the deleted vertices to $I(G')$ so that $I(G)$ is larger in size than half a maximum independent set $I^*(G)$ of G. The method used to construct G' from G varies with $d(v)$ and the structure of a subgraph of G induced by $\{v\}\cup N(v)$. Since G is planar, the minimum degree $d(v)$ is necessarily less than or equal to five. We illustrate all pairs of G and G' in Figs. 2–5. Throughout the description of the algorithm we omit a **begin–end** statement nested in an **if–then–else** statement. In the algorithm, $G=(V,E)$ denotes a graph currently processed and $G'=(V',E')$ a graph reduced from G. The algorithm is as follows.

procedure ISET;
  begin
    procedure INDPT $(G, I(G))$;
      comment Replace the dummy instructions DEGREE3, DEGREE4 and DEGREE5 by the succeeding block with the same label;
      begin
        if $V\neq\emptyset$
          then
          let v be a vertex of minimum degree;
          if $d(v)\leqq 2$
            then
              $G' := G-(\{v\}\cup N(v))$;
              INDPT $(G', I(G'))$;
              $I(G) := I(G')+\{v\}$
            else
              if $d(v)=3$
                then DEGREE3
                else
                  if $d(v)=4$
                    then DEGREE4
                    else comment $d(v)=5$;
                      DEGREE5
          else $I(G) := \emptyset$
      end;
    embed a given planar graph G in the plane;
    INDPT $(G, I(G))$
  end

DEGREE3:
begin
let $N(v)=\{v_1, v_2, v_3\}$;
if $(v_1, v_2), (v_2, v_3), (v_3, v_1) \notin E$
then
comment See Fig. 2;
let G' be the graph obtained from $G-\{v, v_2, v_3\}$ by joining v_1 to all the vertices which were adjacent to v_2 or v_3 in G;
INDPT $(G', I(G'))$;
if $v_1 \in I(G')$
then $I(G) := I(G')+\{v_2, v_3\}$
else $I(G) := I(G')+\{v\}$
else
$G' := G-(\{v\} \cup N(v))$;
INDPT $(G', I(G'))$;
$I(G) := I(G')+\{v\}$
end

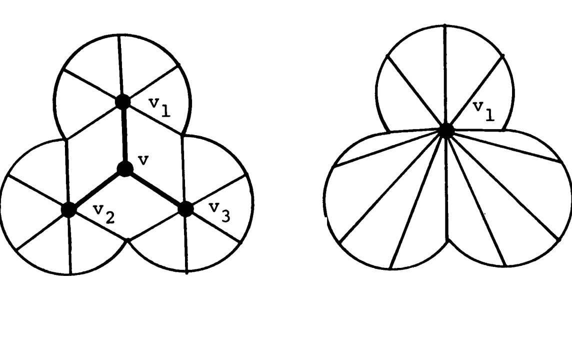


FIG. 2. *G and G′ where* $\delta(G)=3$ *and* $(v_1, v_2), (v_2, v_3), (v_3, v_1) \notin E$.

DEGREE4:
begin
let H be a subgraph of G induced by $\{v\} \cup N(v)$;
if H is isomorphic with H_4
then
comment See Fig. 3;
let vertices in $N(v)=\{v_1, v_2, v_3, v_4\}$ be arranged cyclically counterclockwise about v in the plane embedding of G;
let G' be the graph obtained from $G-\{v, v_2, v_4\}$ by joining v_1 to all vertices which were adjacent to v_2 and joining v_3 to all vertices which were adjacent to v_4 in G;
INDPT $(G', I(G'))$;

if $v_1, v_3 \notin I(G')$
  then $I(G) := I(G') + \{v\}$;
if $v_1 \in I(G')$
  then $I(G) := I(G') + \{v_2\}$;
if $v_1 \notin I(G')$ and $v_3 \in I(G')$
  then $I(G) := I(G') + \{v_4\}$
else comment H contains either H'_4 or H''_4 as a subgraph. (See Fig. 4.)
label vertices in $N(v) = \{v_1, v_2, v_3, v_4\}$ so that H'_4 or H''_4 is a subgraph of the labeled graph H, and $(v_1, v_3) \notin H$;
let G' be the graph obtained from $G - \{v, v_2, v_3, v_4\}$ by joining v_1 to all vertices which were adjacent to v_3 in G;
INDPT $(G', I(G'))$;
if $v_1 \in I(G')$ **then** $I(G) := I(G') + \{v_3\}$
  else $I(G) := I(G') + \{v\}$
end

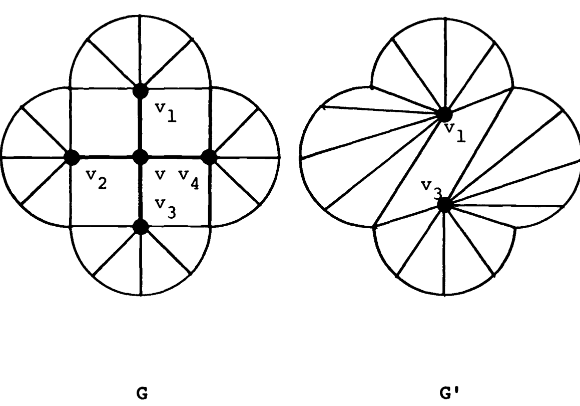


FIG. 3. *G and G' where $\delta(G) = 4$ and the subgraph H of G induced by $\{v\} \cup N(v)$ is isomorphic with H_4.*

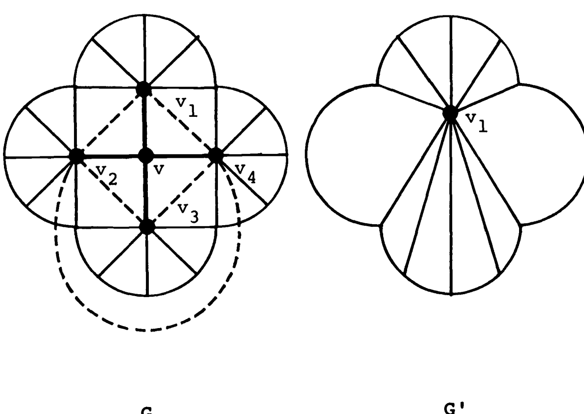


FIG. 4. *G and G' where $\delta(G) = 4$ and H contains H'_4 or H''_4 as a subgraph.*

DEGREE5:
begin
comment See Fig. 5;
let vertices in $N(v)=\{v_1, v_2, v_3, v_4, v_5\}$ be arranged cyclically counterclockwise about v in the plane embedding of G;
wlg assume $(v_1, v_3) \notin E$;
let G' be the graph obtained from $G-\{v, v_2, v_3, v_4, v_5\}$ by joining v_1 to all vertices which were adjacent to v_3;
INDPT $(G', I(G'))$;
if $v_1 \in I(G')$ **then** $I(G) := I(G')+\{v_3\}$
else $I(G) := I(G')+\{v\}$
end

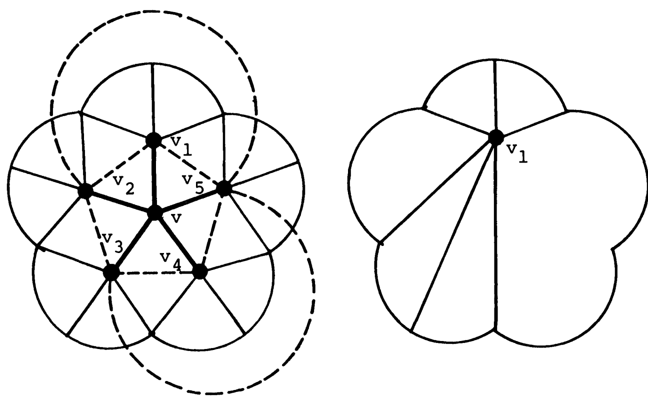


FIG. 5. *G and G' where* $\delta(G)=5$.

3. Correctness of the algorithm. In this section we establish the following theorem.

THEOREM 1. *For any planar graph* $G=(V, E)$, *our algorithm finds an independent set of G with worst case ratio* $>\frac{1}{2}$.

We first present the following result before establishing Theorem 1.

THEOREM 2. *For any planar graph* $G=(V, E)$ *with n vertices, our algorithm finds an independent set of at least* $n/5$ *vertices.*

Proof. We proceed by induction on the number n of vertices of G. One can easily see that for any planar graph having $n \leqq 4$ vertices, our algorithm always finds an independent vertex set of at least one vertex. Hence our claim is true for such a graph.

As the inductive hypothesis we assume that the claim holds for all planar graphs having less than n vertices.

Let $G=(V, E)$ be a planar graph having $|V|=n$ $(\geqq 5)$ vertices. Let v be a vertex of minimum degree in G. Since G is planar, the degree of v is 5 or less [4]. Our algorithm constructs from G a new planar graph $G'=(V', E')$ of fewer vertices than G, and recursively finds an independent set $I(G')$ of G'. We have $|I(G')| \geqq |V'|/5$ by the inductive hypothesis. Although the construction of G' varies with the degree of v, our algorithm always adds at least one vertex to $I(G')$ to form an independent set

$I(G)$ of G, so that $|I(G)| \geqq |I(G')|+1$. Clearly $|V| \leqq |V'|+5$. Combining these three inequalities, we have $|I(G)| \geqq |V|/5$. Q.E.D.

We next establish an upper bound on the size of a maximum independent set $I^*(G)$ of a planar graph G with degree $\delta(G)=5$.

LEMMA 2. *If* $G=(V,E)$ *is a planar graph with* $\delta(G)=5$, *then* $|I^*(G)|<2|V|/5$.

Proof. Let $I^*(G)$ be a maximum independent set of G, and let $B=(V, E_B)$ be a spanning subgraph of G obtained from G by deleting all the edges of both ends in $V-I^*(G)$. Since B is a planar bipartite graph of at least three vertices, we have $|E_B| \leqq 2|V|-4$ [4]. Since $\delta(G)=5$, B is a bipartite graph with partite sets $I^*(G)$ and $V-I^*(G)$, and each vertex in $I^*(G)$ has the same degree in B as it had in G, we have

$$|E_B| = \sum_{v \in I^*(G)} d(G, V) \geqq 5|I^*(G)|.$$

Combining the two inequalities above, we have $|I^*(G)|<2|V|/5$. Q.E.D.

We are now ready to prove Theorem 1.

Proof of Theorem 1. Since the set $I(G)$ found by the algorithm is clearly independent in G, it is sufficient to prove that $|I(G)|/|I^*(G)|>\frac{1}{2}$. We proceed by induction on the number n of vertices of G. For any planar graph G with $n \leqq 4$ vertices, we can easily verify that $|I(G)|/|I^*(G)|>\frac{1}{2}$.

As the inductive hypothesis, we assume that our claim is true for all planar graphs having less than n vertices.

Now suppose that G is a planar graph with n vertices, $n \geqq 5$. Since G is planar, $\delta(G) \leqq 5$. If $\delta(G)=5$, we have $|I(G)| \geqq n/5$ by Theorem 2, and $|I^*(G)|<2n/5$ by Lemma 2. Therefore $|I(G)|/|I^*(G)|>\frac{1}{2}$, as desired. Thus we can assume $\delta(G) \leqq 4$. Let G' be the graph constructed from G by our algorithm; the construction method varies with $\delta(G)$. Since G' has fewer vertices than G, we have $|I(G')|/|I^*(G')|>\frac{1}{2}$ by the inductive hypothesis. Our algorithm always adds at least one vertex to $I(G')$ to form $I(G)$, so that $|I(G)| \geqq |I(G')|+1$. Hence, we shall show that $|I^*(G)| \leqq |I^*(G')|+2$ which, together with the two inequalities above, leads to the desired result $|I(G)|/|I^*(G)|>\frac{1}{2}$.

Let v, v_1, v_2, v_3 and v_4 be vertices of G defined in the algorithm, that is, v is a vertex of minimum degree and $N(v)=\{v_1, \cdots, v_i\}$ where $d(v)=i$. Let $I^*(G)$ be a maximum independent set of graph G. We consider three cases depending on $\delta(G)$.

Case 1. *Either* $\delta(G) \leqq 2$ *or* $\delta(G)=3$ *and* G *has at least one of the edges* (v_1, v_2), (v_2, v_3) *and* (v_3, v_1).

In this case $G'=G-\{v\} \cup N(v)$. Since $I^*(G)-\{v\} \cup N(v)$ is independent in G', and $|(\{v\} \cup N(v)) \cap I^*(G)| \leqq 2$, we have $|I^*(G')| \geqq |I^*(G)-\{v\} \cup N(v)| \geqq |I^*(G)|-2$, as desired.

Case 2. $\delta(G)=3$ *and* (v_1, v_2), (v_2, v_3), $(v_3, v_1) \notin E$. (*See Fig.* 2.)

In this case G' is the graph obtained from $G-S$, $S=\{v, v_2, v_3\}$, by joining v_1 to all the vertices which were adjacent to v_2 or v_3 in G. Clearly $|I^*(G) \cap S| \leqq 2$. We consider two subcases depending on $|I^*(G) \cap S|$ as follows.

Subcase 2.1. $|I^*(G) \cap S|=0$ *or* 1. Clearly $I^*(G)-\{v_1\} \cup S$ is independent in G'. Therefore $|I^*(G')| \geqq |I^*(G)|-2$.

Subcase 2.2. $|I^*(G) \cap S|=2$. Since vertex v_1 of G' is joined to vertices in $N(v_2) \cup N(v_3)$ by edges which might not exist in G, $I^*(G)-S$ is not necessarily independent in G'. However, in this case, $I^*(G) \cap S=\{v_2, v_3\}$, and hence $(N(v_2) \cup N(v_3)) \cap I^*(G)=\emptyset$. Therefore $I^*(G)-S$ is independent in G'. Thus $|I^*(G')| \geqq |I^*(G)|-2$.

Case 3. $\delta(G)=4$. Let H be the subgraph of G induced by $\{v\} \cup N(v)$. Then by Lemma 1, we can assume, without loss of generality, that either H is isomorphic with

graph H_4, or H contains a subgraph isomorphic with graph H_4' or H_4'' (H_4, H_4' and H_4'' are depicted in Fig. 1).

First assume that H is isomorphic with H_4. Let G' be the graph obtained from $G-S$ in the manner described in the algorithm, where $S=\{v, v_2, v_4\}$. (Refer to G and G' in Fig. 3.) Since clearly $|I^*(G)\cap S|\leqq 2$, we consider three subcases.

Subcase 3.1. $|I^*(G)\cap S|=0$. Since $I^*(G)-\{v_1, v_3\}$ is independent in G', $|I^*(G')|\geqq|I^*(G)|-2$.

Subcase 3.2. $|I^*(G)\cap S|=1$. Suppose first that $I^*(G)\cap S=\{v\}$. Then v_1, $v_3\notin I^*(G)$, and hence $I^*(G)-\{v\}$ is independent in G'. Thus $|I^*(G')|\geqq|I^*(G)-1$. Suppose next that $I^*(G)\cap S=\{v_2\}$ or $\{v_4\}$. We can assume without loss of generality that $I^*(G)\cap S=\{v_2\}$. Since $N(v_2)\cap I^*(G)=\varnothing$, $I^*(G)-\{v_2, v_3\}$ is independent in G', and hence $|I^*(G')|\geqq|I^*(G)|-2$.

Subcase 3.3. $|I^*(G)\cap S|=2$. In this case $I^*(G)\cap S$ must be $\{v_2, v_4\}$. Since $N(v_2)\cap I^*(G)=\varnothing$ and $N(v_4)\cap I^*(G)=\varnothing$, $I^*(G)-\{v_2, v_4\}$ is independent in G', so $|I^*(G')|\geqq|I^*(G)|-2$.

Next assume that H contains H_4' or H_4'' as a subgraph. Let G' be the graph obtained from $G-S$ in the manner described in the algorithm, where $S=\{v, v_2, v_3, v_4\}$. (Refer to G and G' in Fig. 4.) Since $|I^*(G)\cap S|\leqq 2$, we consider two further subcases.

Subcase 3.4. $|I^*(G)\cap S|=0$ *or* 1. Since $I^*(G)-\{v_1\}\cup S$ is independent in G', $|I^*(G')|\geqq|I^*(G)|-2$.

Subcase 3.5. $|I^*(G)\cap S|=2$. Suppose first that H contains H_4' as a subgraph, then $I^*(G)\cap S$ must be either $\{v_2, v_3\}$ or $\{v_3, v_4\}$. Since $N(v_3)\cap I^*(G)=\varnothing$ in either case, $I^*(G)-S$ is independent in G', so $|I^*(G')|\geqq|I^*(G)|-2$. Suppose next that H contains H_4'' as a subgraph, then $I^*(G)\cap S$ must be either $\{v_2, v_3\}$ or $\{v_2, v_4\}$. Since $v_1\notin I^*(G)$ in either case, $I^*(G)-S$ is independent in G', so $|I^*(G')|\geqq|I^*(G)|-2$.

Thus the proof is completed. Q.E.D.

The bound on the worst case ratio given in Theorem 1 is sharp in the sense that there exist infinitely many graphs for which our algorithm possibly realizes the bound. We illustrate an example of these graphs G in Fig. 6. For the graph G our algorithm finds, in the worst case, the set of all black vertices as an independent set $I(G)$. Clearly $|I(G)|=(|I^*(G)|+1)/2$.

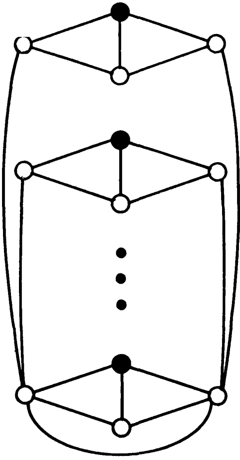


FIG. 6. *An illustrating graph.*

4. Time-complexity. In this section we establish the following theorem.

THEOREM 3. *Our algorithm requires at most $O(|V| \log |V|)$ time on a planar graph $G=(V, E)$.*

We first define some terms. Let u and v be two vertices of a graph $G=(V, E)$. A *vertex-identification* $\langle u, v\rangle$ is an operation on G which identifies u and v, that is, removes u and v and adds a new vertex w adjacent to those vertices to which u or v was adjacent. Our algorithm frequently uses this operation to construct G' from G. Consider a sequence σ of vertex-identifications on G:

$$\sigma = \langle u_1, v_1\rangle\langle u_2, v_2\rangle \cdots \langle u_m, v_m\rangle,$$

where $m \leqq n-1$ and $n=|V|$. Let $G_i=(V_i, E_i)$, $i=1, 2, \cdots, m+1$, be the graph obtained from G by the first $i-1$ vertex-identifications, where $G_1=G$. The process of vertex-identifications can be represented by a *vertex-identification forest* $F(G, \sigma)$, which is a collection of binary trees, recursively defined as follows:

1. If $\sigma=\emptyset$, i.e., an empty sequence, then $F(G, \sigma)$ is a forest with $|V|$ isolated nodes, each corresponding to a vertex of G; and

2. If $\sigma=\sigma' \cdot \langle u, v\rangle$, and in place of vertices u and v a new vertex w is added to the graph resulted from G by the sequence σ', then $F(G, \sigma)$ is a forest formed by adding to $F(G, \sigma')$ a new node w together with arcs (w, u) and (w, v).

Thus each leaf of $F(G, \sigma)$ represents a single vertex of G, and each internal node w represents a vertex of a graph appeared on some stage of the execution of σ; w corresponds to the set of all the vertices of G represented by leaves of $F(G, \sigma)$ that are descendants of w; all these vertices are identified into the single vertex w at the stage. Figure 7 illustrates an example of $F(G, \sigma)$.

We recursively define the "*level*" $l(w)$ and "*degree*" $\deg(w)$ of a node w of $F(G, \sigma)$ as follows:

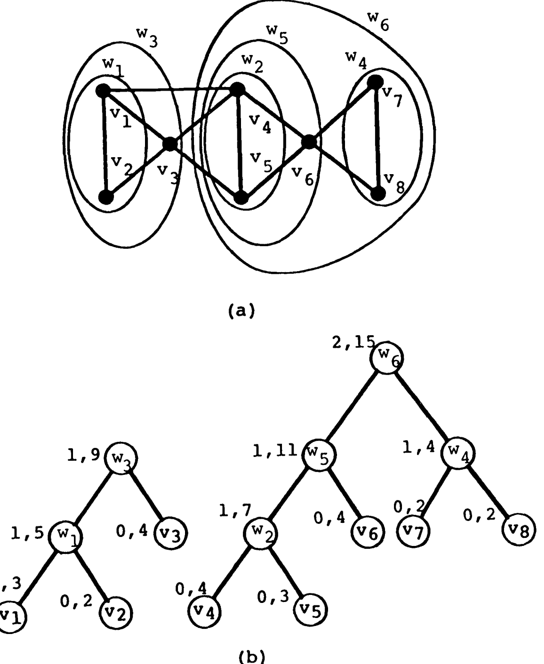


FIG. 7. *A sequence of vertex-identifications* $\sigma=\langle v_1, v_2\rangle\langle v_4, v_5\rangle\langle w_1, v_3\rangle\langle v_7, v_8\rangle\langle w_2, v_6\rangle\langle w_4, w_5\rangle$, (a) *graph G, and* (b) $F(G, \sigma)$, *where the first number associated with a node is the level of the node, and the second is the degree.*

1. For each leaf node w of $F(G, \sigma)$

$$l(w)=0 \quad \text{and} \quad \deg(w)=d(G, w),$$

and

2. if an internal node w of $F(G, \sigma)$ has two sons u and v, then

$$l(w)=\begin{cases} l(v)+1 & \text{if } l(u)=l(v), \\ l(v) & \text{if } l(u)<l(v), \end{cases}$$

$$\deg(w)=\deg(u)+\deg(v).$$

We furthermore call a node w of F a *highest node of level* $l(w)$ if w has no ancestors of level $l(w)$.

As a data structure to represent a graph G, we use an adjacency matrix A together with adjacency lists L. Each adjacency list is doubly linked, and the list for vertex $u \in V$ is denoted by $L(u)$. The $u-v$ element $A(u, v)$ of A is 1 if and only if vertex v is adjacent to vertex u. $A(u, v)$ has also a pointer to the element "v" in $L(u)$ if $(u, v) \in E$. Note that one can initialize the matrix A in $O(|E|)$ time. (See [1, Ex. 2.12].) In addition to A and L, we use two arrays D and DP together with six lists DLIST (i), $0 \leqq i \leqq 5$, so that one can find a vertex of minimum degree in a constant time. An element $D(v)$ of array D contains the value of the degree $d(v)$, $v \in V$. DLIST (i) contains all the vertices of degree i, $0 \leqq i \leqq 5$. $DP(v)$ has a pointer to an element "v" in DLIST $(d(v))$ if $d(v) \leqq 5$. Figure 8 illustrates an example of such a data structure.

LEMMA 3. *Let $G=(V, E)$ be a graph, and let σ be any sequence of edge-deletions on G. There exists an on-line algorithm to execute σ in $O(|E|)$ time.*

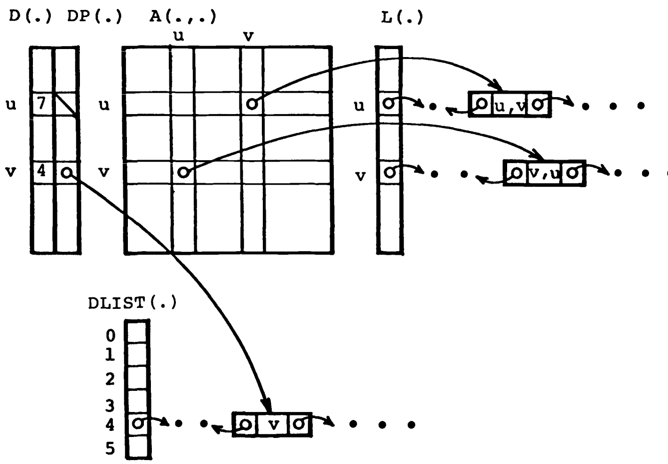


FIG. 8. *Illustration of the data structure for our algorithm.*

Proof. In order to delete an edge, say (u, v), of G, we substitute 0 for elements $A(u, v)$ and $A(v, u)$, and delete elements u from list $L(v)$ and v from $L(u)$. One can directly access these elements in the lists via the pointers in the corresponding elements in A, and one can delete an element in the list in a constant time since it is doubly linked. Thus the execution of each edge-deletion requires a constant time. Of course, σ contains at most $|E|$ edge-deletions. Hence the total amount of time required to execute σ is at most $O(|E|)$. Q.E.D.

LEMMA 4. *Let $G=(V,E)$ be a graph, and let σ be any sequence of vertex-identifications on G. There exists an on-line algorithm to execute σ in at most $O(|E|\log|V|)$ time.*

Proof. Let $\sigma=\langle u_1, v_1\rangle\langle u_2, v_2\rangle\cdots\langle u_m, v_m\rangle$, where $m\leqq n-1$ and $n=|V|$. Let G_i, $i=1,2,\cdots,m+1$, be the graph obtained from G by the first $i-1$ vertex-identifications.

In order to update the data structure for G_{i+1} from that of G_i, we shall modify some adjacency lists and entries of A. Assume $d(G_i, u_i)\leqq d(G_i, v_i)$. Scanning all the elements x of list $L(u_i)$ (i.e., all the vertices x adjacent to u_i), we delete element u_i from $L(x)$, and add element x to $L(v_i)$ if $x\neq v_i$ and x is not in $L(v_i)$. Note that we consider only a simple graph, which has neither multiple edges nor loops. In our data structure, element x in $L(v_i)$ can be directly accessed via the pointer in $A(v_i, x)$ to x. We also modify some entries of A appropriately according to the above modification of lists. Then we remove the list $L(u_i)$, and regard the updated list $L(v_i)$ as a list $L(w_i)$ for a vertex w_i added to G_i in place of u_i and v_i. Hence the time required to execute the vertex-identification $\langle u_i, v_i\rangle$ is proportional to $d(G_i, u_i)$. Therefore the total amount of time $T(\sigma)$ required for σ satisfies

$$T(\sigma)\leqq O\Big(\sum_{1\leqq i\leqq m} \mathrm{MIN}\,(d(G_i, u_i), d(G_i, v_i))\Big). \tag{1}$$

We shall show that the summation in (1) is at most a constant times $|E|\log|V|$.

Let a node u of $F(G,\sigma)$ correspond to a subset U of V, then U is the set of all leaves of $F(G,\sigma)$ that are descendants of u. It follows from the definition that $\deg(u)$ is equal to the total number of edges of G with an end in U, each edge with both ends in U being counted twice. Since all the vertices in U are identified as a single vertex u in G_i, we have

$$d(G_i, u_i)\leqq \deg(u_i) \tag{2}$$

and

$$d(G_i, v_i)\leqq \deg(v_i) \tag{3}$$

for $i=1,2,\cdots,m$. Note that G_i is a simple graph. Let H_j denote the set of all the highest nodes of level j in F, that is, the nodes of level j having no ancestors of level j. Then we claim that

$$\sum_{1\leqq i\leqq m} \mathrm{MIN}\,(d(G_i, u_i), d(G_i, v_i))\leqq \sum_{0\leqq j\leqq h}\sum_{x\in H_j}\deg(x), \tag{4}$$

where h is the highest level of nodes in F.

Proof of the claim. It follows from the definition of "level" that every leaf of F is a highest node of level 0, and that every internal node has at least one son which is highest of some level. For each term $\langle u_i, v_i\rangle$ of σ, F has an internal node having sons u_i and v_i. Thus at least one of u_i and v_i appears in the summations of the right-hand side of (4), that is, for each $i=1,2,\cdots,m$

$$u_i \text{ or } v_i\in \bigcup_{0\leqq j\leqq h} H_j. \tag{5}$$

Combining (2), (3) and (5), we have (4).

In order to complete the proof, we shall show that the right-hand side of (4) is at most $O(|E|\log|V|)$. For $j=0,1,\cdots,h$, every node in H_j is not a descendant of any other node in H_j, so that all the subsets of V corresponding to nodes in H_j are

pairwise disjoint in G. Therefore we have

$$\sum_{x \in H_j} \deg(x) \leqq 2|E|,$$

because each edge of G is counted at most twice in the summation. By the definition of level, each internal node of level j has at least two descendants of level $j-1$. Therefore h is at most $\log_2 |V|$. Hence we have

$$\sum_{0 \leqq j \leqq h} \sum_{x \in H_j} \deg(x) \leqq 2|E| \log |V|,$$

completing the proof. Q.E.D.

THEOREM 4. *For any sequence σ of vertex-identifications and edge-deletions of a graph $G=(V, E)$, there exists an on-line algorithm to execute σ in at most $O(|E| \log |V|)$ time.*

Proof. By Lemma 3, the time required to execute all the edge-deletions in σ is $O(|E|)$. Hence we shall show that the time T' required to execute all the remaining vertex-identifications is at most $O(|E| \log |V|)$. Consider a subsequence σ' of σ consisting of all vertex-identifications. Let G_i be the graph resulted from G after the first i terms of σ, including j vertex-identifications. Let G'_j be the graph resulted from G after the first j vertex-identifications of σ'. Since G_i is a subgraph of G'_j, $d(G_i, v) \leqq d(G'_j, v)$ for every vertex v of G_i. Noting this fact together with (1), we have $T' \leqq T(\sigma')$. Therefore by Lemma 4 $T' \leqq O(|E| \log |V|)$, which establishes our claim. Q.E.D.

We are now ready to prove Theorem 3.

Proof of Theorem 3. One can embed a planar graph G in the plane in $O(|V|)$ time, using the algorithm of Hopcroft and Tarjan [5].

In our algorithm, the problem of finding an independent set of G is reduced to that of finding a smaller graph G', which we can obtain from G by deleting a vertex of minimum degree together with some other modifications. Since we always keep all the vertices of degree i, $i=0, 1, \cdots, 5$, in list DLIST (i) (see Fig. 8), we can find a vertex v of minimum degree (necessarily at most 5) in a constant time. Note that we can delete an element from the doubly linked list DLIST, or insert an element into it, in a constant time, since we can directly access an element v of DLIST via the pointer DP (v). Although the method to form $I(G)$ from $I(G')$ varies with the type of a subgraph of G induced by $\{v\} \cup N(v)$, one can recognize the type in a constant time. Therefore, given $I(G')$ obtained by recursively applying the algorithm, we can form $I(G)$ in a constant time. Since the graph G' reduced from G has fewer vertices than G, such a reduction occurs at most $|V|$ times. Therefore the total amount of time required to these operations is at most $O(|V|)$. All the remaining operations, which are used to construct G' from G, are regarded as a sequence of three kinds of operations: vertex-identification, edge-deletion, and deletion of an isolated vertex. For example, G' of Fig. 3 can be constructed from G by a sequence of four edge-deletions, two vertex-identifications, and one deletion of an isolated vertex. Theorem 4 implies that any sequence of the first two kinds of operations can be executed within the desired time, while one can easily see that any sequence of deletions of an isolated vertex is executed in $O(|V|)$ time. Thus the proof is completed. Q.E.D.

Acknowledgment. We wish to thank Dr. T. Asano and Dr. K. Takamizawa for their valuable suggestions and discussions on the subjects.

REFERENCES

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

[2] K. APPEL AND W. HAKEN, *Every planar map is four colourable, Part I: discharging*, Illinois J. Math., 21 (1977), pp. 429–490.

[3] N. CHIBA, T. NISHIZEKI AND N. SAITO, *A linear 5-coloring algorithm of planar graphs*, J. Algorithms, 2 (1981), pp. 317–327.

[4] F. HARARY, *Graph Theory*, rev. ed., Addison-Wesley, Reading, MA, 1972.

[5] J. E. HOPCROFT AND R. E. TARJAN, *Efficient planarity testing*, J. Assoc. Comput. Mach., 21 (1974), pp. 549–568.

[6] J. E. HOPCROFT AND J. D. ULLMAN, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.

[7] R. J. LIPTON AND R. E. TARJAN, *Applications of a planar separator theorem*, Proc. 18th Annual Symposium on Foundations of Computer Science, IEEE Computer Society, Long Beach, CA, 1977, pp. 162–170.

[8] D. W. MATULA, Y. SHILOACH AND R. E. TARJAN, *Two linear-time algorithms for five-coloring a planar graph*, SIAM J. Alg. Disc. Meth., submitted.

SIAM J. COMPUT.
Vol. 11, No. 4, November 1982
© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1104-0005 $01.00/0

HAMILTON PATHS IN GRID GRAPHS*

ALON ITAI†, CHRISTOS H. PAPADIMITRIOU‡ AND JAYME LUIZ SZWARCFITER§

Abstract. A grid graph is a node-induced finite subgraph of the infinite grid. It is rectangular if its set of nodes is the product of two intervals. Given a rectangular grid graph and two of its nodes, we give necessary and sufficient conditions for the graph to have a Hamilton path between these two nodes. In contrast, the Hamilton path (and circuit) problem for general grid graphs is shown to be NP-complete. This provides a new, relatively simple, proof of the result that the Euclidean traveling salesman problem is NP-complete.

Key words. Hamilton circuit, Hamilton path, grid graphs, rectangular grid graphs, NP-complete problem, Euclidean traveling salesman problem

1. Introduction. Let G^∞ be the infinite graph whose vertex set consists of all points of the plane with integer coordinates and in which two vertices are connected if and only if the (Euclidean) distance between them is equal to 1. A *grid graph* is a finite, node-induced subgraph of G^∞. Thus, a grid graph is completely specified by its vertex set. Let v_x and v_y be the *coordinates* of the vertex v. We say that vertex v is *even* if $v_x+v_y\equiv 0 \pmod 2$; otherwise, v is *odd*. It is immediate that all grid graphs are bipartite, with the edges connecting even and odd vertices.

Let $R(m,n)$ be the grid graph whose vertex set is $V(R(m,n))=\{v: 1\leqq v_x\leqq m$ and $1\leqq v_y\leqq n\}$. A *rectangular graph* is a grid graph which, for some m and n, is isomorphic to $R(m,n)$. Thus m and n, the *dimensions*, specify a rectangular graph up to isomorphism.

Let s and t be distinct vertices of a graph G. We say that the *Hamilton path problem* (G,s,t) has a solution if there exists a Hamilton path from s to t in G. In this paper we examine the Hamilton path problem for grid graphs; rectangular grid graphs were examined first in [LM]. In § 2 we show that the Hamilton path and Hamilton circuit problems for general grid graphs are NP-complete. Consider now a bipartite graph $B=(V^0\cup V^1),E)$. If $|V^0|=|V^1|+1$, then all Hamilton paths of B must start and end at vertices of V^0. If $(R(m,n),s,t)$, with $m\times n$ odd, has a solution, then the number of even vertices is greater by one than that of the odd vertices. Hence, a necessary condition for the solvability of $(R(m,n),s,t)$ is that both s and t be even. In § 3 it is shown that this condition is also *sufficient* for nontrivial (i.e., $m,n>1$) odd rectangular graphs. If $m\times n$ is even, then a solution is possible only if s and t have different parity. However, this condition is not sufficient. There are three families of configurations for which even though s and t have different parity $(R(m,n),s,t)$ has no solution. In § 3 we give the precise necessary and sufficient conditions for a Hamilton path problem to have a solution. Partial results in this direction were first proved in [LM].

* Received by the editors September 22, 1980, and in final form August 25, 1981.

† Department of Computer Science, Technion, Haifa, Israel. Part of this work was conducted while this author was visiting the Electrical Engineering and Computer Science Department, University of California at Berkeley, and the Laboratory for Computer Science, Massachusetts Institute of Technology.

‡ Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139, and National Technical University of Athens, Greece. The work of this author was supported by the National Science Foundation under grant MCS 76-01193.

§ Universidade Federal do Rio de Janeiro, Brasil. Present address: Computer Science Division, University of California, Berkeley, California 94720. The work of this author was supported by the Conselho Nacional de Desenvolvimento Cientifico e Technologico (CNPq), Brasil, processo 574/78.

2. NP-completeness. Before showing that finding Hamilton paths and circuits in grid graphs is NP-complete, we first show several lemmas:

LEMMA 2.1. *The Hamilton circuit problem for planar bipartite graphs with maximum degree* 3 *is* NP-*complete.*

Proof. The Hamilton circuit problem is NP-complete for planar digraphs such that for all vertices v:

$$\text{indegree}(v)+\text{outdegree}(v)=3 \quad \text{(see [GJ], [P]).}$$

To prove the lemma, we conduct for all vertices v the appropriate transformation as illustrated in Fig. 2.1. The resulting graph is planar (the digraph was), bipartite and has maximum degree less than or equal to 3. □

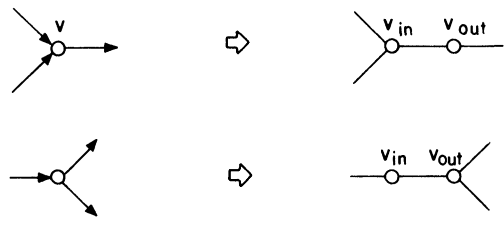


FIG. 2.1

Let $B=(V^0 \cup V^1, E)$ be a bipartite graph, G_1 a rectangular grid graph and let emb be a one-to-one function from $V^0 \cup V^1$ to the vertices of G_1 and from E to paths in G_1. We say that emb is a *parity-preserving embedding* of B into G_1 if:

1. The vertices V^0 are mapped to even vertices of G_1. (If $v \in V^0$, then emb (v) is even.)
2. The vertices of V^1 are mapped to odd vertices of G_1. (If $v \in V^1$, emb (v) is odd.)
3. The edges of B are mapped to vertex-disjoint (except perhaps for endpoints) paths of G_1 (i.e., if $vu \in E(B)$, then emb (vu) is a path P from emb (v) to emb (u), and the intermediate vertices of P do not belong to any other path).

See Fig. 2.2 for an example of a parity-preserving embedding.

LEMMA 2.2. *If B is a bipartite planar graph with n vertices and maximum degree* 3, *then we can construct in polynomial time a parity preserving embedding of B into a rectangular graph $R(kn, kn)$ (for some constant k).*

Proof. It is a quite well-known and straightforward result (see, for example, [S], [V]) that all cubic planar graphs with n nodes can be embedded in $R(n, n)$. Our extra requirement, preserving parity, can be accommodated by multiplying the scale by 3 and moving the vertices "locally" as in Fig. 2.3. □

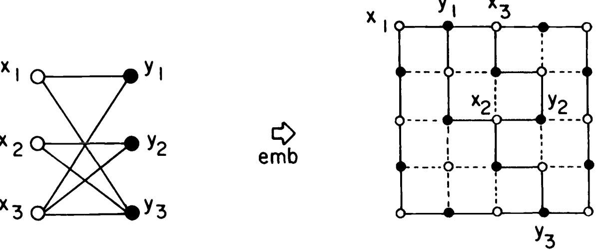


FIG. 2.2

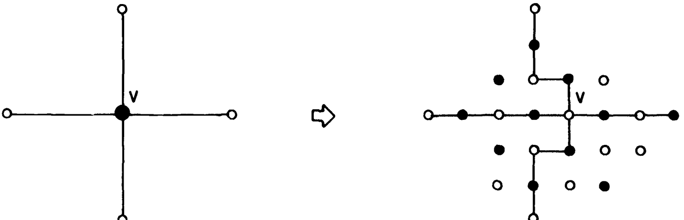


FIG. 2.3

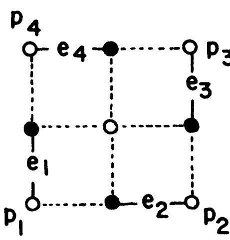


FIG. 2.4

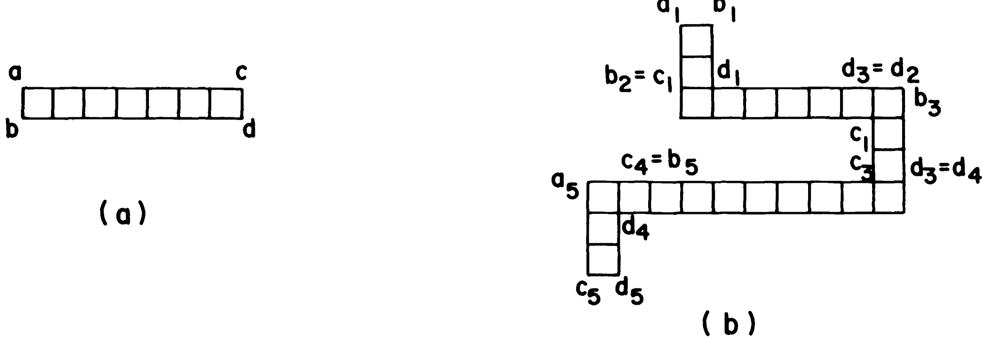


FIG. 2.5

To show NP-completeness of the Hamilton circuit problem for grid graphs, we shall transform an arbitrary planar, bipartite, cubic graph B into a grid graph. Each vertex of B will correspond to a 9-*cluster*, the nine vertices of a square of size 2 (Fig. 2.4).

LEMMA 2.3. *Let C_9 be a 9-cluster, as in Fig. 2.4. Then for all $1 \leqq i < j \leqq 4$, there exists a Hamilton path from p_i to p_j which contains all four edges $\{e_1, e_2, e_3, e_4\}$.*

Proof. By inspection. □

A *strip* is a rectangular graph with minimum dimension 2 (Fig. 2.5a). The strip with corners a, b, c, d (a is adjacent to b and c is adjacent to d) is denoted $S(a, b; c, d)$.

A *tentacle* T is a grid graph which is a union of a series of strips,

$$T\colon S(a_1, b_1; c_1, d_1) \cup S(a_2, b_2; c_2, d_2) \cup \cdots \cup S(a_k, b_k; c_k, d_k),$$

such that both

$$c_i, d_i \in V(S(a_{i+1}, b_{i+1}; c_{i+1}, d_{i+1})), \qquad i = 1, \cdots, k-1$$

and one of

$$a_{i+1}, b_{i+1} \in V(S(a_i, b_i; c_i, d_i)), \qquad i = 1, \cdots, k-1;$$

there is no other intersection between the vertex sets of the strips; and each edge of T belongs to one of the S's.

From the definition the overlap must be in the corners of the strips as in Fig. 2.5a. The vertices a_1, b_1, c_k, d_k are the *corners* of T.

LEMMA 2.4. *Let s and t be corners of a tentacle T. There is a* HP *from s to t in T if and only if s and t have different parity.*

Proof. By an easy induction on the number of strips. □

THEOREM 2.1. *The Hamilton circuit problem for grid graphs is* NP-*complete.*

Proof. Given a planar, degree $\leqq 3$, bipartite graph B, we shall construct a grid graph G_9 such that G_9 has a Hamilton circuit if and only if there exists a Hamilton circuit in B.

First we embed B in a grid graph G_1, as in Lemma 2.2. The graph G_9 will be an induced subgraph of the grid resulting by multiplying the scale of G_1 by 9. Each image of a vertex $w = (w_x, w_y)$ of B corresponds to the following 9-cluster of G_9:

$$\{z \mid w_x \leqq z_x \leqq w_x + 2, \ w_y \leqq z_y \leqq w_y + 2\}.$$

The edges of B are simulated by tentacles. Suppose vu is an edge leading from $v \in V^0(B)$ to $u \in V^1(B)$. Consider the path in G_1 corresponding to vu. G_9 will include the blown up image of this path and another layer to obtain a tentacle. Some care must be taken as to how the tentacle is connected to the 9-clusters corresponding to v and u. By the construction the corners of the 9-cluster corresponding to v are all even. If in G_1 vu leaves v from below, then the corresponding tentacle is connected as in Fig. 2.6a (recall that v is even). The other cases are symmetric (just rotate the figure 90°, 180° or 270°). Since u is odd the connection is completed as in Fig. 2.6b. This concludes the description of G_9. An example is shown in Fig. 2.8.

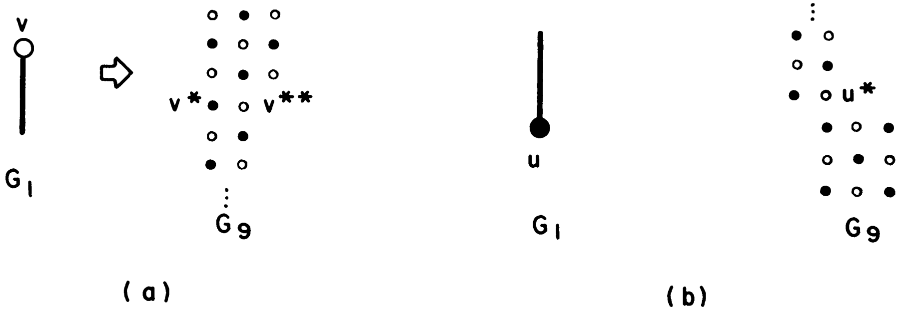


FIG. 2.6

By Lemma 2.4 a tentacle T_{uv} can be covered by two Hamilton paths: the *cross* path from v^* to u^* (Fig. 2.7a) and the *return* path from v^* to v^{**} (Fig. 2.7b).

The following two claims complete the proof of the theorem.

CLAIM 1. *Let HC_B be a Hamilton circuit in B. Then there exists a Hamilton circuit HC_9 in G_9.*

Proof. HC_9 is constructed as follows: If $vu \in HC_B$, the tentacle T_{uv} is covered in HC_9 by a cross path; otherwise, it is covered by a return path. The clusters themselves

FIG. 2.7

FIG. 2.8

are covered as in Lemma 2.3. The partial paths can be connected to constitute a Hamilton circuit. (Some edges of type e_i in Fig. 2.4 must be deleted.)

CLAIM 2. *If HC_9 is a Hamilton circuit in G_9, then there exists a Hamilton circuit HC_B in B.*

Proof. By construction each tentacle T_{vu} is covered either by a cross path or by a return path (connected to v). C_B consists of all edges corresponding to tentacles covered by cross paths. This is a Hamilton circuit because each 9-cluster cannot be covered by HC_B unless it is incident upon exactly two cross paths. □

COROLLARY 1. *The Hamilton path problem for grid graphs is* NP-*complete.*

Proof. Reduction from the Hamilton circuit problem for grid graphs. Let G be a grid graph without degree-1 nodes. Since it is finite, it must have a vertex s of degree 2. Let t be any of the neighbors of s. Then (G, s, t) has a solution if and only if G has a Hamilton circuit. □

By adding two nodes of degree 1 to G, we may also conclude that the Hamilton path problem without specified endpoints is NP-complete.

A *rectangular subgrid graph* is a subgraph (*not* necessarily induced) of G^∞ that has $V(R(m, n))$ as its vertices for some $m, n > 0$.

COROLLARY 2. *The Hamilton circuit and path problems for rectangular subgrid graphs are* NP-*complete.*

Sketch of proof. The grid graph constructed in the proof of the theorem may be considered as a rectangular one minus certain "holes". We can now "fill" these holes with long paths so as to transform the graph into a rectangular *subgrid* one. □

The Euclidean version of the traveling salesman problem was proved NP-complete in [Pa]. It is interesting, however, to notice that the Hamilton circuit problem for grid graphs is a *special case* of the Euclidean traveling salesman problem, with cities the nodes of the grid graph and with length of the tour equal to the number of nodes. We therefore have:

COROLLARY 3. *The Euclidean traveling salesman problem is* NP-*complete.*

We notice that this proof is much simpler than that in [Pa]. It also avoids an annoying complication having to do with the precision in which the distances are calculated (see [Pa]).

3. Hamilton path problems in rectangular graphs.

3.1. Necessary conditions. Let $B = (V^0 \cup V^1, E)$ be a bipartite graph with $|V^0| \geqq |V^1|$. We will think of the vertices of B as colored by two colors, black and white. All the vertices of V^0 will be colored by one color, the *majority color*, and the vertices V^1 by the *minority color*.

The Hamilton path problem (B, s, t) is *color compatible* if

(1) B is *even* ($|V^0| = |V^1|$) *and* s and t have different color *or*

(2) B is *odd* ($|V^0| = |V^1| + 1$) *and* s and t are colored by the majority color (i.e., $s, t \in V^0$).

Since the vertices of any Hamilton path alternate between the two colors, color compatibility is a necessary condition for the existence of a Hamilton path. Another source of necessary conditions arises from the connectivity of the graph. If s or t is a separating vertex (i.e., $G - \{s\}$ or $G - \{t\}$ is not connected), then there exists no s, t Hamilton path in G. For rectangular graphs this implies the following conditions for the graph to have no s, t Hamilton path.

(F1) G is a 1-rectangle, and either s or t is not a corner (Fig. 3.1a).

Also, no s, t Hamilton path exists if $\{s, t\}$ is a separating pair, i.e., $G - \{s, t\}$ is not connected. For rectangular graphs this implies

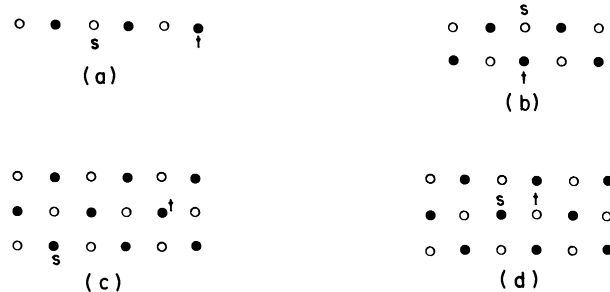


FIG. 3.1

(F2) G is a 2-rectangle, and st is a nonboundary edge (i.e., st is an edge, and it is not on the outermost face, see Fig. 3.1b).

Consider Fig. 3.1c or 3.1d. The vertices s and t are color compatible, the connectivity is greater than two, but still there is no s, t Hamilton path. These cases can be generalized to yield the following condition:

(F3) (G, s, t) is isomorphic to (G', s', t') which satisfies:

1. $G' = R(m, n)$ with $n = 3$ and m even.
2. s' is colored differently from t' and the left corners of G'.
3. $s'_x < t'_x - 1$ (Fig. 3.1c) *or* $s'_y = 2$ and $s'_x < t'_x$ (Fig. 3.1d).

A Hamilton path problem (G, s, t) is *forbidden* if it satisfies one of the conditions (F1), (F2) and (F3).

LEMMA 3.1. *If (G, s, t) is forbidden, then there exists no Hamilton path from s to t in G.*

The proof is a straightforward case analysis. □

We summarize this section with the following definition and theorem:

A Hamilton path problem (G, s, t) is *acceptable* if it is color compatible and not forbidden.

THEOREM 3.1. *If there exists an s, t Hamilton path in G, then (G, s, t) is acceptable.*

3.2. Sufficient conditions. In this section it is shown that all acceptable HP problems have solutions (i.e., acceptability is sufficient). The method of proof is to break large acceptable HP problems into disjoint acceptable subproblems, the HPs of which can be used to construct an HP for the original problem. The two methods, *stripping* and *splitting*, are discussed in the following subsection. We will be done if we show that for prime problems (those which cannot be stripped or split) acceptability implies solvability. However, since the size of these problems is small, their number is finite and can be handled by a case analysis (§ 3.2.2).

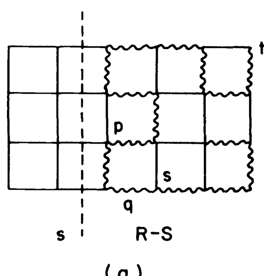
3.2.1. Stripping and splitting. A *separation* of a rectangle R is a partition of R into two subrectangles, i.e., $V(R)$ is a disjoint union of $V(R_1)$ and $V(R_2)$.

A strip S *strips* a Hamilton path problem (R, s, t) if

1. $S, R - S$ is a separation of R,
2. $s, t \in R - S$,
3. $(R - S, s, t)$ is acceptable.

LEMMA 3.2.1. *Let (R, s, t) be an acceptable Hamilton path problem and S strips R. If $(R - S, s, t)$ has a solution, then (R, s, t) also has a solution.*

Proof. Let P be a Hamilton path of $R - S$. Then there exists an edge $pq \in P$ such that pq is on the boundary of $R - S$ facing S. A Hamilton path for (R, s, t) can be obtained by the construction illustrated in Fig. 3.2. □



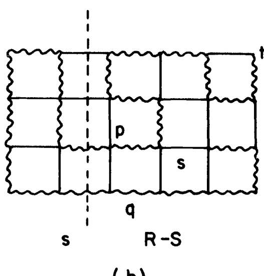


FIG. 3.2

Let $R(m, n)$ be a rectangle with $m \geqq n$. $v, w \in V(R(m, n))$ are called *antipodes* if $v_x \leqq 2$ and $w_x \geqq m-1$.

LEMMA 3.2.2. *Let $(R(m, n), s, t)$ be an acceptable Hamilton path problem which cannot be stripped (by any strip S) and $2 \leqq n \leqq m (n, m) \neq (4, 5)$, $(4, 4)$. Then s and t are antipodes.*

Proof. Without loss of generality, let $s_x \leqq t_x$. Let S be the leftmost strip of R (i.e., $V(S) = \{v \in V(R) \colon v_x \leqq 2\}$). It suffices to show that $s \in S$. Assume to the contrary that $s \notin S$. Let $H_{s,t}$ be the rectangle resulting from deleting S from R. A contradiction will be obtained if $(H_{s,t}, s, t)$ is acceptable. Note that $(H_{s,t}, s, t)$ is color compatible. Now we must show that it is not forbidden.

Case 1. $n \times m$ is odd. $H_{s,t}$ is also odd, so it cannot be forbidden.

Case 2. $m > 5$, $n > 3$. Both the dimensions of $H_{s,t}$ are greater than 3, so it cannot be forbidden.

Case 3. $n = 3$, $m > 5$. If $(H_{s,t}, s, t)$ is forbidden, it must be F3, but then so is (R, s, t).

Case 4. $n = 2$. Since (R, s, t) is not forbidden, neither is $(H_{s,t}, s, t)$. Note that if $m = 5$, $(H_{s,t}, s, t)$ may be F3, but in this case (R, s, t) is F2.

Case 5. $m = 4$, $n = 3$. The only possibility for $(H_{s,t}, s, t)$ to be forbidden is depicted in Fig. 3.3b. However, then (R, s, t) satisfies F3. □

Let (R, s, t) be an acceptable Hamilton path problem and pq an edge. *pq splits* (R, s, t) if there exists a separation of R to R_p and R_q such that:

1. $s, p \in R_p$ and (R_p, s, p) is acceptable, and
2. $q, t \in R_q$ and (R_q, q, t) is acceptable.

The following lemma follows immediately from the definition of splitting.

LEMMA 3.2.3. *Let pq be an edge which splits (R, s, t). If both (R_p, s, p) and (R_q, q, t) have a solution, then so does (R, s, t).*

3.2.2. Prime problems. A Hamilton path problem (R, s, t) is *prime* if it cannot be stripped or split. The following lemma allows us to confine the discussion to a finite number of cases.

LEMMA 3.2.4. *Let $(R(m, n), s, t)$ be an acceptable prime Hamilton path problem, then $(n, m) = (4, 5)$ or $n, m \leqq 3$.*

FIG. 3.3

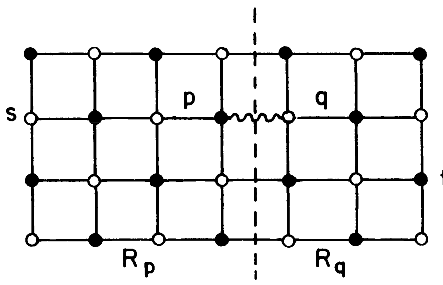


FIG. 3.4

Proof. Assume first that $(n, m) \notin \{(4, 4), (4, 5)\}$. Then s and t are antipodes. Let S be the leftmost strip, then $s \in S$. We show that there exists a split such that $R_p = S$.

Case 1. $m > 5$, $n \geqq 4$. There are at least two vertices, v^i, with $v_x^i = 2$, $i = 1, 2$ and colored differently than s. Let p be a v^i not connected to s by a nonboundary edge of S and q be the adjacent vertex in $R - S$. The edge pq splits R: $R_p = S$, and (R_p, s, p) is acceptable by the construction. As for (R_q, q, t), $q \neq t$, since $t_x \geqq m - 1 > 3 = q_x$; since s and q have the same color, (R_q, q, t) is color compatible; R_q has dimensions $(m-2, n) > 3$; hence, (R_q, q, t) is not forbidden and, therefore, is acceptable.

Case 2. $m > 5$, $n = 3$ (Fig. 3.5a). Without loss of generality, the left corners of R are white. Let $p = (2, 1)$ and $q = (3, 1)$. We show that edge pq splits (R, s, t). Note that p is black and q is white. If m is even, then s is white and t black; otherwise (R, s, t) is not acceptable. Therefore $p \neq s$ and $q \neq t$. Consequently (R_p, s, p) and (R_q, q, t) are color compatible. If m is odd, then both s, t are white. Therefore $p \neq s$, and because $q_x = 3 < t_x$, also $q \neq t$. Again, (R_p, s, p) and (R_q, q, t) are color compatible. In both cases p is a corner of R_p and q a corner of R_q. Hence, the subproblems are not forbidden.

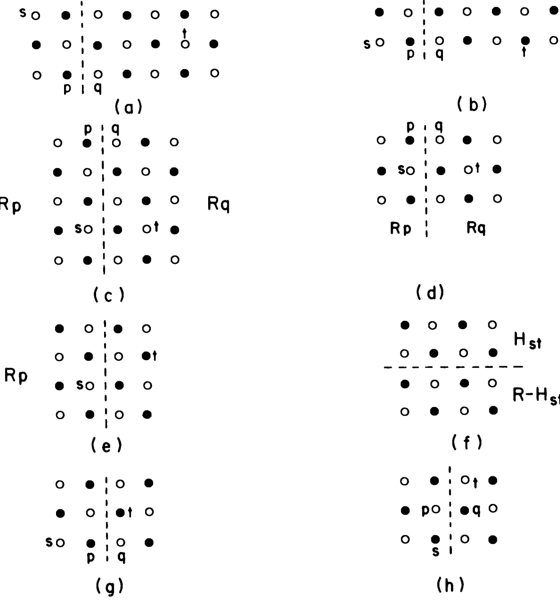


FIG. 3.5

Case 3. $n=3$ (Fig. 3.5b). Without loss of generality s is white. The edge $pq = (2, 1)(3, 1)$ splits (R, s, t) $q \neq t$ since q is white and t is black.

Case 4. $m=5$, $n=5$ (Fig. 3.5c). Let p be a black vertex not connected to s such that $p_x = 2$, $q = (p_x + 1, p_y)$. Since R_q is odd, pq splits (R, s, t).

Case 5. $m=5$, $n=3$ (Fig. 3.5d). Similar to Case 4.

Case 6. $m=4$, $n=4$. If s and t are antipodes, then either $(2, 1)(3, 1)$ or $(2, 4)(3, 4)$ splits (R, s, t) (Fig. 3.5e). If s and t are not antipodes, we may assume that $s_x s_y, t_x, t_y \leqq 2$. Therefore, either the rightmost or the uppermost strip may be stripped off (R, s, t) (Fig. 3.5f).

Case 7. $m=4$, $n=3$.

If s is white, then $pq = (2, 1)(3, 1)$ splits (R, s, t) (Fig. 3.5g). Otherwise, s is black and $s_x = 2$, $t_x = 3$. Therefore, $(2, 2)(3, 2)$ splits (R, s, t) (Fig. 3.5h). □

LEMMA 3.2.5. *Any* $(R(5, 4), s, t)$ *acceptable Hamilton path problem is solvable.*

Proof. It suffices to prove the lemma for prime problems. First, s and t cannot be antipodes. If they were, either edge, $(2, 1)(3, 1)$ or $(2, 4)(3, 4)$, splits (R, s, t). We can then assume that both s, t do not belong, say, to the rightmost strip. Now, if one of s, t belongs to the lowermost strip and the other to the uppermost, then either edge $(4, 2)(4, 3)$ or $(5, 2)(5, 3)$ splits (R, s, t). Therefore without loss of generality we can restrict to the case $s_x, t_x \leqq 3$ and $s_y, t_y \leqq 2$. Then the rightmost strip can be stripped off, except when (R, s, t) is isomorphic to the problem of Fig. 3.6. That is a prime problem, solvable as indicated in the figure.

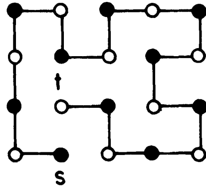
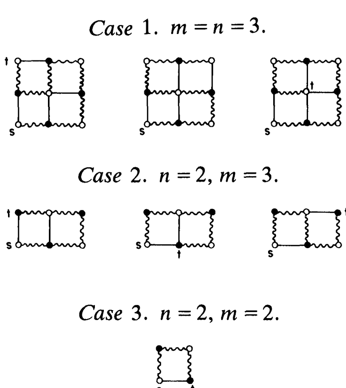


FIG. 3.6

LEMMA 3.2.6. *If* (R, s, t) *is an acceptable prime Hamilton path problem, then it is solvable.*

Proof. From Lemmas 3.2.4 and 3.2.5, we may assume that $n, m \leqq 3$. For all values of m and n, all nonisomorphic problems and their corresponding paths are illustrated in Fig. 3.6.



Following the discussion at the beginning of this section, the preceding lemmas yield the following:

THEOREM 3.2. *There exists a Hamilton path from s to t in R if and only if* (R, s, t) *is acceptable.*

3.3. An algorithm. The proof of Theorem 3.2 is constructive. To decide whether a Hamilton path problem $(R(n, m), s, t)$ has a solution, we check whether it is acceptable. This requires time linear with the representation of n, m, s and t. To find the path itself, we first try to strip off the strips, constructing partial paths, and try to split the problem. This process is repeated until we are left with prime problems, for which a path can be found in constant time. The partial paths are pasted together as in Lemmas 3.2.1, 3.2.3. The entire process takes time linear in the length of the path, $O(nm)$. We note here that the results of this and the previous section leave open the question whether the Hamilton circuit problem is polynomial for grid graphs that are not rectangular, but neither have "holes", i.e., both G and $G^{\infty} - G$ are connected.

REFERENCES

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

[GJ] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, CA, 1979.

[GJT] M. R. GAREY, D. S. JOHNSON AND R. E. TARJAN, *The planar Hamilton circuit problem is NP-complete*, this Journal, 5 (1976), pp. 704–714.

[K1] R. M. KARP, *Reducibility among combinatorial problems*, in Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.

[K2] M. S. KRISHNAMOORTHY, *An NP-hard problem in bipartite graphs*, SIGACT News, 7 (1975), 1, 26.

[LM] F. LUCCIO AND C. MUGNAI, *Hamiltonian paths on a rectangular chessboard*, 16th Annual Allerton Conference, 1978, pp. 73–78.

[Pa] C. H. PAPADIMITRIOU, *The Euclidean traveling salesman problem is NP-complete*, Theoret. Comp. Sci., 4 (1977), pp. 237–244.

[P] J. PLESNIK, *The NP-completeness of the Hamiltonian cycle problem in planar digraphs with degree two*, IPL, to appear.

[S] Y. SHILOACH, *Linear and planar arrangements of graphs*, Ph.D. dissertation, Appl. Math. Dept., Weizmann Institute of Science, Rehovot, Israel, 1976.

[V] L. G. VALIANT, unpublished manuscript, 1979.

SIAM J. COMPUT.
Vol. 11, No. 4, November 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1104-0006 $01.00/0

ALGORITHMS FOR THE SOLUTION OF SYSTEMS OF LINEAR DIOPHANTINE EQUATIONS*

TSU-WU J. CHOU† AND GEORGE E. COLLINS‡

Abstract. Two algorithms for the solution of linear Diophantine systems, which well restrain intermediate expression swell, are presented. One is an extension and improvement of Kannan and Bachem's algorithm for the Smith and the Hermite normal forms of a nonsingular square integral matrix. The complexity of this algorithm is investigated and a polynomial time bound is derived. Also a much better coefficient bound is obtained compared to Kannan and Bachem's analysis. The other is based on ideas of Rosser, which were originally used in finding a general solution with smaller coefficients to a linear Diophantine equation and in computing the exact inverse of a nonsingular square integral matrix. These algorithms are implemented by using the infinite precision integer arithmetic capabilities of the SAC-2 system. Their performances are compared. Finally future studies are mentioned.

Key words. Linear Diophantine system, Hermite normal form, solution module basis, Rosser-type algorithm, Kannan–Bachem algorithm

1. Introduction. Solving the linear Diophantine systems $Ax = b$ means finding an integral vector y and a set of integral vectors $N = \{N_1, \cdots, N_p\}$, where p is the dimension of the null space of the system $Ax = b$, such that the sum of y and any integral linear combination of N_i's is an integral solution of $Ax = b$, and vice versa. The vector y is called a *particular solution*. The pair (y, N) is called a *general solution*.

Algorithms for solving a system of linear Diophantine equations can be found in [1], [2], [7] and [10]. Some other related work can be found in [9], [14] and [15].

The main difficulty in solving systems of linear Diophantine equations is the very rapid growth of the intermediate results. This effect is called *intermediate expression swell* [13]. Frumkin [7] has observed that the order of intermediate expression swell in the algorithm by Bradley [2] can be exponential in the number of equations. Such an algorithm will be impractical even for large computers.

In §§ 2 and 4 we will present two algorithms, LDSMKB and LDSSBR respectively, which control intermediate expression swell very well. The basic ideas for the algorithm LDSMKB come from Kannan and Bachem [9]. They used these ideas in computing the Smith and the Hermite normal forms of a nonsingular square integral matrix A and showed that the lengths of the coefficients of A during the computation of the Hermite normal form of A can be bounded by a polynomial function of order $n^4L(n|A|)$, where n is the common row and column dimension of A, $|A|$ is the largest, in absolute value, element of A and L is the length function defined by equation (1) in § 3.

Without modifying their algorithm, one can derive a better bound of order $n^2L(n|A|)$ on the sizes of the coefficients during the computation. With a slight modification, one can obtain an even better bound of order $nL(n|A|)$. More detailed discussion is given in § 2.

In the algorithm LDSMKB, we extend Kannan and Bachem's ideas with modification to the problem of solving a system of linear Diophantine equations. Analyses of the algorithm LDSMKB are developed in § 3.

* Received by the editors August 7, 1980, and in revised form January 7, 1982. This research was partially supported by the National Science Foundation under grants MCS74-13278 Aol and MCS78-01731.

† Bell Laboratories, Holmdel, New Jersey 07733.

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

The ideas employed in the algorithm LDSSBR to control coefficient growth come from J. B. Rosser. He used the ideas in finding a general solution with much smaller coefficients to a linear Diophantine equation (see [14]) and computing the exact inverse of a nonsingular square integral matrix to minimize computing time (see [15]).

The algorithm LDSSBR restrains coefficient growth very well in the early stage of the algorithm. Although the growth becomes fast in the final stage of the algorithm, it is still quite moderate compared to that in some other algorithms. As our empirical results (see § 5) will show, the length of the largest coefficients while solving the Diophantine system $Ax = b$ is typically bounded by $nL(n|A|)$, where n is the number of variables in the system.

2. Improvement of the Kannan–Bachem algorithm. Usual methods for computing the Hermite normal form of A transform successively the submatrix consisting of the first i rows of A into its Hermite normal form for $i = 1, \cdots, n$. So elements common to rows $1, \cdots, i$ and columns $i+1, \cdots, n$ of A are zero after transforming the first i rows of A into a Hermite normal matrix.

Kannan and Bachem's method successively transforms the $i \times i$ principal minor of A into its Hermite normal form for $i = 1, \cdots, n$. So columns $i+1, \cdots, n$ of A remain unchanged before transforming the $(i+1)\times(i+1)$ principal minor of A. Preconditioning is required to ensure that all the principal minors of A are nonsingular. During the ith execution of the major loop in Kannan and Bachem's algorithm, which transforms the $(i+1)\times(i+1)$ principal minor into its Hermite normal form, two processes are involved, namely, elimination and normalization. Elimination refers to the process of forcing $A_{j,i+1}$, the element in row j and column $i+1$ of A, to zero for $j = 1, \cdots, i$ by a sequence of unimodular column transformations on A, that is, postmultiplication of A by a sequence of unimodular matrices. Note that any unimodular transformation is equivalent to a sequence of elementary column operations, since every unimodular matrix is equal to a product of some sequence of elementary matrices. Normalization refers to the process of making elements to the left of the diagonal nonnegative and less than the diagonal elements to their right. The normalization order is from top to bottom and from left to right. The normalization order in the third execution of the major loop is illustrated by the following figure:

$$\begin{matrix} * & 0 & 0 & 0 \\ 1 & * & 0 & 0 \\ 2 & 3 & * & 0 \\ 4 & 5 & 6 & * \end{matrix}$$

Each * represents a diagonal element. Each positive number denotes the order of the corresponding element in the normalization process.

Let A' be the matrix obtained from A whose $i \times i$ principal minor A_i' is the Hermite normal form of the $i \times i$ principal minor A_i of A. Then $A' = AU$ for some unimodular matrix U. Since A' is obtained from A by a sequence of unimodular transformations on the first i columns of A,

$$U = \begin{bmatrix} U_i & 0 \\ 0 & I \end{bmatrix},$$

where U_i is an $i \times i$ unimodular matrix, and hence, $A_i' = A_i U_i$. Since A_i is nonsingular, $U_i = A_i^{-1} A_i' = \operatorname{adj}(A_i) A_i' / \det(A_i)$. Let u_{jk} and a'_{jk} be the elements in the jth rows and

kth columns of U_i and A_i' respectively. Then $|u_{jk}| \leqq \sum_{l=1}^{i} |a'_{lk}| |\text{adj}(A_i)|/|\det(A_i)|$. Since A_i' is a Hermite matrix, $\sum_{l=1}^{i} |a'_{lk}| = \sum_{l=1}^{i} a'_{lk} \leqq a'_{11} + (a'_{22} - 1) + \cdots + (a'_{ii} - 1) \leqq \det(A_i')$. Furthermore, $\det(A_i') = |\det(A_i U_i)| = |\det(A_i)||\det(U_i)| = |\det(A_i)|$, and hence, $|u_{jk}| \leqq |\text{adj}(A_i)|$. Therefore, $|U_i| \leqq |\text{adj}(A_i)|$. Since every element of the adjoint of A_i is an $(i-1)\times(i-1)$ minor of A whose magnitude is bounded by $(i-1)^{(i-1)/2}|A|^{i-1}$ (see Theorem 3 in § 3), $|U_i| \leqq (i-1)^{(i-1)/2}|A|^{i-1} \leqq (i|A|)^{i-1}$. Since $A' = AU$, $|A'| \leqq n|A||U_i| \leqq (n|A|)^i$. Therefore, the length of the norm of A' is bounded by $iL(n|A|)$.

Using this bound for the lengths of the elements of A' at the end of the ith iteration, Kannan and Bachem derive a bound of order $n^4 L(n|A|)$ on the sizes of the coefficients during the $(i+1)$th iteration. Without modifying their algorithm, one can derive a better bound of order $n^2 L(n|A|)$ on the sizes of the coefficients during the $(i+1)$th iteration by employing once again the fact that A_i' is the Hermite normal form of A_i, so that the product of all the diagonal elements of A_i', being the magnitude of the determinant of A_i, is bounded by $(i|A|)^i$.

One can obtain an even better bound, of order $nL(n|A|)$, by changing the normalization order of Kannan and Bachem's algorithm. The new normalization order is from right to left and from top to bottom. A typical normalization order for the third execution of the major loop of this modified Kannan and Bachem's algorithm is illustrated by the following figure:

$$\begin{matrix} * & 0 & 0 & 0 \\ 4 & * & 0 & 0 \\ 5 & 2 & * & 0 \\ 6 & 3 & 1 & * \end{matrix}$$

This modification enables us to derive a better bound, because column j is normalized by using only the already normalized columns $j+1, \cdots, i+1$ to its right during the ith iteration of the major loop of the modified Kannan and Bachem's algorithm. The derivation of this new bound is very similar to the one given in § 3 for the algorithm LDSMKB.

In the algorithm LDSMKB we extend Kannan and Bachem's ideas with the modification described in the previous paragraph to the problem of solving a system of linear Diophantine equations $Ax = b$ for any $A \in Z(m, n)$, the set of all $m \times n$ matrices over the ring of integers Z, of rank r and $b \in Z^m$. In order to employ Kannan and Bachem's ideas for a square nonsingular preconditioned matrix, the $n \times n$ identity matrix is adjoined to the bottom of A, and we call the new matrix $\bar{A}$. Since $\bar{A}$ is of rank n, there exists a nonsingular submatrix A^* of $\bar{A}$ consisting of r linearly independent rows of A and $n - r$ rows of the $n \times n$ identity matrix. The major work for the algorithm LDSMKB is to transform A^* into a pseudo-Hermite matrix by applying a sequence of unimodular transformations to $\bar{A}$. A square nonsingular matrix is a *pseudo-Hermite matrix*, or in *pseudo-Hermite form*, if it is lower triangular and the absolute value of any off-diagonal element is less than the absolute value of the diagonal element to its right. The transformation of A^* into a pseudo-Hermite matrix enables us to derive a very good bound, of order $n + rL(r|A|)$, on the lengths of the coefficients pertaining to $\bar{A}$.

Some helpful notations are introduced here before we present and analyze the algorithm LDSMKB. Let V be a vector. The ith component of V is denoted by V_i. Let A be an $m \times n$ matrix. The jth column of A is denoted by A_j. The element in the ith row and jth column of A is denoted by $A_{i,j}$. Let $i_1, \cdots, i_s$ and $j_1, \cdots, j_t$ be sequences of integers such that $1 \leqq i_k \leqq m$ and $1 \leqq j_h \leqq n$. Then the matrix consisting

of the elements of A in rows $i_1, \cdots, i_s$ and columns $j_1, \cdots, j_t$ in that order is denoted by

$$A\begin{bmatrix} i_1, & \cdots, & i_s \\ j_1, & \cdots, & j_t \end{bmatrix}.$$

If $s = t$, its determinant is denoted by

$$A\begin{pmatrix} i_1, & \cdots, & i_s \\ j_1, & \cdots, & j_t \end{pmatrix}.$$

Given an $m \times n$ matrix A, let $\bar{A}$ be the $(m+n)\times n$ matrix $\left[\begin{smallmatrix} A \\ I \end{smallmatrix}\right]$, where I is the $n \times n$ identity matrix. Define inductively the *row sequence* of A, $R = (i_1, \cdots, i_n)$, as follows. For $n = 1$, $R = (i)$ where i is the smallest positive integer such that $\bar{A}\left(\begin{smallmatrix} i \\ 1 \end{smallmatrix}\right) \neq 0$. For $n > 1$, let $(i_1, \cdots, i_{n-1})$ be the row sequence of $A\left[\begin{smallmatrix} 1, \cdots, m \\ 1, \cdots, n-1 \end{smallmatrix}\right]$ and let i be the smallest positive integer such that

$$\bar{A}\begin{pmatrix} i_1, & \cdots, & i_{n-1}, & i \\ 1, & \cdots, & & n \end{pmatrix} \neq 0.$$

Then $R = (i_1, \cdots, i_k, i, i_{k+1}, \cdots, i_{n-1})$ where, with $i_0 = 0$, k is the largest integer, $0 \leqq k \leqq n-1$, such that $i_k < i$. The submatrix A^* discussed above will be the matrix $\bar{A}\left[\begin{smallmatrix} i_1, \cdots, i_n \\ 1, \cdots, n \end{smallmatrix}\right]$ where $(i_1, \cdots, i_n)$ is the row sequence of A.

The algorithm has two inputs and two outputs. The inputs are the coefficient matrix A, an integral $m \times n$ matrix, and the right-hand side b, an integral m-vector, of the linear Diophantine system to be solved. The outputs are a particular solution x^* of the Diophantine system $Ax = b$ and a basis N of the solution module of the homogeneous Diophantine system $Ax = 0$ if the Diophantine system $Ax = b$ is consistent. Otherwise, the null list is returned for x^* and N.

The algorithm begins by constructing the $(m+n)\times n$ matrix C whose initial value is $\bar{A} = \left[\begin{smallmatrix} A \\ I \end{smallmatrix}\right]$, where I is the $n \times n$ identity matrix, and the $(m+n)$-vector B whose initial value is $\bar{b} = \left[\begin{smallmatrix} -b \\ 0 \end{smallmatrix}\right]$, where 0 is the zero n-vector.

Let $R_h = (i_1, \cdots, i_h)$ be the row sequence of the submatrix A^h consisting of the first h columns of A. Let D^h denote the square nonsingular matrix $\bar{A}\left[\begin{smallmatrix} i_1, \cdots, i_h \\ 1, \cdots, h \end{smallmatrix}\right]$. Let r_h be the rank of A^h. Then the algorithm computes $R_1 = (i_1)$, where i_1 is the row index of the leading nonzero element of $\bar{A}_1$, that is, the smallest positive integer such that $\bar{A}\left(\begin{smallmatrix} i_1 \\ 1 \end{smallmatrix}\right) \neq 0$. If $i_1 > m$, which implies that A_1 is a zero vector, then $r_1 = 0$; otherwise, $r_1 = 1$. We claim that if $R = (j_1, \cdots, j_n)$ is the row sequence of an $m \times n$ matrix of rank r, then $j_1 < j_2 < \cdots < j_n$ and $j_h > m$ for $r < h \leqq n$. This is obviously true for R_1. Note that $C\left[\begin{smallmatrix} i_1 \\ 1 \end{smallmatrix}\right]$ is obviously a pseudo-Hermite matrix.

Now the algorithm enters a loop which computes r_{k+1} and R_{k+1} and transforms D^{k+1} into a pseudo-Hermite matrix for $k = 1, \cdots, n-1$ inductively. At the beginning of the kth iteration, $C\left[\begin{smallmatrix} i_1, \cdots, i_k \\ 1, \cdots, k \end{smallmatrix}\right]$ is in pseudo-Hermite form where $R_k = (i_1, \cdots, i_k)$ is the row sequence of A^k. To transform D^{k+1} into a pseudo-Hermite matrix, one can first eliminate $C_{i_j,k+1}$ by applying a unimodular transformation to columns C_j and C_{k+1} for $j = 1, \cdots, k$, in that order, and then normalizing elements to the left of the diagonal of $C\left[\begin{smallmatrix} i_1, \cdots, i_k \\ 1, \cdots, k \end{smallmatrix}\right]$ from right to left and from top to bottom.

However, a slight modification can be made to improve the efficiency. The algorithm LDSMKB first finds the row index j of the leading nonzero element of C_{k+1}. If j is in $\{i_1, \cdots, i_k\}$, say $j = i_h$, then the algorithm eliminates $C_{j,k+1}$ by applying a unimodular transformation to C_h and C_{k+1} and repeats the above process. Eventually,

j will be different from any index in $\{i_1, \cdots, i_k\}$, since the j's found in the above process are increasing and C_{k+1} is linearly independent of C_1 through C_k. Let j^* be the last j found. Then j^* is the smallest positive integer such that

$$C\begin{pmatrix} i_1, & \cdots, & i_k, & j^* \\ 1, & \cdots, & & k+1 \end{pmatrix} \neq 0.$$

If $j^* > i_k$, then R_{k+1} is obviously the sequence $(i_1, \cdots, i_k, j^*)$. If, with $i_0 = 0$, h is the largest integer, $0 \leqq h \leqq k-1$, such that $i_h < j^* < i_{h+1}$, then $R_{k+1} = (i_1, \cdots, i_h, j^*, i_{h+1}, \cdots, i_k)$. Such an h exists by the hypothesis that $i_1 < i_2 < \cdots < i_k$. Obviously, R_{k+1} is also an increasing sequence. If $j^* \leqq m$, then A_{k+1} is linearly independent of $A_1, \cdots, A_k$, and hence $r_{k+1} = r_k + 1$. Otherwise, $r_{k+1} = r_k$. Thus if $R_{k+1} = (i'_1, \cdots, i'_{k+1})$ then $i'_h > m$ for $h > r_{k+1}$. For the case that $i_h < j^* < i_{h+1}$, the algorithm rotates columns $h+1, \cdots, k+1$ so that column $k+1$ becomes the $(h+1)$th column, and then the matrix

$$C\begin{bmatrix} i'_1, & \cdots, & i'_{k+1} \\ 1, & \cdots, & k+1 \end{bmatrix}$$

is lower triangular. Since now columns $h+2, \cdots, k+1$ of C were columns $h+1, \cdots, k$ of C at the beginning of the kth iteration, whose elements were not changed during the elimination, the normalization can proceed from column $h+1$, rather than from column k, to column 1.

Let $\bar{A}'$ be the final value of C after finishing the loop $n-1$ times, and let $R_n = (i_1, \cdots, i_n)$. Then a particular solution can be obtained, if the system is consistent, by adding an integral multiple of $\bar{A}'_h$ to B such that $|B_{i_h}| < |\bar{A}'_{i_h,h}|$ for $h = 1, \cdots, r_n$ in that order. Let $(b'_1, \cdots, b'_{m+n})$ be the final value of B. If $b'_i = 0$ for $i = 1, \cdots, m$, then the Diophantine system $Ax = b$ is consistent and the algorithm returns $(b'_{m+1}, \cdots, b'_{m+n})$ and $\bar{A}'\left[\begin{smallmatrix} m+1, \cdots, m+n \\ r_n+1, \cdots, n \end{smallmatrix}\right]$ for x^* and N respectively. Otherwise, the Diophantine system is inconsistent and the algorithm returns the null list for x^* and N.

The algorithm LDSMKB is described as a SAC-2 [6] algorithm in ALDES [12]. The SAC-2 system is a computer-independent system for *s*ymbolic *a*lgebraic *c*omputation. Its predecessor is the SAC-1 system. ALDES is an *AL*gorithm *DES*cription language whose syntax is very self-explanatory.

By SAC-2 convention an algorithm having one output variable is written as a function subprogram; otherwise it is written as a subroutine subprogram. For example, if F is an algorithm having one input variable x and one output variable y, and G is an algorithm having one input variable x and two output variables y and z, then the declarations for F and G are "$y \leftarrow F(x)$" and "$G(x; y, z)$" respectively.

SAC-2 algorithms referenced by LDSMKB can be divided into four categories: list processing algorithms, integer arithmetic algorithms, integral vector algorithms and integral matrix algorithms.

Let S be an arbitrary set called an *atom set*. Elements in S are called *atoms*. A *list over* S is recursively defined to be a finite sequence $(a_1, \cdots, a_n)$, $n \geqq 0$, such that each a_i is an *object*, namely either an atom in S or a list over S. For SAC-2, S is the set of all the β-integers, i.e., integers whose absolute values are less than the integer β. By convention, $\beta = 2^\zeta$ where ζ is three less than the number of bits of a single-precision word in implementation.

Following are specifications of the list processing algorithms.

$a \leftarrow$ FIRST (L)

[First. L is the nonnull list $(a_1, \cdots, a_m)$, $m \geqq 1$. a is the first element of L, namely a_1.]

$L' \leftarrow$ RED (L)

[Reductum. L is the nonnull list $(a_1, \cdots, a_m)$, $m \geqq 1$. L' is the reductum of L, namely the list $(a_2, \cdots, a_m)$.]

SFIRST (L, a)

[Set first. L is a nonnull list. a is an object. The first element of L is changed to a.]

ADV $(L; a, L')$

[Advance. L is a nonnull list. a is the first element of L. L' is the reductum of L.]

$M \leftarrow$ COMP (a, L)

[Composition. a is an object. L is the list $(a_1, \cdots, a_m)$. M is the composition of a and L, namely the list $(a, a_1, \cdots, a_m)$.]

$n \leftarrow$ LENGTH (L)

[Length. L is the list $(a_1, \cdots, a_m)$. n is the length of L, namely m.]

$B \leftarrow$ REDUCT (A, i)

[Reductum. A is a list. i is a nonnegative β-integer not greater than the length of A. $B = A$ if $i = 0$. Otherwise, B is the ith reductum of A.]

$M \leftarrow$ LEROT (L, i, j)

[List element rotation. L is a list $(a_1, \cdots, a_n)$ of objects, $n > 0$. i and j, $1 \leqq i \leqq j \leqq n$, are β-integers. If $i = j$, then $M = L$. Otherwise $M = (a_1, \cdots, a_{i-1}, a_j, a_i, \cdots, a_{j-1}, a_{j+1}, \cdots, a_n)$. L is modified.]

$L \leftarrow$ LEINST (A, i, a)

[List element insertion. A is the list $(a_1, \cdots, a_n)$ of objects. i is a β-integer, $0 \leqq i \leqq n$. a is an object. If $i = 0$, then $L = (a, a_1, \cdots, a_n)$. If $i = n$, then $L = (a_1, \cdots, a_n, a)$. Otherwise, $L = (a_1, \cdots, a_i, a, a_{i+1}, \cdots, a_n)$. A is modified.]

Let a be an integer. For $|a| < \beta$, a is represented as a β-integer. For $|a| \geqq \beta$, let $\sum_{i=0}^{n-1} a_i \beta^i$ be the β-radix representation of a, that is, $a_0, \cdots, a_{n-1}$ are β-integers such that $a_i \geqq 0$ for $i = 0, \cdots, n-2$ and $a_{n-1} > 0$ if a is positive, or $a_i \leqq 0$ for $i = 0, \cdots, n-2$ and $a_{n-1} < 0$ if a is negative. Then a is represented by the list $(a_0, a_1, \cdots, a_n)$.

Following is the algorithm specification of the algorithm IDEGCD.

IDEGCD $(a, b; c, u_1, v_1, u_2, v_2)$

[Integer doubly extended greatest common divisor algorithm. a and b are integers. $c = gcd(a, b)$. $au_1 + bv_1 = c$ and $au_2 + bv_2 = 0$. If $a \neq 0$ and $b \neq 0$ then $u_1 \leqq |b|/2c$, $v_1 \leqq |a|/2c$, $u_2 = -b/c$ and $v_2 = a/c$. Otherwise $u_1 = v_2 = \text{sign}(a)$, $v_1 = \text{sign}(b)$ and $u_2 = -\text{sign}(b)$.]

Let V be a vector in Z^n. Then V is represented by the list $(v_1, \cdots, v_n)$, where each v_i is the integer representation of the ith component of V.

Following are algorithm specifications of integral vector algorithms.

$B \leftarrow$ VIAZ (A, n)

[Vector of integers, adjoin zeros. A is the vector $(a_1, \cdots, a_m)$. n is a nonnegative β-integer. B is the vector $(a_1, \cdots, a_m, 0, \cdots, 0)$ of $m+n$ components. A is modified.]

$W \leftarrow$ VIERED (U, V, i)

[Vector of integers, element reduction. $U = (u_1, \cdots, u_n)$ and $V = (v_1, \cdots, v_n)$ are integral n-vectors. $1 \leqq i \leqq n$. $v_i \neq 0$. $W = U - qV$, where $q = [u_i/v_i]$.]

$B \leftarrow$ VINEG (A)

[Vector of integers negation. A is an integral vector. $B = -A$.]

VIUT $(U, V, i; U', V')$

[Vector of integers, unimodular transformation. $U = (u_1, \cdots, u_n)$ and $V = (v_1, \cdots, v_n)$ are vectors in Z^n with $u_i \neq 0$. $[U', V'] = [U, V]K$ where K is a unimodular matrix, depending on u_i and v_i, whose elements are obtained from IDEGCD.]

Let A be an integral matrix in $Z(m, n)$. Then A is represented by the list $(A_1, \cdots, A_n)$, where each A_i is the list representation of the ith column (as an m-vector) of A.

Following is algorithm specification of the algorithm MIAIM.

$B \leftarrow$ MIAIM (A)

[Matrix of integers, adjoin identity matrix. A is an $m \times n$ matrix of integers. B is the matrix obtained by adjoining an $n \times n$ identity matrix to the bottom of A. A is modified.]

Following is the formal algorithm description of LDSMKB. Note that the *safe* declaration is mainly for run-time efficiency. Without the safe declaration, the validity of the algorithm remains unchanged. The validity proof follows the algorithm description.

LDSMKB $(A, b; x^*, N)$

[Linear Diophantine system solution, modified Kannan and Bachem algorithm. A is an $m \times n$ integral matrix. b is an integral m-vector. If the Diophantine system $Ax = b$ is consistent, then x^* is a particular solution and N is a list of basis vectors of the solution module of $Ax = 0$. Otherwise, x^* and N are null lists. A and b are modified.]

safe $c, C_1, C', C'_1, C^*, h, i, j, k, m, N, n, r,. R', x^*$.

(1) [Adjoin identity matrix to A and zero vector to $-b$.] $n \leftarrow$ LENGTH (A); $C \leftarrow$ MIAIM (A); $B \leftarrow$ VIAZ (VINEG $(b), n)$.

(2) [Initialize.] $m \leftarrow$ LENGTH (b); $C_1 \leftarrow$ FIRST (C); $j \leftarrow 0$; repeat $\{j \leftarrow j+1$; ADV $(C_1; c, C_1)\}$ until $c \neq 0$; $R \leftarrow$ COMP $(j, (\ ))$; if $j \leqq m$ then $r \leftarrow 1$ else $r \leftarrow 0$; $k \leftarrow 1$; if $n = 1$ then go to 5.

(3) [Eliminate column $k+1$ and augment row sequence.] $C^* \leftarrow$ REDUCT (C, k); $C_1^* \leftarrow$ FIRST (C^*); $C' \leftarrow C$; $R' \leftarrow R$; for $h = 1, \cdots, k+1$ do {if $h \leqq k$ then ADV $(R'; i, R')$ else $i \leftarrow m+n+1$; $C'_1 \leftarrow C_1^*$; $j \leftarrow 0$; repeat $\{j \leftarrow j+1$; ADV $(C'_1; c, C'_1)\}$ until $c \neq 0$; if $j \geqq i$ then {if $j = i$ then $\{C_1 \leftarrow$ FIRST (C'); VIUT $(C_1, C_1^*, i; C_1, C_1^*)$; SFIRST $(C', C_1)\}$; $C' \leftarrow$ RED $(C')\}$ else {SFIRST (C^*, C_1^*); $C \leftarrow$ LEROT $(C, h, k+1)$; $R \leftarrow$ LEINST $(R, h-1, j)$; if $j \leqq m$ then $r \leftarrow r+1$; go to 4}}.

(4) [Normalize off-diagonal elements.] for $j = h, \ h-1, \cdots, 1$ do $\{C^* \leftarrow$ REDUCT $(C, j-1)$; ADV $(C^*; T, C')$; $R' \leftarrow$ REDUCT (R, j); while $R' \neq (\ )$ do {ADV $(C'; C'_1, C')$; ADV $(R'; i, R')$; $T \leftarrow$ VIERED $(T, C'_1, i)\}$; SFIRST $(C^*, T)\}$; $k \leftarrow k+1$; if $k < n$ then go to 3.

(5) [Check consistency of the system.] for $j = 1, \cdots, r$ do {ADV $(C; T, C)$; ADV $(R; i, R)$; $B \leftarrow$ VIERED $(B, T, i)\}$; $j \leftarrow 0$; repeat $\{j \leftarrow j+1$; ADV $(B; c, B)\}$ until $j = m \vee c \neq 0$.

(6) [System consistent.] if $c = 0$ then $\{C' \leftarrow C$; while $C' \neq (\ )$ do $\{C'_1 \leftarrow$ FIRST (C'); $C'_1 \leftarrow$ REDUCT (C'_1, m); SFIRST (C', C'_1); $C' \leftarrow$ RED $(C')\}$; $x^* \leftarrow B$; $N \leftarrow C$; return}.

(7) [System inconsistent.] $x^* \leftarrow (\ )$; $N \leftarrow (\ )$; return.

THEOREM 1. *The algorithm* LDSMKB *is valid.*

Proof. Let C be the $(m+n)\times n$ matrix which is initially $\left[\begin{smallmatrix}A\\I\end{smallmatrix}\right]$ and let B be the $(m+n)$-vector which is initially $\left[\begin{smallmatrix}-b\\0\end{smallmatrix}\right]$. Let C' denote the matrix consisting of rows $1,\cdots,m$ of C, let C'' denote the matrix consisting of rows $m+1,\cdots,m+n$ of C, let B' denote the vector consisting of the first m components of B and let B'' denote the vector consisting of the last n components of B. Performing unimodular transformations on C will preserve (1) the unimodularity of C'', (2) the validity of the relation $C'=AC''$ and (3) the consistency of $C'x=B'$. Suppose $\bar{C}$ is obtained from C by performing a unimodular transformation on C. Then $\bar{C}=CE$ for some unimodular matrix E. Let $\bar{C}'$ and $\bar{C}''$ be the submatrices of $\bar{C}$ corresponding to C' and C'' of C. Then $\bar{C}'=C'E$ and $\bar{C}''=C''E$. Since $\det(\bar{C}'')=\det(C'')\det(E)=\pm\det(C'')$, $\bar{C}''$ is unimodular if and only if C'' is unimodular. Since $\bar{C}'-A\bar{C}''=C'E-AC''E=(C'-AC'')E$ and $\det(E)\neq 0$, $\bar{C}'-A\bar{C}''=0$ if and only if $C'-AC''=0$, i.e., $\bar{C}'=A\bar{C}''$ if and only if $C'=AC''$. Since $\bar{C}'=C'E$ and E is unimodular, $\bar{C}'x=B'$ is consistent if and only if $C'x=B'$ is consistent. Obviously, C'' is unimodular and $C'=AC''$ when C is constructed in Step 2. So we can conclude that C'' is unimodular and $C'=AC''$ at any point of the algorithm LDSMKB. Let $\tilde{C}'$ and $\tilde{C}''$ be the final values of C' and C'' respectively. Obviously, C'' is unimodular and $C'=AC''$ when C is constructed in Step 1. So $\tilde{C}''$ is unimodular, $\tilde{C}'=A\tilde{C}''$ and $\tilde{C}'x=b$ is consistent if and only if $Ax=b$ is consistent. Note that $\tilde{C}'$ has the form $[\tilde{C}'_1,\cdots,\tilde{C}'_r,0,\cdots,0]$ where $r=\operatorname{rank}(A)$. We claim that $\{\tilde{C}''_{r+1},\cdots,\tilde{C}''_n\}$ is a basis of the solution module of $Ax=0$. If $x=\sum_{i=r+1}^n z_i\tilde{C}''_i$, then $Ax=A\sum_{i=r+1}^n z_i\tilde{C}''_i=\sum_{i=r+1}^n z_iA\tilde{C}''_i=\sum_{i=r+1}^n z_i\tilde{C}'_i=0$. If x is an integral solution such that $Ax=0$, then $Ax=A\tilde{C}''(\tilde{C}'')^{-1}x=\tilde{C}'(\tilde{C}'')^{-1}x=0$. Let $z=(z_1,\cdots,z_n)^T=(\tilde{C}'')^{-1}x$. Then $0=\tilde{C}'z=\sum_{i=1}^n z_i\tilde{C}'_i=\sum_{i=1}^r z_i\tilde{C}'_i$. Since $\tilde{C}'_1,\cdots,\tilde{C}'_r$ are linearly independent, $z_i=0$ for $i=1,\cdots,r$. Note that $x=\tilde{C}''z=\sum_{i=1}^n z_i\tilde{C}''_i=\sum_{i=r+1}^n z_i\tilde{C}''_i$, an integral linear combination of $\tilde{C}''_{r+1},\cdots,\tilde{C}''_n$, and $\tilde{C}''_{r+1},\cdots,\tilde{C}''_n$ are linearly independent. This proves that $\{\tilde{C}''_{r+1},\cdots,\tilde{C}''_n\}$ is a basis of the solution module of $Ax=0$.

Adding an integral multiple of a column of C to B will preserve (i) the validity of the relation $B'=AB''-b$ and (ii) the consistency of $C'x=B'$. Suppose $\bar{B}$ is obtained from B by adding kC_i, where

$$C_i=\begin{pmatrix}C'_i\\C''_i\end{pmatrix}$$

is the ith column of C, to B. Then $\bar{B}=B+kC_i$. Let $\bar{B}'$ and $\bar{B}''$ be the vectors consisting of the first m elements and the last n elements of $\bar{B}$ respectively. Then $\bar{B}'=B'+kC'_i$ and $\bar{B}''=B''+kC''_i$. Note that $C'_i=AC''_i$ by the fact that $C'=AC''$. Since $\bar{B}'-A\bar{B}''=(B'+kC'_i)-A(B''+kC''_i)=(B'+kAC''_i)-(AB''+kAC''_i)=B'-AB''$, $\bar{B}'=A\bar{B}''-b$ if and only if $B'=AB''-b$. If $x=(x_1,\cdots,x_n)^T$ is a solution of the system $C'x=B'$, then $y=(x_1,\cdots,x_i+k,\cdots,x_n)^T$ is a solution of the system $C'y=\bar{B}'$, since $C'y=C'\{x+(0,\cdots,k,\cdots,0)^T\}=C'x+kC'_i=B'+kC'_i=\bar{B}'$. Similarly, if $y=(y_1,\cdots,y_n)^T$ is a solution of the system $C'y=\bar{B}'$, then $x=(y_1,\cdots,y_i-k,\cdots,y_n)^T$ is a solution of the system $C'x=B'$. This implies $C'x=B'$ is consistent if and only if $C'x=\bar{B}'$ is consistent. Obviously, $B'=AB''-b$ when B is constructed in Step 2. So we can conclude that $B'=AB''-b$ and that $C'x=B'$ is consistent if and only if $Ax=b$ is consistent at any point in the algorithm LDSMKB.

Let $\tilde{B}'$ and $\tilde{B}''$ be the final values of B' and B''. If $\tilde{B}'=0$ then $A\tilde{B}''=b$. Therefore, $\tilde{B}''$ is a particular solution of the system $Ax=b$. If $Ax=b$ is consistent, then $\tilde{C}'x=\tilde{B}'$ is consistent. Let $x^*=(x_1,\cdots,x_n)^T$ be a solution of $\tilde{C}'x=\tilde{B}'$. Then $\tilde{B}'=\tilde{C}'x^*=[\tilde{C}'_1,\cdots,\tilde{C}'_r,0,\cdots,0]\,(x_1,\cdots,x_n)^T=\sum_{j=1}^r x_j\tilde{C}'_j$. We claim $x_1=\cdots=x_r=0$. Let $1\leqq$

$h \leqq r$ and assume $x_1 = \cdots = x_{h-1} = 0$. Then $\tilde{B}' = \sum_{j=h}^{r} x_j \tilde{C}'_j$. Let $R = (i_1, \cdots, i_n)$ be the row sequence of A and let $k = i_h$. Then $\tilde{B}'_k = \sum_{j=h}^{r} x_j \tilde{C}'_{k,j}$. But $\tilde{C}'_{k,j} = 0$ for $j > h$ so $\tilde{B}'_k = x_h \tilde{C}'_{k,h}$. Since $|\tilde{B}'_k| < |\tilde{C}'_{k,h}|$ by virtue of Step 5, $x_h = 0$. By induction on h, therefore, $x_1 = \cdots = x_r = 0$ and $\tilde{B}' = 0$. This completes the proof. □

3. Computing time analyses. The computing time of the algorithm LDSMKB will be analyzed by employing the concepts of dominance and codominance introduced by Collins [4].

Let f and g be 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. Note that f and g are not restricted to functions of one variable, since the elements of S may be n-tuples. 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 $f \prec g$.

Dominance and codominance have the following fundamental properties.

THEOREM 2. *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* $S = S_1 \cup S_2$, *then* $f \preceq g$ *on* S_1 *and* $f \preceq g$ *on* S_2 *implies* $f \preceq g$ *on* S.

Proof. See [5] for a proof.

In SAC-2 an integer is represented in radix form with radix β. It is natural to define the length of an integer a to be the number of β-digits in its β-radix representation and denote it by $L_\beta(a)$, or just $L(a)$ if β is fixed in the context. $L_\beta(a)$ can be expressed by the formula

$$L_\beta(a) = \begin{cases} 1 & \text{if } a = 0, \\ \lfloor \log_\beta (|a|) \rfloor + 1 & \text{if } a \neq 0, \end{cases} \tag{1}$$

where $\lfloor x \rfloor$ is the floor function of the real number x, that is, the largest integer n such that $n \leqq x$. Also,

$$L_\beta(a) = \begin{cases} 1 & \text{if } a = 0, \\ \lceil \log_\beta (|a| + 1) \rceil & \text{if } a \neq 0, \end{cases} \tag{2}$$

where $\lceil x \rceil$ is the ceiling function of the real number x, that is, the smallest integer n such that $n \geqq x$.

The length function has the following properties:

$$L(a \pm b) \preceq L(a) + L(b), \tag{3}$$

$$L(a + b) \sim L(a) + L(b) \quad \text{for } ab \geqq 0, \tag{4}$$

$$L(ab) \sim L(a) + L(b) \quad \text{if } ab \neq 0, \tag{5}$$

$$L\left(\prod_{i=1}^{n} a_i\right) \preceq \sum_{i=1}^{n} L(a_i), \tag{6}$$

$$L\left(\prod_{i=1}^{n} a_i\right) \sim \sum_{i=1}^{n} L(a_i) \quad \text{if } |a_i| > 1 \quad \text{for } 1 \leqq i \leqq n. \tag{7}$$

Properties (6) and (7) hold with n variable, not just for each fixed n.

In this section we will first derive bounds on the coefficients for the algorithm LDSMKB, then use these bounds to derive a bound on the computing time.

The *norm* of a vector $V=(v_1,\cdots,v_n)\in Z^n$, denoted by $|V|$, is defined to be the nonnegative integer $\max_{1\leqq i\leqq n}|v_i|$. The *norm* of a matrix $A=(a_{ij})\in Z(m,n)$, denoted by $|A|$, is defined to be the nonnegative integer $\max_{1\leqq i,j\leqq n}|a_{ij}|$.

THEOREM 3. *Let A be an $n\times n$ matrix, and let d_i be the norm of the ith row of A, i.e., $d_i=\max_{1\leqq j\leqq n}|A_{i,j}|$. Then*

$$|\det(A)|\leqq n^{n/2}\prod_{i=1}^{n} d_i.$$

Proof. This is known as Hadamard's bound. See [8] for a proof.

Steps 3 to 4 form the major loop of the algorithm LDSMKB. Let C^0 be the initial value of the matrix C. Let C^k, $1\leqq k\leqq n-1$, be the value of C at the end of the kth iteration of Step 4. Theorem 4 gives bounds on the coefficients of C^k for $k=1,\cdots,n-1$.

THEOREM 4. *Let r_{k+1} and $(i_1,\cdots,i_{k+1})$ be the rank and the row sequence of $A[^{1,\cdots,m}_{1,\cdots,k+1}]$ respectively with $A\neq 0$. Then*

(1) $C^k_{i,j}=C^0_{i,j}$ *for* $1\leqq i\leqq m+n$ *and* $k+2\leqq j\leqq n$,

(2) $C^k_{i_h,j}=0$ *for* $1\leqq h<j\leqq k+1$,

(3) $|C^k_{i_h,j}|<|C^k_{i_h,h}|$ *for* $1\leqq j<h\leqq k+1$,

(4) $\prod_{h=1}^{k+1}|C^k_{i_h,h}|\leqq r_{k+1}^{r_{k+1}/2}|A|^{r_{k+1}}$,

(5) $|C^k_{i,j}|\leqq(k+1)r_{k+1}^{r_{k+1}/2}|A|^{r_{k+1}+1}$ *for* $1\leqq i\leqq m+n$ *and* $i\notin\{i_1,\cdots,i_{k+1}\}$ *and* $1\leqq j\leqq k+1$.

Proof. Let $k'=k+1$. During the first k iterations of the major loop, no operations have been performed on columns $j>k'$, so (1) is true. Let D' and D denote the submatrices consisting of the first k' columns of C^k and C^0 respectively. Then $D'=DK$ for some $k'\times k'$ unimodular matrix K, since only unimodular transformations have been applied on the first k' columns of C. Let

$$H'=D'\begin{bmatrix} i_1, & \cdots, & i_{k'}\\ 1, & \cdots, & k'\end{bmatrix}.$$

By virtue of the algorithm, H' is a pseudo-Hermite matrix. So (2) and (3) are true. Let

$$H=D\begin{bmatrix} i_1, & \cdots, & i_{k'}\\ 1, & \cdots, & k'\end{bmatrix},$$

$H'=HK$. Since H' is a triangular matrix, $\det(H')=\prod_{h=1}^{k'}D'_{i_h,h}=\prod_{h=1}^{k'}C^k_{i_h,h}$. Also, $\det(H')=\det(HK)=\det(H)\det(K)=\pm\det(H)$. Therefore, $\prod_{h=1}^{k'}|C^k_{i_h,h}|=|\det(H)|$. Note that the hth row of H is the i_hth row of D, which consists of the first k' elements of row i_h of C^0. Since $i_h\leqq m$ for $1\leqq h\leqq r_{k'}$ and $i_h>m$ for $r_{k'}<h\leqq k'$, the norm of the hth row of H is bounded by $|A|$ for $1\leqq h\leqq r_{k'}$ and 1 for $r_{k'}<h\leqq k'$. By Theorem 3, $|\det(H)|\leqq r_{k'}^{r_{k'}/2}|A|^{r_{k'}}$. Thus, (4) is true. Since H is nonsingular, H^{-1} exists. So $K=H^{-1}H'=\operatorname{adj}(H)H'/\det(H)$. Let K_{ij} and H'_{ij} be the elements in the ith rows and jth columns of K and H' respectively. Then $|K_{ij}|\leqq\sum_{l=1}^{k'}|H'_{lj}|\,|\operatorname{adj}(H)|/|\det(H)|$. Since H' is a pseudo-Hermite matrix, $\sum_{l=1}^{k'}|H'_{lj}|\leqq|H'_{11}|+(|H'_{22}|-1)+\cdots+(|H'_{k'k'}|-1)\leqq\prod_{l=1}^{k'}|H'_{ll}|=|\det(H')|=|\det(H)|$. Therefore, $|K|\leqq|\operatorname{adj}(H)|$. Since every element of

adj (H) is the determinant of some $k' \times k'$ minor of H and at most $r_{k'}$ rows of H come from the coefficient matrix A and the rest come from the identity matrix I, $|\text{adj}(H)| \leqq r_{k'}^{r_{k'}/2}|A|^{r_{k'}}$. Therefore, $|K| \leqq r_{k'}^{r_{k'}/2}|A|^{r_{k'}}$. Now $D' = DK$, hence $|D| \leqq k'|D||K| \leqq k'|A|(r_{k'}^{r_{k'}/2}|A|^{r_{k'}}) \leqq k' r_{k'}^{r_{k'}/2}|A|^{r_{k'}+1}$. So (5) is true. □

Let $(i_1, \cdots, i_k)$ be the row sequence of

$$A\begin{bmatrix} 1, & \cdots, & m \\ 1, & \cdots, & k \end{bmatrix}.$$

During the kth iteration of the major loop, Step 3 eliminates elements in column $k+1$ of C by a sequence of unimodular transformations. The jth iteration of the for-loop in Step 3 will change only columns j and $k+1$ of C. Let $\bar{C}^{k,j}$ be the value of C after the jth iteration of the for-loop during the kth iteration of the major loop. The following theorem gives bounds on the coefficients of $\bar{C}_j^{k,j}$ and $\bar{C}_{k+1}^{k,j}$.

THEOREM 5. *Let $d_h = |C_{i_h,h}^{k-1}|$ for $h = 1, \cdots, k$. If the rank of $A[{}^{1,\cdots,m}_{1,\cdots,k}]$ is r_k and $A \neq 0$, then*

$$(1) \qquad |\bar{C}_{i,j}^{k,j}|, |\bar{C}_{i,k+1}^{k,j}| \leqq |A| d_h \prod_{t=1}^{j} (2d_t) \text{ for } i = i_h \in \{i_{j+1}, \cdots, i_k\},$$

$$(2) \qquad |\bar{C}_{i,j}^{k,j}|, |\bar{C}_{i,k+1}^{k,j}| \leqq k|A| r_k^{r_k/2}|A|^{r_k+1} \prod_{t=1}^{j} (2d_t) \text{ for } i \notin \{i_1, \cdots, i_k\}.$$

Proof by induction on j. Let $k' = k+1$, and let $d_0 = k r_k^{r_k/2}|A|^{r_k+1}$. The theorem is obviously true for $k = 1$ if VIUT is not applied in the first iteration of the for-loop in Step 3. If VIUT is applied in the first iteration of the for-loop in Step 3, then $\bar{C}_{i,1}^{k,1} = u_1 C_{i,1}^{k-1} + v_1 C_{i,k'}^{k-1}$ and $\bar{C}_{i,k'}^{k,1} = u_2 C_{i,1}^{k-1} + v_2 C_{i,k'}^{k-1}$, where u_1, v_1, u_2 and v_2 are integers obtained by applying IDEGCD to $C_{i_1,1}^{k-1}$ and $C_{i_1,k'}^{k-1}$. If $C_{i_1,k'}^{k-1} \neq 0$, then $|u_1| \leqq |C_{i_1,k'}^{k-1}| = |C_{i_1,k'}^{0}| \leqq |A|$ and $|v_1| \leqq |C_{i_1,1}^{k-1}| = d_1$. Therefore, $|\bar{C}_{i,1}^{k,1}| \leqq |A||C_{i,1}^{k-1}| + d_1|A| \leqq 2|A| \max\{|C_{i,1}^{k-1}|, d_1\}$. If $C_{i_1,k'}^{k-1} = 0$, then $|u_1| = |\text{sign}(C_{i_1,1}^{k-1})| = 1$ and $|v_1| = |\text{sign}(C_{i_1,k'}^{k-1})| = 0$. Again, $|\bar{C}_{i,1}^{k,1}| = |C_{i,1}^{k-1}| \leqq 2|A| \max\{|C_{i,1}^{k-1}|, d_1\}$. If $i = i_h \in \{i_2, \cdots, i_k\}$, then $|C_{i,1}^{k-1}| < |C_{i_h,h}^{k-1}| = d_h$. Hence, $|\bar{C}_{i,1}^{k,1}| \leqq 2|A| \max\{d_h, d_1\} \leqq |A| d_h(2d_1)$. If $i \notin \{i_1, \cdots, i_k\}$, then $|C_{i,1}^{k-1}| \leqq d_0$. Hence, $|\bar{C}_{i,1}^{k,1}| \leqq |A| d_0(2d_1)$. So the theorem is true for $\bar{C}_{i,1}^{k,1}$. With a similar argument, the theorem is also true for $\bar{C}_{i,k}^{k,1}$.

Now assume the induction hypothesis is true for $j \geqq 1$. The theorem is obviously true for $k = j+1$ if VIUT is not applied in the $(j+1)$th iteration of the for-loop in Step 3. If VIUT is applied in the $(j+1)$th iteration of the for-loop in Step 3, then $\bar{C}_{i,j+1}^{k,j+1} = u_1 C_{i,j+1}^{k-1} + v_1 \bar{C}_{i,k'}^{k,j}$ and $\bar{C}_{i,k'}^{k,j+1} = u_2 C_{i,j+1}^{k-1} + v_2 \bar{C}_{i,k'}^{k,j}$, where u_1, v_1, u_2 and v_2 are integers obtained by applying IDEGCD to $a_1 = C_{i_{j+1},j+1}^{k-1}$ and $a_2 = \bar{C}_{i_{j+1},k'}^{k,j}$. Let $d = |A| d_{j+1} \prod_{t=1}^{j} (2d_t)$. If $a_2 \neq 0$, then $|u_1| \leqq |a_2| \leqq d$ by induction hypothesis, and $|v_1| \leqq |a_1| = d_{j+1}$. If $a_2 = 0$, then $|u_1| = |\text{sign}(a_1)| = 1$ and $|v_1| = |\text{sign}(a_2)| = 0$. For either case, $|\bar{C}_{i,j+1}^{k,j+1}| \leqq d|C_{i,j+1}^{k-1}| + d_{j+1}|\bar{C}_{i,k'}^{k,j}|$. If $i = i_h \in \{i_{j+2}, \cdots, i_k\}$, then $|C_{i,j+1}^{k-1}| < |C_{i_h,h}^{k-1}| = d_h$ and $|\bar{C}_{i,k'}^{k,j}| \leqq |A| d_h \prod_{t=1}^{j} (2d_t)$ by induction hypothesis. Thus, $|\bar{C}_{i,j+1}^{k,j+1}| \leqq 2|A| d_h d_{j+1} \prod_{t=1}^{j} (2d_t) = |A| d_h \prod_{t=1}^{j+1} (2d_t)$. If $i \notin \{i_1, \cdots, i_k\}$, then $|C_{i,j+1}^{k-1}| \leqq d_0$ by Theorem 4(5), and $|\bar{C}_{i,k'}^{k,j}| \leqq |A| d_0 \prod_{t=1}^{j} (2d_t)$. Thus, $|\bar{C}_{i,j+1}^{k,j+1}| \leqq 2|A| d_0 d_{j+1} \prod_{t=1}^{j} (2d_t) = |A| d_0 \prod_{t=1}^{j+1} (2d_t)$. So the theorem is true for $\bar{C}_{i,j+1}^{k,j+1}$. Similarly for $\bar{C}_{i,k'}^{k,j+1}$. □

COROLLARY. *During the kth iteration of Step 3 of the algorithm* LDSMKB, *if $A \neq 0$ then the norm of C is bounded by $2^k k r_k^{r_k}|A|^{2r_k+2}$.*

Proof. The hth column of C, $1 \leqq h \leqq k$, is changed only once during the kth iteration of Step 3, and $\bar{C}_h^{k,h}$ is the new value of the hth column of C. By Theorem 5 and the fact that $|\bar{C}_{i_h,h}^{k,h}| = \gcd(C_{i_h,h}^{k-1}, \bar{C}_{i_h,k'}^{k,h-1}) \leqq |C_{i_h,h}^{k-1}| = d_h$, $|C| \leqq k r_k^{r_k/2}|A|^{r_k+2} \prod_{t=1}^{k} (2d_t)$ during the kth iteration of Step 3. Since $\prod_{t=1}^{k} (2d_t) = 2^k \prod_{t=1}^{k} |C_{i_t,t}^{k-1}| \leqq 2^k r_k^{r_k/2}|A|^{r_k}$ by Theorem 4(4), $|C| \leqq 2^k k r_k^{r_k}|A|^{2r_k+2}$. □

Let $R_{k+1}=(i_1,\cdots,i_{k+1})$ be the row sequence of $A[{}^{1,\cdots,m}_{1,\cdots,k+1}]$. Let $\tilde{C}^{k,p}_j = (\tilde{C}^{k,p}_{1,j},\cdots,\tilde{C}^{k,p}_{m+n,j})$ be the jth column of C after the i_pth element, $j<p\leqq k+1$, of the jth column of C is normalized during the kth iteration of the major loop. The following theorem gives bounds on $\tilde{C}^{k,p}_{i,j}$.

THEOREM 6. *Let $d_h=|C^k_{i_h,h}|$ for $h=1,\cdots,k+1$. If $A\neq 0$, then*

$$(1)\qquad |\tilde{C}^{k,p}_{i,j}|\leqq 2^{k+1}(k+1)r_{k+1}^{r_{k+1}}|A|^{2(r_{k+1}+1)}d_h\prod_{t=j+2}^{p}(2d_t)$$

$$\text{if } i=i_h\in\{i_{p+1},\cdots,i_{k+1}\},$$

$$(2)\qquad |\tilde{C}^{k,p}_{i,j}|\leqq 2^{k+1}(k+1)^2 r_{k+1}^{3r_{k+1}/2}|A|^{3(r_{k+1}+1)}\prod_{t=j+2}^{p}(2d_t)$$

$$\text{if } i\notin\{i_1,\cdots,i_{k+1}\}.$$

Proof by induction on p. Let $\tilde{C}^k_j=(\tilde{C}^k_{1,j},\cdots,\tilde{C}^k_{m+n,j})$ be the jth column of C before the normalization of the jth column begins. Let $d=2^k(k+1)r_{k+1}^{r_{k+1}}|A|^{2(r_{k+1}+1)}$. Then $|\tilde{C}^k_{i,j}|\leqq 2^k k r_k^{r_k}|A|^{2r_k+2}\leqq d$ by Corollary 5.1. Let $s=i_{j+1}$. Then $\tilde{C}^{k,j+1}_{i,j}=\tilde{C}^k_{i,j}-qC^k_{i,j+1}$, where $q=[\tilde{C}^k_{s,j}/C^k_{s,j+1}]$. Since $|\tilde{C}^k_{i,j}|\leqq d$ and $|q|\leqq|\tilde{C}^k_{s,j}|\leqq d$, $|\tilde{C}^{k,j+1}_{i,j}|\leqq d+d|C^k_{i,j+1}|$. If $i=i_h\in\{i_{j+2},\cdots,i_{k+1}\}$, then $|C^k_{i,j+1}|<|C^k_{i,h}|=d_h$. Hence, $|\tilde{C}^{k,j+1}_{i,j}|\leqq d(1+d_h)\leqq 2d\,d_h$. If $i\notin\{i_1,\cdots,i_{k+1}\}$, then by Theorem 4(5) $|\tilde{C}^{k,j+1}_{i,j}|\leqq 2d|C^k_{i,j+1}|\leqq 2d(k+1)r_{k+1}^{r_{k+1}/2}|A|^{r_{k+1}+1}=2d'$, where $d'=2^k(k+1)^2r_{k+1}^{3r_{k+1}/2}|A|^{3(r_{k+1}+1)}$. Therefore, the theorem is true for $p=j+1$.

Now assume inductively that the theorem is true for $p\geqq j+1$. Let $s=i_{p+1}$. Then $\tilde{C}^{k,p+1}_{i,j}=\tilde{C}^{k,p}_{i,j}-[\tilde{C}^{k,p}_{s,j}/C^k_{s,p+1}]C^k_{i,p+1}$ and hence

$$|\tilde{C}^{k,p+1}_{i,j}|\leqq|\tilde{C}^{k,p}_{i,j}|+|\tilde{C}^{k,p}_{s,j}||C^k_{i,p+1}|\leqq|\tilde{C}^{k,p}_{i,j}|+2d\,d_{p+1}\left\{\prod_{t=j+2}^{p}(2d_t)\right\}|C^k_{i,p+1}|.$$

If $i=i_h\in\{i_{p+2},\cdots,i_{k+1}\}$, then

$$|\tilde{C}^{k,p}_{i,j}|\leqq 2d\,d_h\prod_{t=j+2}^{p}(2d_t)$$

and $|C^k_{i,p+1}|<|C^k_{i,h}|=d_h$. Hence, $|\tilde{C}^{k,p+1}_{i,j}|\leqq 2d\,d_h(1+d_{p+1})\prod_{t=j+2}^{p}(2d_t)\leqq 2d\,d_h\prod_{t=j+2}^{p+1}(2d_t)$. If $i\notin\{i_1,\cdots,i_{k+1}\}$, then $|\tilde{C}^{k,p}_{i,j}|\leqq 2d'\prod_{t=j+2}^{p}(2d_t)$ by induction hypothesis and $|C^k_{i,p+1}|\leqq(k+1)r_{k+1}^{r_{k+1}/2}|A|^{r_{k+1}+1}$ by Theorem 4(5). Hence, $|\tilde{C}^{k,p+1}_{i,j}|\leqq 2d'(1+d_{p+1})\prod_{t=j+2}^{p}(2d_t)\leqq 2d'\prod_{t=j+2}^{p+1}(2d_t)$. Therefore, the theorem is true for $p+1$. □

COROLLARY 6.1. *During the kth iteration of Step 4 of the algorithm* LDSMKB, *if $A\neq 0$, then $|C|\leqq 2^{2k}(k+1)^2r_{k+1}^{2r_{k+1}}|A|^{4(r_{k+1}+1)}$, where r_{k+1} is the rank of $A[{}^{1,\cdots,m}_{1,\cdots,k+1}]$.*

Proof. Setting $p=k+1$ in Theorem 6, and using Theorem 4(4), we find $|C|\leqq 2d'\prod_{t=j+2}^{k+1}(2d_t)\leqq 2^kd'\prod_{t=1}^{k+1}d_t\leqq 2^kd'r_{k+1}^{r_{k+1}}|A|^{r_{k+1}+1}=2^{2k}(k+1)^2r_{k+1}^{2r_{k+1}}|A|^{4(r_{k+1}+1)}$. □

Since $k\leqq n-1$, $r_k\leqq r_{k+1}\leqq\operatorname{rank}(A)=r$ and $n^2<2^{n+1}$ for $n>0$, we have the following theorem immediately from the corollaries of Theorems 5 and 6.

THEOREM 7. *At any point of the algorithm* LDSMKB, *if $A\neq 0$, then $|C|<2^{3n}(r|A|)^{4(r+1)}$, where $r=\operatorname{rank}(A)$.*

The next theorem gives bounds on elements of the vector B.

THEOREM 8. *Let $(i_1,\cdots,i_n)$ be the row sequence of A. Let C^* be the final value of C. Let $e_h=|C^*_{i_h,h}|$ for $h=1,\cdots,n$. Let B^k be the value of B right after the kth application of the algorithm* VIERED *in Step 6. Then, if $A\neq 0$,*

(1) $|B_i^k| < e_h$ *if* $i = i_h \in \{i_1, \cdots, i_k\}$,
(2) $|B_i^k| \leqq 2^k|b|e_h$ *if* $i = i_h \in \{i_{k+1}, \cdots, i_n\}$, *and*
(3) $|B_i^k| \leqq 2^k|b|n(r|A|)^{r+1}$ *if* $i \notin \{i_1, \cdots, i_n\}$.

Proof by induction on k. Let B^0 be the initial value of B, i.e., $B^0 = [{}^{-b}_{0}]$. For $k = 1$, let $t = i_1$. Since $B^1 = B^0 - [B_t^0/C_{t,1}^*]C_1^*$, $B_i^1 = B_i^0 - [B_t^0/C_{t,1}^*]C_{i,1}^*$. If $i \in \{i_1\}$, then $B_i^1 = B_t^0 - [B_t^0/C_{t,1}^*]C_{t,1}^*$, and hence, $|B_i^1| < |C_{t,1}^*| = e_1$. Therefore, (1) is true for $k = 1$. Note that $|B_i^1| \leqq |B_i^0| + |B_t^0||C_{i,1}^*| \leqq |b| + |b||C_{i,1}^*|$. If $i = i_h \in \{i_2, \cdots, i_n\}$, then $|C_{i,1}^*| < |C_{i,h}^*| = e_h$. Therefore, $|B_i^1| \leqq |b| + |b|e_h \leqq 2|b|e_h$. That is, (2) is true for $k = 1$. If $i \notin \{i_1, \cdots, i_n\}$ then by Theorem 4(5), $|B_i^1| \leqq 2|b||C_{i,1}^*| \leqq 2|b|n(r|A|)^{r+1}$ so (3) is true for $k = 1$.

Now assume the theorem is true for $k = p \geqq 1$. Let $t = i_{p+1}$. Since $B^{p+1} = B^p - [B_t^p/C_{t,p+1}^*]C_{p+1}^*$, $B_i^{p+1} = B_i^p - [B_t^p/C_{t,p+1}^*]C_{i,p+1}^*$. If $i = i_h \in \{i_1, \cdots, i_p\}$, then since $C_{i,p+1}^* = 0$, $|B_i^{p+1}| = |B_i^p| < e_h$ by induction hypothesis. If $i = i_{p+1} = t$, then $|B_i^{p+1}| = |B_t^p - [B_t^p/C_{t,p+1}^*]C_{t,p+1}^*| < |C_{t,p+1}^*| = e_{p+1}$. Therefore, (1) is true for $k = p+1$. If $i = i_h \in \{i_{p+2}, \cdots, i_n\}$, then by the induction hypothesis $|[B_t^p/C_{t,p+1}^*]| \leqq |B_t^p|/|C_{t,p+1}^*| \leqq (2^p|b|e_{p+1})/e_{p+1} = 2^p|b|$. Thus, $|B_i^{p+1}| \leqq |B_i^p| + 2^p|b||C_{i,p+1}^*| \leqq 2^p|b|e_h + 2^p|b||C_{i,h}^*| = 2^p|b|e_h + 2^p|b|e_h = 2^{p+1}|b|e_h$. Therefore, (2) is true for $k = p+1$. If $i \notin \{i_1, \cdots, i_n\}$, then $|B_i^p| \leqq 2^p|b|n(r|A|)^{r+1}$ by the induction hypothesis (3), and $|C_{i,p+1}^*| \leqq n(r|A|)^{r+1}$ by Theorem 4(5). As shown in the proof for (2) $|[B_t^p/C_{t,p+1}^*]| \leqq 2^p|b|$. Therefore, $|B_i^{p+1}| \leqq 2^p|b|n(r|A|)^{r+1} + 2^p|b|n(r|A|)^{r+1} = 2^{p+1}|b|n(r|A|)^{r+1}$. This completes the proof. □

COROLLARY. *At any point of the algorithm* LDSMKB, *if* $A \neq 0$, *then* $|B| \leqq 2^{2n}|b|(r|A|)^{r+1}$.

Proof. By Theorem 4(4), $e_h \leqq (r|A|)^r$ for $h = 1, \cdots, n$. Therefore by Theorem 8, at any point of the algorithm LDSMKB, $|B| \leqq 2^n|b|n(r|A|)^{r+1} = 2^n n|b|(r|A|)^{r+1} < 2^{2n}|b|(r|A|)^{r+1}$. □

Before we analyze the computing time of LDSMKB, we list theorems for the computing times of those SAC-2 algorithms referenced by LDSMKB. Proofs of these theorems are given in [3].

THEOREM 9. *The computing times of the algorithms* FIRST, RED, SFIRST, ADV, COMP *are all codominant with* 1.

THEOREM 10. *Let L be the list* $(a_1, \cdots, a_n)$. *Then* $t_{\text{LENGTH}}(L) \sim n+1$, $t_{\text{REDUCT}}(L, i) \sim i+1$, $t_{\text{LEROT}}(L, i, j) \leqq j$ *and* $t_{\text{LEINST}}(L, i, a) \sim i+1$.

THEOREM 11. *Let* $U \in Z^m$ *and* $V \in Z^m$. *Then* $t_{\text{VIAZ}}(V, n) \sim m+n$, $t_{\text{VINEG}}(V) \leqq mL(|V|)$, $t_{\text{VIUT}}(U, V, i) \leqq mL(|U|)L(|V|)$ *and* $t_{\text{VIERED}}(U, V, i) \leqq mL(|U|)L(|V|)$.

THEOREM 12. *Let A be an* $m \times n$ *integral matrix. Then* $t_{\text{MIAIM}}(A) \sim n(m+n)$.

Finally, Theorem 13 gives a bound on the computing time of the algorithm LDSMKB.

THEOREM 13. *Let* $A \in Z(m, n)$, *with* $A \neq 0$, *and let* $b \in Z^n$. *Then* $t_{\text{LDSMKB}}(A, b) \leqq n^3(m+n)\{n + rL(r|A|)\}^2 + r(m+n)L(|b|)\{n + rL(r|A|)\}$.

Proof. Let t_i be the computing time of Step i. Obviously, $t_1 \leqq n(m+n) + mL(|b|)$ and $t_2 \sim m$ by Theorems 10, 11, and 12. Let $t_{i,k}$, $3 \leqq i \leqq 4$ and $1 \leqq k \leqq n-1$, be the computing time of Step i in the kth iteration of the major loop. The computing time for VIUT is the most significant one in Step 3. VIUT is called at most k times during the kth iteration. Therefore, by Theorems 7 and 11, $t_{3,k} \leqq k(m+n)\{L(d^*)\}^2$, where $d^* = 2^{3n}(r|A|)^{4(r+1)}$. The computing time for VIERED is the most significant one in Step 4. VIERED is called no more than k^2 times during the kth iteration. So $t_{4,k} \leqq k^2(m+n)\{L(d^*)\}^2$ by Theorems 7 and 11. Thus,

$$t_3 + t_4 = \sum_{k=1}^{n-1} (t_{3,k} + t_{4,k}) \leqq \sum_{k=1}^{n-1} \{k(m+n) + k^2(m+n)\}\{L(d^*)\}^2 \sim n^3(m+n)\{L(d^*)\}^2.$$

t_5 is codominant with the computing time for the for-loop and the repeat-loop in Step 5. By Theorems 7 and 11 and the corollary of Theorem 8, $t_5 \leqq r(m+n)L(d^*) \times L(e^*)+(m+n)$, where $e^* = 2^{2n}|b|(r|A|)^{r+1}$. Obviously, $t_6 \sim m(n-r)+1$ and $t_7 \sim 1$. Therefore, $t_{\mathrm{LDSMKB}}(A, b) \leqq n^3(m+n)\{L(d^*)\}^2 + r(m+n)L(d^*)L(e^*)$. Since

$$L(d^*) \sim n + rL(r|A|) \quad \text{and} \quad L(e^*) \sim n + L(|b|) + rL(r|A|) \sim L(d^*) + L(|b|),$$

$$t_{\mathrm{LDSMKB}}(A, b) \leqq n^3(m+n)\{L(d^*)\}^2 + r(m+n)\{L(d^*)\}^2 + r(m+n)L(d^*)L(|b|)$$

$$\sim n^3(m+n)\{n + rL(r|A|)\}^2 + r(m+n)L(|b|)\{n + rL(r|A|)\}. \qquad \square$$

4. A Rosser-type algorithm. Consider the following linear Diophantine equation:

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = b. \tag{8}$$

Without loss of generality, let us assume $a_1 \geqq a_2 \geqq \cdots \geqq a_n \geqq 0$ and $a_1 > 0$, since if $a_i < 0$ we can replace x_i by $-x_i$, if $a_i < a_j$ for some $i < j$ we can interchange x_i and x_j and if $a_1 = 0$ the equation becomes trivial. Rosser's algorithm begins with the following matrix:

$$C = \begin{bmatrix} a_1 & a_2 & \cdots & a_n \\ 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}.$$

Let $c_{i,j}$ be the element in the ith row and jth column of C, and let C_j be the jth column of C. Then the algorithm consists of the following steps:

(1) while $c_{1,2} \neq 0$ do $\{C_1 \leftarrow C_1 - \lfloor c_{1,1}/c_{1,2} \rfloor C_2$; sort $C_1, \cdots, C_n$ in descending order according to their leading elements$\}$.

(2) At this point the matrix C has the form

$$\begin{bmatrix} c & 0 & \cdots & 0 \\ U_1 & U_2 & \cdots & U_n \end{bmatrix},$$

where $U_j \in Z^n$ and $c = gcd(a_1, \cdots, a_n)$. If $c \nmid b$ then (8) has no solution; otherwise, $X = qU_1 + y_2U_2 + \cdots + y_nU_n$, where $X = (x_1, \cdots, x_n)^T$, $q = b/c$ and $y_2, \cdots, y_n$ are arbitrary integers, is a general solution of (8).

One may easily verify that the matrix $U = [U_1, \cdots, U_n]$ is unimodular. Since $c_{1,1}$ and $c_{1,2}$ are the largest and the second largest elements in the first row of C, the integer $\lfloor c_{1,1}/c_{1,2} \rfloor$ computed in Step 1 is usually small. So Rosser's algorithm usually will find a U with evenly small elements, especially when n is large, while other methods usually will find a U with some large elements and the rest quite small, including many zeros and ones.

In solving the linear Diophantine system $Ax = b$, one would like to compute a unimodular matrix U such that $A' = AU$ is a *column echelon matrix*, i.e., if $A' \in Z(m, n)$, $r = \text{rank}(A')$ and k_j, $1 \leqq j \leqq r$, is the row index of the leading nonzero element of column j of A', then $1 \leqq k_1 < \cdots < k_r$ and columns $r+1, \cdots, n$ of A' are zero. In the algorithm LDSSBR, U is obtained by executing statements similar to those in Step 1 of Rosser's algorithm r (the rank of A) times. In fact, U is the product of r unimodular matrices $U_1, \cdots, U_r$ associated with executions of Step 1 of Rosser's algorithm. The sizes of the elements in the U_i's will directly affect the rate of coefficient growth in solving a linear Diophantine system. Since, in general, Rosser's algorithm computes U_i's with evenly small elements, the algorithm controls the coefficient growth very well.

Three additional SAC-2 algorithms are referenced by LDSSBR.

$B \leftarrow$ MINNCT (A)

[Matrix of integers, nonnegative column transformation. $A = (a_{ij})$ is an $m \times n$ integral matrix. $B = (b_{ij})$ is the $m \times n$ integral matrix with $b_{ij} = a_{ij}$ if $a_{1j} \geqq 0$ and $b_{ij} = -a_{ij}$ if $a_{1j} < 0$. A is modified.]

$B \leftarrow$ MICS (A)

[Matrix of integers column sort. A is an integral matrix with nonnegative elements in the first row. B is an integral matrix obtained by sorting columns of A such that elements of the first row are in descending order. A is modified.]

$B \leftarrow$ MICINS (A, V)

[Matrix of integers column insertion. A is an $m \times n$ integral matrix represented by the list $(A_1, A_2, \cdots, A_n)$, where A_i is the list $(a_{1i}, \cdots, a_{mi})$ representing column i of A and $a_{11} \geqq a_{12} \geqq \cdots \geqq a_{1n}$. $V = (v_1, \cdots, v_m)$ is an integral vector with $v_1 < a_{11}$. Let i be the largest integer such that $a_{1i} \geqq v_1$. Then B is the matrix represented by the list $(A_1, \cdots, A_i, V, A_{i+1}, \cdots, A_n)$. A is modified.]

Following is the formal algorithm description of LDSSBR.

LDSSBR $(A, b; x^*, N)$

[Linear Diophantine system solution, based on Rosser's ideas. A is an $m \times n$ integral matrix. b is an integral m-vector. If the Diophantine system $Ax = b$ is consistent, then x^* is a particular solution and N is a list of basis vectors of the solution module of $Ax = 0$. Otherwise, x^* and N are null lists. A and b are modified.]

safe $b_1, C', C_2, m, N, n, s, x^*$.

(1) [Initialize.] $n \leftarrow$ LENGTH (A); $m \leftarrow$ LENGTH (b).
(2) [Adjoin identity matrix to A and zero vector to $-b$.] $C \leftarrow$ MIAIM (A); $B \leftarrow$ VIAZ (VINEG (b), n).
(3) [Sort columns of C.] $C \leftarrow$ MINNCT (C); $C \leftarrow$ MICS (C).
(4) [Pivot row zero.] $C_1 \leftarrow$ FIRST (C); if FIRST $(C_1) = 0$ then go to 6.
(5) [Eliminate pivot row.] repeat {$B \leftarrow$ VIERED $(B, C_1, 1)$; $C \leftarrow$ RED (C); if $C = (\ )$ then $s \leftarrow 0$ else {$C_2 \leftarrow$ FIRST (C); $s \leftarrow$ FIRST (C_2); if $s \neq 0$ then {$C_1 \leftarrow$ VIERED $(C_1, C_2, 1)$; $C \leftarrow$ MICINS (C, C_1); $C_1 \leftarrow C_2$}}} until $s = 0$; $n \leftarrow n - 1$.
(6) [System inconsistent?] ADV $(B; b_1, B)$; if $b_1 \neq 0$ then {$x^* \leftarrow (\ )$; $N \leftarrow (\ )$; return}.
(7) [Remove pivot row.] $C' \leftarrow C$; while $C' \neq (\ )$ do {$C_1 \leftarrow$ FIRST (C'); $C_1 \leftarrow$ RED (C_1); SFIRST (C', C_1); $C' \leftarrow$ RED (C')}; $m \leftarrow m - 1$.
(8) [Finished?] if $m > 0$ then {if $n > 0$ then go to 3 else go to 6}; $x^* \leftarrow B$; $N \leftarrow C$; return.

Step 1 computes the number of variables n and the number of equations m in the system. Step 2 constructs the matrix $C = \left[\begin{smallmatrix} A \\ I \end{smallmatrix}\right]$, where I is the identity matrix in $Z(n, n)$, and the vector $B = \left[\begin{smallmatrix} -b \\ 0 \end{smallmatrix}\right]$, where 0 is the zero vector in Z^n. Steps 3 to 8 form a loop, which computes a unimodular matrix U, such that AU is a column echelon matrix, by repeatedly executing a step similar to Step 1 in Rosser's algorithm (described in this section), checks the consistency of the system and computes a particular solution if there exists one. Step 3 makes the elements in the first row nonnegative and sorts them in descending order by performing two kinds of elementary column operations: multiplying a column by -1 and interchanging two columns. This is a preparatory step for Step 5 which employs Rosser's ideas. Step 4 checks whether the first row is zero and, if so, skips the execution of Step 5. Step 5 basically does two things: (1) reduces the size of the first element of B by repeatedly subtracting multiples of the first column of C from B, and (2) performs Step 1 of Rosser's algorithm. Note that two kinds of elementary column operations are involved in this step, i.e., subtracting

a multiple of a column from another column and interchanging two columns. Also note that, because the first column of C is no longer useful in later computations after leaving the repeat-loop, it is deleted from C through the algorithm RED before leaving the repeat-loop. Step 6 checks the consistency of the system. If it is inconsistent, then Step 6 returns the null list for x^* and N. Step 7 removes the first row of C, whose elements are zero. The deletions of these unnecessary columns and rows make this algorithm more efficient.

5. Empirical observations. We observed that the first k diagonal elements of C after the kth iteration are small integers (a typical example is given in the Appendix). An explanation is given below.

THEOREM 14 (Cauchy–Binet theorem). *Let H and K be $k \times n$ and $n \times k$ matrices respectively. If $G = HK$, then*

$$\det(G) = \sum_{1 \leqq i_1 < \cdots < i_k \leqq n} H\begin{pmatrix} 1, & \cdots, & k \\ i_1, & \cdots, & i_k \end{pmatrix} K\begin{pmatrix} i_1, & \cdots, & i_k \\ 1, & \cdots, & k \end{pmatrix}.$$

Proof. See [11, p. 37] or [8, p. 9] for a proof.

Let $Q_{k,n}$, with $1 \leqq k \leqq n$, be the set of all k-tuples $(i_1, \cdots, i_k)$ of integers such that $1 \leqq i_1 < \cdots < i_k \leqq n$. Let $S \in Z(k, n)$ with $k \leqq n$. If

$$S\begin{pmatrix} 1, & \cdots, & k \\ i_1, & \cdots, & i_k \end{pmatrix} = 0$$

for all $(i_1, \cdots, i_k) \in Q_{k,n}$, then let $d_k(S) = 0$; otherwise, let $d_k(S)$ be the greatest common divisor of all $S\binom{1,\cdots,k}{i_1,\cdots,i_k}$ such that $(i_1, \cdots, i_k) \in Q_{k,n}$. $d_k(S)$ is called the kth *determinantal divisor of S*.

THEOREM 15. *Let $S \in Z(k, n)$. If $S' = SU$ and U is a unimodular matrix, then $d_k(S') = d_k(S)$.*

Proof. Since $d_k(S) = 0$ if and only if rank $(S) < k$ and rank $(S') =$ rank (S), $d_k(S') = 0$ if and only if $d_k(S) = 0$. Let us assume $d_k(S) \neq 0$. For any $(j_1, \cdots, j_k) \in Q_{k,n}$, let

$$G = S'\begin{bmatrix} 1, & \cdots, & k \\ j_1, & \cdots, & j_k \end{bmatrix}, \quad H = \begin{bmatrix} 1, & \cdots, & k \\ 1, & \cdots, & n \end{bmatrix}, \quad K = U\begin{bmatrix} 1, & \cdots, & n \\ j_1, & \cdots, & j_k \end{bmatrix}.$$

Then, by Theorem 14,

$$S'\begin{pmatrix} 1, & \cdots, & k \\ j_1, & \cdots, & j_k \end{pmatrix} = \sum_{(i_1,\cdots,i_k) \in Q_{k,n}} S\begin{pmatrix} 1, & \cdots, & k \\ i_1, & \cdots, & i_k \end{pmatrix} U\begin{pmatrix} i_1, & \cdots, & i_k \\ j_1, & \cdots, & j_k \end{pmatrix}.$$

Since

$$d_k(S) \mid S\begin{pmatrix} 1, & \cdots, & k \\ i_1, & \cdots, & i_k \end{pmatrix} \quad \text{for any } (i_1, \cdots, i_k) \in Q_{k,n},$$

$$d_k(S) \mid S'\begin{pmatrix} 1, & \cdots, & k \\ j_1, & \cdots, & j_k \end{pmatrix} \quad \text{for any } (j_1, \cdots, j_k) \in Q_{k,n},$$

and hence, $d_k(S) \mid d_k(S')$. Similarly, $d_k(S') \mid d_k(S)$, since $S = S'U^{-1}$ and U^{-1} is unimodular. Since $d_k(S)$ and $d_k(S')$ are positive, $d_k(S) = d_k(S')$. □

Suppose S' is lower triangular and of full row rank. Then $S'\left[\begin{smallmatrix} 1,\cdots,k \\ 1,\cdots,k \end{smallmatrix}\right]$ is the only $k \times k$ minor of S' with nonzero determinant. Let $s_1, \cdots, s_k$ be the diagonal elements of S'. Then

$$\prod_{i=1}^{k} |s_i| = \left| S'\begin{pmatrix} 1, & \cdots, & k \\ 1, & \cdots, & k \end{pmatrix} \right| = d_k(S') = d_k(S)$$

by Theorem 15. If $k < n$, then $d_k(S)$ is the greatest common divisor of the determinants of all the $k \times k$ minors of S. Suppose S is a random matrix. Then

$$S\begin{pmatrix} 1, & \cdots, & & k \\ 1, & \cdots, & k-1, & h \end{pmatrix} \quad \text{for } h = k, k+1, \cdots, n$$

are random integers whose greatest common divisor, say $d'_k(S)$, in general will be small (see [10, p. 301]). Since $d_k(S) \leqq d'_k(S)$, the s_i's in general are small.

To the algorithm LDSSBR, let S be the matrix

$$C^0\begin{bmatrix} i_1, & \cdots, & i_k \\ 1, & \cdots, & n \end{bmatrix}$$

for the kth iteration, where $C^0 = [{}^A_I]$ and i_h is the hth element of the row sequence of A. To the algorithm LDSMKB, let S be the matrix

$$C^0\begin{bmatrix} i_1, & \cdots, & i_k \\ 1, & \cdots, & k+1 \end{bmatrix}.$$

Then our observation follows from the above argument.

Finally, we will present several tables of empirical results (Tables 1–5) which indicate some aspects of the performances of the algorithms LDSSBR and LDSMKB.

Symbols in these tables have the following meanings:

n—number of variables in the system,
m—number of equations in the system,
r—rank of the coefficient matrix,
d_0—length, in bits, of the norm of the coefficient matrix,
e_0—length, in bits, of the norm of the right-hand side,
d—length, in bits, of the longest integer occurring in the matrix C during the computation,
e—length, in bits, of the longest integer occurring in the vector B during the computation,
$\bar{d}$—length, in bits, of the longest integer in the basis obtained,
$\bar{e}$—length, in bits, of the longest integer in the particular solution obtained,
t—VAX execution time, in seconds,
τ—ratio of LDSSBR time to LDSMKB time.

These tables are obtained by applying LDSSBR and LDSMKB to sets of randomly generated systems; that is, coefficients in these systems are randomly chosen from prespecified intervals. For any given n, m, d_0 and e_0 in these tables, random systems were generated until a consistent one was found. About 40 per cent of the random systems generated were inconsistent.

TABLE 1

	LDSSBR					LDSMKB				
n	d	e	$\bar{e}$	t	d	e	$\bar{e}$	t	$\bar{d}$	τ
5	41	35	35	.73	87	50	49	.86	41	.85
7	60	59	59	2.45	128	71	71	2.87	60	.85
9	82	82	82	6.02	170	92	91	6.64	82	.91
11	100	97	97	12.24	204	110	110	13.45	100	.91
13	122	119	119	21.82	248	132	132	24.69	122	.88
15	149	149	149	38.69	306	160	160	45.54	149	.85

Notes: $m = r = n - 1$ *and* $d_0 = e_0 = 10$.

By examining the tables, we find that the ratio of the computing time of LDSSBR to that of LDSMKB ranged from 0.85 to 3.47. In all cases LDSSBR found smaller solutions than LDSMKB did; that is, the norms of the particular solutions obtained by LDSSBR were smaller than the norms of those obtained by LDSMKB. In all cases where $n-r>1$, LDSSBR also obtained smaller solution module bases in the same sense.

TABLE 2

	LDSSBR				LDSMKB					
n	d	e	$\bar{e}$	t	d	e	$\bar{e}$	t	$\bar{d}$	τ
5	77	76	76	1.97	164	96	96	1.35	77	1.46
7	119	118	118	7.56	260	138	137	4.69	119	1.61
9	161	159	159	19.61	329	180	178	12.14	161	1.62
11	202	201	200	38.26	416	222	222	25.01	202	1.53
13	247	246	246	69.39	512	268	267	52.95	247	1.31
15	288	285	285	119.13	590	308	305	96.30	288	1.24

Notes: $m=r=n-1$ *and* $d_0=e_0=20$.

TABLE 3

	LDSSBR				LDSMKB					
n	d	e	$\bar{e}$	t	d	e	$\bar{e}$	t	$\bar{d}$	τ
5	161	160	160	7.89	343	201	201	2.60	161	3.03
7	239	238	238	25.74	514	278	278	10.46	239	2.46
9	320	317	317	58.64	670	360	355	26.84	320	2.18
11	403	403	403	120.43	832	443	441	65.24	403	1.85
13	485	484	484	206.63	999	525	524	136.48	485	1.51
15	567	566	566	359.11	1,169	607	605	261.65	567	1.37

Notes: $m=r=n-1$ *and* $d_0=e_0=40$.

TABLE 4

	LDSSBR				LDSMKB					
n	d	e	$\bar{e}$	t	d	e	$\bar{e}$	t	$\bar{d}$	τ
5	239	238	238	15.89	531	299	298	4.58	239	3.47
7	358	356	356	49.15	771	418	418	17.60	358	2.79
9	483	480	480	118.03	1,023	542	541	50.10	483	2.36
11	601	599	599	229.39	1,260	661	661	120.78	601	1.90
13	723	722	722	411.54	1,491	783	783	258.27	723	1.59
15	849	847	847	761.56	1,752	908	908	519.64	849	1.47

Notes: $m=r=n-1$ *and* $d_0=e_0=60$.

TABLE 5

	LDSSBR					LDSMKB				
m	d	e	$\bar{d}$	$\bar{e}$	t	d	e	$\bar{d}$	$\bar{e}$	t
1	60	60	13	12	3.78	178	116	57	116	4.40
2	72	70	30	25	11.76	298	174	115	174	9.26
3	85	85	41	38	17.81	414	235	177	235	16.17
4	99	96	64	64	30.24	533	294	235	293	27.97
5	125	122	87	86	49.29	659	358	298	358	40.76
6	152	151	124	123	72.99	780	414	355	413	55.74
7	171	168	151	150	97.23	902	476	416	476	72.82
8	225	223	225	223	129.15	1,022	539	479	537	93.01
9	321	320	321	320	171.23	1,138	602	541	602	110.43
10	602	602	602	601	255.71	1,257	661	602	657	126.30

Notes: $n = 11$, $r = m$ and $d_0 = e_0 = 60$.

6. Conclusions. The algorithm LDSSBR is very simple. However, analysis of this algorithm is not so simple as one might suppose. Analysis of LDSSBR was attempted by Chou [3], but he was unable to get polynomial space and time bounds for the algorithm LDSSBR.

The algorithm LDSSBR mainly depends on Rosser's algorithm for solving the linear Diophantine equation $a_1x_1 + \cdots + a_nx_n = c$ as described in § 4. The extended Euclidean algorithm is a special case of Rosser's algorithm with $n = 2$ and $c = gcd(a_1, \cdots, a_n)$. Detailed analyses of the Euclidean algorithm are found in [5] and [10]. No analyses of the general case with $n > 2$ have been found. Studies should be done on the general case in order to provide a better understanding of the complexity of the algorithm LDSSBR.

As observed in § 4, LDSSBR in general obtained particular solutions with smaller norms than LDSMKB did. Given a basis and a particular solution, an optimal particular solution with respect to a definition of the norm of a vector can be obtained by adding an integral linear combination of the basis vectors to the particular solution. When the basis consists of only one vector, the problem is simple, since there is only one multiplier to be determined. However, when the basis consists of more than one basis vector, the problem becomes much more difficult. Algorithms for computing an optimal particular solution, as well as the complexities of such algorithms, can be investigated. If no polynomial time bounded algorithms can be found, algorithms for computing nearly optimal particular solutions could be sought.

Appendix. Consider

$$A = \begin{bmatrix} -7 & -18 & -1 & -8 & 4 \\ -20 & -11 & 7 & 29 & -5 \\ -15 & 19 & -27 & -17 & 21 \\ -4 & 14 & 16 & -11 & -18 \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} -24 \\ -6 \\ 21 \\ 8 \end{bmatrix}$$

as a randomly generated 4×5 matrix and 4-vector whose entries are five bits or less in length.

The matrix C and the vector B at the end of the kth iteration of the major loop of LDSSBR are shown below. The first five columns are the matrix C. The last column is the vector B.

$k=0$:

−7	−18	−1	−8	4	24
−20	−11	7	29	−5	6
−15	19	−27	−17	21	−21
−4	14	16	−11	−18	−8
1	0	0	0	0	0
0	1	0	0	0	0
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0

$k=1$:

1	0	0	0	0	0
−44	83	42	−19	14	−69
47	−107	25	−25	−20	30
58	−4	−23	47	−98	60
−1	0	−1	0	2	0
1	−1	0	0	−1	2
0	2	−1	0	0	0
−2	2	1	−1	2	−2
−1	0	0	−2	−3	−1

$k=2$:

1	0	0	0	0	0
−44	1	0	0	0	0
47	72	−653	2	83	−57
58	−313	239	−293	564	154
−1	5	7	6	−13	−2
1	−2	−4	−3	6	2
0	−3	15	−4	3	2
−2	5	−1	5	−10	−2
−1	9	−4	6	−15	−4

$k=3$:

1	0	0	0	0	0
−44	1	0	0	0	0
47	72	1	0	0	0
58	−313	38,030	139,981	−108,167	23,616
−1	5	−775	−2,852	2,204	−481
1	−2	352	1,293	−1,000	219
0	−3	317	1,154	−896	193
−2	5	−648	−2,385	1,843	−402
−1	9	−989	−3,654	2,819	−618

$k=4$:

1	0	0	0	0	0
−44	1	0	0	0	0
47	72	1	0	0	0
58	−313	38,030	1	0	0
−1	5	−775	−8,612	25,840	−6,129
1	−2	352	40,350	−121,069	28,718
0	−3	317	199,388	−598,258	141,903
−2	5	−648	−2,229	6,688	−1,586
−1	9	−989	211,893	−635,779	150,803

$\{(25{,}840, -121{,}069, -598{,}258, 6{,}688, -635{,}779)^T\}$ is the basis of the solution module of $Ax=0$. $(-6{,}129, 28{,}718, 141{,}903, -1{,}586, 150{,}803)^T$ is the particular solution of $Ax=b$.

The matrix C and the vector B at the end of the kth iteration of the major loop of LDSMKB are shown below.

$$k=0: \quad \begin{array}{rrrrrr}
-7 & -18 & -1 & -8 & 4 & 24 \\
-20 & -11 & 7 & 29 & -5 & 6 \\
-15 & 19 & -27 & -17 & 21 & -21 \\
-4 & 14 & 16 & -11 & -18 & -8 \\
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0
\end{array}$$

$$k=1: \quad \begin{array}{rrrrrr}
1 & 0 & -1 & -8 & 4 & 24 \\
-78 & -283 & 7 & 29 & -5 & 6 \\
-113 & -403 & -27 & -17 & 21 & -21 \\
-48 & -170 & 16 & -11 & -18 & -8 \\
5 & 18 & 0 & 0 & 0 & 0 \\
-2 & -7 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0
\end{array}$$

$$k=2: \quad \begin{array}{rrrrrr}
1 & 0 & 0 & -8 & 4 & 24 \\
0 & 1 & 0 & 29 & -5 & 6 \\
1{,}126 & 157 & 11{,}007 & -17 & 21 & -21 \\
-310 & -42 & -3{,}014 & -11 & -18 & -8 \\
-14 & -2 & -137 & 0 & 0 & 0 \\
7 & 1 & 69 & 0 & 0 & 0 \\
-29 & -4 & -283 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0
\end{array}$$

$$k=3: \quad \begin{array}{rrrrrr}
1 & 0 & 0 & 0 & 4 & 24 \\
0 & 1 & 0 & 0 & -5 & 6 \\
0 & 0 & 1 & 0 & 21 & -21 \\
457{,}006 & 3{,}871 & -133{,}660 & -635{,}779 & -18 & -8 \\
-9{,}796 & -83 & 2{,}865 & 13{,}628 & 0 & 0 \\
6{,}493 & 55 & -1{,}899 & -9{,}033 & 0 & 0 \\
14{,}993 & 127 & -4{,}385 & -20{,}858 & 0 & 0 \\
-7{,}912 & -67 & 2{,}314 & 11{,}007 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0
\end{array}$$

$$k=4: \quad \begin{array}{rrrrrr}
1 & 0 & 0 & 0 & 0 & 24 \\
0 & 1 & 0 & 0 & 0 & 6 \\
0 & 0 & 1 & 0 & 0 & -21 \\
0 & 0 & 0 & 1 & 0 & -8 \\
9{,}636 & 3{,}369 & -8{,}415 & -8{,}612 & 25{,}840 & 0 \\
-45{,}148 & -15{,}785 & 39{,}427 & 40{,}350 & -121{,}069 & 0 \\
-223{,}097 & -78{,}001 & 194{,}827 & 199{,}388 & -598{,}258 & 0 \\
2{,}494 & 872 & -2{,}178 & -2{,}229 & 6{,}688 & 0 \\
-237{,}089 & -82{,}893 & 207{,}046 & 211{,}893 & -635{,}779 & 0
\end{array}$$

The basis of the solution module of $Ax=0$ is $\{(25{,}840, -121{,}069, -598{,}258, 6{,}688, -635{,}779)^T\}$. The particular solution $(-497{,}089, 2{,}329{,}029, 11{,}508{,}805, -128{,}658, 12{,}230{,}604)^T$ is obtained from eliminating the first four elements of B.

REFERENCES

[1] W. A. BLANKINSHIP, *Algorithm* 288, *solution of simultaneous linear Diophantine equations* [F4], Comm. ACM, 9 (1966), pp. 514.

[2] G. H. BRADLEY, *Algorithms for Hermite and Smith normal matrices and linear Diophantine equations*, Math. Comp., 25 (1971), pp. 897–907.

[3] T.-W. J. CHOU, *Algorithms for the solution of systems of linear Diophantine equations*, Ph.D. thesis, Computer Science Dept., Univ. of Wisconsin, Madison, 1979.

[4] G. E. COLLINS, *The calculation of multivariate polynomial resultants*, J. Assoc. Comput. Mach., 18 (1971), pp. 515–532.

[5] ———, *The computing time of the Euclidean algorithm*, this Journal, 3 (1974), pp. 1–10.

[6] ———, *The* SAC-2 *Manual*, version 1.

[7] M. A. FRUMKIN, *An application of modular arithmetic to the construction of algorithms for solving systems of linear equations*, Soviet Math. Dokl., 17 (1976), pp. 1165–1168.

[8] F. R. GANTMACHER, *The Theory of Matrices*, Vol. 1, Chelsea, New York, 1959.

[9] R. KANNAN AND A. BACHEM, *Polynomial algorithms for computing the Smith and Hermite normal forms of an integer matrix*, this Journal, 8 (1979), pp. 499–507.

[10] D. E. KNUTH, *The Art of Computer Programming, Vol.* 2: *Seminumerical Algorithms*, Addison-Wesley, Reading, MA, 1969.

[11] ———, *The Art of Computer Programming, Vol.* 1: *Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1973.

[12] R. G. K. LOOS, *The algorithm description language ALDES* (*report*), SIGSAM Bull., 10 (1976), pp. 14–38.

[13] M. T. MCCLELLAN, *The exact solution of systems of linear equations with polynomial coefficients*, J. Assoc. Comput. Mach., 20 (1973), pp. 563–588.

[14] J. B. ROSSER, *A note on the linear Diophantine equation*, Amer. Math. Monthly, 48 (1941), pp. 662–666.

[15] ———, *A method of computing exact inverses of matrices with integer coefficients*, J. Res. Nat. Bur. Standards, 49 (1952), pp. 349–358.

SIAM J. COMPUT.
Vol. 11, No. 4, November 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1104-0007 $01.00/0

SCHEDULING THE OPEN SHOP TO MINIMIZE MEAN FLOW TIME*

JAMES O. ACHUGBUE† AND FRANCIS Y. CHIN‡

Abstract. It is shown that the problem of scheduling a two-processor n-job open shop nonpreemptively in order to minimize mean flow time is NP-complete even if input length is measured by the sum of the task lengths. The proof is similar in approach to that used by Garey, Johnson and Sethi to show NP-completeness of the two-processor flow shop mean flow problem. We assume previous results from their paper where possible and concentrate on those elements of the proof that are distinct from theirs.

In addition, bounds are derived for the mean flow times of arbitrary and shortest processing time (SPT) first schedules for m-processor n-job systems in terms of the mean flow time of an optimal schedule.

Key words. scheduling, open shop, mean flow time, optimal, NP-complete, approximate solution, multi-processors

1. Introduction. An *open shop* consists of $m \geqq 1$ processors each of which performs a different task, and $n \geqq 1$ jobs each consisting of m tasks. Task j of job i is to be performed on processor j. Let $S(a)$ and $F(a)$ be the *start* and *finish* (or *flow*) times of job (or task) a. Then a schedule for the shop is given by specifying for each processor, the start $S(a)$ and the finish time $F(a)$ of each task, a, to be processed on it. It is necessary that no processor be assigned more than one task and no job be assigned to more than one processor at any time. We shall consider only *nonpreemptive* schedules in which a task is not interrupted once its execution has begun. The *finish time* of a schedule (or *schedule length*) is maximum $\{F(a)|$ for all tasks, $a\}$, and the *mean flow time* is defined as the summation of $F(a)$, over all jobs, a.

The open shop is similar to the processor bound systems studied in [1], [2], [10], the flow shop [1], [3], [4], [5], [6], [7], [11], [13] and others and the job shop [5], [7], [11]. The common assumption in each of these is that a processor performs a specific type of task and hence tasks can only be executed on the specified processors. The only difference between the open shop and the flow and job shops is that in an open shop no restrictions are placed on the order in which the tasks of any job are to be processed.

The problem of finding minimal length schedules for the open shop has been studied by Gonzalez and Sahni [12] and Gonzalez [9]. In [12], a linear algorithm is presented for the 2-processor preemptive and nonpreemptive systems, and for $m \geqq 3$ an efficient algorithm is given for preemptive schedules, while the nonpreemptive problem is shown to be NP-complete. In [9], Gonzalez presents a faster algorithm for the preemptive case when $m \geqq 3$.

In [8], Gonzalez shows the minimal mean flow time problem for an m-processor flow shop, arbitrary m, to be NP-complete. When $m = 1$ the problem reduces to that of scheduling n independent tasks on one processor to minimize mean flow time. A well-known solution to this problem reported by Smith [14] is to schedule the tasks in order of nondecreasing execution time. We will show in § 2 that the problem is NP-complete even for only two processors. Thus, our result is not covered by [8] which assumes an arbitrary number of processors. Our approach is similar to that

* Received by the editors September 4, 1979, and in final revised form November 25, 1981.

† Department of Mathematical and Computer Sciences, Michigan Technological University, Houghton, Michigan 49931.

‡ Department of Computing Science, The University of Alberta, Edmonton, Alberta, Canada T6G 2H1.

applied in [7] to show NP-completeness of the 2-processor flow shop mean flow problem but there are considerable differences (see § 2). The reduction is from 3-PARTITION.

3-PARTITION. Given positive integers n, B, and a set A of integers $\{a_1, \cdots, a_{3n}\}$ with $\sum_{i=1}^{3n}(a_i) = nB$ and $B/4 < a_i < B/2$, $1 \leqq i \leqq 3n$, does there exist a partition of A into 3-element sets such that the sum of the three elements of each partition is B?

3-PARTITION is known [7] to be NP-complete even when the input length is measured by the sum of the a_i. Given an instance of 3-PARTITION it will be sufficient for our purposes to construct in time polynomial in nB a 2-processor open shop problem with a bound D such that there exists a schedule for the open shop with mean flow time not exceeding D if and only if the 3-PARTITION problem has a solution.

In § 3, we derive tight bounds on the mean flow time of an arbitrary schedule and an SPT schedule as compared to the optimal mean flow time.

2. Complexity of mean flow schedules. We now show that the 2-processor problem is NP-complete. We must stress that although our method is similar to that of [7], the similarity is only in approach. The proof in [7] for the flow shop problem does not apply to the open shop. The fact that tasks composing a job in a flow shop must be executed in a fixed order simplifies that proof a great deal. Since we do not have a fixed order of execution for tasks in a particular job, more care is needed in setting up our task system and our proof is more complex. We will concentrate on those elements of the proof that are distinct from that of [7].

The open shop to be constructed consists of a large number of jobs but their number and the sum of their lengths will be polynomial in nB. We use four types of jobs, the T, U, X and Y-jobs. U-jobs are further divided into V and W-jobs.

For convenience in specifying the different types of jobs, a job will be given as an ordered pair (a, b) where a is the time taken by the job on processor 1 and b is the time taken on processor 2. Also, following the notation in [7], the ith task of job J, will be denoted by $J[i]$.

There are $n+1$ T-jobs, $T_0 = (0, g)$ and $T_i = (t, g)$, $1 \leqq i \leqq n$, where t is a much larger integer than g. In the system under construction, we will ensure that the best schedules (with smallest mean flow time) execute the T-task on processor 1 before the corresponding T-task on processor 2. Thus, in the best schedules, the positioning of the T-jobs alone will leave n slots of size $t-g$ (see Fig. 1) which will be used to test if the 3-PARTITION problem has a solution.

t | t | t | t | x | x | ...
g | g | g | g | g | y | y | ...

FIG. 1. *Schedule when A has a 3-partition, $n = 4$. X, Y and T jobs indicated. Each hatched area takes u jobs, 3 of type W with total time $3v + B$ and u-3 of type V.*

The slots to be created by the T-jobs will be filled by the U-jobs consisting of $V_{i,j} = (0, v)$, $1 \leqq i \leqq n$, $1 \leqq j \leqq (u-3)$, and $W_i = (0, v + a_i)$, $1 \leqq i \leqq 3n$. The U-jobs have no tasks on the first processor. The different parameters of the system will be so balanced that each slot on the second processor must be occupied by 3 W-tasks and $(u-3)$ V-tasks, the a_i element in each of the 3 W-tasks thus forming a partition of the 3-PARTITION solution.

Now in order to produce the slots as indicated, the T-tasks on processor 1 must be processed in the first nt time units. To ensure that this is indeed the case, we assign

$X_i = (x, 0)$, where x is very large compared to the other task times considered so far. The effect is that delaying the execution of the X-tasks for even one unit after time nt will force total flow to exceed the given bound and similarly, for the Y-tasks on the second processor.

Note that it is necessary to show that for each T-job, the task on the second processor will be done after that on the first processor and it is not yet clear from the above that this will be the case. This problem requires very careful balancing of the task time variables and in order to solve it we had to use different task assignments from those of [7]. We achieve the required results by having g barely larger than u (in [7] $u \gg g = 1$) and using a tighter lower bound for the flow times of the U-tasks than that given in [7].

THEOREM 1. *The 2-processor open shop nonpreemptive mean flow scheduling problem is* NP-*complete, even if input length is measured by the sum of the task lengths.*

Proof. Given a 3-PARTITION problem with n, B and the set A, consider the 2-processor flow shop specified in Fig. 2.

T-jobs: $T_0 = (0, g)$, $\quad T_i = (t, g)$, $1 \leqq i \leqq n$,
X-jobs: $X_i = (x, 0)$, $\quad 1 \leqq i \leqq h$,
Y-jobs: $Y_i = (0, y)$, $\quad 1 \leqq i \leqq h$,
U-jobs: $V_{i,j} = (0, v)$, $\quad 1 \leqq i \leqq n$, $1 \leqq j \leqq (u-3)$
and $W_i = (0, v + a_i)$, $\quad 1 \leqq i \leqq 3n$,

The bound D is given as $D = X_e + Y_e + T_e + U_e$, with

$$X_e = \sum_{i=1}^{h} (nt + ix),$$

$$Y_e = \sum_{i=1}^{h} (nt + g + iy),$$

$$T_e = \sum_{i=0}^{n} (g + it), \text{ and}$$

$$U_e = 3nB + \sum_{i=0}^{n-1} \left(\sum_{j=1}^{u} (g + it + jv) \right); \text{ where}$$

$u = 3nB + 3n + 1$, $\quad g = 3nB + 3n + 6$,
$v = n(B+1)g$, $\quad t = uv + g + B$,
$x = 2(n+h)t$, $\quad y = x$,
$s = (n+1)g + 3nB + nug + n(n-1)u(g+B)/2$ and
$h = s + 1$.

FIG. 2. *Open shop specifications.*

Suppose A has a 3-partition. We use the schedule suggested in Fig. 1. We schedule, on the first processor, tasks $T_i[1]$, $1 \leqq i \leqq n$, followed by $X_i[1]$, $1 \leqq i \leqq h$, and on the second processor, tasks $T_i[2]$, $0 \leqq i \leqq n$, followed by $Y_i[2]$, $1 \leqq i \leqq h$. This yields the template of Fig. 1. In each of the n hatched areas on the second processor we place $(u-3)$ V-type tasks followed by the three W-tasks corresponding to the three elements of one of the 3-element partitions. Clearly, the X, Y and T-jobs contribute X_e, Y_e and T_e to the mean flow time. Since there are exactly three W-type tasks with sum $(3v + B)$ in each hatched area, the contribution of the ith hatched area, $0 \leqq i \leqq (n-1)$, is less than $(3B + \sum_{j=1}^{u} (g + it + jv))$. (Note that the three W-tasks will be the last to be executed in each hatched area.) Hence the U-jobs' contribution to the mean flow

time is less than U_e, and the bound D is not exceeded by the mean flow time of the schedule.

Now, suppose there exists a schedule with mean flow time not exceeding the bound D. For brevity we shall refer to such a schedule as a *good* schedule. We complete the proof of the theorem by showing that the desired partition of A must exist.

We first show by proving the following claims that if there is a good schedule, then it can be reduced to a good schedule with the structure of Fig. 1. (Note that $S(a)$, $F(a)$ are defined in the introduction.)

(XY1) In a good schedule, $S(X_i[1]) \geqq nt$ and $S(Y_i[2]) \geqq (nt+g)$, $1 \leqq i \leqq h$. Hence, all $T[1]$-tasks are executed before any $X[1]$-tasks and all $T[2]$- and $U[2]$-tasks are executed before any $Y[2]$-task. Without loss of generality, we may assume that the $X[1]$-, $Y[2]$- and T-tasks are executed in increasing order of their index i.

(XY2) In a good schedule, $S(X_1[1]) = nt$ and $S(Y_1[2]) = nt+g$. In addition, $F(T_i[1]) = ti$, $0 \leqq i \leqq n$.

(UT1) A good schedule can be reduced in polynomial time to one in which $F(T_i[1]) \leqq S(T_i[2])$, $0 \leqq i \leqq n$.

(UT2) A good schedule can be reduced in polynomial time to one in which $S(T_i[2]) - F(T_i[1]) < v + B/2$, $0 \leqq i \leqq n$.

(UT3) In a good schedule satisfying all previous claims, at most $(u+1)$ and at least $(u-1)$ U-tasks are executed between $T_i[2]$ and $T_{i+1}[2]$, $0 \leqq i < n$.

(UT4) In a good schedule satisfying all previous claims, EXACTLY u U-tasks are executed between $T_i[2]$ and $T_{i+1}[2]$, $0 \leqq i < n$.

(UT5) In a good schedule satisfying all previous claims, $S(T_i[2]) = ti$, $0 \leqq i \leqq n$.

Claims XY1 and XY2 establish the fact that the $T[1]$-tasks are done in the first nt time units and are immediately followed by the $X[1]$-tasks. In addition, the $Y[2]$-tasks are the final tasks to be done on processor 2 and they must start precisely at time $(nt+g)$, thus leaving no idle time on that processor.

Claim UT1 says that the T-jobs are executed first on processor 1 and then on processor 2.

Claims UT2, UT3 and UT4 correspond roughly to [7, Claim U1]. Essentially, what we need is to show that task $T_i[2]$ is preceded by exactly iu U-tasks on the second processor. However, our choice of task lengths necessitated by Claim UT1 does not allow the straightforward arguments of [7, Claim U1] to work here. For clarity, we present our proof in three simple stages.

Finally, Claim UT5 ensures that there is no delay between the executions of the component tasks of each T-job.

Now, a good schedule which satisfies the above claims is similar in structure to Fig. 1. Given such a schedule, a partition of A into 3-element sets A_i, $1 \leqq i \leqq n$, is obtained by setting $A_i = \{a_k | S(T_{i-1}[2]) < S(W_k[2]) < S(T_i[2])\}$. By Claim UT4, we know that there are u U-tasks, $U_i[2]$, with $S(T_{i-1}[2]) < S(U_i[2]) < S(T_i[2])$. Since the $V[2]$-type tasks have length v and the $W[2]$-tasks have length less than $(v+B/2)$ and greater than $(v+B/4)$ and these u tasks cover an interval of $t-g = uv+B$ without leaving an idle period, the u tasks must contain exactly three $W[2]$-tasks with total length $3v+B$. Hence, the open shop has a schedule with mean flow time not exceeding the deadline D if and only if the set A has the required partition.

In proving the claims, their effects are taken to be cumulative. In other words, when proving any claim it is assumed that we have at hand a good schedule which satisfies the previously proven claims. Furthermore, bear in mind that for any schedule, A, there exists a schedule, B, with mean flow time of B not exceeding that of A, in

which $S(a)$, $F(a)$ are integers for any task, a. This is obvious since all task times are integers. Thus, in the following we consider $S(a)$ and $F(a)$ to be integers.

We omit the proofs of Claims XY1, XY2, which are straightforward and similar to [7, Claims X2, X3]. The key point here is that the variables x, y are sufficiently large to ensure that the X- and Y-jobs are done last and the start of their execution cannot be delayed by even one unit beyond a certain point. Complete details of these proofs are given in [1].

Claim UT1. A good schedule can be reduced in polynomial time to one in which $F(T_i[1]) \leqq S(T_i[2])$, $0 \leqq i \leqq n$.

Given a good schedule, we construct another with the stated property in polynomial time. Recall that as a consequence of the XY-Claims we have a good schedule in which $F(T_i[1]) = ti$, $0 \leqq i \leqq n$.

We scan the schedule from right to left and perform the following operation, OP, whenever we encounter a $T_i[2]$ with $F(T_i[2]) \leqq S(T_i[1])$.

(OP) Find a V-task, V_c, such that $(ti+g) \leqq F(V_c) < (ti+v+g)$. Remove $T_i[2]$ from the schedule, shift the following tasks up to and including V_c to the left to take up the interval vacated by $T_i[2]$, and insert $T_i[2]$ after V_c. If among the shifted tasks a $T_j[2]$ conflicts with $T_j[1]$, exchange the $T_j[2]$ with the following task.

We must prove the existence of V_c and show that OP does not increase mean flow time. Consider the interval $[ti-v-B/2, t(i+1))$. If there is any task of length g in this interval, it must be a $T_j[2]$ task where $j<i$. Thus we can exchange it with its predecessor without generating conflicts or increasing flow time. This may be repeated until the task executes before time $ti-v-B/2$. Thus we can ensure that no task of length g executes in the time interval $[ti-v-B/2, t(i+1))$ (see Fig. 3). Now the

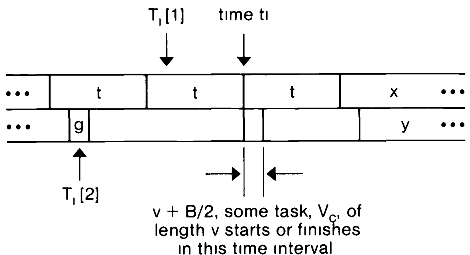


FIG. 3

interval $[ti-v-B/2, t(i+1))$ has length $t+v+B/2$. Even if all the W-tasks were in this interval, their total length is less than $t-3v$ and hence there is still room for at least four V-tasks in the interval. Since the V-tasks are smaller than any W-task, they will be the first to be done in the interval. Thus, we conclude that within $[ti-v-B/2, ti+v+B/2)$ there are only V-tasks or parts of V-tasks, one of which will be identified as task V_c.

Now, consider the first task $V_j[2]$ to finish after time ti. By above arguments this task and the one following it must be V-tasks. $V_j[2]$ finishes before $(ti+v+B/2)$ (which is less than $(ti+v+g)$). If $F(V_j[2])-ti \geqq g$, then choose $V_c = V_j[2]$, otherwise choose the task following it as V_c. Hence, $F(V_c)-ti < v+g$. In either case, $ti+g \leqq F(V_c) < ti+v+g$.

We proceed to remove $T_i[2]$ from the schedule, shift the following tasks, say J in number, up to and including V_c to the left by a distance of g and insert $T_i[2]$ after V_c. Since $F(V_c) \geqq ti + g$, after performing OP there can be no conflict between $T_i[2]$ and $T_i[1]$. However, we may have to resolve some conflicts between some $T_j[2]$ and $T_j[1]$ where the $T_j[2]$ is among the tasks shifted left. We do this by exchanging the $T_j[2]$ with the following task. (It is clear that the following task is a V-task by the same line of argument that was used to determine V_c; the idea is that following $T_j[2]$ is some interval containing only U-jobs, which is sufficiently large that V-tasks must be included and they will be the first executed in the interval to reduce mean flow time.)

Now, among the shifted tasks there will be at most one $T[2]$ task for every sub-interval $[t(j-1), tj]$ of the interval originally covered by the tasks. The same interval must contain at least u U-tasks as well for the following reason. Recall that there is no idle time on processor 2 during the first $(nt+g)$ time units. The number of U-tasks executed in the interval is at least $J = (t-g)/(v+B/2)$ since every task executed before $(nt+g)$ on processor 2 has length not exceeding $(v+B/2)$. Now

$$\begin{aligned} v = n(B+1)g > nBg > nBu > uB/2 \\ \Rightarrow B > uB/2 - v - B/2 \\ \Rightarrow uv + B > (u-1)(v+B/2) \\ \text{or } t-g > (u-1)(v+B/2) \\ \text{or } J > u-1. \end{aligned}$$

Hence, there are at most $\lfloor J/u \rfloor$ $T[2]$-tasks among the shifted tasks.

We are now ready to compute the effects of OP on the mean flow time of the schedule. The flow time of job T_i increases by at most $(v+g)$. The J tasks shifted to the left lose at least Jg flow time. In addition, the intermediate $T[2]$-tasks may gain at most $\lfloor J/u \rfloor v$ in exchanges to resolve conflicts. Thus, we must show $Jg \geqq v + g\lfloor J/u \rfloor v$, where $J \geqq u$. (Since the J tasks cover at least the interval $[t(i-1), ti]$, J must be no less than u.) Let $J = ku + d$, $k \geqq 1$ and $0 \leqq d < u$.

$$\begin{aligned} Jg \geqq v + g + \lfloor J/u \rfloor v \\ \Leftrightarrow (ku+d)g \geqq v + g + kv \\ \Leftrightarrow 3nkB + 3nk + k + d \geqq nkB + nk + nB + n + 1, \qquad k \geqq 1, \end{aligned}$$

which is easily seen to be true. There are no other changes to the mean flow time. Hence, a good schedule can be reduced to one satisfying the claim. □

Claim UT2. A good schedule can be reduced in polynomial time to one for which $S(T_i[2]) - F(T_i[1]) < v + B/2$, $0 \leqq i \leqq n$.

Consider a good schedule satisfying all previous claims. $S(T_i[2]) - F(T_i[1]) \leqq t - g$. Furthermore, in the interval $(ti, ti+t)$ there is no task of size g on processor 2 other than $T_i[2]$. If $S(T_i[2]) - F(T_i[1]) > v + B/2$, then the task preceding $T_i[2]$, a $U[2]$-task, can be exchanged with $T_i[2]$ with no increase in mean flow time, and no conflict in the execution of job T_i since that task must start after $ti = F(T_i[1])$. □

Claim UT3. In a good schedule satisfying all previous claims, at most $(u+1)$ and at least $(u-1)$ U-tasks execute between $T_i[2]$ and $T_{i+1}[2]$, $0 \leqq i < n$.

By Claim UT2, the interval between $T_i[2]$ and $T_{i+1}[2]$ is no smaller than $(t - g - v - B/2)$ and no larger than $(t - g + v + B/2)$.

Suppose there are fewer than $(u-1)$ U-tasks in the interval. Their total length is at most $nB+(u-2)v$. Now

$$\begin{aligned} v=n(B+1)g&>nB-B/2\\ &\Rightarrow uv+B/2-v>nB+uv-2v\\ &\Rightarrow t-g-v-B/2>nB+(u-2)v \quad \text{(by definition of } t\text{)}. \end{aligned}$$

Thus, there will be idle time on processor 2 and by Claim XY2 the schedule cannot be a good one.

Similarly, suppose there are more than $(u+1)$ U-tasks. Their total length is at least $(u+2)v$. But

$$\begin{aligned} v&>3B/2\\ &\Rightarrow uv+2v>uv+v+3B/2\\ &\Rightarrow (u+2)v>t-g+v+B/2 \quad \text{(by definition of } t\text{)}. \end{aligned}$$

Hence, at most $(u+1)$ U-tasks can be present in the interval. □

Claim UT4. In a good schedule satisfying all previous claims, EXACTLY u U-tasks are executed between $T_i[2]$ and $T_{i+1}[2]$, $0\leqq i\leqq n$.

To begin with, let us consider in what ways this claim can be violated. If there are less than iu U-tasks preceding $T_i[2]$ then the total length of tasks preceding $T_i[2]$ (including $T_j[2]$, $j<i$) on processor 2 is at most $(ig+(iu-1)v+nB)$. This is less than the time interval ti which it has to cover as

$$ig+(iu-1)v+nB<it=i(uv+g+B).$$

Thus, we must have at least iu U-tasks preceding $T_i[2]$.

Now, let k_i, $0\leqq i\leqq(n-1)$, be the number of U-tasks in the interval between $T_i[2]$ and $T_{i+1}[2]$. Suppose $k_p=u+1=k_q$, and $k_i=u$ for $p<i<q$. Then, there are $((q+1-p)u+2)$ U-tasks between $T_p[2]$ and $T_{q+1}[2]$ with total length at least $((q+1-p)u+2)v$. Together with the tasks $T_i[2]$, $p\leqq i\leqq q$, we get total length at least $((q+1-p)u+2)v+(q+1-p)g$ which is greater than $(q+1-p)t+v+B/2$. Since the latter value expresses the maximum time interval available for the tasks, as implied by Claim UT2, this is not possible.

Collecting these facts together, namely, that $k_i=(u-1)$, u or $(u+1)$, that $\sum_{j=1}^{i}(k_j)\geqq iu$, and that it is not possible to have $k_p=k_q=(u+1)$ and $k_i=u$ for $p<i<q$, we can conclude that the $(u+1)$ and $(u-1)$ values of k_i occur alternately starting with a $(u+1)$ and finishing with a $(u-1)$, with the u values interspersed.

We now show that the bound D is exceeded if k_1 is not equal to u for $0\leqq i\leqq(n-1)$. A new lower bound U_c for the flow of the U-jobs can now be computed as

$$U_c=\sum_{i=0}^{n-1}\left(\sum_{j=1}^{k_i}(g+jv+ti+E_i)\right).$$

The term $E_i\geqq 0$ will compensate for the fact that the $T_i[2]$ cannot start early enough after a $k_p=u+1$ and before the following $k_q=u-1$.

$$U_c-U_e=-3nB+\sum_{i=0}^{n-1}\left(\sum_{j=1}^{k_i}(g+jv+ti+E_i)-\sum_{j=1}^{u}(g+jv+ti)\right).$$

With respect to the sequence k_i, outside of those subsequences starting with a $(u+1)$ and ending with a $(u-1)$, the E_i term is zero and the corresponding summations in

the above cancel out. However, we show that if there is at least one subsequence $(u+1), u, \cdots, u, (u-1)$ then $U_c + T_c - U_e - T_e > 0$, where T_c is a corresponding bound for the T-jobs.

Let $k_p, \cdots, k_q$ be one such subsequence. Task $T_i[2]$, $p < i \leqq q$, cannot start until after

$$pt + (i-p)g + ((i-p)u+1)v = ti + v - (i-p)B.$$

Thus $E_p = 0$ and $E_i = v - (i-p)B$, $p < i \leqq q$.

In the expression for $(U_c - U_e)$ the summations corresponding to the subsequence are

$$\sum_{j=1}^{u+1} (g + jv + pt) - \sum_{j=1}^{u} (g + jv + pt) + \sum_{i=p+1}^{q-1} \left(\sum_{j=1}^{u} (v - (i-p)B) \right)$$
$$+ \sum_{j=1}^{u-1} (g + qt + jv + v - (q-p)B) - \sum_{j=1}^{u} (g + qt + jv).$$

After simplification, this becomes

$$-(q-p)(g+uB) - \sum_{i=p+1}^{q-1} (u(i-p)B),$$

where the summation in the second term is zero if $q = p+1$. However, on computing the lower bound T_c for the T-jobs, again the values to be summed outside the subsequences are the same as in the summation for T_e. For the subsequence above, we get

$$\sum_{i=p+1}^{q} (g + ti + v - (i-p)B) - \sum_{i=p+1}^{q} (g + ti)$$
$$= v - (q-p)B + \sum_{i=p+1}^{q-1} (v - (i-p)B)$$

as the difference in the two summations. Since $v - (q-p)B > (q-p)(g+uB) + 3nB$ and $v - (i-p)B > u(i-p)B$, $p < i < q$, the overall bound is strictly greater than D. Note that we have taken care of the $3nB$ term in the definition of U_e, while the bounds for X and Y-tasks remain X_e and Y_e.

Thus if there is at least one subsequence of the type described the flow time is strictly greater than D. Hence, $k_i = u$ for $0 \leqq i \leqq (n-1)$. $\square$

Claim UT5. In a good schedule satisfying the previous claims, $S(T_i[2]) = ti$, $0 \leqq i \leqq n$.

Given a good schedule which satisfies the previous claims, assume that $S(T_i[2]) > ti$ for some i, $0 \leqq i \leqq n$. Then the following lower bounds for the jobs' flow time are easily determined; X_e for the X-jobs, Y_e for the Y-jobs, $(T_e + 1)$ for the T-jobs, and finally at least $(U_e + u - 3nB)$ for the U-jobs (the u term is introduced by the delay on the u U-jobs between $T_i[2]$ and $T_{i+1}[2]$). Adding these together gives a bound of $(D + u + 1 - 3nB)$ which is greater than D. Hence, the claim must be true.

This concludes the proof of the theorem. $\square$

3. Heuristic solutions. We have seen that the 2-processor n-job mean flow time scheduling problem for the open shop is NP-complete. In this section, we derive tight bounds on the mean flow time of an arbitrary schedule and of a schedule obtained using the shortest processing time (SPT) first heuristic as compared to the mean flow

time of an optimal schedule. Recall that the SPT heuristic is optimal for the 1-processor case.

Let the n jobs be J_i, $1 \leqq i \leqq n$. The jth task of the ith job is $J_i[j]$ and has execution time $t_i[j]$. Let $T_i = \sum_{j=1}^{m} (t_i[j])$ and $T = \sum_{i=1}^{n} (T_i)$. T_i is the total processing time for the ith job, while T is the total time for all the n jobs. For any schedule Q the mean flow time of Q will be denoted by mft(Q). We continue to use $S(a)$ and $F(a)$ for the start and finish time of task/job a. (If Q is subscripted, S and F may be given the same subscript to indicate clearly that we are considering start and finish times with respect to the particular schedule Q.)

THEOREM 2. *Let Q_0 be an optimal mean flow schedule for an m-processor open shop with n jobs. Let Q be an arbitrary schedule. Then* $(\text{mft}(Q)/\text{mft}(Q_0)) \leqq n$.

Proof. Without loss of generality, we may assume that the jobs are completed in the order $J_1, J_2, \cdots, J_n$. In the worst case, no task of job J_i is started before J_{i-1} has been completed. Hence,

$$F(J_i) \leqq \sum_{k=1}^{i} (T_k).$$

Now,

$$\text{mft}\,(Q) = \sum_{i=1}^{n} (F(J_i)) \leqq \sum_{i=1}^{n} \left(\sum_{k=1}^{i} (T_k) \right)$$

$$= \sum_{i=1}^{n} (n+1-i)T_i \leqq nT,$$

$$\text{mft}\,(Q_0) = \sum_{i=1}^{n} (F_0(J_i)) \geqq \sum_{i=1}^{n} (T_i) = T \geqq \text{mft}\,(Q)/n.$$

The bound given above is asymptotically tight as illustrated by the following example.

Example 1. $m = 2$. The jobs are $J_1, J_2, \cdots, J_{n+1}$, where

$$t_i[1] = t_i[2] = 1, \qquad 1 \leqq i \leqq n;$$

$$t_{n+1}[1] = x, \qquad t_{n+1}[2] = 1, \qquad x > n.$$

The schedules Q and Q_0 are given in Fig. 4.

$$\text{mft}\,(Q) = (x+1) + \sum_{i=1}^{n} (x+i)$$

$$= (n+1)x + n(n+1)/2 + 1,$$

$$\text{mft}\,(Q_0) = \sum_{i=1}^{n-1} (i+1) + n + x + (n+1)$$

$$= x + \frac{n(n+5)}{2}.$$

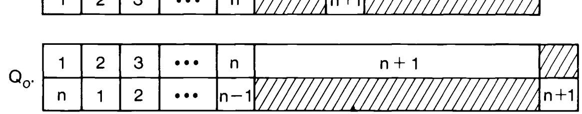


FIG. 4. *Schedules for Example* 1. *Tasks for ith job are indicated by i.*

Therefore,

$$\frac{\text{mft}(Q)}{\text{mft}(Q_0)}=\frac{(n+1)x+n(n+1)/2+1}{x+n(n+5)/2},$$

which approaches $n+1$ as x approaches infinity.

We now consider the case for the SPT heuristic, in which jobs are processed in order of nondecreasing processing time. The rule is normally implemented as follows: Suppose that the jth processor is available, and jobs J_i and J_k have no tasks currently under execution and their tasks for the jth processor have not yet been executed, then $J_i[j]$ is chosen to execute before $J_k[j]$ if $T_i \leqq T_k$.

THEOREM 3. *Let Q_o be an optimal mean flow time schedule for an m-processor n-job flow shop. Let Q_s be a schedule constructed with the* SPT *heuristic. Then,* $(\text{mft}(Q_s)/\text{mft}(Q_o)) \leqq m$.

Proof. Assume $T_1 \leqq T_2 \leqq \cdots \leqq T_n$.

$$\text{mft}(Q_s)=\sum_{i=1}^{n}(F_s(J_i)) \leqq \sum_{i=1}^{n}\left(\sum_{k=1}^{i}(T_k)\right).$$

Let $(q(1), q(2), \cdots, q(n))$, a permutation of the first n integers, be the order in which the jobs are completed in schedule Q_o. Then

$$F_o(J_{q(i)}) \geqq \sum_{k=1}^{i}\left(\frac{T_{q(k)}}{m}\right) \geqq \sum_{k=1}^{i}\left(\frac{T_k}{m}\right).$$

Therefore,

$$\text{mft}(Q_o)=\sum_{i=1}^{n}(F_o(j_{q(i)})) \geqq \sum_{i=1}^{n}\sum_{k=1}^{i}\left(\left(\frac{T_k}{m}\right)\right) \geqq \frac{\text{mft}(Q_s)}{m}. \qquad \square$$

As for the previous theorem, the bound is asymptotically tight as illustrated by Example 2.

Example 2. There are $m+1$ jobs, $J_1, \cdots, j_{m+1}$ for m processors, where

$$t_1[j]=\begin{cases}1 & \text{for } j=1,\\ 0 & \text{for } 2 \leqq j \leqq m,\end{cases}$$

$$\underset{2 \leqq i \leqq m}{t_i[j]}=\begin{cases}2 & \text{for } j=1, \text{ or } j=i,\\ 0 & \text{for } 2 \leqq j \leqq m, j \neq i,\end{cases}$$

$$t_{m+1}[j]=\begin{cases}x & \text{for } j=1\ (x>2),\\ 2 & \text{for } j=2,\\ 0 & \text{for } 3 \leqq j \leqq m.\end{cases}$$

The SPT and optimal schedules, Q_s, Q_o, are given in Fig. 5.

$$\text{mft}(Q_s)=1+(x+3)+\sum_{i=1}^{m-1}(x+1+2i)=mx+m^2+3.$$

$$\text{mft}(Q_o)=1+2m+2+x+\sum_{i=1}^{m-1}2(i+1)=x+m(m+3)+1.$$

Therefore,

$$\frac{\text{mft}(Q_s)}{\text{mft}(Q_o)}=\frac{mx+m^2+3}{x+m(m+3)+1}$$

which approaches m as x approaches infinity. $\square$

4. Conclusions. We have shown NP-completeness for the 2-processor minimal mean flow time open shop scheduling problem. We can also conclude that the problem remains NP-complete when the number of processors $m > 2$. Thus the relaxation of the constraint that each job's tasks be processed in the same processor order in the flow shop model, yielding an open shop, still leaves us with an intrinsically difficult problem.

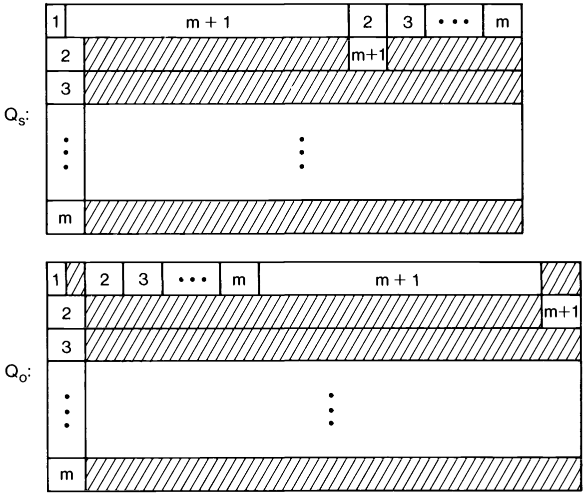


FIG. 5. *Schedules for Example* 2. *Tasks for ith job indicated by i.*

We have also derived tight bounds for the mean flow time of an arbitrary schedule and for an SPT schedule in terms of the optimal mean flow time. Since the number of jobs is usually much larger than the number of processors, the bounds indicate an advantage of SPT schedules over arbitrary schedules.

REFERENCES

[1] J. O. ACHUGBUE, *The complexity of some deterministic scheduling problems*, Ph.D. Thesis, Dept. of Computing Science, University of Alberta, Edmonton 1980.

[2] J. O. ACHUGBUE AND F. Y. CHIN, *On optimal schedules for processor bound systems*, Tech. Rep. 79-9, Dept. of Computing Science, University of Alberta, Edmonton, 1979.

[3] ———, *Complexity and solutions of some three-stage flow shop scheduling problems*, Maths Oper. Res., May (1982).

[4] F. Y. CHIN AND L. TSAI, *On J-maximal and J-minimal flow shop schedules*, J. Assoc. Comput. Mach., 28 (1981), pp. 462–476.

[5] R. W. CONWAY, W. L. MAXWELL AND L. W. MILLER, *Theory of Scheduling*, Addison-Wesley, Reading, MA 1967.

[6] E. G. COFFMAN JR., ed., *Computer and Jobshop Scheduling Theory*, John Wiley, New York, 1976.

[7] M. R. GAREY, D. S. JOHNSON AND R. SETHI, *The complexity of flow shop and jobshop scheduling*, Maths. Oper. Research 1 (1976), pp. 117–129.

[8] T. GONZALEZ, *NP-hard shop problems*, Tech. Rep. CS-79-35, Computer Science Dept., Pennsylvania State Univ., University Park, 1979.

[9] ———, *A note on open shop preemptive schedules*, Tech. Rep. 214, Computer Science Dept., Penn State Univ., Dec. 1976.

[10] D. K. GOYAL, *Scheduling processor bound systems*, Proc. Sixth Texas Conference on Computing Systems, 1977, pp. 7B-21-7B-25.

[11] T. GONZALEZ AND S. SAHNI, *Flowshop and jobshop schedules: Complexity and approximation*, Operations Research, 26 (1978), pp. 36–52.
[12] ———, *Open shop scheduling to minimize finish time*, J. ACM, 23 (1976), pp. 665–679.
[13] S. M. JOHNSON, *Optimal two- and three-stage production schedules with setup times included*, Nav. Res. Logist. Quart., 1 (1954), pp. 61–68.
[14] W. E. SMITH, *Various optimizers for single state production*, Nav. Res. Logist. Quart., 3 (1956), pp. 59–66.

SIAM J. COMPUT.
Vol. 11, No. 4, November 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1104-0008 $01.00/0

ON CONSTRUCTING MINIMUM SPANNING TREES IN k-DIMENSIONAL SPACES AND RELATED PROBLEMS*

ANDREW CHI-CHIH YAO†

Abstract. The problem of finding a minimum spanning tree connecting n points in a k-dimensional space is discussed under three common distance metrics: Euclidean, rectilinear, and L_∞. By employing a subroutine that solves the post office problem, we show that, for fixed $k \geqq 3$, such a minimum spanning tree can be found in time $O(n^{2-a(k)}(\log n)^{1-a(k)})$, where $a(k) = 2^{-(k+1)}$. The bound can be improved to $O((n \log n)^{1.8})$ for points in 3-dimensional Euclidean space. We also obtain $o(n^2)$ algorithms for finding a farthest pair in a set of n points and for other related problems.

Key words. algorithm, minimum spanning tree, nearest neighbor, post office problem

1. Introduction. Given an undirected graph with a weight assigned to each edge, a minimum spanning tree (MST) is a spanning tree whose edges have a minimum total weight among all spanning trees. The classical algorithms for finding MST were given by Dijkstra [7], Kruskal [13], Prim [14], and Sollin [4, p. 179]. It is well known (e.g., see Aho, Hopcroft and Ullman [1]) that, for a graph with n vertices, an MST can be found in $O(n^2)$ time. (All time bounds discussed in this paper are for the worst-case behavior of algorithms.) For a sparse graph with e edges and n vertices, it was shown by Yao [16] that an MST can be found in time $O(e \log \log n)$. More studies of MST algorithms can also be found in Cheriton and Tarjan [6], Kerschenbaum and Van Slyke [11].

An interesting application of MST occurs in connection with hierarchical clustering analysis in pattern recognition (see, for example, Dude and Hart [9, Ch. 6], Zahn [20]). In this application, n vertices $V = \{\tilde{v}_1, \tilde{v}_2, \cdots, \tilde{v}_n\}$ are given, each a k-tuple of numbers. The graph is understood to be a complete graph $G(V)$ on these n vertices, with the weight on each edge $\{\tilde{v}_i, \tilde{v}_j\}$ being $d(\tilde{v}_i, \tilde{v}_j)$ where d is a certain metric function computable from the components of $\tilde{v}_i$ and $\tilde{v}_j$. A simple way to find an MST in this case is to compute all the weights $d(\tilde{v}_i, \tilde{v}_j)$, and then use an $O(n^2)$ MST algorithm for general graphs. However, as there are only kn input parameters, it is interesting to find out if there are algorithms which take only $o(n^2)$ time. Several empirically good algorithms were proposed in Bentley and Friedman [2], where a list of references to other applications of finding MST in k-dimensional spaces can also be found. Shamos and Hoey [16] gave an $O(n \log n)$ algorithm for n points in the plane ($k = 2$) with Euclidean metric. No algorithm, however, is known to have a guaranteed bound of $o(n^2)$ when $k \geqq 3$.

In this paper, we consider three common metrics in k-dimensional spaces, namely, the rectilinear (L_1), the Euclidean (L_2), and the L_∞ metric. We use E_p^k ($p = 1, 2, \infty$) to denote the space of all k-tuples of real numbers with the L_p-metric, i.e., the distance between two points $\tilde{x}$ and $\tilde{y}$ is given by $d_p(\tilde{x}, \tilde{y}) = (\sum_{i=1}^k |x_i - y_i|^p)^{1/p}$. (It is agreed that $d_\infty(\tilde{x}, \tilde{y}) = \max_i |x_i - y_i|$.) We give new algorithms which construct, for a given set V of n points in E_p^k, an MST for the associated complete graph $G(V)$. The algorithms work in time $O(n^{2-a(k)}(\log n)^{1-a(k)})$, where $a(k) = 2^{-(k+1)}$ for any fixed $k \geqq 3$. Fast algorithms for related geometric problems are also given using similar techniques.

* Received by the editors November 3, 1980. This research was supported in part by National Science Foundation under grant MCS 72-03752 A03.

† Computer Science Department, Stanford University, Stanford, California 94305.

The main results of this paper are summarized in the following theorem. Sections 2–5 are devoted to a proof of it.

THEOREM 1. *Let $k \geqq 3$ be a fixed integer, $a(k) = 2^{-(k+1)}$, and all points to be considered are in E_p^k with $p \in \{1, 2, \infty\}$. Then each of the following problems can be solved in time $O(n^{2-a(k)}(\log n)^{1-a(k)})$. For the case when $k = 3$ and $p = 2$, the bound can be improved to $O((n \log n)^{1.8})$.*

MST-*problem* — *Let V be a set of n points; find a minimum spanning tree on V.*

NFN-*problem* — *(nearest foreign neighbor). Let $V_1, V_2, \cdots, V_l$ be disjoint sets of points, $V = \bigcup_i V_i$, and $|V| = n$. For each V_i and every $\tilde{x} \in V_i$, find a $\tilde{y} \in V - V_i$ such that $d_p(\tilde{x}, \tilde{y}) = \min \{d_p(\tilde{x}, \tilde{z}) \mid \hat{z} \in V - V_i\}$.*

GN-*problem* — *(geographic neighbor). Let V be a set of n points. For any $\tilde{x} \in V$, let $N(\tilde{x}) = \{\tilde{v} \mid v_i \geqq x_i$ for all $1 \leqq i \leqq k, \tilde{v} \neq \tilde{x}, \tilde{v} \in V\}$. For each $\tilde{x} \in V$, find a $\tilde{y} \in N(\tilde{x})$ such that $d_p(\tilde{x}, \tilde{y}) = \min \{d_p(\tilde{x}, \tilde{v}) \mid \tilde{v} \in N(x)\}$ if $N(\tilde{x}) \neq \varnothing$.*

AFP-*problem* — *(all farthest points)* [3]. *Let V be a set of n points. For each $\tilde{x} \in V$, find a $\tilde{y} \in V$ such that $d_p(\tilde{x}, \tilde{y}) = \max \{d_p(\tilde{x}, \tilde{v}) \mid \tilde{v} \in V\}$.*

FP-*problem* — *(farthest pair)* [3]. *Let V be a set of n points; find $\tilde{x}, \tilde{y} \in V$ such that $d_p(\tilde{x}, \tilde{y}) = \max \{d_p(\tilde{u}, \tilde{v}) \mid \tilde{u}, \tilde{v} \in V\}$.*

In § 6, we briefly describe, for the L_2 and the L_∞ metric, how to obtain $o(kn^2)$ algorithms when k is allowed to vary with n.

A remark on the model of computation: We assume a random access machine with arithmetic on real numbers, and charge uniform cost for all access and arithmetic operations [1]. In this paper, we often carry out computations of $d_p(\tilde{x}, \tilde{y})$, which involves an apparent square root operation when $p = 2$. However, since our construction of MST depends only on the linear ordering among the edge weights, we can replace $d_p(\tilde{x}, \tilde{y})$ throughout by some monotone function of $d_p(\tilde{x}, \tilde{y})$. In particular, $d_2(\tilde{x}, \tilde{y})$ may be replaced by $(d_2(\tilde{x}, \tilde{y}))^2 = \sum (x_i - y_i)^2$ everywhere to produce a valid algorithm without square root operations. We shall, however, retain the original form of the algorithm for clarity and for consistency with the cases $p = 1, \infty$.

2. The post-office problem and its applications. In this section we review solutions to the post-office problem, and show how it can be used to prove Theorem 1 for the AFP, FP and NFN problems.

The *post-office problem* can be stated as follows. Given a set of n points $\tilde{v}_1, \tilde{v}_2, \cdots, \tilde{v}_n$ in E_p^k, we wish to preprocess them so that any subsequent query of the following form can be answered quickly:

nearest-point query. Given a point $\tilde{x}$, find a nearest $\tilde{v}_i$ to $\tilde{x}$ (i.e., $d_p(\tilde{x}, \tilde{v}_i) \leqq d_p(\tilde{x}, \tilde{v}_j)$ for all j).

This problem was mentioned in Knuth [12] for the case of points in the Euclidean plane ($k = p = 2$). For this special case, several solutions were given by Dobkin and Lipton [8] and Shamos [15]. For example, it is known that with an $O(n^2)$-time preprocessing, any nearest-point query can be answered in $O(\log n)$ time [15]. A solution for the k-dimensional Euclidean space was given in Dobkin and Lipton [8], where it was shown that it is possible to preprocess n points such that any subsequent nearest-point query can be answered in $O(2^k \log n)$ time. Their technique is quite general, and applies equally well if we wish to answer "farthest-point" queries—Given $\tilde{x}$, find a farthest $\tilde{v}_i$ to $\tilde{x}$—instead of nearest-point queries. The preprocessing procedure was not discussed in great detail in [8]. A detailed study [19] gives the following result.

DEFINITION. We shall use $b(k) = 2^{k+1}$, and $a(k) = b(k)^{-1} = 2^{-(k+1)}$.

LEMMA 2.1. *Let $k \geqq 3$ be a fixed integer, and $p \in \{1, 2, \infty\}$. There is an algorithm which preprocesses n points in E_p^k in time $O(n^{b(k)})$ such that each subsequent nearest-point query can be answered in $O(\log n)$ time. In the special case $k = 3, p = 2$, the preprocessing time can be improved to $O(n^5 \log n)$ with a query response-time $O((\log n)^2)$. The preceding statements remain true if the farthest-point query is used in place of the nearest-point query.*

We shall now demonstrate the use of Lemma 2.1 by applying it to solve the MST problem in a special case. It also gives us some insight into the connection between MST and some typical nearest-neighbor problem [3], [16].

Consider the case when V consists of two widely separated clusters A and B. For definiteness, assume that $d_p(A, B) > n \cdot (\text{diam}(A) + \text{diam}(B))$[1]. In this case any MST on V consists of the union of an MST for A and an MST for B, plus a shortest edge between A and B. Thus, to be able to solve the MST problem efficiently, we have to be able to solve the following problem efficiently:

Problem RMST. Given two well-separated sets A and B in E_p^k, with $|A| = |B| = n$, find a shortest edge between A and B.

This problem looks very similar to the problem of finding the closest pair in a set, which has an $O(n \log n)$-time algorithm. However, there does not seem to be any simple divide-and-conquer $o(n^2)$ solution. We shall presently give an $o(n^2)$-time algorithm employing the post-office problem as a subroutine.

Consider the following algorithm.

(S1) Divide B into $r = \lceil n/q \rceil$ sets $B_1, B_2, \cdots, B_r$, each with at most q points (q to be determined).
(S2) For each $1 \leqq i \leqq r$, preprocess B_i for nearest-point queries as in Lemma 2.1.
(S3) For each $\tilde{x} \in A$ and each $1 \leqq i \leqq r$, find a point $\tilde{y}(\tilde{x}, i) \in B_i$ that is nearest to $\tilde{x}$ among all points in B_i.
(S4) For each $\tilde{x} \in A$, find a $\tilde{z}(\tilde{x}) \in B$ nearest to x by comparing $\tilde{y}(\tilde{x}, i)$ for all $1 \leqq i \leqq r$.
(S5) Find a shortest such edge $\{\tilde{x}, \tilde{z}(\tilde{x})\}$.

The time taken is dominated by (S2) and (S3), i.e.,

$$O(r \cdot q^{b(k)} + nr \log q).$$

Choosing $q = (\log n)^{(b(k))^{-1}}$ gives the time $O(n(n \log n)^{1-(b(k))^{-1}})$. Thus, we have found an algorithm that solves RMST in time $O(n^{2-a(k)}(\log n)^{1-a(k)})$. For the case $k = 3$ and $p = 2$, one can choose $q = (n \log n)^{1/5}$ to obtain an $O((n \log n)^{1.8})$ algorithm.

We wish to make two observations concerning the above procedure. Firstly, the AFP- and FP-problems can be solved with the same time bounds by very similar procedures (employing farthest-point queries and preprocessing, of course). We will thus consider that Theorem 1 has been proved for these problems. Secondly, the RMST-problem is a type of nearest-neighbor problem with some restrictions on the "legal" neighbors. It is reasonable to expect more such problems can be solved with similar techniques. The NFN- and GN-problems are problems of this type, and we will see that their efficient solutions enable the MST problem to be solved efficiently. We shall give a fast algorithm for NFN-problems presently, leaving the more involved proof of Theorem 1 for MST and GN to the later sections.

[1] We use the notation $d_p(A, B) = \min\{d_p(\tilde{u}, \tilde{v}) \mid \tilde{u} \in A, \tilde{v} \in B\}$, $d_p(\tilde{u}, S) = \min\{d_p(\tilde{u}, \tilde{v}) \mid \tilde{v} \in S\}$, and diam $(S) = \max\{d_p(\tilde{u}, \tilde{v}) \mid \tilde{u}, \tilde{v} \in S\}$.

We are given disjoint sets $V_1, V_2, \cdots, V_l$ with a total of n points in $V = \bigcup_i V_i$. For a point $\tilde{x} \in V_i$, every point $\tilde{y} \in V - V_i$ is a *foreign neighbor* of $\tilde{x}$. Let $q = \lceil (n \log n)^{a(k)} \rceil$; call a set V_i *small* if $|V_i| < q$, and *large* if $|V_i| \geqq q$. We partition V into $r = O(n/q)$ parts $B_1, B_2, \cdots, B_r$, where each part (call it a *block*) either is the union of several small V_i or is totally contained in some large V_i. Furthermore, each part contains at most $2q$ points, and except possibly for B_r, at least q points. The above partition can be accomplished in $O(n)$ time by breaking each large V_i into several blocks and grouping small V_i into blocks of appropriate sizes. We now preprocess each block B_i so that, for any query point $\tilde{x}$, a point nearest to $\tilde{x}$ in B_i can be found in $O(\log q)$ time. According to Lemma 2.1, this preprocessing can be accomplished in time $O(rq^{b(k)})$ for all blocks B_i. We are now ready to find, for each point $\tilde{x} \in V$, a nearest foreign neighbor $\tilde{y}$, i.e., $d_p(\tilde{x}, \tilde{y}) = \min \{d_p(\tilde{x}, \tilde{z}) \mid \tilde{z} \in V - V_i\}$, when $\tilde{x} \in V_i$. Assume that $\tilde{x} \in V_i$ and $\tilde{x} \in B_t$. Let us find, for each block B_j that is disjoint from V_i, a point $\tilde{z}(\tilde{x}, j)$ nearest to $\tilde{x}$ among all points in B_j. Then we find a nearest foreign neighbor $\tilde{y}$ from the points $\tilde{z}(\tilde{x}, j)$ and points in $B_t - V_i$ by computing and comparing their distances to $\tilde{x}$. The running time for finding $\tilde{y}$, for each $\tilde{x}$, is thus $O(r \log q + (r+q))$. In summary, the total running time of the above procedure for NFN is $O(n + rq^{b(k)} + nr \log q + nq)$, which is $O(n^{2-a(k)}(\log n)^{1-a(k)})$. As before, an $O((n \log n)^{1.8})$ algorithm can be obtained for the case $k = 3$ and $p = 2$.

This proves Theorem 1 for the NFN-problem. An interesting connection exists between MST- and NFN-problems. In fact, in Sollin's algorithm[4, p. 179], an MST can be found essentially by solving NFN-problems $O(\log n)$ times. Thus, we have shown that an MST can be found in $\log n \times O(n^{2-a(k)}(\log n)^{1-a(k)})$-time. The $\log n$ factor can be avoided by reducing MST to a generalized version of the GN-problem, which can be solved in time $O(n^{2-a(k)}(\log n)^{1-a(k)})$. The proof requires additional techniques beyond the simple application of post-office problems to small parts of V. We shall illustrate the ideas for two dimensions in the next section, and complete the proof in §§ 4 and 5.

3. An illustration in two dimensions. We illustrate the ideas of our MST-algorithms with an informal description for 2-dimensional Euclidean space. Let us first consider a special type of nearest-neighbor problem. Let $\tilde{p}$ be any point in the plane. We divide the plane into eight regions relative to $\tilde{p}$ as shown in Fig. 1. The regions are formed by four lines passing through $\tilde{p}$ and having angles of 0°, 45°, 90°, and 135°, respectively, with the x-axis. We number the regions counterclockwise as shown in Fig. 1, and use $R_l(\tilde{p})$ to denote the set of points in the lth region (including its boundary), for $1 \leqq l \leqq 8$.

LEMMA 3.1. *If $\tilde{q}$ and $\tilde{q}'$ are two points in $R_l(\tilde{p})$ for some l, then $d_2(\tilde{q}, \tilde{q}') < \max \{d_2(\tilde{p}, \tilde{q}), d_2(\tilde{p}, \tilde{q}')\}$.*

Proof. Consider the triangle $\tilde{p}\tilde{q}\tilde{q}'$ (see Fig. 1). Since $\angle \tilde{q}\tilde{p}\tilde{q}' \leqq 45° < \pi/3$, its opposite side $\tilde{q}\tilde{q}'$ cannot be the longest side of the triangle. □

Let V be a set of n distinct points in the plane. For each point $\tilde{v} \in V$, let $N_l(\tilde{v})$ be those points of V, excluding $\tilde{v}$ itself, that are in the lth region relative to $\tilde{v}$. That is,

$$N_l(\tilde{v}) = V \cap R_l(\tilde{v}) - \{\tilde{v}\} \quad \text{for } 1 \leqq l \leqq 8.$$

A point $\tilde{u}$ in $N_l(\tilde{v})$ is said to be a *nearest neighbor to $\tilde{v}$ in the lth region* if $d_2(\tilde{v}, \tilde{u}) = \min \{d_2(\tilde{v}, \tilde{w}) \mid \tilde{w} \in N_l(\tilde{v})\}$. Note that such a nearest neighbor does not exist if $N_l(\tilde{v}) = \varnothing$, and may not be unique when it exists. Now, consider the following computational problem.

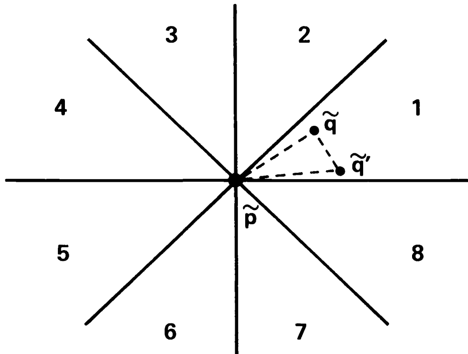


FIG. 1. *Regions* $R_l(\tilde{p})$ *for* $1 \leqq l \leqq 8$.

The eight-neighbors problem (ENP). Given a set V of n points in the plane, find for each $\tilde{v} \in V$ and $1 \leqq l \leqq 8$ a nearest neighbor to $\tilde{v}$ in the lth region, if it exists.

We first show that, once the eight-neighbors problem is solved for V, it takes very little extra effort to find an MST on V. To see this, we form E, the set of edges defined by:

$$E = \{\{\tilde{v}, \tilde{u}\} \mid \tilde{v} \in V \text{ and } \tilde{u} \text{ is a nearest neighbor to } \tilde{v} \text{ selected by ENP}\}.$$

We assert that the set of edges E contains an MST on V. As E contains at most $8n$ edges, we can then construct an MST for the sparse graph (V, E) in $O(n \log \log n)$ steps [17], a very small cost.

THEOREM 3.2. *The set of edges E contains an MST on V.*

Proof. Let T be a set of edges that form an MST on V. We will show that, for any edge $\{\tilde{v}, \tilde{w}\}$ that is in T but not in E, we can replace $\{\tilde{v}, \tilde{w}\}$ by an edge in E and still maintain an MST. This would prove the theorem since we can perform this operation on T repeatedly until all edges in T are from E.

Let $\{\tilde{v}, \tilde{w}\}$ be an edge in $T - E$. Assume $\tilde{w} \in R_l(\tilde{v})$. Then $N_l \neq \varnothing$, and there is a nearest neighbor $\tilde{u}$ to $\tilde{v}$ in $N_l(\tilde{v})$ such that $\{\tilde{v}, \tilde{u}\} \in E$. Clearly $\tilde{u} \neq \tilde{w}$ and $d_2(\tilde{v}, \tilde{u}) \leqq d_2(\tilde{v}, \tilde{w})$. Let us delete $\{\tilde{v}, \tilde{w}\}$ from T. Then T is separated into two disjoint subtrees with $\tilde{v}$ and $\tilde{w}$ belonging to different components. Now, $\tilde{u}$ and $\tilde{w}$ must be in the same component. For if they were not, $\{\tilde{u}, \tilde{w}\}$ would be a shorter connecting edge for the two subtrees than $\{\tilde{v}, \tilde{w}\}$ by Lemma 3.1, contradicting the fact that T is an MST. Therefore $\tilde{u}$ is in the same subtree as $\tilde{w}$, and adding the edge $\{\tilde{v}, \tilde{u}\}$ to $T - \{\tilde{v}, \tilde{w}\}$ results in a spanning tree with total weight no greater than that of T. □

We now proceed to solve the eight-neighbors problem. We will find a nearest neighbor to each point in the first region. The procedure can be simply adapted to find nearest neighbors in the lth region for other l. As demonstrated earlier, the MST problem can be thus solved in a total of $8 \cdot f(n) + O(n \log \log n)$ steps, if the first-region nearest neighbors can be found in $f(n)$ steps.

To study the first regions, it is convenient to tilt the y-axis by 45° clockwise (see Fig. 2). That is, transform the coordinates (x_1, x_2) of a point v into (x_1', x_2'), defined by

$$\begin{pmatrix} x_1' \\ x_2' \end{pmatrix} = \begin{pmatrix} 1 & -1 \\ 0 & \sqrt{2} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}.$$

In the new coordinates, a point $\tilde{u} = (u_1', u_2')$ is in the first region relative to $v = (v_1', v_2')$ if and only if $(u_1' \geqq v_1') \wedge (u_2' \geqq v_2')$.

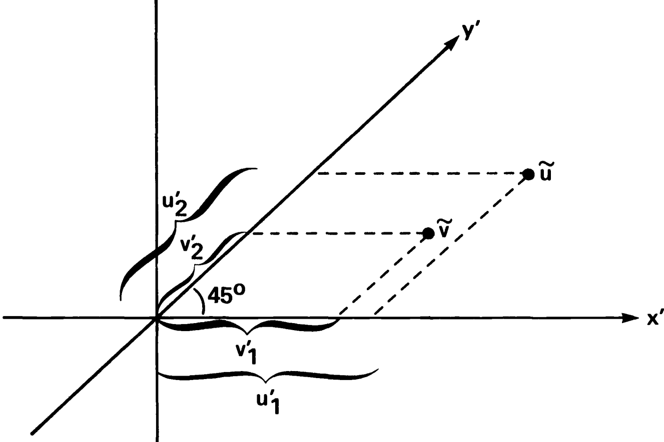


FIG. 2. *New coordinate system.*

For simplicity we assume that all the $2n$ coordinates x'_1, x'_2 of points $\tilde{x} \in V$ are distinct numbers. This restriction shall be removed in the general algorithm in § 3. Let us first sort the points according to their first coordinates x'_1, and divide them into $s = (n/q)^{1/2}$ consecutive groups each with $\approx qs$ points (Fig. 3), q to be determined later. Then for each of these s groups we sort the points in ascending order of the coordinates x'_2, and divide them into s consecutive groups with $\approx q$ points each (Fig. 4).

The set V is thus divided into s^2 "cells". For any $\tilde{v} \in V$, the cells can be classified into three classes by their position relative to $\tilde{v}$: class 1, cells all of whose points are in $N_1(\tilde{v})$; class 2, cells with no points in $N_1(\tilde{v})$; and class 3, the remaining cells. A useful observation is that the number of cells in class 3 is at most $2 \times s$. This can be understood as follows: If we draw a horizontal and a vertical line through v, only those cells that are "hit" can be in class 3, and there are at most $2 \times s$ of them. We can now try to find a nearest neighbor for $\tilde{v}$ in $N_1(\tilde{v})$ using the following strategy: we examine each cell in turn for cells in class 3, and compute $d_2(\tilde{v}, \tilde{u})$ for all $\tilde{u}$ in the

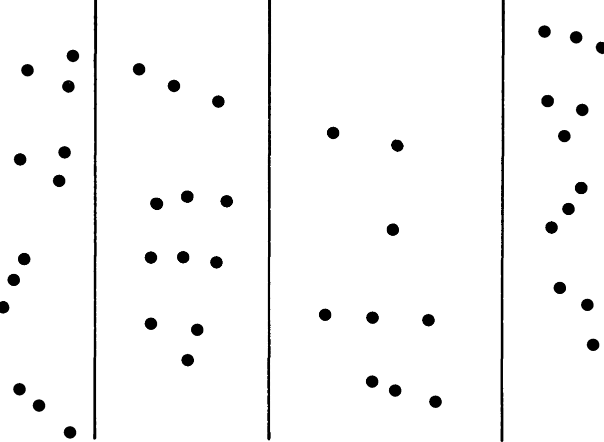


FIG. 3. *Division of points into s groups according to values of* x'_1.

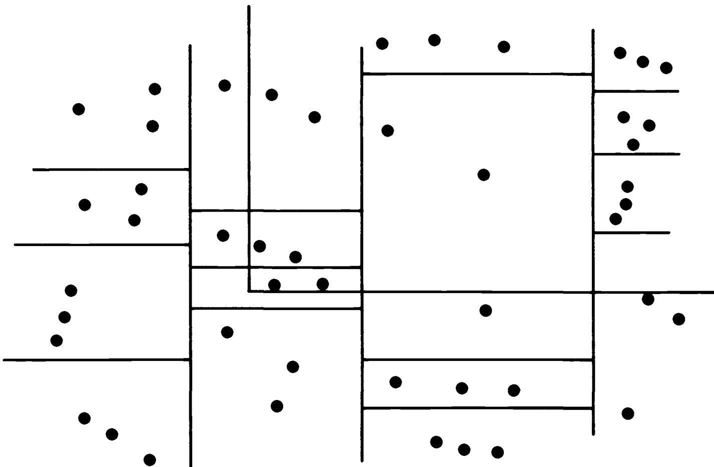


FIG. 4. *Completing the division of V into s^2 cells.*

cell; for a cell in class 2, we ignore it; for a cell C in class 1, we compute $\tilde{u}$ and $d_2(\tilde{v}, \tilde{u})$ defined by $d_2(\tilde{v}, \tilde{u}) = \min \{d_2(\tilde{v}, \tilde{x}) | \tilde{x} \in C\}$. A nearest point can now be found by selecting the point $\tilde{u}$ with minimum $d_2(\tilde{v}, \tilde{u})$ from the preceding calculations. The cost is $O(2s \cdot q + \#$ of class 1 cells $\times a) = O(2sq + s^2 a) = O(n/s + 2an/q)$, where a is the cost of computing $d_2(v_i, C)$ for a cell C of q points. If we have to compute $d_2(\tilde{v}, \tilde{u})$ for each $\tilde{u} \in C$, then $a = O(q)$, and the total cost would be $O(n)$, and we have not made any progress. However, we know from the post-office problem that we can lower a to $\log q$ if we are willing to preprocess the set C (in $O(q^2)$ time). So let us do the following: (i) preprocess every cell C to facilitate the computing of $d_2(\tilde{v}, C)$ (cost $O((n/q) \cdot q^2) = O(nq)$); (ii) for each $\tilde{v}$, compute the nearest neighbor in the above manner in time $O(n/s + (n/q) \log q)$. The total cost is then $O(nq + n^2/s + (n^2/q) \log q)$. Take $q = n^{1/3}$ and obtain an algorithm that runs in time $O(n^{5/3} \log n)$. This gives an $o(n^2)$ algorithm for finding an MST in two dimensions. We shall generalize the ideas to general k.

4. Reduction of MST to a general GN-problem. We shall prove Theorem 1 for the MST- and GN-problems in this and the next sections. Without loss of generality, we shall assume that the n given points in V are all distinct.

In this section we reduce the finding of MST in E_p^k to a version of the geographic-neighbor problem. We assume that $p \in \{1, 2, \infty\}$ throughout the rest of the paper.

We make E_p^k a vector space by defining $\tilde{x} + \tilde{y} = (x_1 + y_1, x_2 + y_2, \cdots, x_k + y_k)$ and $c\tilde{x} = (cx_1, cx_2, \cdots, cx_k)$, where c is any real number and x_i, y_i are the components of $\tilde{x}$ and $\tilde{y}$. We shall refer to any element of E_p^k as a *point* or a *vector*. The jth component of a vector $\tilde{z}$ will be denoted as z_i without further explanation. The *inner product* of two vectors $\tilde{x}$ and $\tilde{y}$ is $\tilde{x} \cdot \tilde{y} = \sum_{i=1}^k x_i y_i$, and the *norm* of $\tilde{x}$ is $\|\tilde{x}\| = (\tilde{x} \cdot \tilde{x})^{1/2}$. A *unit vector* $\tilde{x}$ is a vector with $\|\tilde{x}\| = 1$. Notice that all these definitions are independent of p.

Vectors $\tilde{b}_1, \tilde{b}_2, \cdots, \tilde{b}_j$ are *linearly independent* if $\sum_{i=1}^j \lambda_i \tilde{b}_i = 0$ implies all $\lambda_i = 0$. A set of k linearly independent vectors in E_p^k is called a *basis* (of E_p^k). Let $B = \{\tilde{b}_1, \tilde{b}_2, \cdots, \tilde{b}_k\}$ be a basis of E_p^k. The *convex cone* of B is $\operatorname{Conv}(B) = \{\sum_{i=1}^k \lambda_i \tilde{b}_i \,|\, \lambda_i \geqq 0 \text{ for all } i\}$. For any $\tilde{x} \in E_p^k$, *the region B of $\tilde{x}$* is defined as

$$R(B; \tilde{x}) = \{\tilde{y} \,|\, \tilde{y} - \tilde{x} \in \operatorname{Conv}(B)\}.$$

Let V be a set of n distinct vectors in E_p^k. Denote by $N(B, \tilde{v})$ the set $V \cap \{\tilde{u} \mid \tilde{u} \in R(B; \tilde{v}) - \{\tilde{v}\}\}$, for each $\tilde{v} \in V$. We shall say that $\tilde{w}$ is a *geographic neighbor to $\tilde{v}$ in region B* if $\tilde{w} \in N(B; v)$ and $d_p(\tilde{w}, \tilde{v}) \leqq d_p(\tilde{u}, \tilde{v})$ for all $\tilde{u} \in N(B; \tilde{v})$.

The GGN-*problem* (*general geographic neighbor*). Given a basis B and a set V of n distinct vectors in E_p^k, find, for each $\tilde{v} \in V$, a geographic neighbor to $\tilde{v}$ in region B, if one exists.

Notice that this reduces to the GN-problem when $B = \{\tilde{b}_1, \tilde{b}_2, \cdots, \tilde{b}_k\}$ with $b_{ij} = \delta_{ij}$. The rest of this section is devoted to showing the following theorem, which states that, if there is a fast algorithm to solve the GGN-problem, then one can solve the MST-problem efficiently.

THEOREM 4.1. *Let $k \geqq 2$ be a fixed integer. Suppose there is an algorithm that solves the* GGN-*problem for n given points in E_p^k in at most $f(n)$ steps. Then a minimum spanning tree for n points in E_p^k can be found in $O(f(n) + n \log\log n)$ steps.*

Define the *angle* between two nonzero vectors $\tilde{x}$ and $\tilde{y}$ as $\Theta(\tilde{x}, \tilde{y}) = \cos^{-1}(\tilde{x} \cdot \tilde{y}/\|\tilde{x}\| \cdot \|\tilde{y}\|)$, $0 \leqq \Theta(\tilde{x}, \tilde{y}) \leqq \pi$. For any basis B of E_p^k, the *angular diameter* of B is defined by $\text{Ang}(B) = \sup\{\Theta(\tilde{x}, \tilde{y}) \mid \tilde{x}, \tilde{y} \in \text{Conv}(B)\}$. It can be shown that $\text{Ang}(B) = \max\{\Theta(\tilde{b}_i, \tilde{b}_j) \mid \tilde{b}_i, \tilde{b}_j \in B\}$, although we shall not use that fact.

Let $\mathscr{B}$ be a finite family of basis of E_p^k. We call $\mathscr{B}$ a *frame* if $\bigcup_{B \in \mathscr{B}} \text{Conv}(B) = E_p^k$. The *angular diameter* of a frame $\mathscr{B}$ is given by $\text{Ang}(\mathscr{B}) = \max\{\text{Ang}(B) \mid B \in \mathscr{B}\}$. For example, let $\tilde{b}_1 = (1, 0)$, $\tilde{b}_2 = (-1, 1)$, $\tilde{b}_3 = (0, -1)$, $\tilde{b}_4 = (-\frac{1}{2}, -1)$ as shown in Fig. 5, then

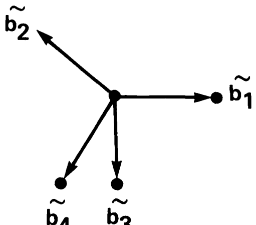


FIG. 5. *Illustration of "basis" and "frame".*

$B_1 = \{\tilde{b}_1, \tilde{b}_2\}$, $B_2 = \{\tilde{b}_2, \tilde{b}_3\}$, $B_3 = \{\tilde{b}_4, \tilde{b}_1\}$ are bases of E_p^2, and $\mathscr{B} = \{B_1, B_2, B_3\}$ a frame; $\Theta(B_1) = \Theta(B_2) = 3\pi/4$, $\Theta(B_3) = 2\pi/3$, and $\Theta(\mathscr{B}) = 3\pi/4$.

Intuitively, the convex cone of a basis B has a "narrow" angular coverage if $\text{Ang}(B)$ is small. The following result asserts that a frame exists in which every basis is narrow, and such a frame can be constructed.

LEMMA 4.2. *For any $0 < \psi < \pi$, one can construct in finite steps a frame $\mathscr{B}$ of E_p^k such that* $\text{Ang}(\mathscr{B}) < \psi$.

Proof. See Appendix. □

We consider the following MST-algorithm. Let us construct a frame $\mathscr{B}$ of E_p^k such that $\text{Ang}(B) < \sin^{-1}(\frac{1}{2}k^{-(1/2+1/p)})$. Next, for each $B \in \mathscr{B}$, we solve the GGN-problem—for each $\tilde{v} \in V$, find a geographic neighbor $\tilde{u}$ to $\tilde{v}$ in region B if it exists—and form the set $E(B)$, the collection of all such edges $\{\tilde{u}, \tilde{v}\}$. Clearly, $|\bigcup_{B \in \mathscr{B}} E(B)| \leqq n \cdot |\mathscr{B}| = O(n)$. We now claim that $\bigcup_{B \in \mathscr{B}} E(B)$ contains an MST on V. If this is true, then we can find an MST in an additional $O(n \log\log n)$ steps. The total time taken by the MST algorithm is then $O(f(n) + n \log\log n)$. It remains to prove the following result.

LEMMA 4.3. *$\bigcup_{B \in \mathscr{B}} E(B)$ contains an* MST *on V.*

Proof. The proof is almost identical to the proof of Theorem 3.2, except that we need to establish the next lemma. □.

LEMMA 4.4. *Let* $\tilde{x}$, $\tilde{y}$, $\tilde{z}$ *in* E_p^k *satisfy* $\Theta(\tilde{x}-\tilde{z}, \tilde{y}-z)<\sin^{-1}(\frac{1}{2}k^{-(1/2+1/p)})$, *then* $d_p(\tilde{x}, \tilde{y})<\max\{d_p(\tilde{y}, \tilde{z}), d_p(\tilde{x}, \tilde{z})\}$.

Proof. Use α, β, γ to denote angles as shown in Fig. 6. By assumption,

$$\sin\alpha<\tfrac{1}{2}k^{-(1/2+1/p)}. \tag{1}$$

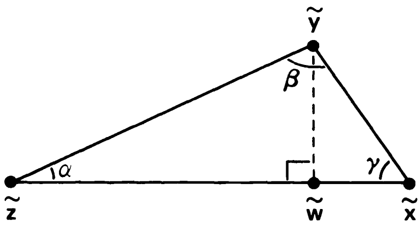


FIG. 6. *Illustration for the proof of Lemma* 4.4.

Without loss of generality, assume that $\alpha+\beta>\pi/2$. Let $\tilde{w}$ be the projection of $\tilde{y}$ on the segment from $\tilde{z}$ to $\tilde{x}$. By the triangle inequality satisfied by metric d_p, we have

$$d_p(\tilde{z}, \tilde{w})+d_p(\tilde{w}, \tilde{y})\geqq d_p(\tilde{y}, \tilde{z}),$$

$$d_p(\tilde{x}, \tilde{w})+d_p(\tilde{w}, \tilde{y})\geqq d_p(\tilde{x}, \tilde{y}).$$

Thus,

$$d_p(\tilde{z}, \tilde{w})+d_p(\tilde{x}, \tilde{w})\geqq d_p(\tilde{x}, \tilde{y})+(d_p(\tilde{y}, \tilde{z})-2d_p(\tilde{w}, \tilde{y})). \tag{2}$$

But since $\tilde{w}$ is on the segment $\tilde{z}$ to $\tilde{x}$, we have $d_p(\tilde{x}, \tilde{z})=d_p(\tilde{z}, \tilde{w})+d_p(\tilde{x}, \tilde{w})$. Therefore, if we can further show that

$$d_p(\tilde{y}, \tilde{z})-2d_p(\tilde{w}, \tilde{y})>0, \tag{3}$$

then (2) implies $d_p(\tilde{x}, \tilde{z})>d_p(\tilde{x}, \tilde{y})$, proving the lemma.

To prove formula (3), we notice that for any positive l, and $\tilde{u}$, $\tilde{v}$ in E_l^k,

$$k^{1/l}\max_i|\tilde{u}_i-\tilde{v}_i|\geqq d_l(\tilde{u}, \tilde{v})\geqq\max_i|\tilde{u}_i-\tilde{v}_i|. \tag{4}$$

This leads to

$$k^{1/p}\|\tilde{u}-\tilde{v}\|\geqq d_p(\tilde{u}, \tilde{v})\geqq k^{-1/2}\|\tilde{u}-\tilde{v}\|. \tag{5}$$

In particular,

$$\begin{aligned} d_p(\tilde{y}, \tilde{w})&\leqq k^{1/p}\|\tilde{y}-\tilde{w}\|,\\ d_p(\tilde{y}, \tilde{z})&\geqq k^{-1/2}\|\tilde{y}-\tilde{z}\|. \end{aligned} \tag{6}$$

Now, clearly by (1),

$$\|\tilde{y}-\tilde{w}\|=(\sin\alpha)\|\tilde{y}-\tilde{z}\|<\tfrac{1}{2}k^{-(1/2+1/p)}\|\tilde{y}-\tilde{z}\|. \tag{7}$$

Formula (3) follows from (6) and (7). □

5. An algorithm for the general geographic neighbor problem.

5.1. An outline. As shown in the preceding section, the MST-problem can be reduced to the GGN-problem, and the GN-problem is a special case of the GGN-

problem. In this section, we shall give an asymptotically fast algorithm for the GGN-problem, which completes the proof of Theorem 1.

Given a basis B and a set V of n points in E_p^k, the algorithm works in two phases.

Preprocessing phase. (A) Partition V in $O(kn \log n)$ steps into $r = \lceil n/q \rceil$ subsets $V_1, V_2, \cdots, V_r$, each with at most q points (q to be determined later). The division will be such that, for any $\tilde{x} \in E_p^k$, all but a fraction $r^{-1/k}$ of the subsets V_j have the property that the entire set V_j is either in region B of $\tilde{x}$ or outside of region B.

(B) Preprocess each V_j in $O(q^{b(k)})$ steps such that, for any new point $\tilde{x} \in E^k$, a nearest point $\tilde{u}$ in V_j can be found in $O(\log q)$ steps.

Finishing phase. (C) For each $\tilde{v} \in V$, we find a geographic neighbor in region B as follows. We examine the r sets $V_1, V_2, \cdots, V_r$ in turn. For each V_j, we perform a test which puts V_j into one of the three categories. A category-1 V_j has all its points in region B of $\tilde{v}$, a category-2 V_j has all its points outside of region B. The nature of a category-3 V_j is unimportant, except that there are at most $r^{1-k^{-1}}$ V_j in this category; we consider the V_j that contains $\tilde{v}$ itself to be of category 3 independent of the above division. As we shall see later, the test will be easy to carry out, in fact in $O(k)$ time per test. For a category-1 V_j, we find a nearest $\tilde{w}$ in V_j in $O(\log q)$ time. For a category-2 V_j, nothing need be done. For a category-3 V_j, we find a nearest $\tilde{w}(\neq \tilde{v}) \in V_j$ in region B, if it exists, by finding all the $\tilde{z} \in V_j$ that are in region B and computing and comparing $d_p(\tilde{z}, \tilde{v})$ for all such $\tilde{z}$. Call $\tilde{w}$ a candidate from V_j. After all the V_j have been so processed, we compare $d_p(\tilde{w}, \tilde{v})$ for all the candidates $\tilde{w}$ obtained (at most r of them), and find a nearest one $\tilde{u}$ to $\tilde{v}$. This $\tilde{u}$ is the geographic neighbor we seek for $\tilde{v}$. Return "nonexistent" if no candidate $\tilde{w}$ exists from any V_j.

In the above description, three points need further elaboration: how step (A) is accomplished, how we check a subset V_i for its category, and how q is chosen. We shall deal with the first two points in § 5.2, and the last point in § 5.3.

5.2. A set partition theorem. We shall show that step (A) of the preprocessing phase in § 5.1 can be accomplished. The key is the following result in Yao and Yao [20]. For completeness, a proof is included.

For any finite set F of points in E^k, let $\text{high}_l(F) = \max\{x_l \mid \tilde{x} \in F\}$ and $\text{low}_l(F) = \min\{x_l \mid \tilde{x} \in F\}$, for $1 \leqq l \leqq k$.

LEMMA 5.1 [20].[2] *Let q and k be positive integers, and F a set of n points in E^k. Then, in $O(kn \log n)$ steps, the following can be done.*

(i) *F is partitioned into $r = \lceil n/q \rceil$ sets $F_1, F_2, \cdots, F_r$, each with at most q points,*

(ii) *The $2kr$ numbers $\text{high}_l(F_i)$, $\text{low}_l(F_i)$, $1 \leqq i \leqq r$, and $1 \leqq l \leqq k$, are computed,*

(iii) *The partition satisfies the condition that, for any $\tilde{y} \in E^k$, there exist at most $k\lceil r^{1/k} \rceil^{k-1}$ sets F_i such that there exists an l with $\text{low}_l(F_i) < y_l \leqq \text{high}_l(F_i)$.*

Proof. We shall prove Lemma 5.1 for the case $k = 3$; the extension to general k is obvious. For the moment, let us assume further that $n = qm^3$ for some integer m. We use the following procedure to partition F.

(a) Sort the points of F in ascending order according to the first components into a sequence $\tilde{x}_1, \tilde{x}_2, \cdots, \tilde{x}_n$. Divide the sorted sequence into m consecutive parts of equal size, that is,

$$G_1 = \{x_j \mid 1 \leqq j \leqq n/m\}, \quad G_2 = \{X_j \mid n/m + 1 \leqq j \leqq 2n/m\}, \quad \cdots, \quad G_m.$$

[2] This lemma was proved in [20] with $q = n^{1/k}$.

(b) For each $1 \leqq i \leqq m$, sort the points in G_i according to the 2nd components; divide the sorted sequence of G_i into m consecutive parts of equal size, $G_{i1}, G_{i2}, \cdots, G_{im}$.

(c) For each $1 \leqq i, j \leqq m$, sort the points in G_{ij} according to their 3rd components; divide the sorted sequence of G_{ij} into m consecutive parts of equal size, $G_{ij1}, G_{ij2}, \cdots, G_{ijm}$.

(d) Rename the m^3 sets G_{ijl} as $F_1, F_2, \cdots, F_r$ where $r = n/q = m^3$.

(e) Compute $\text{high}_l(F_i)$, $\text{low}_l(F_i)$ for $1 \leqq i \leqq r$, $1 \leqq l \leqq 3$ according to the definitions. The above procedure takes $O(n \log n)$ steps, and each F_i contains exactly q points. It remains to show that property (iii) in Lemma 5.1 is satisfied.

Let $\tilde{y} \in E^3$. We shall prove that, for each $1 \leqq l \leqq 3$, there are at most m^2 F_i with $\text{low}_l(F_i) < y_l \leqq \text{high}_l(F_i)$. The proof is based on the following properties of the partition, where $1 \leqq i, j \leqq m$:

$$\text{(5.1)} \quad \text{low}_1(G_1) \leqq \text{high}_1(G_1) \leqq \text{low}_1(G_2) \leqq \text{high}_1(G_2) \leqq \cdots \leqq \text{low}_1(G_m) \leqq \text{high}_1(G_m),$$

$$\text{(5.2)} \quad \text{low}_2(G_{i1}) \leqq \text{high}_2(G_{i1}) \leqq \text{low}_2(G_{i2}) \leqq \text{high}_2(G_{i2}) \leqq \cdots \leqq \text{low}_2(G_{im}) \leqq \text{high}_2(G_{im}),$$

$$\text{(5.3)} \quad \text{low}_3(G_{ij1}) \leqq \text{high}_3(G_{ij1}) \leqq \text{low}_3(G_{ij2}) \leqq \text{high}_3(G_{ij2}) \leqq \cdots \leqq \text{low}_3(G_{ijm}) \leqq \text{high}_3(G_{ijm}).$$

For $l = 1$, according to (5.1), there is at most one j such that

$$\text{low}_1(G_j) < y_1 \leqq \text{high}_1(G_j).$$

Thus, only the m^2 G_{jts}, $1 \leqq t, s \leqq m$, can have $\text{low}_1(G_{jts}) < y_1 \leqq \text{high}_1(G_{jts})$. This proves our assertion for $l = 1$. We now prove the case for $l = 2$. For each i, by (5.2), there is at most one j such that $\text{low}_2(G_{ij}) < y_2 \leqq \text{high}_2(G_{ij})$. Thus, for each i, only the m G_{ijt}, $1 \leqq t \leqq m$, may have $\text{low}_2(G_{ijt}) < y_2 \leqq \text{high}_2(G_{ijt})$. Therefore, at most m^2 G_{ijt} can have $\text{low}_2(G_{ijt}) < y_2 \leqq \text{high}_2(G_{ijt})$. A similar proof works for $l = 3$, making use of formula (5.3).

This proves that, when $k = 3$, and $n = qr = qm^3$ for some integer m, Lemma 5.1 is true. We now drop the restriction on n (still $k = 3$). In this situation, $r = \lceil n/q \rceil$. Let $m = \lceil r^{1/k} \rceil$, and use the same procedure. At most $3m^2$ G_{ijt} will satisfy (iii) by the same proof. This completes the proof for $k = 3$. □

We now extend the above result. Let $B = \{\tilde{b}_1, \tilde{b}_2, \cdots, \tilde{b}_k\}$ be a basis of E^k; for any $\tilde{x} \in E^k$, we shall define a k-tuple $(x'_1, x'_2, \cdots, x'_k)$ by $\tilde{x} = \sum_{i=1}^k x'_i \tilde{b}_i$. For any finite set F of points, define for each $1 \leqq l \leqq k$,

$$\text{high}_l(B; F) = \max\{x'_l \mid \tilde{x} \in F\},$$

$$\text{low}_l(B; F) = \min\{x'_l \mid \tilde{x} \in F\}.$$

THEOREM 5.2. *Let q, n, k $(q, k \leqq n)$ be positive integers, B a basis of E^k, and V a set of n points in E^k. Then, in $O(kn \log n + k^2 n + k^3)$ steps, we can accomplish the following*:

(i) *V is partitioned into $r = \lceil n/q \rceil$ sets $V_1, V_2, \cdots, V_r$, each with at most q points.*

(ii) *The $2kr$ numbers* $\text{high}_l(B, V_i)$, $\text{low}_l(B, V_i)$, $1 \leqq i \leqq r$, $1 \leqq l \leqq k$, *are computed.*

Furthermore, the partition satisfies the condition:

(iii) *For any k-tuple of numbers* $(y_1, y_2, \cdots, y_k)$, *there exist at most* $k\lceil r^{1/k}\rceil^{k-1}$ V_i *such that there exists an l that satisfies*:

$$\text{low}_l(B; V_i) < y_l \leqq \text{high}_l(B; V_i).$$

Before proving this theorem, let us check that this partition fulfills the requirements of step (i) in the preprocessing phase (see § 5.1).

LEMMA 5.3. *A point* $\tilde{y}$ *is in the region B to* $\tilde{x}$, *i.e.*, $\tilde{y} \in R(B; \tilde{x})$, *if and only if* $y'_l \geqq x'_l$ *for all* $1 \leqq l \leqq k$.

Proof. The lemma follows from the equation $\tilde{y} - \tilde{x} = \sum_{l=1}^{k} (y'_l - x'_l)\tilde{b}_l$. □

LEMMA 5.4. *If* $\tilde{x} \in E^k$, *B a basis, and F a finite set of points in* E^k, *then either*

(i) $x'_l \leqq \text{low}_l(B; F)$ *for all* $1 \leqq l \leqq k$, *in which case all points in F are in region B to* $\tilde{x}$, *or*

(ii) $\exists l$, $x'_l > \text{high}_l(B; F)$, *in which case none of the points in F is in region B to* $\tilde{x}$, *or*

(iii) *none of the above*; *there exists an l such that* $\text{low}_l(A; F) < x'_l \leqq \text{high}_l(B; F)$.

Proof. This is an immediate consequence of Lemma 5.3. □

There are two consequences of Lemma 5.4 of interest to us. Firstly, the lemma shows that the requirements of step (A) in § 5.1 are satisfied. For any $\tilde{x}$, a V_j such that *neither* all points of V_j are in $R(B; \tilde{x})$ *nor* none are in $R(B; \tilde{x})$ must satisfy the condition that $\text{low}_l(B, V_j) < x'_l \leqq \text{high}_l(B; V_j)$ for some l, because of Lemma 5.4. By Theorem 5.2, there are at most about $r^{1-1/k}$ such V_j. This proves the claim. Secondly, Lemma 5.4 gives a simple way to detect most of the V_j that satisfy $V_j \subseteq R(B; \tilde{x})$ or $V_j \cap R(B; \tilde{x}) = \emptyset$. Namely, compare x'_l with $\text{high}_l(B; V_j)$ and $\text{low}_l(B; V_j)$ for all l, and determine whether case (i), (ii), or (iii) applies in Lemma 5.4. The test only takes $O(k)$ for each i and j, can be conveniently used in step (C) in the procedure in § 5.1.

We now turn to the proof of Theorem 5.2.

Proof of Theorem 5.2. Let M be the $k \times k$ matrix (b_{ij}) (recall that $\tilde{b}_i = (b_{i1}, b_{i2}, \cdots, b_{ik})$), and M^{-1} be its inverse. We use the following procedure to partition V.

(1) Compute M^{-1} in $O(k^3)$ steps (see, e.g., [1]).

(2) Compute, for each $\tilde{x} \in V$, the k-tuple $(x'_1, x'_2, \cdots, x'_k)$ by $(x'_1, x'_2, \cdots, x'_k) = (x_1, x_2, \cdots, x_k) \cdot M^{-1}$. This takes $O(k^2 n)$ steps.

(3) Consider the set $F = \{(x'_1, x'_2, \cdots, x'_k) | \tilde{x} \in V\}$. We now use the procedure in Lemma 5.1 to divide F into r parts $F_1, F_2, \cdots, F_r$. Let V_i be the subset of V obtained from F_i by replacing every $(x'_1, \cdots, x'_k)$ by the corresponding $\tilde{x}$.

(4) Set $\text{high}_l(B; V_i) \leftarrow \text{high}_l(F_i)$, and $\text{low}_l(B; V_i) \leftarrow \text{low}_l(F_i)$.

The procedure clearly takes $O(kn \log n + k^2 n + k^3)$ steps. The quantities $\text{high}_l(B; V_i)$ and $\text{low}_l(B; V_i)$ are correctly computed by their definitions. Items (i) and (ii) in Theorem 5.2 are obviously true, and (iii) is true because of the properties of $\text{high}_l(F_i)$, $\text{low}_l(F_i)$ stated in Lemma 5.1. □

5.3. Finishing the proof. We now analyze the running time of the algorithm for fixed k and choose q. The preprocessing phase takes time $O(n \log n + r \cdot q^{b(k)})$. In the finishing phase, the running time is dominated by the search for candidates $\tilde{w}$, which is of order $n[(\# \text{ of category-1 } V_j) \cdot \log q + (\# \text{ of category-3 } V_j) \cdot q]$. The last expression is bounded by $n(r \log q + r^{1-k^{-1}} \cdot q)$. The total running time of the algorithm is thus $O(n \log n + r \cdot q^{b(k)} + nr \log q + nqr^{1-k^{-1}})$. Remembering that $b(k) = 2^{k+1}$ and $r = O(n/q)$, we optimize the expression by choosing $q \approx (n \log n)^{a(k)}$. This gives a time $O(n^{2-a(k)} (\log n)^{1-a(k)})$. The improved time bound for the special case $k = 3$, $p = 2$ can be similarly obtained.

6. Discussions. We have shown that, for fixed k and $p \in \{1, 2, \infty\}$, there are $o(n^2)$-time algorithms for a number of geometric problems in E_p^k, including the minimum spanning tree problem. We shall now argue that, when $p \in \{2, \infty\}$, $o(kn^2)$ algorithms exist for all k and n. As are typical for results under fixed k assumptions, the algorithms given in the paper have $o(n^2)$ time bounds when k is allowed to grow slowly with n. In fact, a close examination shows that, if $k \leqq \frac{1}{2} \log \log n$, the algorithms still run in time $o(n^2)$. For $k > \frac{1}{2} \log \log n$, it can be shown [18] that the computation of the distances between all points can be done in $o(kn^2)$ time when $p \in \{2, \infty\}$. Since all problems considered in this paper have $O(n^2)$ algorithms once all the distances are known, the previous statement provides algorithms that run in time $o(kn^2)$.

The efficiency of our algorithms is dependent on the solution to the post-office problem (or its farthest-point analogue). For example, suppose the nearest-point query could be answered in $O(log\, n)$ time after an $O(n^\beta)$-time preprocessing, $\beta \geqq 2$. A simple adaptation of the algorithm would give an $O(n^{2-\beta^{-1}}(\log n)^{1-\beta^{-1}})$-time solution to the NFN-problem, which in turn implies an $O((n \log n)^{2-\beta^{-1}})$-time solution to the MST-problem (see the remark at the end of § 2). If $1 < \beta < 2$, the following modification would also give an $O(n^{2-\beta^{-1}}(\log n)^{1-\beta^{-1}})$-algorithm for the NFN-problem (and hence an $O((n \log n)^{2-\beta^{-1}})$-algorithm for finding MST). We first divide V into $r \approx n/(n \log n)^{\beta^{-1}}$ blocks $B_1, B_2, \cdots$ as before. Each block is preprocessed, and for each $\tilde{x}$, a nearest point in every block not containing x is found. Now, for every point $\tilde{x} \in B_i$, we need to find for it a nearest "foreign" neighbor in B_i. Instead of using brute force (computing the distance from each $\tilde{x} \in B_i$ to every other point in B_i) as was done previously, we divide B_i into r subblocks, preprocess each subblock, and find for $\tilde{x}$ a nearest point in every subblock in B_i. To compute a nearest foreign neighbor to $\tilde{x}$ in the subblock containing $\tilde{x}$, we shall again break the subblocks. This process continues until the size of the subblocks is less than n^δ, where $\delta = 1 - \beta^{-1}$, at which point we compute all distances between points in the same subblock. During the above process, we have located, for each $\tilde{x}$, a set of points containing a nearest foreign neighbor $\tilde{u}$ to $\tilde{x}$. It is then simple to locate such a $\tilde{u}$. This is a brief outline of an $O(n^{2-\beta^{-1}}(\log n)^{1-\beta^{-1}})$-algorithm for NFN-problems, $1 < \beta < 2$. However, it seems unlikely that a nearest-point query can be answered in $O(\log n)$ time with an $O(n^\beta)$-preprocessing, $\beta < 2$, when $k \geqq 3$.

We conclude this paper with the following open problems.

(1) Improve the bounds obtained in this paper.

(2) Analyze the performance of new or existing fast heuristic algorithms for MST-problems. For example, can one show that the AMST-algorithm in [2] always constructs a spanning tree with length at most 5% over the true MST?

(3) Prove bounds on average running time of MST-algorithms for some natural distributions.

(4) Extend results in this paper to L_p-metric for general p.

Appendix. The existence and construction of "narrow" frames—Proof of Lemma 4.2.

LEMMA 4.2. *For any* $0 < \psi < \pi$, *one can construct in finite steps a frame* $\mathscr{B}$ *of* E_p^k *such that* $\operatorname{Ang}(\mathscr{B}) < \psi$.

As the discussion is independent of p, we shall use E^k instead of E_p^k.

We begin with the concept of a "simplex" familiar in Topology (see, e.g., [10]). Let $\tilde{p}_0, \tilde{p}_1, \cdots, \tilde{p}_j$ be $j+1$, $0 \leqq j \leqq k$, points in E^k, where the vectors $\tilde{p}_i - \tilde{p}_0$, $1 \leqq i \leqq j$, are linearly independent. We shall call the set $\{\sum_{i=0}^{j} \lambda_i p_i \mid \lambda_i \geqq 0 \text{ for all } i, \text{ and } \sum_i \lambda_i = 1\}$ a (geometric) *j-simplex in* E^k, denoted by $\langle \tilde{p}_0, \tilde{p}_1, \cdots, \tilde{p}_j \rangle$. Informally, it is the convex

hull formed by vertices $\tilde{p}_0, \tilde{p}_1, \cdots, \tilde{p}_j$ on the minimal linear subspace containing them (see Fig. A). The *diameter* of a simplex s is $\operatorname{diam}(s) = \sup \{\|\tilde{x} - \tilde{y}\| \mid \tilde{x}, \tilde{y} \in s\}$.

The following two lemmas give the connection between simplices and bases. Let $\hat{\varepsilon}$ be a k-tuple $(\varepsilon_1, \varepsilon_2, \cdots, \varepsilon_k)$, where $\varepsilon_i \in \{-1, 1\}$ for all i. Denote by $H(\hat{\varepsilon})$ the hyperplane $\{x \mid \sum_i \varepsilon_i x_i = 1\}$ in E^k.

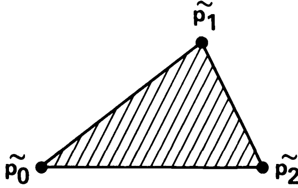


FIG A. *A 2-simplex in* E^2

LEMMA A1. *Let* $s = \langle \hat{p}_0, \hat{p}_1, \cdots, \hat{p}_{k-1} \rangle$ *be a* $(k-1)$*-simplex in* E^k, *where* $\tilde{p}_i \in H(\hat{\varepsilon})$ *for every* i. *Then the set* $B(s) = \{\tilde{p}_0, \tilde{p}_1, \cdots, \tilde{p}_k\}$ *is a basis. Furthermore, the angle* $\varphi = \operatorname{Ang}(B(s))$ *satisfies* $\cos \varphi \geqq 1 - \frac{1}{2}k(\operatorname{diam}(s))^2$.

Proof. Suppose $\sum_{i=0}^{k-1} \lambda_i \tilde{p}_i = 0$. We shall show that $\lambda_i = 0$ for all i. If $\sum_{i=0}^{k-1} \lambda_i = 0$, then $\sum_{i=1}^{k-1} \lambda_i (\tilde{p}_i - \tilde{p}_0) = \sum_{i=0}^{k-1} \lambda_i \tilde{p}_i = 0$. This implies $\lambda_i = 0$ for all i, by the definition of simplex. If $\sum_{i=0}^{k-1} \lambda_i = \Lambda \neq 0$, then $\tilde{v} = \sum_{i=0}^{k-1} (\lambda_i / \Lambda) \tilde{p}_i = 0$. But it is easy to check that $v \in H(\hat{\varepsilon})$, a contradiction.

We have thus shown that $B(s)$ is a basis. To prove the rest of the lemma, let $\tilde{x}$ and $\tilde{y}$ be any two nonzero vectors in $\operatorname{Conv}(B(s))$. We shall prove that $\cos \theta(\tilde{x}, \tilde{y}) \geqq 1 - \frac{1}{2}k(\operatorname{diam}(s))^2$. Without loss of generality, we can assume that $\tilde{x}, \tilde{y} \in s$. Then

$$(\operatorname{diam}(s))^2 \geqq (\tilde{x} - \tilde{y}) \cdot (\tilde{x} - \tilde{y}) = \|\tilde{x}\|^2 + \|\tilde{y}\|^2 - 2\|\tilde{x}\| \cdot \|\tilde{y}\| \cos \theta(\tilde{x}, \tilde{y})$$
$$\geqq 2\|\tilde{x}\| \cdot \|\tilde{y}\| (1 - \cos \theta(\tilde{x}, \tilde{y})).$$

It follows that

$$\cos \theta(\tilde{x}, \tilde{y}) \geqq 1 - \frac{(\operatorname{diam}(s))^2}{2\|\tilde{x}\| \cdot \|\tilde{y}\|}. \tag{A1}$$

As can be easily verified, $\tilde{x}, \tilde{y} \in H(\hat{\varepsilon})$, which implies

$$\|\tilde{x}\|^2 = \sum_i x_i^2 \geqq \frac{1}{k} \Big(\sum_i \varepsilon_i x_i \Big)^2 = \frac{1}{k}.$$

Therefore, $\|\tilde{x}\| \geqq 1/\sqrt{k}$, and similarly $\|\tilde{y}\| \geqq 1/\sqrt{k}$. Formula (A1) then implies

$$\cos \theta(\tilde{x}, \tilde{y}) \geqq 1 - \frac{k}{2} (\operatorname{diam}(s))^2.$$

This proves Lemma A1. □

We shall use $B(s)$ to denote the basis corresponding to simplex s.

LEMMA A2. *Let* $s \subseteq H(\hat{\varepsilon})$ *be a simplex,* $\mathcal{S}$ *a finite collection of simplices, and* $s = \bigcup_{s' \in \mathcal{S}} s'$. *Then* $\operatorname{Conv}(B(s)) = \bigcup_{s' \in \mathcal{S}} \operatorname{Conv}(B(s'))$.

Proof. It is easy to see that $\operatorname{Conv}(B(s)) \supseteq \bigcup_{s' \in \mathcal{S}} \operatorname{Conv}(B(s'))$. To prove the converse, let $s = \langle \tilde{p}_0, \tilde{p}_1, \cdots, \tilde{p}_{k-1} \rangle$, where each $\tilde{p}_i \in H(\hat{\varepsilon})$. If a point $\tilde{u} \in \operatorname{Conv}(B(s))$, then $\tilde{u} = \sum_{i=0}^{k-1} \lambda_i \tilde{p}_i$, where $\lambda_i \geqq 0$. We shall prove that $\tilde{u} \in \operatorname{Conv}(B(s'))$ for some $s' \in \mathcal{S}$. It is trivial if $\tilde{u} = 0$. Otherwise, the point $(1/\sum_i \lambda_i)\tilde{u} \in s = \bigcup_{s' \in \mathcal{S}} s'$, and hence $(1/\sum_i \lambda_i)\tilde{u} \in s'$ for some $s' \in \mathcal{S}$. This implies $\tilde{u} \in \operatorname{Conv}(B(s'))$. □

The above lemmas suggest that we may try to construct a frame with narrow bases, by first constructing a family of simplices all with small diameters. We use the following scheme:

Let $\tilde{e}_i$ denote the unit vector in E^k, whose ith component is 1 and all others are 0.

For each of the 2^k k-tuples $\hat{\varepsilon} = (\varepsilon_1, \varepsilon_2, \cdots, \varepsilon_k)$, where $\varepsilon_i = \pm 1$, do the following.

(a) Let $s = \langle \varepsilon_1 \tilde{e}_1, \varepsilon_2 \tilde{e}_2, \cdots, \varepsilon_k \tilde{e}_k \rangle$. (Clearly $s \subseteq H(\hat{\varepsilon})$.)

(b) Construct a finite family $\mathcal{S}$ of simplices all contained in $H(\hat{\varepsilon})$ such that $s = \bigcup_{s' \in \mathcal{S}} s'$ and $\operatorname{diam}(s') < (2(1-\cos\psi)/k)^{1/2}$ for all $s' \in \mathcal{S}$.

(c) Form $B(s')$ for all $s' \in \mathcal{S}$.

The collection $\mathcal{B}$ of all the $B(s')$ constructed this way is clearly a frame because of Lemma A2. Using Lemma A1, it is easy to verify that $\operatorname{Ang}(B(s')) < \psi$ for all s'. Thus, such a construction would give a frame satisfying the conditions in Lemma 4.2. It remains to show that step (b) above can be carried out.

A procedure in topology ([10, p.209, Thm. 5–20]), known as *barycentric subdivision*, guarantees that step (b) can be accomplished in a finite number of steps. For completeness, we shall give a brief description below.

There is a basis procedure, called *first barycentric subdivision* (FBS), which, for a given j-simplex s, constructs in finite steps a family $\mathcal{S}$ of simplices such that $s = \bigcup_{s' \in \mathcal{S}} s'$ and $\max_{s' \in \mathcal{S}} (\operatorname{diam}(s')) \leqq (j/(j+1))(\operatorname{diam}(s))$. If we apply this procedure iteratively, at each iteration we apply FBS to every simplex present, then all the simplices will have a diameter less than any prescribed positive number after enough iterations. This then constitutes a procedure for step (b).

Finally, we describe the FBS procedure. For a proof that it produces simplices with the desired properties, see [10]. Let $s = \langle \tilde{p}_0, \tilde{p}_1, \cdots, \tilde{p}_j \rangle$, the point $\tilde{c}(s) = (1/(j+1)) \sum_{i=0}^{j} \tilde{p}_i$ is called the *centroid* of simplex s. For any t distinct integers $0 \leqq i_1, i_2, \cdots, i_t \leqq j$, let $\tilde{p}_{i_1 i_2 \cdots i_t} = \tilde{c}(\langle \tilde{p}_{i_1}, \tilde{p}_{i_2}, \cdots, \tilde{p}_{i_t} \rangle)$. For each $\sigma = (i_0, i_1, \cdots, i_j) \in \Sigma$, where Σ is the set of all permutations of $(0, 1, 2, \cdots, j)$, let $s'(\sigma)$ denote the simplex $\langle \tilde{p}_{\sigma_0}, \tilde{p}_{\sigma_1}, \cdots, \tilde{p}_{\sigma_j} \rangle$ with $\sigma_t = i_0 i_1, \cdots, i_t$. The FBS of s is defined by

$$\mathcal{S} = \{ s'(\sigma) \mid \sigma \in \Sigma \}.$$

REFERENCES

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] J. L. Bentley and J. H. Friedman, *Fast algorithms for constructing minimal spanning trees in coordinate spaces*, Rep. STAN-CS-75-529, Stanford Computer Science Department, Stanford, CA, January 1976.

[3] J. L. Bentley and M. I. Shamos, *Divide-and-conquer in multidimensional space*, in Proc. 8th Annual ACM Symposium on Theory of Computing, 1976, pp. 220–230.

[4] C. Berge and A. Ghouila-Houri, *Programming, Games, and Transportation Networks*, John Wiley, New York, 1965.

[5] R. C. Buck, *Partition of space*, Amer. Math. Monthly, 50 (1943), pp. 541–544.

[6] D. Cheriton and R. E. Tarjan, *Finding minimum spanning trees*, this Journal, 5 (1976), pp. 724–742.

[7] E. W. Dijkstra, *A note on two problems in connection with graphs*, Numer. Math., 1 (1959), pp. 269–271.

[8] D. Dobkin and R. J. Lipton, *Multidimensional search problems*, this Journal, 5 (1976), pp. 181–186.

[9] R. O. Dude and P. E. Hart, *Pattern Classification and Science Analysis*, John Wiley, New York, 1973.

[10] J. G. Hocking and G. S. Young, *Topology*, Addison-Wesley, Reading, MA, 1961.

[11] A. Kerschenbaum and R. Van Slyke, *Computing minimum spanning trees efficiently*, in Proc. 25th Annual Conference of the ACM, 1972, pp. 518–527.

[12] D. E. KNUTH, *The Art of Computer Programming, Vol. 3, Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[13] J. B. KRUSKAL, JR7., *On the shortest spanning subtree of a graph and the travelling salesman problem*, Problem. Amer. Math. Soc., 7 (1956), pp. 48–50.

[14] R. C. PRIM, *Shortest connection networks and some generalizations*, Bell System Tech. J., 36 (1957), pp. 1389–1401.

[15] M. I. SHAMOS, *Geometric complexity*, Proc. 7th Annual ACM Symposium on Theory of Computing, 1975, pp. 224–233.

[16] M. I. SHAMOS AND D. J. HOEY, *Closest-point problems*, Proc. 16th Annual IEEE Symposium on Foundations of Computer Science, 1975, pp. 151–162.

[17] A. C. YAO, *An* $O(|E| \log\log |V|)$ *algorithm for finding minimum spanning trees*, Inform. Process. Lett., 4 (1975); pp. 21–23.

[18] ———, *On computing the distance matrix of n points in k dimensions*, IBM San Jose Research Center Technical Report, to appear.

[19] ———, *On the preprocessing cost in multidimensional search*, IBM San Jose Research Center Technical Report, to appear.

[20] A. C. YAO AND F. F. YAO, *On computing the rank function for a set of vectors* Rep. UIUCDCS-R-75-699, Computer Science Department, University of Illinois, Illinois, Februrary, 1975.

[21] C. T. ZAHN, *Graph-theoretical method for detecting and describing gestalt clusters*, IEEE Trans. Comput., C-20 (1970), pp. 68–86.

SIAM J. COMPUT.
Vol. 11, No. 4, November 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1104-0009 $01.00/0

SOME AREA-TIME TRADEOFFS FOR VLSI*

RICHARD P. BRENT† AND LESLIE M. GOLDSCHLAGER‡

Abstract. Area-time bounds on VLSI circuits for context-free language recognition, for the evaluation of propositional calculus formulae and for set equality and disjointness questions, are considered. In all cases, a lower bound $AT^{2\alpha}=\Omega(n^{1+\alpha})$ is proved, where A is the chip area, T the execution time, and $0\leqq\alpha\leqq 1$. Similar results were known for computations with $\Omega(n)$-bit outputs, but the computations considered here have only 1-bit outputs. Upper bounds are also discussed.

Key words. area-time tradeoffs, VLSI, formula evaluation, circuit-value problem, context-free language recognition, set equality, set disjointness, computational complexity, crossing sequences, models of computation

1. Introduction. Very large scale integrated circuit chips present an economic method of building general-purpose parallel machines as well as special-purpose chips which could be used as plug-in modules to conventional computers [9], [15], [17], [18], [21], [22].

The main complexity measures relevant to VLSI technology are the execution time of the algorithm ("parallel time") and the chip area required for the implementation. This chip area consists of both the active processing elements (transistors) and the wires used to interconnect them. If λ is the narrowest wire width allowed by the technology of the day, then chip area can usefully be measured in units of λ^2.

Because speed increases obtained through parallelism often require a large volume of information transfer between the active processing elements, it has frequently been observed that the resource needs of parallel algorithms are dominated by data communication. For VLSI circuits this phenomenon manifests itself in the chip area required for interconnections between active processing elements [15], [22], [23].

It is reasonable to expect that, as one increases the degree of parallelism used to solve a problem in order to decrease the execution time, the required inter-processor communication will increase. In a VLSI setting, we could therefore expect to find tradeoffs between time and chip area. Such tradeoffs have been reported for the discrete Fourier transform, matrix multiplication, Gaussian elimination, transitive closure, sorting and permutation problems, and for integer addition and multiplication [1], [5], [6], [20], [21], [22], [24], [25].

This paper gives area-time tradeoffs for context-free language recognition, finding the truth value of a formula given the values of its variables, determining whether two sets are equal (or disjoint), and determining if all elements of a set have distinct values. These problems differ from those mentioned above in that they have only a one-bit output. After our results were announced in [3], we learned that similar results were obtained independently by Lipton and Sedgewick [14].

Preliminary results and definitions are given in §2, lower bounds on area-time products are given in §3, and upper bounds are considered in §4. Finally, in §5, we mention some open problems.

* Received by the editors January 5, 1981 and in revised form January 15, 1982.

† Department of Computer Science, Australian National University, Canberra, Australia.

‡ Department of Computer Science, University of Queensland, St. Lucia, Australia. Now at Basser Department of Computer Science, Sydney University, Sydney, Australia.

2. Definitions and basic results.

DEFINITION 2.1. Let G denote a context-free grammar and $L(G)$ the language generated by G [11]. Then CFMEMBER $= \{\langle G, s\rangle | s \in L(G)\}$.

DEFINITION 2.2. Let F denote a formula of the propositional calculus in disjunctive normal form (DNF), and let ϕ denote a truth assignment to the variables of F. Then FVAL $= \{\langle F, \phi\rangle | F$ is true under $\phi\}$.

DEFINITION 2.3. A circuit α is a sequence $(\alpha_1, \cdots, \alpha_n)$ where each α_i is either a variable $x_1, x_2, \cdots$ or a gate AND (j, k), OR (j, k) or NOT (j) where $j \leqq k < i$. Let ϕ be a truth assignment to the variables of α, and ϕ' be the natural extension of ϕ to the gates of α [13]. Then the circuit value problem CVP $= \{\langle \alpha, \phi\rangle | \phi'(\alpha_n) = \text{true}\}$.

Let $X = (x_1, \cdots, x_n)$ and $Y = (y_1, \cdots, y_n)$ be sequences of integers in the range 0 to $n-1$ and $N = \{1, 2, \cdots, n\}$.

DEFINITION 2.4.

$$\text{DISTINCT} = \{\langle X\rangle | \text{ for all } i, j \in N, i \neq j \Rightarrow x_i \neq x_j\},$$

$$\text{DISJOINT} = \{\langle X, Y\rangle | \text{ for all } i, j \in N, x_i \neq y_j\},$$

$$\text{NONDISJOINT} = \{\langle X, Y\rangle | \text{ for some } i, j \in N, x_i = y_j\},$$

$$\text{EQUAL} = \{\langle X, Y\rangle | a \in X \text{ iff } a \in Y\}.$$

The VLSI model we use is that of [6]; see also [1], [2], [4], [5], [20], [21], [22], [23]. Essentially, the model is a collection of processing elements, some of which receive inputs or produce outputs, and which are connected by wires with finite width. For completeness we state the model here. Comments and justification are given following the statement of each assumption A1–A9. Our lower bound results are insensitive to minor details of the model.

A1. The computation is performed in a convex planar region R of area A.

Because of heat dissipation, packing, and testing requirements, a two-dimensional planar model is reasonable. The convexity assumption is not restrictive in the sense that almost all existing chips or useful modular designs do have convex boundaries for packaging or modularity reasons.

A2. Wires have minimal width $\lambda > 0$.

λ is assumed constant, but in applications of our results it will of course depend on the technology. We also assume R has width at least λ in every direction.

A3. At most $\nu \geqq 2$ wires can overlap (or intersect) at any point in R.

A chip may consist of ν layers. Wire crossings through different layers are allowed. In fact, transistors are typically formed by crossovers of wires. Since $\nu \geqq 2$, the graph of wires (edges) and gates (nodes) need not be planar in a graph-theoretic sense.

A4. I/O ports each contain a $\lambda \times \lambda$ square and thus have area at least $\rho \geqq \lambda^2$. An I/O port can be multiplexed to handle more than one input or output variable.

If R is a complete chip, ρ will be large compared to λ^2. If R is only part of a chip and I/O is to other regions on the chip, ρ could be of order λ^2. We do not require each input (or output) variable to appear on a distinct input (or output) port, as required in [21]. I/O ports may be multiplexed as they often are in practice.

A5. A bit requires minimal time $\tau > 0$ to propagate along a wire or to be transmitted through an I/O port. The time for one gate computation and an arbitrary fanout of the result is included in τ.

Since dimensions are limited by the minimal wire width λ and minimal gate area, a minimal propagation time is reasonable. We do not need to assume that the

propagation time increases with the length of the wire. With the (small) sizes of chips we now have or anticipate, the propagation time, which is the time needed to charge or discharge a wire, is limited by the wire capacitance rather than the velocity of light. A longer wire will generally have a larger capacitance and thus require a larger driver to maintain constant propagation time, but the driver area need not exceed a fixed percentage of the wire area and so can be ignored if λ is increased slightly. A thorough discussion of this point may be found in [2]. Although it would be reasonable to assume bounded fanout, we do not need this assumption for proving lower bounds. When proving upper bounds in §4, we do assume bounded fanout.

A6. The times and locations at which input and output bits are available are fixed and independent of the values of the input bits.

In the terminology of [14], the input-output schedule is "where-oblivious" and "when-oblivious".

A7. Storage for one bit of information takes area at least $\beta > 0$.

β is typically several times larger than λ^2.

A8. Each input bit is available only once.

There is no free memory outside R. If the same input bit is required at different times, it must be stored within R, taking area at least β (see A7).

A9. Computation is synchronous with clock period at least τ.

This assumption, not required in [5], [6], simplifies the definition of "crossing sequence" given below.

DEFINITION 2.5. If V is any subset of the processing elements (i.e. gates and I/O ports), then a *bisection* of V is a cut of V into disjoint sets B and C such that, for some chord L perpendicular to a diameter D of the chip, the elements in B lie (at least in part) on one side of L and those in C lie (at least in part), on the other side of L.

As in [5], [6], we shall assume that processing elements can be shrunk to points and the chord L perturbed slightly so that L intersects only wires and not processing elements.

DEFINITION 2.6. Given a bisection as above, the (maximal) *information transfer* across the cut during a computation of time T is $I = WT/\tau$, where W is the number of wires which intersect L.

(Informally, I is the maximum number of bits which can be transferred in either direction between the processing elements in B and those in C. This definition is closer to those implicit in [5], [6] than those given in [1], [21], [22].)

DEFINITION 2.7. The *crossing sequence* associated with a given bisection and computation is the sequence of (binary) values on the wires crossing the cut (with some fixed ordering) at intervals of the clock period during the computation.

THEOREM 2.8. *Given a set V of processing elements, if there is a bisection of V with information transfer I, then*

$$AT^2 = \Omega(I^2).$$

Proof. Let L be the length of the chord associated with the bisection of V. Then, by Assumptions A1–A3, with W as in Definition 2.6, $A = \Omega(L^2)$ and $L \geqq \lambda W/\nu$, so $AT^2 = \Omega(W^2T^2) = \Omega(I^2)$.

(The proof is similar to that of Thompson [21], [22] except that, because of our Assumption A1 and Definition 2.6, we do not need to use the concept of "bisection width" or to minimize over a class of bisections. Our argument is used in the proof of [6, Thm. 3.1].)

Results similar to Theorem 2.8 have been used in the following way [1], [5], [6], [21], [22]: one considers the bisection of V into two regions B and C with a split of inputs $x_1, \cdots, x_n$ between them and an arbitrary distribution of outputs $y_1, \cdots, y_m$. Assuming that $|Y_C| \geqq |Y_B|$, the idea is to prove that there must be a sizeable information transfer I between X_B and Y_C. If one can show that I is proportional to n, then an area-time tradeoff of the form $AT^2 = \Omega(n^2)$ follows from Theorem 2.8.

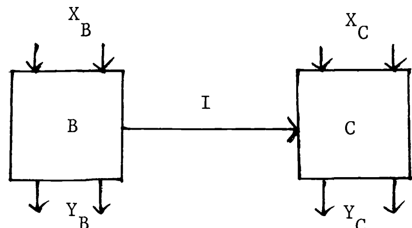


FIG. 2.1. *One-way information flow.*

For the problems considered in this paper, there is only one output, whose Boolean value answers the membership question posed by the instance of the problem which appears on the inputs. When the circuit produces only one output, the proof techniques required to bound the information transfer differ from the techniques used in [2], [5], [6], [20], [21], [22], [23], [24], [25]. In particular, it is necessary to consider the information transfer in both directions I_B and I_C (see Fig. 2.2). The total information

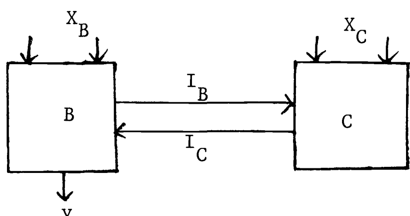


FIG. 2.2. *Two-way information flow.*

transfer $I = I_B + I_C$. Our technique may be regarded as a hybrid of the "crossing sequence" argument used to obtain lower bounds for Turing machine computations [10] and the "information flow" arguments previously used to obtain lower bounds on AT^2 for VLSI computations. All such techniques have the inherent limitation of yielding lower bounds on AT^2 no larger than cn^2, since all the inputs and outputs can be trivially redistributed with I proportional to n. This limitation results from ignoring the processing performed in B and C.

3. Lower bounds.

THEOREM 3.1. *For any* VLSI *circuit of area A which accepts as input a propositional calculus formula in disjunctive normal form with up to n literals and a set of truth values of its variables, and has as output the truth value of the formula in* (*worst-case*) *time T,*

$$AT^2 = \Omega(n^2).$$

Proof. Without loss of generality we assume that the formula has exactly n literals, $n/2$ variables, and that n is divisible by 8. Let M be the maximum number of input variables sharing (or multiplexing) one input port. If $M \geqq n/4$ the result is immediate, for $T = \Omega(M)$. Thus, we may assume that $M < n/4$. It follows that there is a bisection of the circuit into two parts B and C so that the input ports for between $n/8$ and $3n/8$ of the variables x_i are in B, and those for the remaining variables are in C. We denote the variables in B by $b_1, \cdots, b_k, \cdots$ and those in C by $c_1, \cdots, c_k, \cdots$, where $k = n/8$, and write $\mathbf{b} = (b_1, \cdots, b_k)$, $\mathbf{c} = (c_1, \cdots, c_k)$. The $2k$ variables not in $\mathbf{b}$ or $\mathbf{c}$ are denoted by $\mathbf{d} = (d_1, \cdots, d_{2k})$. Assume without loss of generality that the output port is in B.

Consider the formula

$$f(\mathbf{b}, \mathbf{c}) = \bigvee_{i=1}^{k} (b_i \& \bar{c}_i) \bigvee_{i=1}^{k} (\bar{b}_i \& c_i) \bigvee_{i=1}^{2k} (d_i \& \bar{d}_i)$$

which is in disjunctive normal form with $8k = n$ literals and $4k = n/2$ variables. (Intuitively, $f(\mathbf{b}, \mathbf{c})$ returns the truth value of $\mathbf{b} \neq \mathbf{c}$. By assumption A6, the bisection into B and C is independent of f.) Let I be the information transfer between B and C during a computation of $f(\mathbf{b}, \mathbf{c})$. Assume, by way of contradiction, that $I < k$. There are at most $2^I < 2^k$ crossing sequences (of bits across the cut), but there are 2^k possible values of $\mathbf{b}$, so there exist distinct $\mathbf{b}^{(1)}, \mathbf{b}^{(2)}$ such that the computations of $f(\mathbf{b}^{(1)}, \mathbf{b}^{(1)})$ and $f(\mathbf{b}^{(1)}, \mathbf{b}^{(2)})$ give identical crossing sequences. It follows that the circuit computes identical values for $f(\mathbf{b}^{(1)}, \mathbf{b}^{(1)})$ and $f(\mathbf{b}^{(1)}, \mathbf{b}^{(2)})$, contrary to the definition of f. Thus $I \geqq k = \Omega(n)$, and the result follows from Theorem 2.8.

THEOREM 3.2. *With the assumptions of Theorem* 3.1, $A = \Omega(n)$.

Proof. Without loss of generality we may assume that there are $n/2$ variables and that n is divisible by 8. Suppose that there are P input ports. If $P \geqq n/4$ the result is immediate, for $A = \Omega(P)$. Thus, we may assume that $P < n/4$. At most P truth values can be read in simultaneously, so at some time during the computation, the truth values of a subset B of variables will have been read in, where $n/8 \leqq |B| \leqq 3n/8$. Let C be the complementary set of variables, and suppose $B = \{b_1, \cdots, b_k, \cdots\}$, $k = n/8$. Using the same function f as in proof of Theorem 3.1, we see that the output $f(\mathbf{b}, \mathbf{c})$ distinguishes between all 2^k possible vectors $\mathbf{b} = (b_1, \cdots, b_k)$ for suitable choice of $\mathbf{c} = (c_1, \cdots, c_k)$. Thus, the circuit must have at least 2^k internal states, i.e., it must store at least $k = \Omega(n)$ bits of information. The result now follows from Assumptions A7 and A8.

THEOREM 3.3. *With the assumptions of Theorem* 3.1,

$$AT^{2\alpha} = \Omega(n^{1+\alpha}) \quad \textit{for all } \alpha \in [0, 1].$$

Proof. The result follows by combining the bounds of Theorems 3.1 and 3.2 (as in the proof of [6, Thm. 3.3]).

COROLLARY 3.4. *For* VLSI *circuits which evaluate Boolean formulae in general (not necessarily disjunctive normal) form,*

$$AT^{2\alpha} = \Omega(n^{1+\alpha}) \quad \textit{for all } \alpha \in [0, 1].$$

COROLLARY 3.5. *For* VLSI *computation of the circuit value problem* CVP,

$$AT^{2\alpha} = \Omega(n^{1+\alpha}) \quad \textit{for all } \alpha \in [0, 1].$$

THEOREM 3.6. *Any* VLSI *circuit which computes* DISTINCT *has* $AT^{2\alpha} = \Omega(n^{1+\alpha})$ *for all* $\alpha \in [0, 1]$, *where n is the number of elements in the input sequence.*

Proof. Call an element of an input sequence a *block*, and note that each block consists of at least $\lceil \log_2 n \rceil$ bits.

We proceed much as in the proof of Theorem 3.1, taking $k = n/4$, and consider any bisection of the least significant input bits of each block into two parts B and C, $k \leqq |B| \leqq 3k$. Let $\mathbf{b} = (b_1, \cdots, b_k)$ denote a subset of the bits in B, and similarly for $\mathbf{c} = (c_1, \cdots, c_k)$. Now for each i, $1 \leqq i \leqq k$, if b_i or c_i is the least significant bit of block x_i, then assume that the other bits of x_i equal the binary representation of $i-1$. Let I be the information transfer between B and C. If $I < k$, then there exist distinct $\mathbf{b}^{(1)}$, $\mathbf{b}^{(2)}$ such that $\mathbf{b} = \mathbf{b}^{(1)}$ and $\mathbf{c} = \bar{\mathbf{b}}^{(1)}$ results in the same crossing sequence across the cut as $\mathbf{b} = \mathbf{b}^{(2)}$ and $\mathbf{c} = \bar{\mathbf{b}}^{(2)}$, where $\bar{\mathbf{b}}$ is $(\bar{b}_1, \cdots, \bar{b}_k)$. So the circuit computes the same output for $\mathbf{b} = \mathbf{b}^{(1)}$ and $\mathbf{c} = \bar{\mathbf{b}}^{(1)}$, as for $\mathbf{b} = \mathbf{b}^{(1)}$ and $\mathbf{c} = \bar{\mathbf{b}}^{(2)}$. But this is a contradiction, as in the former case all blocks are distinct, while in the latter there is at least one pair of blocks which have identical input values. Hence $I \geqq k$.

The remainder of the proof is similar to the proofs of Theorems 3.1–3.3.

THEOREM 3.7. *Any* VLSI *circuit which computes* DISJOINT (or NONDISJOINT) *has* $AT^{2\alpha} = \Omega(n^{1+\alpha})$ *for all* $\alpha \in [0, 1]$, *where n is the number of elements in the input sequences.*

Proof. As in the proof of Theorem 3.6, taking $k = n/8$, consider any bisection of the least significant input bits of each block into two parts B and C, $2k \leqq |B| \leqq 6k$. At least half of the least significant bits of X must be in one part (say B), and thus at least half of the least significant bits of Y must be in the other part (C). Let $\mathbf{b}$ and $\mathbf{c}$ (respectively) denote k-subsets of these bits, assume that the most significant bits of their corresponding blocks are as above, and that all remaining blocks in X equal $2k$ and in Y equal $2k+1$. Choose $\mathbf{b}^{(1)}$ and $\mathbf{b}^{(2)}$ as above. Now $\mathbf{b} = \mathbf{b}^{(1)}$ and $\mathbf{c} = \bar{\mathbf{b}}^{(1)}$ produce the same output as $\mathbf{b} = \mathbf{b}^{(1)}$ and $\mathbf{c} = \bar{\mathbf{b}}^{(2)}$, but in the former case X and Y are disjoint, whereas in the latter they have some element in common.

THEOREM 3.8. *Any* VLSI *circuit which computes* EQUAL *has* $AT^{2\alpha} = \Omega(n^{1+\alpha})$ *for all* $\alpha \in [0, 1]$, *where n is the number of elements in the input sequences.*

Sketch of proof. The proof is similar to that of Theorem 3.7, but now $\mathbf{b} = \mathbf{b}^{(1)}$ and $\mathbf{c} = \mathbf{b}^{(1)}$ produces the same crossing sequence as $\mathbf{b} = \mathbf{b}^{(2)}$ and $\mathbf{c} = \mathbf{b}^{(2)}$, and so the circuit produces the same output for $\mathbf{b} = \mathbf{b}^{(1)}$ and $\mathbf{c} = \mathbf{b}^{(1)}$ as for $\mathbf{b} = \mathbf{b}^{(1)}$ and $\mathbf{c} = \mathbf{b}^{(2)}$.

Remark. The proof of Theorem 3.8 breaks down if we allow probabilistic algorithms which have a positive (albeit very small) probability of error [M. O. Rabin, personal communication, August 1981].

THEOREM 3.9. *Let* $m = O(n)$. *There is a context-free grammar G with the following property. Any* VLSI *circuit which, given a string* $s = s_1 s_2 \cdots s_m$, *determines whether*

$$\langle G, s \rangle \in \text{CFMEMBER},$$

has $AT^{2\alpha} = \Omega(n^{1+\alpha})$ *for all* $\alpha \in [0, 1]$.

Proof. Let G be the context-free grammar with start symbol S, nonterminal symbols D, E, F, terminal symbols 0, 1, \$, ¢, and productions

$$S \to EF¢E$$

$$F \to 0F0|1F1|¢E\$E$$

$$E \to ED¢|\varepsilon$$

$$D \to D0|D1|\varepsilon \qquad \text{(where } \varepsilon \text{ is the empty string).}$$

It is easy to see that G generates the language

$$L(G) = \{x_1¢\cdots x_p¢\$y_1^R¢\cdots y_q^R¢ \mid p > 0, q > 0, x_i = y_j \text{ for some } i \leqq p, j \leqq q\}$$

where each x_i, y_j is a (possibly empty) string of binary digits and y_j^R is the reverse of y_j. $L(G)$ is just a generalization of NONDISJOINT. Hence, a circuit which can determine if $\langle G, s\rangle \in$ CFMEMBER, i.e., if $s \in L(G)$, can also determine if $\langle X, Y\rangle \in$ NONDISJOINT provided $\langle X, Y\rangle$ is encoded as the string $x_1¢ \cdots x_n¢\$y_1^R¢ \cdots y_n^R¢$. Hence, the result follows from Theorem 3.7.

4. Upper bounds. The following theorem shows that, for several of the problems considered in §3, the exponent of n in the lower bound $AT^2 = \Omega(n^2)$ is the best possible.

THEOREM 4.1. *There is a* VLSI *circuit which solves the problems of Theorem* 3.1, 3.6, 3.7 *or* 3.8 *with*

$$A = O(n^2 \log^3 n)$$

and

$$T = O(\log n).$$

Proof. We shall consider the problem of Theorem 3.1 (evaluating propositional calculus formulae in disjunctive normal form with up to n literals); the problems of computing DISTINCT, DISJOINT and EQUAL may be dealt with in a similar fashion.

A literal is a variable or its negation, so any formula with at most n literals has at most n variables. We shall construct a circuit for which there are n input ports along the north edge, the ith such port giving the Boolean value of the variable x_i, for $i = 1, 2, \cdots, n$. A literal is of the form x_i or $\bar{x}_i$, so can be encoded by $\lceil \log_2 n \rceil + 1$ bits. Since the formula is in disjunctive normal form, it can be encoded simply as a string of literals separated by encoded operators "$\&$" and "V". The encoded formula will enter the west edge of our circuit through $O(n \log n)$ input ports. The overall layout is illustrated in Fig. 4.1.

To simplify the description we first assume unbounded fan-out. Consider the evaluation of a single literal $p_i = N_i x_i$, where N_i may be either the identity or negation operator. It is easy to construct a circuit, of area $O(n \log n)$ and delay $O(\log n)$, which has a grid of n lines running north to south (of which the ith is to be selected), and along the south edge a fan-in tree whose result may be complemented (depending on N_i) and fed out along the east edge. This is illustrated in Fig. 4.2.

It remains to specify an evaluation tree which inputs the value of each literal and the operators, and outputs the value of the formula. This is illustrated in Fig. 4.3, where the wires between each node transmit six bits $x\theta_x y\theta_y z\theta_z$, where x, y and z can be 0 or 1 and θ_x, θ_y and θ_z can be encodings of $\&$ or V.

Initially, the ith node of the evaluation tree carries the signal $1 \,\&\, 1 \,\&\, p_i\theta_i$, where p_i is the value of the ith literal and θ_i the operator to its right (irrelevant if $i = n$).

Each node of the evaluation tree takes two six-bit signals f_1 and f_2 representing partially evaluated formulae, and evaluates as much as possible of the concatenated formula $f_1 f_2$. It is straightforward to verify by induction that each six-bit signal must have one of the forms

$$xVyVz\&,$$

$$1\&yVz\theta,$$

or

$$1\&1\&z\theta$$

where x, y, z are 0 or 1 and θ is $\&$ or V.

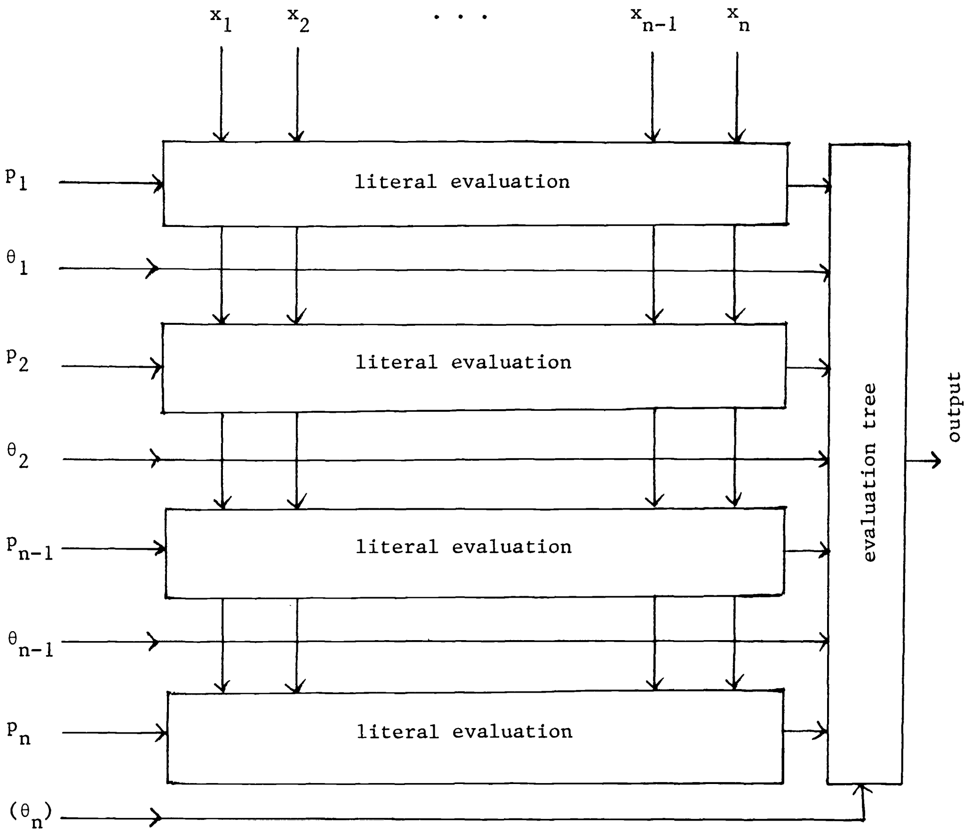


FIG. 4.1. *Overall layout.*

The root of the evaluation tree evaluates the final six-bit signal to give the values of the formula.

The circuit constructed above has width $O(n)$, height $O(n \log n)$, and delay $O(\log n)$. However, we have neglected the problem of fanout. In order to reduce the fanout of the input lines entering along the west edge (i.e., the literal encodings) we must separate each line by a distance $O(\log n)$ to accommodate fanout trees of width $O(n)$ and height $O(\log n)$ (see [4]). This makes the height of the complete circuit $O(n \log^2 n)$. A similar problem arises with the lines (carrying variable values) which enter along the north edge and must be fanned out to the sub-circuits evaluating each literal. We separate each line by a distance $O(\log n)$ to allow space for fanout trees. Thus, the circuit has been "stretched" by a factor $O(\log n)$ in both directions, giving it a width of $O(n \log n)$, height of $O(n \log^2 n)$, and area $O(n^2 \log^3 n)$. (A more detailed analysis might lower the exponent of $\log n$, but we are more interested in the exponent of n, which is optimal by the results of §3.)

Remark 4.2. The problems of evaluating DISTINCT, DISJOINT and EQUAL can clearly be solved rather easily if a sorting circuit is available. Hence, upper bounds on the area and time required for these problems may be obtained from the corresponding upper bounds for sorting networks. In particular, these problems can be solved with $AT^{2\alpha} = O(n^{1+\alpha} \log^c n)$ for some constant c. See, for example, Thompson [23].

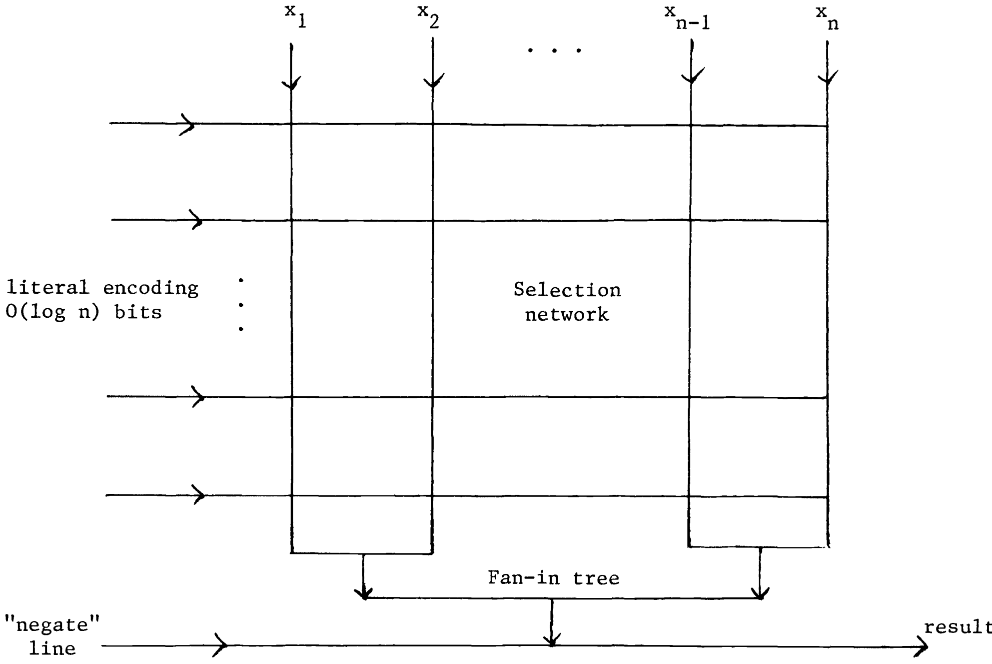


FIG. 4.2. *Literal evaluation (fan-out restrictions ignored).*

In [5], [6] it was shown that the exponent $1+\alpha$ in the lower bound $AT^{2\alpha}=\Omega(n^{1+\alpha})$ was sharp (for all $\alpha \in [0, 1]$) for the problem of binary multiplication of n-bit numbers. Except for the cases covered by Theorem 4.1 and Remark 4.2, and the trivial case $\alpha=0$, we do not know if the same is true for the problems considered in §3. In fact, this is unlikely for the context-free language recognition problem, as the best known serial algorithm uses n by n matrix multiplication [11], [16], [20]. Combining ideas of Preparata and Vuillemin [18] and Ruzzo [19], it may be shown that the context-free language problem can be solved by a VLSI circuit having $T=O(\log^2 n)$ and $A=O(n^c)$ for some constant c. (We believe that $c \leqq 8$, but have not yet worked out all the details. c can be reduced by the method of [12], at the expense of increasing T to $O(n)$.) Recognizing regular expressions appears to be a much easier problem [7], [8].

5. Conclusion. We have given lower bounds $AT^{2\alpha}=\Omega(n^{1+\alpha})$ for several natural recognition problems. The case $\alpha=\frac{1}{2}$ is interesting, as the area-time product AT can be viewed as the "rental cost" for a VLSI chip for the duration of the computation. On the other hand, for VLSI circuits which allow pipelining, AT overestimates the cost since portions of the chip can be re-used for subsequent computations before earlier computations are completed. In these cases the area alone (i.e., the case $\alpha=0$) may be a better measure of cost.

Possible areas for future research include:

a) Finding a technique to prove lower bounds better than $AT^{2\alpha}=\Omega(n^{1+\alpha})$.

b) Obtaining sharper upper bounds for problems of practical interest, e.g., the context-free language recognition problem.

each line carries 6 bits

result

FIG. 4.3. *Final evaluation tree.*

c) Extending our results to probabilistic algorithms (see the remark following the proof of Theorem 3.8).

Acknowledgments. We are grateful to Al Borodin for suggesting the set equality and disjointness problems, and for fruitful discussions. We also thank the referees and Professor W. L. Ruzzo for their comments which helped us to sharpen Theorem 3.9 and to correct the proof of Theorem 4.1.

REFERENCES

[1] H. ABELSON AND P. ANDREAE, *Information transfer and area-time tradeoffs for* VLSI *multiplication*, Comm. ACM, 23 (1980), pp. 20–23.

[2] G. BILARDI, M. PRACCHI AND F. P. PREPARATA, *A critique and appraisal of* VLSI *models of computation*, VLSI Systems and Computations, H. T. Kung et al., eds., Computer Science Press, Rockville, MD, 1981, pp. 81–88.

[3] R. P. BRENT AND L. M. GOLDSCHLAGER, *Some area-time tradeoffs for* VLSI, Report 22, Dept. Computer Science, University of Queensland, Australia, August 1980.

[4] R. P. BRENT AND H. T. KUNG, *On the area of binary tree layouts*, Inform. Proc. Letters, 11 (1980), pp. 46–68.

[5] ——— *The chip complexity of binary arithmetic*, Proc. 12th Annual ACM Symposium on Theory of Computing, New York, April 1980, pp. 190–200.

[6] ——— *The area-time complexity of binary multiplication*, J. Assoc. Comput. Mach., 28 (1981), pp. 521–534.

[7] R. W. FLOYD AND J. D. ULLMAN, *The compilation of regular expressions into integrated circuits*, Proc. 21st Annual IEEE Symposium on Foundations of Computer Science, New York, 1980, pp. 260–269.

[8] M. J. FOSTER AND H. T. KUNG, *Recognize regular languages with programmable building blocks*, VLSI 81, J. P. Gray, ed., Academic Press, New York, 1981, pp. 75–84.

[9] L. J. GUIBAS, H. T. KUNG AND C. D. THOMPSON, *Direct* VLSI *implementation of combinatorial algorithms*, Proc. Conference on VLSI: Architecture, Design, Fabrication, California Inst. of Technology, Jan. 1979.

[10] F. C. HENNIE, *On-line Turing machine computations*, IEEE Trans. Electronic Computers, EC-15 (1966), pp. 35–44.

[11] J. E. HOPCROFT AND J. D. ULLMAN, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.

[12] S. R. KOSARAJU, *Speed of recognition of context-free languages by array automata*, this Journal, 4 (1975), pp. 331–340.

[13] R. E. LADNER, *The circuit value problem is log space complete for* P, SIGACT News, 7, 1, Jan 1975, pp. 18–20.

[14] R. J. LIPTON AND R. SEDGEWICK, *Lower bounds for* VLSI, Proc. 13th Annual ACM Symposium on Theory of Computing, New York, 1981, pp. 300–307.

[15] C. A. MEAD AND L. C. CONWAY, *Introduction to* VLSI *Systems*, Addison-Wesley, Reading, MA, 1980.

[16] V. YA. PAN, *New fast algorithms for matrix operations*, this Journal, 9 (1980), pp. 321–342.

[17] F. P. PREPARATA AND J. E. VUILLEMIN, *The cube-connected-cycles: a versatile network*, Proc. 20th IEEE Symposium on Foundations of Computer Science, New York, Oct. 1979, pp. 140–147.

[18] ——— *Area-time optimal* VLSI *networks for multiplying matrices*, Inform. Proc. Letters, 11 (1980), pp. 77–80.

[19] W. L. RUZZO, *On uniform circuit complexity*, Proc. 20th IEEE Symposium on Foundations of Computer Science, Oct. 1979, pp. 312–318.

[20] J. E. SAVAGE, *Area-time tradeoffs for matrix multiplication and related problems in* VLSI *models*, J. Comput. System Sci., 22 (1981), pp. 230–242.

[21] C. D. THOMPSON, *Area-time complexity for* VLSI, Proc. 11th Annual ACM Symposium on Theory of Computing, New York, April, 1979, pp. 81–88.

[22] ——— *A complexity theory for* VLSI, Report TR-CS-80-140 (Ph.D. thesis), Dept. Computer Science, Carnegie-Mellon University, Pittsburgh, August 1980.

[23] ——— *The* VLSI *complexity of sorting*, in VLSI Systems and Computations, H. T. Kung et al., eds., Computer Science Press, Rockville, MD, 1981, pp. 108–118.

[24] J. E. VUILLEMIN, *A combinatorial limit to the computing power of* VLSI *circuits*, Proc. 21st Annual IEEE Symposium on Foundations of Computer Science, New York, 1980, pp. 294–300.

[25] A. C. YAO, *The entropic limits on* VLSI *computations*, Proc. 13th Annual ACM Symposium on the Theory of Computing, New York, 1981, pp. 308–311.

SIAM J. COMPUT.
Vol. 11, No. 4, November 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1104-0010 $01.00/0

A PARTIAL ANALYSIS OF HEIGHT-BALANCED TREES UNDER RANDOM INSERTIONS AND DELETIONS*

KURT MEHLHORN†

Abstract. We describe a fringe analysis of AVL-trees (and 2-3-trees and HB-trees) under random insertions and deletions. Previously, only the case of random insertions was dealt with.

Key words. AVL-trees, random insertions and deletions, fringe analysis.

1. Introduction. Balanced tree schemes such as 2-3-trees [1], B-trees [2], AVL-trees [12] and BB[α]-trees [17] are very popular and useful data structures for manipulation of ordered lists of keys. Their worst case behavior for a single search, insertion or deletion is well understood (Knuth [12], Mehlhorn [14]). Recently, there has been progress in the study of the worst case behavior of sequences of such operations (Blum and Mehlhorn [3], Brown and Tarjan [6], Mehlhorn [16], Huddleston and Mehlhorn [9]); in these papers bounds on the total cost of rebalancing operations required to process a sequence of searches, insertions and deletions are derived. In contrast, very little is known about the average case behavior of balanced tree schemes, i.e., about their behavior for random inputs. To analyse the storage utilization in 2-3-trees and B-trees under random insertions, Yao [19] introduced the concept of fringe analysis. Subsequently, Brown [4] applied this type of analysis to AVL-trees under random insertions.

In this paper, we will carry out a fringe analysis of AVL-trees and 2-3-trees under random insertions and deletions. Our model of randomness is the following: At any instant of time, we are equally likely to perform an insertion or deletion, and each external node (leaf) is equally likely to be split in the case of an insertion or removed in the case of a deletion.

AVL-trees can be used to store ordered sets S of keys. In terms of the key space, our randomness assumption may be formulated as follows: At any instant of time, we are equally likely to perform an insertion or deletion. In the case of an insertion, the new key goes into each of the gaps between the keys present which equal probability. In the case of a deletion, each key present is chosen for deletion with equal probability. This assumption corresponds to assumption (I_0, D_r) in Knuth [13], and was used before in Flajolet et al. [8]. As shown by Knott's phenomenon [11], our assumption is not equivalent to the condition that keys are drawn independently from a uniform distribution. The latter assumption, called (I_r, D_r) in [13], leads to an extremely involved analysis even for very simple algorithms (Jonassen and Knuth [10]).

The fringe of a tree is obtained by deleting all nodes which are not close to the leaves, e.g., by deleting all nodes of height $\geqq k$ or by deleting all nodes which have at least k leaves below them. Note that only a small percentage of the total number of nodes will be deleted in this way, and hence fringe analysis can give valuable insights. The fringe is a system of trees of small height; in particular, it is a system of trees drawn from a finite collection C of trees. The composition of the fringe can be described by counting, for each tree of that finite collection, the number of times it occurs in the fringe. We can now study the effect of random insertions and deletions on the composition of the fringe.

* Received by the editors December 19, 1979, and in revised form February 12, 1982.

† Fachbereich 10, Angewandte Mathematik und Informatik, Universität des Saarlandes, 6600 Saarbrücken, West Germany.

If our finite collection C of trees is "closed", i.e., if the effect of an insertion or deletion on a tree $T \in C$ is determined by T alone and transforms T into other (maybe several) trees of C, then the effect of an insertion or deletion on the composition of the fringe can be studied without reference to the entire tree. Hence the study of random trees reduces to the study of a relatively simple Markov chain.

If *insertion is the only operation* under consideration, then closed classes are known for some balanced tree schemes. The subtrees of height k form a closed class for 2-3-trees and B-trees (Yao [19], Brown [5]). The subtrees with three or fewer leaves form a closed class for AVL-trees (Brown [4]). No other closed classes are known for AVL-trees. Even worse, if we use B-trees with the overflow mechanism (Bayer and McCreight [2]), i.e., if a node is only split if the brother pages are also full, then no closed classes exist.

If insertions and deletions are the operations under consideration, then the situation is completely hopeless. Closed classes do not exist for any balanced tree scheme.

In this paper we show how to do fringe analysis without the burden of finding a closed class of subtrees, and (partially) analyse AVL-trees under random insertions and deletions. The same approach can be applied to obtain a higher order analysis of AVL-trees under insertions only (Mehlhorn [15]), and to B-trees with the overflow mechanism (Eisenbart and Mehlhorn [7]).

The idea behind our approach is quite simple. Suppose that the fringe consists of two types of trees, type I and type II, and suppose further that the effect of a deletion from a type I tree depends on the environment of that tree in the entire tree, say whether the brother is type II or not. Since the information about the type of the brother is lost when we pass to the fringe, we introduce an (unknown) probability for the event that the brother of a type I tree is a type II tree. A key observation is the fact that this probability, though unknown, cannot assume arbitrary values between 0 and 1 (Lemma 6 below). Recurrence relations describing the effect of a random insertion or deletion on the composition of the fringe are then derived in the standard way, i.e., by recurrence relations for the behavior of a Markov chain with unknown transition probabilities (§ 2). These recurrence relations can be solved very easily under the (unjustified) assumption that the probability mentioned above does not depend on the size of the trees. Fortunately, one can show that these solutions are also valid without that assumption (§ 3). In § 4 we use our results to obtain bounds on the number of balanced nodes in random AVL-trees, and on the total number of nodes in random 2-3-trees and HB-trees.

2. A partial analysis of AVL-trees. Let T be an AVL-tree. The *fringe* of T is obtained by deleting all nodes which have more than three leaves below them. (See Fig. 1 for an example.) This definition of fringe is due to Brown [4]. The fringe of an AVL-tree consists of two types of trees, subtrees with 3 leaves form the first type and

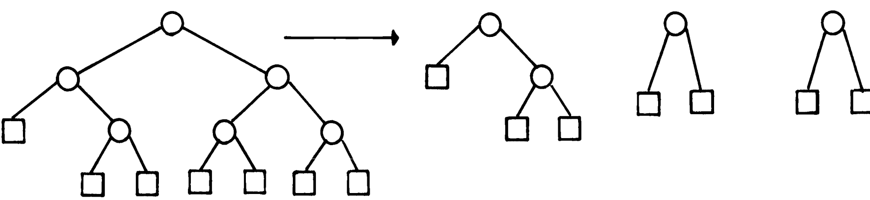


FIG. 1. An AVL-*tree and its fringe.*

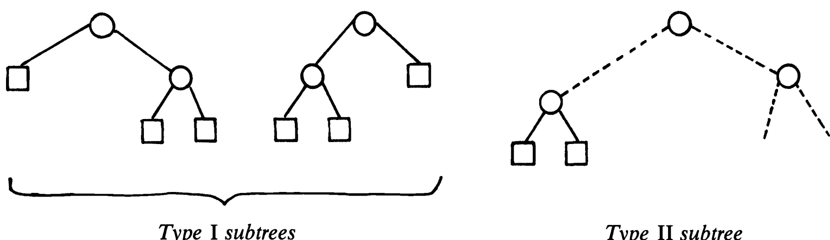


FIG. 2. *The subtrees in the fringe.*

subtrees with 2 leaves form the second type (see Fig. 2). Note that the brother of the root of a type II subtree is an internal node and therefore has two sons. Brown calls type I and type II subtrees M- and N-subtrees respectively.

DEFINITION 1. For an AVL-tree T let $a_1(T)$ $(a_2(T))$ be the number of type I (type II) subtrees in the fringe of T.

LEMMA 1. *Let T be an AVL-tree with n leaves. Then* $3a_1(T)+2a_2(T)=n$.

Proof. A type I subtree has 3 leaves and a type II subtree has 2 leaves, and every leaf is in a type I subtree or a type II subtree. □

Next we need to study the effect of an insertion into (or a deletion from) an AVL-tree T on the number of type I and type II subtrees. We need some notation first. The height of a node is the length (= number of edges) of the longest path to a leaf. The height of a tree is the height of its root. The balance factor $b(v)$ of an internal node v is the height of the left son of v minus the height of the right son of v. Finally, we assume that the insertion and deletion algorithms are as described in Knuth [12].

LEMMA 2. *Let T be an AVL-tree. Suppose that a new leaf is inserted into T and tree T' is obtained after rebalancing.*

a) *If the insertion is into a type* I *subtree, then* $a_1(T')=a_1(T)-1$ *and* $a_2(T')=a_2(T)+2$.
b) *If the insertion is into a type* II *subtree, then* $a_1(T')=a_1(T)+1$ *and* $a_2(T')=a_2(T)-1$.

Proof. This result is proved in Brown [4]. □

Deletion is somewhat harder to deal with. We first prove a general lemma on the effect of a rotation or double rotation on the composition of the fringe.

LEMMA 3. *Let T_1 and T_2 be AVL-trees of height $h+2$ and h respectively. Let u be a new node and consider the tree T consisting of root u, left subtree T_1 and right subtree T_2. Either a rotation or double rotation about u transforms T into an AVL-tree T'.*

If $h \geqq 1$ then $a_1(T')=a_1(T)$ *and* $a_2(T')=a_2(T)$.

Proof. Let T_{11} and T_{12} be the left and right subtrees of T_1. At least one of them has height $h+1$; the other one has height $h+1$ or h.

Case 1. T_{11} has height $h+1$. Then a rotation (see Fig. 3) rebalances the tree. Since T_{11}, T_{12} and T_2 have height at least 1 and hence at least two leaves each, the composition of the fringe is not changed.

Case 2. T_{11} has height h and T_{12} has height $h+1$. Let T_{121} and T_{122} be the left and right subtrees of T_{12}. One of them has height h, the other one has height h or $h-1$. A double rotation rebalances the tree (cf. Fig. 4). If $h \geqq 2$ or both T_{121} and T_{122} have height h, then all four trees have at least two leaves each, and hence the composition of the fringe is not changed. Otherwise, we have $h=1$ and either T_{121}

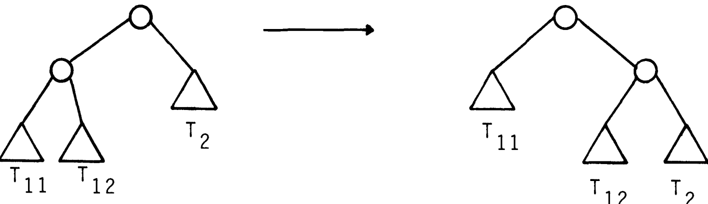


FIG. 3. *A rotation.*

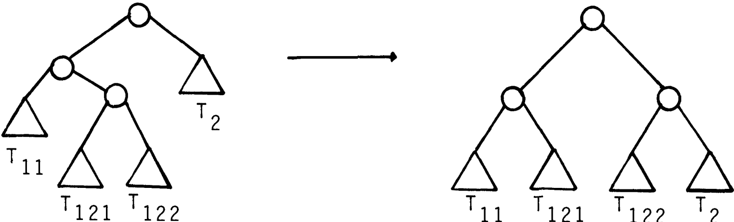


FIG. 4. *A double rotation* (DR).

or T_{122} is a single leaf. Figure 5 shows that the composition of the fringe is not changed in this case. □

Upon the deletion of a leaf, the father of that leaf is replaced by the other subtree, and then the tree is rebalanced along the path back to the root (cf. Knuth [12] for more detail).

LEMMA 4. *Let T be an AVL-tree with at least three leaves. Suppose that a leaf is deleted from T and tree T' is obtained after rebalancing.*

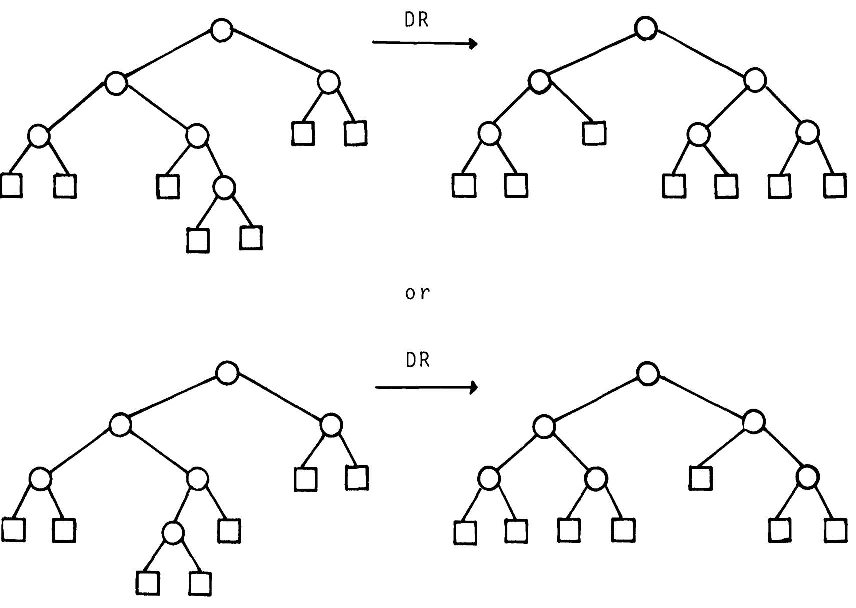


FIG. 5. $a_1(T') = a_1(T) = 1$, $a_2(T') = a_2(T) = 2$.

a) *If the deletion is from a type* I *subtree, then* $a_1(T') = a_1(T) - 1$ *and* $a_2(T') = a_2(T) + 1$.

b) *If the deletion is from a type* II *subtree, then*

b1) *If the brother of that type* II *subtree is a type* I *subtree, then* $a_1(T') = a_1(T) - 1$ *and* $a_2(T') = a_2(T) + 1$.

b2) *If the brother of that type* II *subtree is not a type* I *subtree, then* $a_1(T') = a_1(T) + 1$ *and* $a_2(T') = a_2(T) - 2$.

Proof. a) Deletion of a leaf from a type I subtree transforms that tree into a subtree with two leaves; call its root u (cf. Fig. 6). Let v be the father of u. We distinguish cases according to the *old* balance $b(v)$ of node v. We assume w.l.o.g. that u is the left son of v.

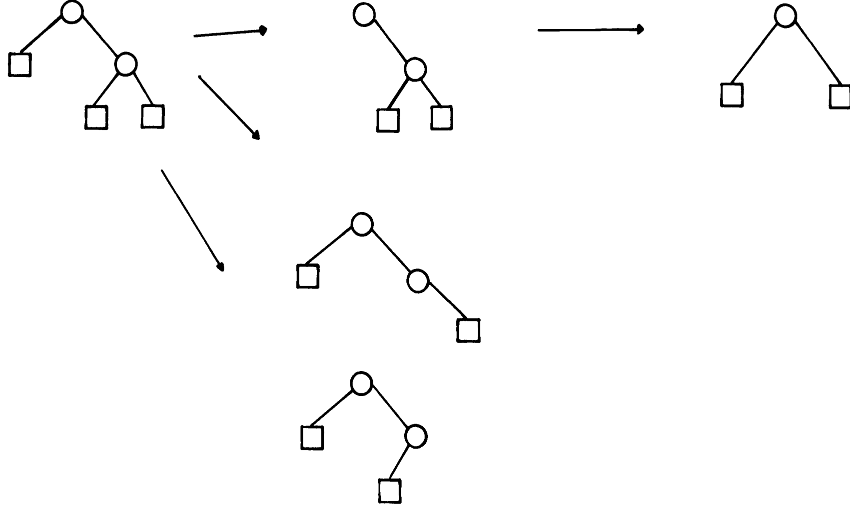


FIG. 6. *Deletion of a leaf from a type* I *subtree.*

Case 1. $b(v) = 0$. Then rebalancing is completed by changing $b(v)$ to -1. Hence $a_1(T') = a_1(T) - 1$ and $a_2(T') = a_2(T) + 1$.

Case 2. $b(v) = +1$. Then the other son of v has height 1 and hence is a type II subtree. The new balance of u is 0 and its new height is 2. Also there might be rebalancing necessary higher up in the tree. By Lemma 3, none of these rebalancing operations changes the composition of the fringe. Hence $a_1(T') = a_1(T) - 1$ and $a_2(T') = a_2(T) + 1$.

Case 3. $b(v) = -1$. Then the other son of v has height 3, and hence rebalancing about v is necessary. By Lemma 3 the rebalancing operation about v does not change the composition of the fringe. After rebalancing v, rebalancing higher up in the tree might be required. Again, it does not affect the composition of the fringe. Hence $a_1(T') = a_1(T) - 1$ and $a_2(T') = a_2(T) + 1$.

b) Deletion of a leaf from a type II subtree transforms the subtree into a single leaf. Let v be the father of that leaf. Assume w.l.o.g. that this leaf is the left son. Before the deletion, the balance parameter $b(v)$ is either 0 or -1. If it were $+1$ then v would be the root of a type I subtree.

b1) The other son of v is a type I subtree. Then $b(v) = -1$ before the deletion and $b(v) = -2$ after the deletion. A rotation or double rotation about v will generate

two type II subtrees. Subsequent rebalancing operations higher up in the tree will not change the composition of the fringe. Hence $a_1(T')=a_1(T)-1$ and $a_2(T')=a_2(T)+1$.

b2) The other son of v is not a type I subtree.

Case 1. $b(v)=0$. Then the other son of v is a type II subtree. After the deletion, v is the root of a type I subtree. No rebalancing is required. Hence $a_1(T')=a_1(T)+1$ and $a_2(T')=a_2(T)-2$.

Case 2. $b(v)=-1$. Then the other son of v has height 2, but it is not the root of a type I subtree. Hence its two sons are type II subtrees. A rotation about v will be performed. Hence $a_1(T')=a_1(T)+1$ and $a_2(T')=a_2(T)-2$. □

In the remainder of this section, we will set up recurrence relations in order to study the effect of random insertions and deletions on the composition of the fringe of AVL-trees. We review our randomness assumption.

1) For $n \geqq 3$, insertion and deletion are equally likely. Only insertions occur for $n=2$.
2) In the case of insertion, each one of the n leaves is equally likely to be split.
3) In the case of deletion, each leaf is deleted with equal probability.

Let k_n be the number of AVL-trees with n leaves. Let $T_{n,j}$ represent the jth AVL-tree with n leaves in some arbitrary ordering of the n-leaf AVL-trees. The above randomness assumptions define a Markov chain with states $T_{n,j}$. There is a transition from $T_{n,j}$ to $T_{n+1,k}$ if insertion into $T_{n,j}$ and subsequent rebalancing can generate $T_{n+1,k}$, and there is a transition to $T_{n-1,i}$ if deletion from $T_{n,j}$ and subsequent rebalancing yields $T_{n-1,i}$. For each leaf of $T_{n,j}$ we have two transitions, one corresponding to splitting that leaf (insertion) and one corresponding to deleting that leaf. Each transition has probability $1/(2n)$. Let $q_{n,j}$ be the stationary (conditional) probability of being in tree $T_{n,j}$, under the assumption of being in a tree with n leaves.

DEFINITION 2.

$$a_1(n)\equiv \sum_{j=1}^{k_n} q_{n,j}a_1(T_{n,j}), \qquad a_2(n)\equiv \sum_{j=1}^{k_n} q_{n,j}a_2(T_{n,j}),$$

where $a_1(n)$ ($a_2(n)$) is the average number of type I (type II) subtrees in a random AVL-tree with n leaves.

We proceed to derive recurrence relations for $a_2(n)$.

$$\begin{aligned} a_2(n) &= \sum_{j=1}^{k_n} q_{n,j}\cdot a_2(T_{n,j}) \\ &= \sum_{j=1}^{k_n} a_2(T_{n,j})\left[\sum_{i=1}^{k_{n-1}} q_{n-1,i}\cdot \text{Prob}\,(T_{n-1,i}\to T_{n,j}) + \sum_{h=1}^{k_{n+1}} q_{n+1,h}\cdot \text{Prob}\,(T_{n+1,h}\to T_{n,j})\right], \end{aligned}$$

where Prob $(T\to T')$ denotes the probability of obtaining T' after a random insertion into or deletion from T. This probability is always a multiple of $1/(2n)$, where n is the number of leaves of T.

$$\begin{aligned} &= \sum_{i=1}^{k_{n-1}} q_{n-1,i}\cdot \sum_{j=1}^{k_n} \text{Prob}\,(T_{n-1,i}\to T_{n,j})\cdot a_2(T_{n,j}) \\ &\quad + \sum_{h=1}^{k_{n+1}} q_{n+1,h}\cdot \sum_{j=1}^{k_n} \text{Prob}\,(T_{n+1,h}\to T_{n,j})\cdot a_2(T_{n,j}). \end{aligned}$$

There are $n-1$ different ways of inserting a new leaf into $T_{n-1,i}$. Out of these $n-1$ different ways, $3a_1(T_{n-1,i})$ correspond to insertions into type I subtrees (and then $a_2(T_{n,j})=a_2(T_{n-1,i})+2$ by Lemma 2), and $2a_1(T_{n-1,i})$ correspond to insertions into

type II subtrees (and then $a_2(T_{n,j}) = a_2(T_{n-1,i}) - 1$). Hence

$$\sum_{i=1}^{k_{n-1}} q_{n-1,i} \cdot \sum_{j=1}^{k_n} \text{Prob}\,(T_{n-1,i} \to T_{n,j}) \cdot a_2(T_{n,j})$$
$$= \sum_{i=1}^{k_{n-1}} q_{n-1,i} \cdot \frac{1}{2}\left[\frac{3a_1(T_{n-1,i})}{n-1}(a_2(T_{n-1,i})+2) + \frac{2a_2(T_{n-1,i})}{n-1}(a_2(T_{n-1,i})-1)\right]$$
$$= \frac{1}{2}\left[a_2(n-1) + \frac{6a_1(n-1)}{n-1} - \frac{2a_2(n-1)}{n-1}\right].$$

There are $n+1$ different ways of deleting a leaf from $T_{n+1,h}$. Exactly $3a_1(T_{n+1,h})$ of these deletions are deletions from type I subtrees, and then $a_2(T_{n,j}) = a_2(T_{n+1,h}) + 1$. The other $2a_2(T_{n+1,h})$ deletions are deletions from type II subtrees.

DEFINITION 3. Let $p(T_{n,j})$ be the probability that the brother of a type II subtree is a type I subtree in $T_{n,j}$.

LEMMA 5.

$$0 \leqq p(T_{n,j}) \leqq \min\,(1, a_1(T_{n,j})/a_2(T_{n,j})).$$

Proof. Since $p(T_{n,j})$ is a probability, we have $0 \leqq p(T_{n,j}) \leqq 1$. Furthermore, $p(T_{n,j})$ is the fraction of type II subtrees of $T_{n,j}$ whose brothers are type I subtrees of $T_{n,j}$. Hence $a_2(T_{n,j}) \cdot p(T_{n,j}) \leqq a_1(T_{n,j})$. □

Out of the $2a_2(T_{n+1,h})$ deletions from type II subtrees, exactly $2a_2(T_{n+1,h})p(T_{n+1,h})$ are deletions from type II subtrees whose brothers are type I subtrees. In this case we have $a_2(T_{n,j}) = a_2(T_{n+1}, h) + 1$. The remaining $2a_2(T_{n+1,h}) \cdot (1 - p(T_{n+1,h}))$ deletions give $a_2(T_{n,j}) = a_2(T_{n+1,h}) - 2$. Hence

$$\sum_{h=1}^{k_{n+1}} q_{n+1,h} \cdot \sum_{j=1}^{k_n} \text{Prob}\,(T_{n+1,h} \to T_{n,j}) \cdot a_2(T_{n,j})$$
$$= \sum_{h=1}^{k_{n+1}} q_{n+1,h} \cdot \frac{1}{2}\left[\frac{3a_1(T_{n+1,h})}{n+1}(a_2(T_{n+1,h})+1)\right.$$
$$+ \frac{2a_2(T_{n+1,h})}{n+1}(p(T_{n+1,h})(a_2(T_{n+1,h})+1)$$
$$\left. + (1-p(T_{n+1,h}))(a_2(T_{n+1,h})-2))\right]$$
$$= \frac{1}{2}\left[a_2(n+1) + \frac{3a_1(n+1)}{n+1} + \frac{2a_2(n+1)}{n+1}(3p(n+1)-2)\right],$$

where

$$p(n+1) \equiv \left(\sum_{h=1}^{k_{n+1}} q_{n+1,h} \cdot a_2(T_{n+1,h})p(T_{n+1,h})\right) \Big/ a_2(n+1)$$

is the probability that the brother of a type II subtree is a type I subtree in a random AVL-tree with $n+1$ leaves.

LEMMA 6.

$$0 \leqq p(n) \leqq \min\,(1, a_1(n)/a_2(n)).$$

Proof. $0 \leqq p(n) \leqq 1$ is obvious. Also

$$\begin{aligned} p(n) &= \frac{1}{a_2(n)} \cdot \sum_{j=1}^{k_n} a_2(T_{n,j}) p(T_{n,j}) \cdot q_{n,j} \\ &\leqq \frac{1}{a_2(n)} \sum_{j=1}^{k_n} q_{n,j} a_1(T_{n,j}) \qquad \text{(by Lemma 5)} \\ &= a_1(n)/a_2(n). \end{aligned}$$

□

Putting the derivations together, we arrive at our main equation:

$$\text{(M)} \qquad \begin{aligned} 2a_2(n) &= a_2(n-1) + \frac{6a_1(n-1)}{n-1} - \frac{2a_2(n-1)}{n-1} \\ &\quad + a_2(n+1) + \frac{3a_1(n+1)}{n+1} + \frac{2a_2(n+1)}{n+1}(3p(n+1)-2), \end{aligned}$$

where $0 \leqq p(n+1) \leqq \min(1, a_1(n+1)/a_2(n+1))$.

With the use of Lemma 1, we can eliminate $a_1(n-1)$, $a_1(n+1)$ from equation (M) and obtain equations (M′) and (M″):

$$\text{(M}'\text{)} \qquad 2a_2(n) = a_2(n-1) - \frac{6a_2(n-1)}{n-1} + a_2(n+1) + \frac{6a_2(n+1)}{n+1}[p(n+1)-1] + 3,$$

or

$$\text{(M}''\text{)} \qquad a_2(n+1)\left[1 + \frac{6p(n+1)-6}{n+1}\right] = 2a_2(n) - a_2(n-1)\left[1 - \frac{6}{n-1}\right] - 3,$$

where

$$0 \leqq p(n+1) \leqq \min\left(1, \frac{n+1-2a_2(n+1)}{3a_2(n+1)}\right).$$

In the next section we will derive bounds on the solutions of equation (M″).

3. Solving the equation. In this section we will solve equation (M″); more precisely, we will show $c_1 \cdot n \leqq a_2(n) \leqq c_2 \cdot n$ for suitable constants c_1 and c_2. We proceed in two steps. In the first step, we solve the equation under the simplifying, however unjustified, assumption $p(n) = p$ and $a_2(n) = a_2 \cdot n$. In the second step, we try to show that the solutions obtained in the first step suffice to describe the system, even without the additional assumption.

Suppose $p(n) = p$ and $a_2(n) = a_2 \cdot n$ for all n. We want to stress again that this assumption is completely unjustified. Then equation (M″) transforms into

$$a_2 \cdot (n+1) \cdot \left[\frac{n+6p-5}{n+1}\right] = 2a_2 \cdot n - a_2 \cdot (n-1) \cdot \left[\frac{n-7}{n-1}\right] - 3$$

or

$$a_2(6p-5) = 7a_2 - 3 \quad \text{or} \quad a_2 = \frac{3}{12-6p}.$$

The extremal values of p are 0 and $\min(1, a_1(n)/a_2(n))$. Using Lemma 1 and $a_2(n) = a_2 \cdot n$, we obtain $0 \leqq p \leqq \min(1, (1-2a_2)/3a_2)$. For $p = 0$ we have $a_2 = \frac{1}{4}$ and for $p = (1-2a_2)/3a_2$ we have $a_2 = \frac{5}{16}$. Hence our simplifying assumptions imply $n/4 \leqq a_2(n) \leqq 5n/16$.

In our main theorem, we show that the above inequality is essentially true even without the simplifying assumption:

THEOREM 1. a) *For* $n \geqq 7$, $a_2(n) \geqq n/4$;

b) $\lim_{n\to\infty} \sup a_2(n)/n \leqq \frac{5}{16}$.

Proof. Up to symmetry, there is only one AVL-tree with 7 leaves (Fig. 7). Hence $a_2(7)=2$.

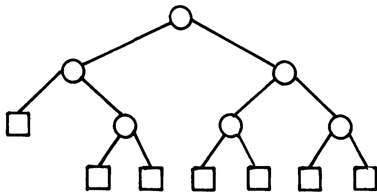


FIG. 7. *The only (up to symmetry)* AVL-*tree with* 7 *leaves.*

Define $a_2(n)=b_2(n)\cdot n$. Then equation (M″) transforms into (M‴):

$$\text{(M''')} \qquad b_2(n+1)[n+6p(n+1)-5]=2b_2(n)\cdot n-b_2(n-1)[n-7]-3.$$

We will show $b_2(n) \geqq \frac{1}{4}$ for $n \geqq 7$, and $\lim \sup b_2(n) \leqq \frac{5}{16}$.

LEMMA 7. $0 \leqq b_2(n) \leqq \frac{1}{2}$ *for all* n.

Proof. $a_1(n) \geqq 0$ and $a_2(n) \geqq 0$ are obvious. An application of Lemma 1 finishes the proof. □

We will first show $b_2(n) \geqq \frac{1}{4}$ for $n \geqq 7$. This was already shown for $n=7$. Lemma 8 below shows that once $b_2(n)$ goes below $\frac{1}{4}$, it will be monotonically decreasing. Hence by Lemma 7 the sequence $b_2(n)$ converges to some number between 0 and $\frac{1}{4}$ (exclusive). A contradiction is then derived in Lemma 9.

LEMMA 8. *For* $n \geqq 7$, *if* $b_2(n)<b_2(n-1)$ *and* $b_2(n)<\frac{1}{4}$, *then* $b_2(n+1) \leqq b_2(n)+(12b_2(n)-3)/(n-5)$.

Proof. Since $p(n+1) \geqq 0$, equation (M‴) implies

$$(n-5)b_2(n+1) \leqq 2nb_2(n)-(n-7)b_2(n-1)-3$$

or

$$(n-5)(b_2(n+1)-b_2(n)) \leqq (n-7)(b_2(n)-b_2(n-1))+(12b_2(n)-3).$$

Thus

$$b_2(n+1) \leqq b_2(n)+(12b_2(n)-3)/(n-5).$$ —

LEMMA 9. $b_2(n) \geqq \frac{1}{4}$ *for all* $n \geqq 7$.

Proof. Assume otherwise. Since $b_2(7) \geqq \frac{1}{4}$, there is at least $n_0>7$ with $b_2(n_0)<\frac{1}{4} \leqq b_2(n_0-1)$. Let $\varepsilon=3-12b_2(n_0)<0$. A simple induction argument based on Lemma 8 shows $b_2(n+1)<b_2(n)-\varepsilon/(n-5)$ for all $n \geqq n_0$. Hence $b_2(n+1) \leqq b_2(n_0)-\varepsilon \cdot \sum_{i=n_0}^{n} 1/(i-5)$ for $n \geqq n_0$. Thus $b_2(n)$ must become negative, a contradiction to Lemma 7. □

It remains to prove $\lim \sup b_2(n) \leqq \frac{5}{16}$; $\lim \sup b_2(n)$ exists, since $b_2(n)$ is bounded. We will first prove the analogue of Lemma 8.

LEMMA 10. *For* $n \geqq 9$, *if* $b_2(n)>b_2(n-1)$ *and* $b_2(n)>\frac{5}{16}$, *then*

$$b_2(n+1) \geqq b_2(n)+(16b_2(n)-5)/(n-9).$$

Proof. Since $p(n+1) \leqq ((n+1)-2a_2(n+1))/3a_2(n+1) = (1-2b_2(n+1))/3b_2(n+1)$, equation (M''') implies

$$(n-5+6 \cdot (1-2b_2(n+1))/3b_2(n+1)) \cdot b_2(n+1) \geqq 2nb_2(n)-(n-7)b_2(n-1)-3$$

or

$$(n-9)(b_2(n+1)-b_2(n)) \geqq (n-7)(b_2(n)-b_2(n-1))+(16b_2(n)-5),$$

and hence $b_2(n+1) \geqq b_2(n)+(16b_2(n)-5)/(n-9)$. □

LEMMA 11.

$$\limsup b_2(n) \leqq \tfrac{5}{16}.$$

Proof. Assume otherwise; say $b = \limsup b_2(n) > \frac{5}{16}$. Then there is either an $n_0 \geqq 9$ such that $b_2(n_0) > b_2(n_0-1)$ and $b_2(n_0) > \frac{5}{16}$, or $b_2(n) > \frac{5}{16}$ and $b_2(n) \leqq b_2(n-1)$ for all $n \geqq 9$. In either case we will derive a contradiction.

Case 1. *There is an* $n_0 > 9$ *such that* $b_2(n_0) > b_2(n_0-1)$ *and* $b_2(n_0) > \frac{5}{16}$. Let $\varepsilon = 16b_2(n_0)-5$. A simple induction argument based on Lemma 10 shows that $b_2(n+1) > b_2(n)+\varepsilon/(n-9)$ for all $n \geqq n_0$. Hence $b_2(n)$ is unbounded, a contradiction to Lemma 7.

Case 2. $b_2(n) > \frac{5}{16}$ *and* $b_2(n) < b_2(n-1)$ *for all* $n \geqq 9$. Then $b_2(n)$ is nonincreasing, and hence $b = \lim b_2(n)$. Let $\varepsilon = b - \frac{5}{16} > 0$. Then there is n_0 such that for $n \geqq n_0$ we have $b_2(n)-b_2(n+1) \leqq \varepsilon$. Hence the next to last inequality in Lemma 10 implies for $n > n_0$

$$\begin{aligned}(n-9)&(b_2(n+1)-b_2(n))\\ &\geqq (n-9)(b_2(n)-b_2(n-1))+2(b_2(n)-b_2(n-1))+(16b_2(n)-5)\\ &\geqq (n-9)(b_2(n)-b_2(n-1))-2\varepsilon+16\varepsilon,\end{aligned}$$

and thus

$$b_2(n+1)-b_2(n) \geqq (b_2(n)-b_2(n-1))+(14\varepsilon)/(n-9).$$

This shows that the difference $b_2(n+1)-b_2(n)$ must become unbounded, a contradiction to Lemma 7.

In either case we have derived a contradiction. Hence $\limsup b_2(n) \leqq \frac{1}{16}$. □

This finishes the proof of Theorem 1. □

4. Applications. In this section we apply our results to derive bounds on the number of balanced nodes in random AVL-trees, and we indicate how to extend the results to HB-trees (Ottman and Six [18], Mehlhorn [14]) and 2-3-trees (Aho, Hopcroft and Ullman [1]).

THEOREM 2. *Let* $\bar{B}(n)$ *denote the average number of balanced nodes* (= *nodes of balance* 0) *in a random* (*in the sense of* § 1)AVL-*tree with n leaves. Then*

$$4n/9 \leqq \bar{B}(n) \leqq 7n/8+o(n).$$

Proof. Let T be an AVL-tree with a_1 type I subtrees and a_2 type II subtrees, and n leaves. Then $3a_1+2a_2=n$. T contains a_1+a_2 balanced nodes in the fringe and a_1+a_2-1 nodes which are not in the fringe.

Out of these a_1+a_2-1 nodes, all may be balanced. Hence T contains at most $2a_1+2a_2-1=2n/3+2a_2/3-1$ balanced nodes. Passing to averages yields $\bar{B}(n) \leqq \frac{7}{8}n+o(n)$ by Theorem 1.

Next we want to derive a lower bound on the number of balanced nodes. The brother of a type II subtree is either a type I subtree or a type II subtree or a tree

consisting of two type II subtrees. Since the former situation arises at most a_1 times, the latter two situations must occur at least $a_2 - a_1$ times. Every three occurrences lead to at least one balanced node. This yields another $(a_2 - a_1)/3$ balanced nodes. Passing to averages yields

$$\bar{B}(n) \geqq a_1(n) + a_2(n) + (a_2(n) - a_1(n))/3$$
$$\geqq 2n/9 + 8a_2(a)/9 \geqq 4n/9. \qquad \square$$

Theorem 2 shows that $0.44n \leqq \bar{B}(n) \leqq 0.875n + o(n)$. Previously AVL-trees were analyzed under random insertions only. In that case, Brown [4] showed $0.47n \leqq \bar{B}(n) \leqq 0.85n$ and Mehlhorn [15] showed $0.51n \leqq \bar{B}(n) \leqq 0.81n$.

In the remainder of this section, we indicate how to extend our results to 2-3-trees and HB-trees. In a 2-3-tree, all leaves are on the same level, and every interior node has degree either two or three (Aho, Hopcroft and Ullman [1]). We obtain the fringe by deleting all nodes of height at least two, i.e., we do a first-order analysis in the sense of Yao [19]. The fringe consists of two types of trees: nodes with three sons (type I) and nodes with two sons (type II) (Fig. 8).

FIG. 8. *The subtrees in the fringe of a* 2-3-*tree.*

The crucial observation is that Lemma 2 remains true if we replace "AVL-tree" by "2-3-tree" and that Lemma 4 remains true if we replace "AVL-tree" by "2-3-tree" and "if the brother" by "if a brother" in b1) and "if the brother" by "if the brothers" in b2). The simple proofs are left to the readers.

Next we need to define $p(T)$ as the probability that one of the brothers of a type II subtree is a type I subtree. Then Lemma 5 transforms into $0 \leqq p(T) \leqq \min(1, 2 \cdot a_1(T)/a_2(T))$, since one type I tree can serve as the brother of two type II trees. Nevertheless, we obtain the same recurrence relation for $a_2(n)$ as before; however, we have the weaker condition $0 \leqq p(n) \leqq \min(1, 2a_1(n)/a_2(n))$ on $p(n)$.

Proceeding as above, we obtain $a_2(n) \geqq n/4$ and $a_2(n) \leqq 7n/20 + o(n)$. The proofs are almost literally the same; in the proofs of Lemmas 10 and 11, one has to use the weaker restriction on $p(n)$. These bounds on $a_2(n)$ can be used to derive bounds on the expected number of nodes in a random 2-3-tree and hence on the storage utilization. (cf. Yao [19]). If the fringe of a 2-3-tree T consists of $a_1(a_2)$ nodes with 3(2) sons, then the number of internal nodes of T lies between $(3(a_1 + a_2) - 1)/2$ and $2(a_1 + a_2) - 1$. Also $3a_1 + 2a_2$ is equal to the number of leaves of T. Hence the expected number of nodes in a random 2-3-tree with n leaves is at least $(3(a_1(n) + a_2(n)) - 1)/2 = (n + a_2(n) - 1)/2 \geqq (5n - 4)/8 \approx 0.625n$, and at most $2(a_1(n) + a_2(n)) - 1 \leqq 2(n + a_2(n))/3 - 1 \leqq 9/10n + o(n) \approx 0.9n$. The obvious bounds are $(n-1)/2$ and $n - 1$. We summarize the discussion in

THEOREM 3. *Let $\bar{N}(n)$ be the expected number of nodes in a random (in the sense of* § 1) 2-3-*tree with n leaves. Then*

$$(5n - 4)/8 \leqq \bar{N}(n) \leqq (9/10) \cdot n + o(n).$$

For HB-trees (Ottman and Six [18]), one obtains the fringe by deleting all nodes with at least four leaves below them. Figure 9 shows the trees which appear in the

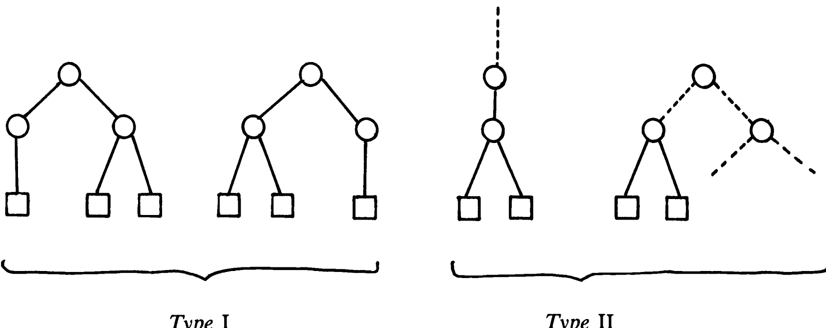


FIG. 9. *The subtrees in the fringe of an* HB-*tree.*

fringe. Lemmas 1–6 stay true with "AVL-tree" replaced by "HB-tree" throughout. Hence Theorem 1 gives the number of type I and type II subtrees in a random HB-tree. This leads to

THEOREM 4. *Let* $\bar{N}(n)$ *be the expected number of nodes in a random (in the sense of* § 1) HB-*tree with n leaves. Then*

$$(9/8)\cdot n - o(n) \leqq \bar{N}(n) \leqq 14n/9.$$

Proof. Let T be an HB-tree with a_1 type I subtrees and a_2 type II subtrees and n leaves. Since T has n leaves, there are exactly $n-1$ nodes with two sons. It remains to derive bounds on the number of nodes with one son. Since every type I subtree contains one node with only one son, there are at least a_1 nodes with only one son. Passing to averages gives $N(n) \geqq n-1+a_1(n) \geqq (9/8)n - o(n)$. For the upper bound on the number of nodes with one son, we use the following correspondence between AVL-trees and HB-trees. If one deletes all nodes with only one son from an HB-tree by combining these nodes with their fathers, then one obtains an AVL-tree T' (cf. Fig. 10). Furthermore, this correspondence preserves the composition of the fringe; i.e., type I(II) subtrees in the HB-tree sense are mapped onto type I(II) subtrees in the AVL-tree sense, and the number of nodes with one son in T is equal to the number of unbalanced nodes in T'. So T' is an AVL-tree with n leaves, a_1 type I subtrees and a_2 type II subtrees. In the proof of Theorem 2 we have shown that T' contains at least $a_1+a_2+(a_2-a_1)/3$ balanced nodes. Thus T contains at most $n-1-a_1-a_2-(a_2-a_1)/3$ nodes with only one son. Passing to averages give $\bar{N}(n) \leqq 2n-2-a_1(n)-a_2(n)-(a_2(n)-a_1(n))/3 \leqq 14n/9$. □

The proof of Theorem 4 may seem unnecessarily complicated to some readers. Why not use the correspondence between AVL-trees and HB-trees directly to lift

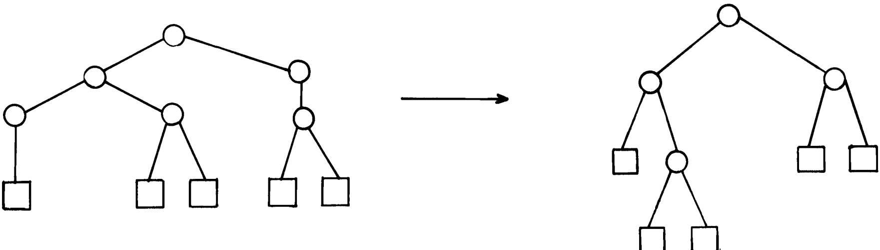


FIG. 10. *An* HB-*tree and the corresponding* AVL-*tree.*

the average case results? This approach does not work because the correspondence described above is a static one; i.e., it is not compatible with the rebalancing operations.

5. Conclusion. We have shown how to extend "fringe analysis" to the analysis of balanced tree schemes under insertion and deletion.

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
[2] R. BAYER AND E. MCCREIGHT, *Organization and maintenance of large ordered indices*, Acta Inform., 1 (1972), pp. 173–189.
[3] N. BLUM AND K. MEHLHORN, *On the average number of rebalancing steps in weight-balanced trees*, Theoret. Comput. Sci., 11 (1980), pp. 303–320.
[4] M. R. BROWN, *A partial analysis for height-balanced trees*, this Journal, 8 (1979), pp. 33–41.
[5] ———, *Some observations on random* 2–3 *trees*, Inform. Proc. Letters, 9 (1979), pp. 57–59.
[6] M. R. BROWN AND R. E. TARJAN, *Design and analysis of a data structure for representing sorted lists*, this Journal, 9 (1980), pp. 594–614.
[7] B. EISENBART AND K. MEHLHORN, *Allgemeine Fringe-Analyse und ihre Anwendung auf B-Bäume mit Overflow*, Typescript, Oct. 1980.
[8] P. FLAJOLET, J. FRANCON AND J. VUILLEMIN, *Sequence of operations analysis for dynamic data structures*, J. Algorithms, (1980), pp. 111–114.
[9] S. HUDDLESTON AND K. MEHLHORN, *Robust balancing in B-trees*, in 5th GI-Conference in Theoretical Computer Science 1981, LNCS 104 (1981), pp. 234–244.
[10] A. JONASSEN AND D. E. KNUTH, *A trivial algorithm whose analysis isn't*, J. Comput. Systems Sci., 16 (1978), pp. 301–322.
[11] G. D. KNOTT, *Deletion in binary storage trees*, Ph.D. thesis, Computer Science Dept., Stanford Univ., Stanford, CA, 1975.
[12] D. E. KNUTH, *The Art of Computer Programming, Vol.* 3: *Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.
[13] ———, *Deletions that preserve randomness*, IEEE Trans. Software Engrg., SE 3 (1977), pp. 351–359.
[14] K. MEHLHORN, *Effiziente Algorithmen*, Studienbücher·Informatik, Teubner-Verlag, Leipzig, 1977.
[15] ———, *A partial analysis of height-balanced trees*, Techn. Bericht A 79/13, FB 10, Univ. d. Saarlandes, June 1979.
[16] ———, *A new representation for linear lists*, Techn. Bericht A 22/79, FB 10, Univ. d. Saarlandes, Dec. 1979.
[17] J. NIEVERGELT AND E. M. REINGOLD, *Binary search trees of bounded balance*, this Journal, 2 (1973), pp. 33–43.
[18] TH. OTTMAN AND H. W. SIX, *Eine neue Klasse von ausgeglichenen Bäumen*, Angewandte Informatik, Heft, 9 (1976), pp. 395–400.
[19] A. C.-C. YAO, *On random* 2–3 *trees*, Acta Inform., 9 (1978), pp. 159–170.

SIAM J. COMPUT.
Vol. 11, No. 4, November 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1104-0011 $01.00/0

THE CATEGORY-THEORETIC SOLUTION OF RECURSIVE DOMAIN EQUATIONS*

M. B. SMYTH† AND G. D. PLOTKIN†

Abstract. Recursive specifications of domains plays a crucial role in denotational semantics as developed by Scott and Strachey and their followers. The purpose of the present paper is to set up a categorical framework in which the known techniques for solving these equations find a natural place. The idea is to follow the well-known analogy between partial orders and categories, generalizing from least fixed-points of continuous functions over *cpos* to initial ones of continuous functors over ω-categories. To apply these general ideas we introduce Wand's **O**-categories where the morphism-sets have a partial order structure and which include almost all the categories occurring in semantics. The idea is to find solutions in a derived category of embeddings and we give order-theoretic conditions which are easy to verify and which imply the needed categorical ones. The main tool is a very general form of the limit-colimit coincidence remarked by Scott. In the concluding section we outline how compatibility considerations are to be included in the framework. A future paper will show how Scott's universal domain method can be included too.

Key words. Domains, semantics, data-types, category, partial-order, fixed-point, computability

1. Introduction. Recursive specifications of domains play a crucial role in denotational semantics as developed by Scott and Strachey and their followers (Gordon [13], Milne and Strachey [26], Stoy [39], Tennent [40], [41]). For example, the equation

$$D \cong \mathrm{At} + (D \to D) \tag{1}$$

is just what is needed for the semantics of an untyped λ-calculus for computing over a domain, At, of atoms. Again, the simultaneous equations

$$T \cong \mathrm{At} \times F, \tag{2}$$

$$F \cong 1 + (T \times F) \tag{3}$$

specify a domain, T, of all finitarily branching trees and another, F, of forests of such trees. And recursively specified data types are also very useful [10], [19], [20].

The first tools for solving such equations were provided by Scott using his inverse limit constructions [33]. Later he showed how the inverse limits could be entirely avoided by using a universal domain and the ordinary least fixed point construction [34]. A systematic exposition of the inverse limit method was given by Reynolds [30], and the categorical aspects (already mentioned by Scott) were emphasized by Wand [43]. All of these treatments stuck to one category, such as, for example, **CL**, the category of countably based continuous lattices and continuous functions, although the details did not change much in other categories. Then Wand [44], gave an abstract treatment based on **O**-categories where the morphism sets are provided with a suitable order-theoretic structure. The relation between the category-theoretic treatment and the universal domain method has, until now, remained rather obscure.

The purpose of the present paper is to set up a categorical framework in which all known techniques for solving domain equations find a natural place. The idea as set out in § 2 is to follow the well-known analogy between partial orders and categories, and generalize from least fixed points to initial fixed points. These are constructed

* Received by the editors January 2, 1979, and in revised form February 17, 1982. This work was partially supported by a grant from the Science and Engineering Research Council.

† Department of Computer Science, University of Edinburgh, Edinburgh, Scotland EH9 3JZ.

using the "basic lemma" which plays an organizational role: Most of the solution methods considered appear as ways of ensuring that the hypotheses of the lemma are fulfilled. Just as continuous functions over complete partial orders always have least fixed points, so continuous functors over ω-categories (as defined below) always have initial fixed points, which can be constructed by using the basic lemma; this seems to formalize some hints of Lawvere mentioned by Scott in [33]. The same idea appears in [1], [2].

All this is very general, and we introduce **O**-categories in § 3 in order to apply the basic lemma to the construction of the domains needed in denotational semantics. Here we are clearly greatly indebted to Wand [44], [45] who introduced **O**-categories, and indeed our work arose partly as an attempt to simplify and clarify his treatment. The idea is to apply the basic lemma not to a given **O**-category, **K**, but rather to a derived category, $\mathbf{K}^E$, of embeddings (equivalently, projections). We then look for easily verified conditions on **K** (whether categorical or order-theoretic) which imply the needed conditions on $\mathbf{K}^E$.

Our main tool is Theorem 2, which establishes a very general form of the limit-colimit coincidence remarked by Scott [33] and also gives an order-theoretic characterization of the relevant categorical limits. This improves Wand's work by removing the need for his troublesome "Condition A" (appearing in [44] rather than [45] which incorporates some of the ideas of the present paper); more positively we also introduce a useful notion of duality for **O**-categories.

With the aid of Theorem 2 (and the easy Theorem 1), one sees that simple conditions on an **O**-category, **K**, (mainly that it has all $\boldsymbol{\omega}^{\mathrm{op}}$-limits) ensure that $\mathbf{K}^E$ is an ω-category. Again with the aid of (Lemma 4 and) Theorem 3, one sees how to take any mixed contravariant-covariant functor over **K** (like the function-space one) which satisfies an order-continuity property (usually evident), and turn it into a covariant-continuous one over $\mathbf{K}^E$.

Section 4 presents several examples of useful categories which may be handled by the methods of §§ 2 and 3.

The method of universal domains, in relation to the ideas presented here, is treated in the sequel to this paper. An indication of our approach may be found in Plotkin and Smyth [28] (which may also be of help in getting a general overview of our results).

There is, however, one aspect of Scott's presentation of the universal domain approach [34] which must receive some mention here: the question of computability. The results presented in this paper would lose much of their point if we were forced to invoke a universal domain to handle computability. In the concluding § 5, we indicate briefly how this topic can in fact be handled at the level of generality of this paper; for a more detailed treatment we refer to Smyth [38].

We assume the reader possesses a basic knowledge of category theory; any gaps can be filled by consulting Arbib and Manes [4], Herrlich and Strecker [16], or MacLane [21].

2. Initial fixed points. In the categorical approach to recursive domain specifications we try to regard all equations such as (1) or (2) and (3) above as being of the form

$$X \cong F(X), \tag{4}$$

where X ranges over the objects of a category **K**, say, and $F: \mathbf{K} \to \mathbf{K}$ is an endofunctor of that category. For example, in the case of (1) we could take X to range over the objects of **K**, At to be a fixed object of **K**, and + and $\to$ to be covariant sum and

function-space functors over $\mathbf{K}$; then $F:\mathbf{K}\to\mathbf{K}$ is defined by:

$$F(X) \underset{\text{def}}{=} \text{At}+(X\to X). \tag{5}$$

Let us spell the meaning of (5) out in detail. Recall that if $F_i:\mathbf{K}\to\mathbf{K}_i$ $(i=1,n)$ are functors then their *tupling* $F=\langle F_1,\cdots,F_n\rangle$: $\mathbf{K}\to\mathbf{K}_1\times\cdots\times\mathbf{K}_n$ is defined by putting for each object, X, of $\mathbf{K}$:

$$F(X)=\langle F_1(X),\cdots,F_n(X)\rangle$$

and for each morphism $f:X\to Y$ of $\mathbf{K}$:

$$F(f)=\langle F_1(f),\cdots,F_n(f)\rangle.$$

Then the functor F defined by (5) is just

$$F=+\circ\langle K_{\text{At}},\to\circ\langle \text{id}_{\mathbf{K}},\text{id}_{\mathbf{K}}\rangle\rangle$$

where $K_{\text{At}}:\mathbf{K}\to\mathbf{K}$ is the constantly-At functor and $\text{id}_{\mathbf{K}}:\mathbf{K}\to\mathbf{K}$ is the identity functor.

Simultaneous equations are handled using product categories. For example, (2) and (3) can be regarded as having the form:

$$X\cong F_0(X,Y), \tag{6}$$

$$Y\cong F_1(X,Y), \tag{7}$$

where X, Y range over a category $\mathbf{K}$ (such as $\mathbf{CL}$) and F_0 and F_1 are bifunctors over $\mathbf{K}$ being defined by

$$F_0 \underset{\text{def}}{=} \times\circ\langle K_{\text{At}},\pi_1\rangle,\qquad F_1\underset{\text{def}}{=}+\circ\langle K_1,\times\circ\langle\pi_0,\pi_1\rangle\rangle$$

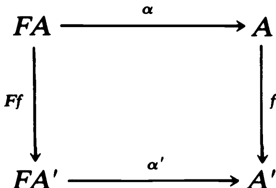
where At, 1 are objects of $\mathbf{K}$, and the $\pi_i:\mathbf{K}\times\mathbf{K}\to\mathbf{K}$ $(i=0,1)$ are the projection functors. Then (2) and (3) are put into the form (4) by using the product category $\mathbf{K}\times\mathbf{K}$ and taking F to be $\langle F_0,F_1\rangle$. Clearly this idea works for n simultaneous equations $X_i=F_i(X_1,\cdots,X_n)$ $(i=1,n)$ where X_i ranges over $\mathbf{K}_i$ $(i=1,n)$ and $F_i:\mathbf{K}_1\times\cdots\times\mathbf{K}_n\to\mathbf{K}_i$; we just take $\mathbf{K}$ to be $\mathbf{K}_1\times\cdots\times\mathbf{K}_n$ and F to be $\langle F_1,\cdots,F_n\rangle$.

Let us now decide what a solution to (4) might be and which particular ones we want. In the case where $\mathbf{K}$ is a partial order, F is then just a monotonic function, solutions are just fixed points of F (that is, elements A of K such that $A=F(A)$), and we can look for least solutions. Further, we can define *prefixed points* as elements A such that $F(A)\sqsubseteq A$, and it turns out that the least prefixed point, if it exists, is always the least fixed point as well. In the categorical case we need to know the isomorphism as well as the object:

DEFINITION 1. Let $\mathbf{K}$ be a category and $F:\mathbf{K}\to\mathbf{K}$ be an endofunctor. Then a *fixed point* of F is a pair (A,α) where A is an object of K and $\alpha:FA\cong A$ is an isomorphism of K; a *prefixed point* is a pair (A,α) where A is an object of $\mathbf{K}$ and $\alpha:FA\to A$ is a morphism of $\mathbf{K}$.

We also call prefixed points of F, *F-algebras* (same as *F-dynamic* of Arbib and Manes [4]). The F-algebras are the objects of a category:

DEFINITION 2. Let (A,α) and (A',α') be F-algebras. A *morphism* $f:(A,\alpha)\to(A',\alpha')$ (*F-homomorphism*) is just a morphism $f:A\to A'$ in $\mathbf{K}$ such that the following commutes:



It is easily verified that this gives a category: the identity and composition are both inherited from **K**. Following on the above remarks on partial orders, we look for initial F-algebras rather than just initial fixed points of F and this is justified by the following lemma (which also appears in Arbib [5], and in Barr [8], where it is credited to Lambek).

LEMMA 1. *The initial F-algebra, if it exists, is also the initial fixed point.*

Proof. Let (A, α) be the initial F-algebra. We only have to prove that α is an isomorphism. Now as (A, α) is an F-algebra so is $(FA, F\alpha)$ and so there is an F-homomorphism $f:(A, \alpha) \to (FA, F\alpha)$; one also easily sees that $\alpha:(FA, F\alpha) \to (A, \alpha)$ is an F-homomorphism and so $\alpha \circ f: (A, \alpha) \to (A, \alpha)$ is also one and it must be id_A, the identity on A as (A, α) is initial. Then as $f:(A, \alpha) \to (FA, F\alpha)$ we also have $f \circ \alpha = (F\alpha) \circ (Ff) = F(\alpha \circ f) = F(\mathrm{id}_A) = \mathrm{id}_{FA}$, which shows that α is an isomorphism with two-sided inverse f. □

Note that we have to do more than specify an object A such that $A \cong F(A)$ when looking for the initial fixed point. First we have to specify an isomorphism $\alpha : FA \cong A$, and secondly we must establish the initiality property. Both are vital in applications. When giving the semantics of programming languages using recursively specified domains the isomorphism is needed just to be able to make the definitions. Initiality is closely connnected to structural induction principles and both can be used for making proofs about elements of the specified domains. When the equations are used to specify data-type definitions within a language following the approach in Lehmann and Smyth [19], [20], the isomorphism carries the basic operations, and initiality is again essential for proofs. (The paper [20] also contains more information on simultaneous equations and on equations with parameters; in many ways, it is a companion to the present paper.)

When K is a partial order, the least fixed point can, as is well known, be constructed as $\bigsqcup_K F^n(\perp)$, the l.u.b. of the increasing sequence $\langle F^n(\perp)\rangle_{n\in\omega}$ where $\perp$ is the least element of K. This works if the least element exists, the l.u.b. exists and F preserves the l.u.b.—that is, $F(\bigsqcup_K F^n(\perp)) = \bigsqcup_K F(F^n(\perp))$. Our basic lemma merely generalizes these remarks to the case of a category.

First we give some notation and terminology which are not quite standard. By an $\boldsymbol{\omega}$*-chain* in a category **K** we understand a diagram of the form $\Delta = D_0 \to^{f_0} D_1 \to^{f_1} \cdots$ (that is, a functor from $\boldsymbol{\omega}$ to **K**); for $m \leqq n$, we write $f_{mn}: D_m \to D_n$ for the morphism $f_{n-1} \circ \cdots \circ f_m$. Dually an $\boldsymbol{\omega}^{\mathrm{op}}$*-chain* in a category **K** is a diagram of the form $\Delta = D_0 \leftarrow^{f_0} D_1 \leftarrow^{f_1} \cdots$ (that is, a functor from $\boldsymbol{\omega}^{\mathrm{op}}$ to **K**); for $m \geqq n$ we have the evident $f_{mn}: D_m \to D_n$. By the *mediating* morphism between a limiting (colimiting) μ over a diagram Δ and any other cone ν over Δ, we must understand the unique morphism given by universality from (to) the vertex of ν to (from) the vertex of μ.

Notation. The initial object of a category **K** is written as $\perp_{\mathbf{K}}$ and the unique arrow from it to an object A is written as $\perp_A$. If $\Delta = D_0 \to^{f_0} D_1 \to^{f_1} \cdots$ is an $\boldsymbol{\omega}$-chain in K and $\mu : \Delta \to A$ is a cone over Δ, then Δ^- is the $\boldsymbol{\omega}$-chain $D_1 \to^{f_1} D_2 \to^{f_2} \cdots$ and $\mu^- : \Delta^- \to A$ is the cone $\langle \mu_{n+1}\rangle_{n\in\omega}$; if, further, $F:\mathbf{K} \to \mathbf{L}$ is a functor, then $F\Delta$ is the $\boldsymbol{\omega}$-chain $FD_0 \to^{Ff_0} FD_1 \to^{Ff_1}$ and $F\mu : F\Delta \to FA$ is the cone $\langle F\mu_n\rangle_{n\in\omega}$.

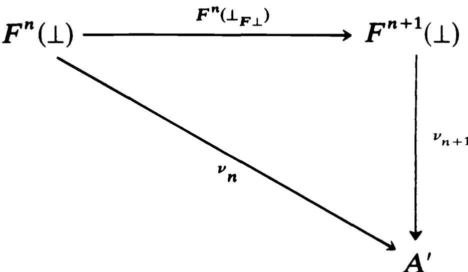
Now given a functor $F: \mathbf{K} \to \mathbf{K}$ we can define the $\boldsymbol{\omega}$-chain $\Delta = \langle F^n(\perp_{\mathbf{K}}), F^n(\perp_{F\perp})\rangle$ (if $\perp_{\mathbf{K}}$ exists) and try to justify the calculation analogous to that for partial orders:

$$F \lim_{\to} \Delta \cong \lim_{\to} F\Delta = \lim_{\to} \Delta^- = \lim_{\to} \Delta.$$

The basic lemma gives conditions for this to work and characterizes $\lim_{\to} \Delta$ (with an appropriate morphism) as the initial F-algebra.

LEMMA 2 (basic lemma). *Let* $\mathbf{K}$ *be a category with initial object* $\perp_{\mathbf{K}}$ *and let* $F: \mathbf{K} \to \mathbf{K}$ *be a functor. Define the* $\boldsymbol{\omega}$*-chain* Δ *to be* $\langle F^n(\perp_{\mathbf{K}}), F^n(\perp_{F\perp})\rangle$. *Suppose that both* $\mu: \Delta \to A$ *and* $F\mu: F\Delta \to FA$ *are colimiting cones. Then the initial F-algebra exists and is* (A, α) *where* $\alpha: FA \to A$ *is the mediating morphism from* $F\mu$ *to* μ^-.

Proof. Let $\alpha': FA' \to A'$ be any F-algebra. We show there is a unique F-homomorphism $f: (A, \alpha) \to (A', \alpha')$. First suppose f is such a homomorphism. Define a cone $\nu: \Delta \to A'$ by putting $\nu_0 = \perp_{A'}: \perp \to A'$ and $\nu_{n+1} = \alpha' \circ F(\nu_n)$. To see ν is a cone we prove by induction on n that the following diagram commutes.



This is clear for $n = 0$. For $n+1$ we have: $\nu_{n+2} \circ F^{n+1}(\perp_{F\perp}) = \alpha' \circ F(\nu_{n+1}) \circ F^{n+1}(\perp_{F\perp}) = \alpha' \circ F(\nu_{n+1} \circ F^n(\perp_{F\perp})) = \alpha' \circ F(\nu_n)$ (by induction assumption) $= \nu_{n+1}$. Now the uniqueness of f will follow when we show it is the mediating morphism from μ to ν; here we use induction on n to show $\nu_n = f \circ \mu_n$. This is clear for $n = 0$. For $n+1$ we have: $f \circ \mu_{n+1} = f \circ \alpha \circ F(\mu_n)$ (by the definition of α) $= \alpha' \circ F(f) \circ F(\mu_n)$ (f is a homomorphism) $= \alpha' \circ F(f \circ \mu_n) = \alpha' \circ \nu_n$ (by induction assumption) $= \nu_{n+1}$.

Secondly, to show that f exists, let it be the mediating morphism from μ to ν (so that $\nu_n = f \circ \mu_n$ for all integers, n) where ν is defined as above. We will show that $f \circ \alpha$ and $\alpha' \circ Ff$ are both mediating arrows from $F\mu$ to ν^-, thus demonstrating that they are equal and that f is an F-homomorphism as required.

In the first case, $(f \circ \alpha) \circ F\mu_n = f \circ \mu_{n+1}$ (by definition of α) $= \nu_{n+1}$ (by definition of f). In the second case, $(\alpha' \circ Ff) \circ F\mu_n = \alpha' \circ F(f \circ \mu_n) = \alpha' \circ F\nu_n$ (by definition of f) $= \nu_{n+1}$ (by definition of ν). This concludes the proof. $\square$

In the case of partial orders, our method of constructing least fixed-points always works if K is an ω-complete partial order and $F: K \to K$ is ω-continuous. Here an *ω-complete pointed partial order* (cpo) is a partial order which has l.u.b.s of all increasing ω-sequences and which has a least element; it is termed an "ω-complete partial order" or even just a "complete partial order" elsewhere. Also a function $F: K \to L$ of partial orders is *ω-continuous* if and only if it is monotonic and preserves l.u.b.s of increasing ω-sequences, that is, if whenever $\langle x_n \rangle_{n \in \omega}$ is an increasing ω-sequence such that $\bigsqcup_K x_n$ exists, then $F(\bigsqcup_K x_n) = \bigsqcup_L F(x_n)$. We make analogous definitions for categories:

DEFINITION 3. A category, K, is an *ω-complete pointed category* (shortened below to *ω-category*) if and only if it has an initial element, and every ω-chain has a colimit.

DEFINITION 4. Let $F: \mathbf{K} \to \mathbf{L}$ be a functor. It is *ω-continuous* if and only if it preserves $\boldsymbol{\omega}$-colimits; that is, whenever Δ is an $\boldsymbol{\omega}$-chain and $\mu: \Delta \to A$ is a colimiting

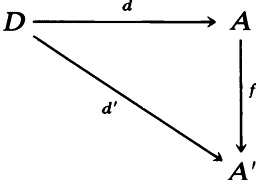
cone, then $F\mu: F\Delta \to FA$ is also a colimiting cone. (The reader is warned that this definition is dual to the notion of continuity of functors usual in category theory (MacLane [21]); this is done in order to maintain the analogy with partial orders.)

Clearly, when $\mathbf{K}$ is an ω-category and $F: \mathbf{K} \to \mathbf{K}$ is ω-continuous, the conditions of the basic lemma are satisfied. In § 3 we will give the conditions for this to be the case. In the sequel to this paper (see also Plotkin and Smyth [28]), we will show that, in the presence of a universal object, the conditions of the basic lemma may be satisfied without requiring that $\mathbf{K}$ be an ω-category and that F be ω-continuous. Usually, we can completely avoid direct verification of the conditions of the basic lemma, or of whether something is an ω-category or is ω-continuous. Of course sometimes, as in the case of **Set**, it is already known that we have an ω-category and that such functors as $+$ and $\times$ are ω-continuous (see Lehmann and Smyth [20]). One case in which there is, so far, no alternative to direct verification is with Lehmann's category, **Dom** of small ω-categories and ω-continuous functors [18] (in this case **Dom**-categories might provide a good general setting).

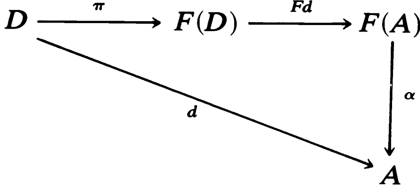
Note that it is only necessary to check that the basic categories are ω-categories and the basic functors are ω-continuous; one easily proves that any denumerable product of ω-categories is an ω-category, that all constant and projection functors are ω-continuous, and that composition and tupling preserve ω-continuity. Thus to solve (1) one only needs to check that $+$ and $\to$ are ω-continuous; for (2) and (3) one looks at $+$ and $\times$.

The original work on models of the pure λ-calculus (Scott [34], Wadsworth [42]) did not solve (4) as $\lim_{\to} \langle F^n(\perp), F^n(\perp_{F\perp})\rangle$, but rather as $\lim_{\to} \langle F^n(D), F^n(\pi)\rangle$ for an object D and a morphism $\pi: D \to F(D)$. It turns out that this solution is, essentially, the initial fixed point of a functor F_π, derived from F, over the comma category $(D\downarrow\mathbf{K})$ of "objects over D" (see MacLane [21]). The analogous idea in partial orders is that of a least fixed point greater than some fixed element, d.

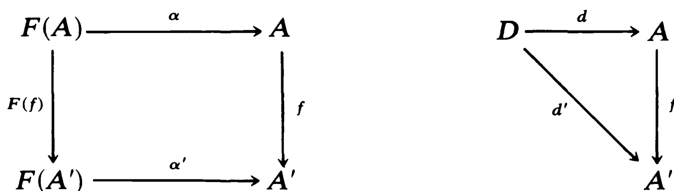
The comma category $(D\downarrow\mathbf{K})$ has as objects pairs (A, d) where A is an object of $\mathbf{K}$ and $d: D \to A$; the morphisms $f: (A, d) \to (A', d')$ are the morphisms $f: A \to A'$ of $\mathbf{K}$ such that the following diagram commutes.



Now the endofunctor $F_\pi: (D\downarrow\mathbf{K}) \to (D\downarrow\mathbf{K})$ can be obtained by putting $F(A, d) = (FA, (Fd)\circ\pi)$ for objects and $F_\pi(f) = F(f)$ for morphisms. Then one can see that an F_π-algebra is a pair $((A, d), \alpha)$ where A is an object of $\mathbf{K}$, and $d: D \to A$ and $\alpha: FA \to A$ and the following diagram commutes.



A homomorphism f: $((A, d), \alpha) \to ((A', d'), \alpha')$ is a morphism $f: A \to A'$ such that the following two diagrams commute.



Let us assume, for simplicity, that **K** is an ω-category and F is ω-continuous. Then $(D\downarrow\mathbf{K})$ is also an ω-category. Its initial object is (D, id_D). For colimits suppose $\Delta = \langle (A_n, d_n), f_n\rangle$ is an $\boldsymbol{\omega}$-chain in $D\downarrow\mathbf{K}$. Then it is straightforward to check that $\mu:\langle A_n, f_n\rangle \to A$ is a (colimiting) cone in **K** if and only if $\mu:\Delta\to(A, \mu_0\circ d_0)$ is in $(D\downarrow\mathbf{K})$ (and they have the same mediating morphisms). This makes it easy to show that F is ω-continuous.

Now, applying the basic lemma to $(D\downarrow\mathbf{K})$ and F_π, we have to find a colimiting cone $\mu:\langle F_\pi^n(\perp), F_\pi^n(\perp_{F_\pi(\perp)})\rangle\to(A, d)$. One sees, by induction on n, that $F_\pi^n(\perp_{(D\downarrow\mathbf{K})})$ is $\langle F^n(D), d_n\rangle$ where $d_n = F^{n-1}(\pi)\circ\cdots\circ\pi: D\to F^n(D)$ and that $F_\pi^n(\perp_{F_\pi(\perp)}) = F^n(\pi)$. So from the above remarks one can take μ to be a colimiting cone, $\mu:\langle F^n(D), F^n(\pi)\rangle\to A$, also defining A, and put $d = \mu_0\circ d_0$. Then, by the basic lemma, the initial F_π-algebra is $((A, d), \alpha)$ where α is the mediating morphism from $F\mu$ to μ^- (which can be taken in **K**). Thus we see that $A = \lim_\to \langle F^n(D), F^n(\pi)\rangle$, together with its colimiting cone, determines the initial F_π-algebra. Thus we have characterized the original inverse limit construction in universal terms.

3. O-categories. In most of the categories used for the denotational semantics of programming languages, the hom-sets have a natural partial order structure. When solving recursive domain equations only the projections are considered, and they are easily defined in terms of the partial order. Wand [44] introduced **O**-categories to study such categories at a suitably abstract level. We now present a view of his work as providing theorems and definitions which facilitate the application of the basic lemma, as outlined in the introduction.

DEFINITION 5. A category, **K**, is an **O**-category if and only if (i) every hom-set is a partial order in which every ascending ω-sequence has a l.u.b. and (ii) composition of morphisms is an ω-continuous operation with respect to this partial order.

Note that if **K** is an **O**-category, so is $\mathbf{K}^{\mathrm{op}}$, and if **L** is another so is $\mathbf{K}\times\mathbf{L}$. Here the orders are inherited, and in the case of $\mathbf{K}^{\mathrm{op}}$ we have $f^{\mathrm{op}}\sqsubseteq g^{\mathrm{op}}$ if and only if $f\sqsubseteq g$, for any morphisms f and g of **K** (in general we will omit the superscript when writing morphisms in opposite categories).

As it happens **O**-categories are a particular case of a general theory of **V**-categories where **V** is any closed category (MacLane [21]); here **O** is the category whose objects are those partial orders with l.u.b.s of all increasing ω-sequences and whose morphisms are the ω-continuous functions between the partial orders. We will not use any of the general theory but just take over the idea of endowing the morphism sets with extra structure—in this case that of being an object in **O**.

DEFINITION 6. Let K be an **O**-category and let $A\to^f B\to^g A$ be arrows such that $g\circ f = \mathrm{id}_A$ and $f\circ g\sqsubseteq \mathrm{id}_B$. Then we say that $\langle f, g\rangle$ is a *projection pair from* A *to* B, that g is a *projection* and that f is an *embedding*.

LEMMA 3. *Let* $\langle f, g\rangle$ *and* $\langle f', g'\rangle$ *both be projection pairs from* A *to* B, *in an* **O**-*category* K. *Then* $f\sqsubseteq f'$ *if and only if* $g\sqsupseteq g'$.

Proof. If $f\sqsubseteq f'$ then $g\sqsupseteq g\circ f'\circ g'\sqsupseteq g\circ f\circ g' = g'$. Conversely, if $g\sqsupseteq g'$ then $f = f\circ g'\circ f'\sqsubseteq f\circ g\circ f'\sqsubseteq f'$. □

So, in particular, it follows that one half of a projection pair determines the other; if f is an embedding we write f^R for the corresponding projection which we call the *right adjoint* of f; if g is a projection we write g^L for the corresponding embedding which we call the *left adjoint* of g. (Our use of the term "adjoint" is only a matter of convenience; when the **K**-objects are posets and the morphisms are monotonic maps, it agrees with a standard usage.)

Given any **O**-category **K** we can now form the subcategory, $\mathbf{K}^E$, of the embeddings. For the identity morphism $\mathrm{id}_A : A \to A$ is an embedding with $\mathrm{id}_A^R = \mathrm{id}_A$, and if $A \to^f B \to^{f'} C$ are embeddings, so is $(f' \circ f)$ with $(f' \circ f)^R = f^R \circ f'^R$. Equally, we can form $\mathbf{K}^P$, the subcategory of the projections. We do *not* try to take either of these as **O**-categories under the induced ordering; indeed they are, in general, not **O**-categories.

3.1. Duality for O-categories. Our discussion of adjoints shows that we have the duality $\mathbf{K}^E \cong (\mathbf{K}^P)^{\mathrm{op}}$ (and so too $\mathbf{K}^P \cong (\mathbf{K}^E)^{\mathrm{op}}$). There is another kind of duality arising from the fact that an embedding in $\mathbf{K}^{\mathrm{op}}$ is just a projection in **K**. Thus $(\mathbf{K}^{\mathrm{op}})^E \cong \mathbf{K}^P$ (and so $(\mathbf{K}^{\mathrm{op}})^P \cong \mathbf{K}^E$). Thus an $\boldsymbol{\omega}$-chain in $(\mathbf{K}^{\mathrm{op}})^E$ can be considered as an $\boldsymbol{\omega}^{\mathrm{op}}$-chain in $\mathbf{K}^P$, and a colimiting cone in $(\mathbf{K}^{\mathrm{op}})^E$ can be identified with a limiting cone in $\mathbf{K}^P$. We therefore have the following dualities (for an **O**-category **K**): embedding/projection, $\boldsymbol{\omega}$-chain in $\mathbf{K}^E/\boldsymbol{\omega}^{\mathrm{op}}$-chain in $\mathbf{K}^P$, colimiting cone in $\mathbf{K}^E$/limiting cone in $\mathbf{K}^P$. A further example, **O**-colimit/**O**-limit, is provided by Definition 7.

These observations will be used in the proof of Theorem 2.

Our first theorem is trivial but does illustrate the idea of transferring properties from **K** to $\mathbf{K}^E$.

THEOREM 1. *Let* **K** *be an* **O**-*category which has a terminal object,* $\perp$, *and in which every* $\hom(A, B)$ *has a least element,* $\perp_{A,B}$. *Suppose too that composition is left-strict in the sense that for any* $f: A \to B$ *we have* $\perp_{B,C} \circ f = \perp_{A,C}$. *Then* $\perp$ *is the initial object of* $\mathbf{K}^E$.

Proof. First, if $f, f' : \perp \to A$ are both embeddings, then they have a common right adjoint, as $\perp$ is terminal in K, and so, by Lemma 3, they are equal. Second, $\perp_{\perp,A} : \perp \to A$ is an embedding with right adjoint $\perp_{A,\perp}$. For $\perp_{A,\perp} \circ \perp_{\perp,A} : \perp \to \perp$ must be $\mathrm{id}_\perp$ the unique map from $\perp$ to $\perp$ and $\perp_{\perp,A} \circ \perp_{A,\perp} = \perp_{A,A}$ (by left-strictness)$\sqsubseteq \mathrm{id}_A$. □

To make the connection with the basic lemma, we need to be able to relate **O**-notions (expressed in terms of the ordering of hom-sets) to ω-notions (expressed in terms of $\boldsymbol{\omega}^{\mathrm{op}}$-limits/$\boldsymbol{\omega}$-colimits). This is the main purpose of Theorem 2. Another way to view this result, exemplified further in the ensuing discussion, is that it is concerned with the correspondence between local properties (that is, properties local to particular hom-sets) and global properties of the category. Yet another way to regard Theorem 2 is to note that it contains the most complete and general formulation of the limit-colimit coincidence, remarked in Scott [33], that we have been able to develop.

DEFINITION 7. Let **K** be an **O**-category and $\mu : \Delta \to A$ a cone in $\mathbf{K}^E$, where Δ is the ω-chain $\langle A_n, f_n \rangle$. Then μ is an **O**-*colimit* of Δ provided that $\langle \mu_n \circ \mu_n^R \rangle_n$ is increasing with respect to the ordering of $\hom(A, A)$ and $\bigsqcup_n \mu_n \circ \mu_n^R = \mathrm{id}_A$. Dually an **O**-*limit* of an $\boldsymbol{\omega}^{\mathrm{op}}$-chain Γ in K^P is a cone $\nu : B \to \Gamma$ in **K** such that $\langle \nu_n^L \circ \nu_n \rangle_n$ is increasing and $\bigsqcup_n \nu_n^L \circ \nu_n = \mathrm{id}_B$.

Obviously, μ is an **O**-colimit of Δ if μ^R is an **O**-limit of Δ^R, where we define Δ^R to be $\langle A_n, f_n^R \rangle$ and μ^R to be $\langle \mu_n^R \rangle_{n \in \omega}$.

THEOREM 2. *Let* **K** *be an* **O**-*category and* Δ *be an* $\boldsymbol{\omega}$-*chain in* $\mathbf{K}^E$. *Consider the six properties*

(a) Δ *has a colimit in* **K**,

(b) Δ^R *has a limit in* $\mathbf{K}$,

(c) Δ *has an* $\mathbf{O}$*-colimit*,

(d) Δ^R *has an* $\mathbf{O}$*-limit*,

(e) Δ *has a colimit in* $\mathbf{K}^E$,

(f) Δ^R *has a limit in* $\mathbf{K}^P$.

We have: properties (a)–(d) *are equivalent* (*to each other*); (a) *implies* (e); *and* (e) *is equivalent to* (f). *Indeed, a cone* $\mu:\Delta\to A$ (*a cone* $\nu:A\to\Delta^R$) *is colimiting* (*limiting*) *in* $\mathbf{K}$ *if and only if* $\mu(\nu)$ *is an* $\mathbf{O}$*-colimit* ($\mathbf{O}$*-limit*); *any colimiting* (*limiting*) *cone of* $\Delta(\Delta^R)$ *in* $\mathbf{K}$ *is also a colimiting* (*limiting*) *cone of* $\Delta(\Delta^R)$ *in* $\mathbf{K}^E(\mathbf{K}^P)$; *and* $\mu:\Delta\to A$ *is colimiting in* $\mathbf{K}^E$ *if and only if* μ^R *is limiting in* $\mathbf{K}^P$.

Proof. We establish Propositions A–E which are jointly equivalent to the result. We always suppose that Δ has the form $\langle A_n, f_n\rangle$. Note that $\langle \mu'_n\circ\mu_n^R\rangle_{n\in\omega}$ is increasing for any cones $\mu:\Delta\to A$ and $\mu':\Delta\to A'$ *in* $\mathbf{K}$, as we have: $\mu'_n\circ\mu_n^R = (\mu'_{n+1}\circ f_n)\circ(\mu_{n+1}\circ f_n)^R = \mu'_{n+1}\circ(f_n\circ f_n^R)\circ\mu_{n+1}^R \sqsubseteq \mu'_{n+1}\circ\mu_{n+1}^R$.

PROPOSITION A. *If* $\mu:\Delta\to A$ *is an* $\mathbf{O}$*-colimit, then* μ *is colimiting in both* $\mathbf{K}$ *and* $\mathbf{K}^E$.

Proof of Proposition (A). Choose a cone $\mu':\Delta\to A$ in $\mathbf{K}$, and suppose $\theta:A\to A'$ is a morphism from μ to μ' (i.e., for all n, $\theta\circ\mu_n=\mu'_n$). Then θ is determined by:

$$\theta = \theta\circ\bigsqcup \mu_n\circ\mu_n^R = \bigsqcup(\theta\circ\mu_n)\circ\mu_n^R = \bigsqcup \mu'_n\circ\mu_n^R.$$

This proves uniqueness; for existence we can define θ as $\bigsqcup \mu'_n\circ\mu_n^R$, since the above remark shows that $\langle\mu'_n\circ\mu_n^R\rangle$ is increasing, and calculate:

$$\theta\circ\mu_m = \left(\bigsqcup_{n\geqq m}\mu'_n\circ\mu_n^R\right)\circ\mu_m = \bigsqcup_{n\geqq m}\mu'_n\circ\mu_n^R\circ\mu_n\circ f_{mn} = \bigsqcup_{n\geqq m}\mu'_n\circ f_{mn} = \mu'_m.$$

So μ is colimiting in $\mathbf{K}$. For $\mathbf{K}^E$ it only remains to show that if μ' is actually a cone in $\mathbf{K}^E$ then θ as defined above is an embedding. By the remark above, $\langle\mu_n\circ\mu'^R_n\rangle_{n\in\omega}$ is increasing; to show that θ is an embedding, we prove that it has the right adjoint $\theta^R=\bigsqcup\mu_n\circ\mu'^R_n$.

On the one hand we have:

$$(\bigsqcup\mu_n\circ\mu'^R_n)\circ(\bigsqcup\mu'_n\circ\mu_n^R) = \bigsqcup\mu_n\circ(\mu'^R_n\circ\mu'_n)\circ\mu_n^R = \bigsqcup\mu_n\circ\mu_n^R = \mathrm{id}_A;$$

on the other hand we have:

$$(\bigsqcup\mu'_n\circ\mu_n^R)\circ(\bigsqcup\mu_n\circ\mu'^R_n) = \bigsqcup\mu'_n\circ(\mu_n^R\circ\mu_n)\circ\mu'^R_n = \bigsqcup\mu'_n\circ\mu'^R_n \sqsubseteq \mathrm{id}_{A'}. \quad \square$$

Dually, we have Proposition B:

PROPOSITION B. *If* $\nu:A\to\Delta^R$ *is an* $\mathbf{O}$*-limit, then* ν *is limiting in both* $\mathbf{K}$ *and* $\mathbf{K}^P$.

PROPOSITION C. *If* $\nu:A\to\Delta^R$ *is limiting in* $\mathbf{K}$, *then each* ν_n *is a projection and* ν *is an* $\mathbf{O}$*-limit of* Δ^R.

Proof of Proposition (C). For each A_m we can define a cone $\nu^{(m)}:A_m\to\Delta^R$ in $\mathbf{K}$ by:

$$\nu_n^{(m)} = \begin{cases} f_{mn} & (m\leqq n),\\ (f_{nm})^R & (m>n).\end{cases}$$

To see that $\nu^{(m)}$ is a cone, we first check that if $r\geqq\max(m,n)$ then $\nu_n^{(m)}=f_{nr}^R\circ f_{mr}$. For if $m\leqq n$, then $f_{nr}^R\circ f_{mr}=f_{nr}^R\circ(f_{nr}\circ f_{mn})=\nu_n^{(m)}$; if $m>n$ then $f_{nr}^R\circ f_{mr}=$

$(f_{mr} \circ f_{nm})^R \circ f_{mr} = f_{nm}^R = \nu_n^{(m)}$. Now we see that $\nu^{(m)}$ is a cone, as follows:

$$\begin{aligned} f_n^R \circ \nu_{n+1}^{(m)} &= f_n^R \circ (f_{(n+1)r}^R \circ f_{mr}) \qquad \text{(by the above with } r = \max(m, n+1)) \\ &= (f_{(n+1)r} \circ f_n)^R \circ f_{mr} = f_{nr}^R \circ f_{mr} \\ &= \nu_n^{(m)} \qquad \text{(by the above).} \end{aligned}$$

Now as $\nu : A \to \Delta^R$ is a limiting cone there is, for each m, a mediating morphism $\theta_m : A_m \to A$ from $\nu^{(m)}$ to ν. So we have for all m and n: $\nu_n \circ \theta_m = \nu_n^{(m)}$. Putting n equal to m we find that $\nu_m \circ \theta_m = \mathrm{id}_{A_m}$, which is half the proof that ν_m is a projection with $\nu_m^L = \theta_m$.

Next we connect up the θ_m's by showing that $\theta_m = \theta_{m+1} \circ f_m$, which holds since $\theta_{m+1} \circ f_m$ mediates between $\nu^{(m)}$ and ν as can be seen from:

$$\begin{aligned} \nu_n \circ (\theta_{m+1} \circ f_m) &= \nu_n^{(m+1)} \circ f_m \qquad (\theta_{m+1} \text{ mediates between } \nu^{(m+1)} \text{ and } \nu) \\ &= f_{nr}^R \circ f_{(m+1)r} \circ f_m \qquad (\text{with } r = \max(m+1, n)) \\ &= f_{nr}^R \circ f_{mr} = \nu_n^{(m)}. \end{aligned}$$

This, in turn, enables us to show that $\langle \theta_m \circ \nu_m \rangle_{m \in \omega}$ is increasing: $\theta_m \circ \nu_m = \theta_{m+1} \circ f_m \circ f_m^R \circ \nu_{m+1} \sqsubseteq \theta_{m+1} \circ \nu_{m+1}$. Consequently, as $\mathbf{K}$ is an $\mathbf{O}$-category, we may define $\theta : A \to A$ by: $\theta = \bigsqcup_{m \in \omega} \theta_m \circ \nu_m$. To finish the proof, we show that $\theta = \mathrm{id}_A$ (as then we also have $\theta_m \circ \nu_m \sqsubseteq \theta = \mathrm{id}_A$, completing the proof that $\nu_m^L = \theta_m$). This follows from the fact that θ mediates between ν and itself as is shown by:

$$\nu_n \circ \theta = \nu_n \circ \bigsqcup_{m \geqq n} \theta_m \circ \nu_m = \bigsqcup_{m \geqq n} (\nu_n \circ \theta_m) \circ \nu_m = \bigsqcup_{m \geqq n} \nu_n^{(m)} \circ \nu_m = \bigsqcup_{m \geqq n} f_{nm}^R \circ \nu_m = \nu_n.$$

□

By duality:

PROPOSITION D. *If $\mu : \Delta \to A$ is colimiting in $\mathbf{K}$, then each μ_n is an embedding, and μ is an $\mathbf{O}$-colimit.*

PROPOSITION E. *$\mu : \Delta \to A$ is colimiting in $\mathbf{K}^E$ if and only if $\mu^R : A \to \Delta^R$ is limiting in $\mathbf{K}^P$.*

Proof of Proposition E. Obvious. □

This completes the proof of Theorem 2. □

Our first main use of Theorem 2 will be to establish the evident corollary that if $\mathbf{K}$ is an $\mathbf{O}$-category which has all $\boldsymbol{\omega}^{\mathrm{op}}$-limits, then $\mathbf{K}^E$ has all $\boldsymbol{\omega}$-colimits. The second main use concerns functors, but first a definition and another corollary prove convenient.

DEFINITION 8. An $\mathbf{O}$-category $\mathbf{K}$ is said to have *locally determined ω-colimits of embeddings* provided that, whenever Δ is an $\boldsymbol{\omega}$-chain in $\mathbf{K}^E$ and $\mu : \Delta \to A$ is a cone in $\mathbf{K}^E$, μ is colimiting in $\mathbf{K}^E$ if and only if μ is an $\mathbf{O}$-colimit. (Note that, by Theorem 2, only half of the implication can ever be in doubt.)

COROLLARY (to Theorem 2). *Suppose that the $\mathbf{O}$-category $\mathbf{K}$ has all $\boldsymbol{\omega}^{\mathrm{op}}$-limits (i.e., every $\boldsymbol{\omega}^{\mathrm{op}}$-chain in $\mathbf{K}$ has a limit). Then $\mathbf{K}$ has locally determined $\boldsymbol{\omega}$-colimits of embeddings.*

Proof. Suppose that $\mu : \Delta \to A$ is colimiting in $\mathbf{K}^E$. Let $\nu' : A' \to \Delta^R$ be a limit with respect to $\mathbf{K}$ for Δ^R. By Theorem 2, ν' is an $\mathbf{O}$-limit for Δ^R. Thus, ν'^L is an $\mathbf{O}$-colimit for Δ, so that by Theorem 2, ν'^L is colimiting in $\mathbf{K}^E$. Hence μ is isomorphic with ν'^L and must itself be an $\mathbf{O}$-colimit (the property of being an $\mathbf{O}$-colimit is trivially invariant under isomorphism of cones). □

By duality, the conclusion of the corollary also follows from the assumption that **K** has all **ω**-colimits, but that is not so useful. At present we lack an example of an **O**-category which does not have locally determined **ω**-colimits of embeddings.

As mentioned in the Introduction, a major reason for introducing $\mathbf{K}^E$ is to enable us to consider contravariant functors on **K** as covariant ones on $\mathbf{K}^E$; the remainder of this section develops the idea. We consider throughout (that is, to the end of the section) three **O**-categories **K**, **L**, **M** and a covariant functor $T:\mathbf{K}^{\mathrm{op}}\times\mathbf{L}\to\mathbf{M}$. Purely covariant functors are included by suppressing **K** (i.e., taking it to be the trivial one-object category), contravariant ones by suppressing **L**, and mixed ones by taking **K** and **L** to be product categories as required.

DEFINITION 9. The functor T is *locally monotonic* if and only if it is monotonic on the hom-sets; that is, for $f, f':A\to B$ in $\mathbf{K}^{\mathrm{op}}$, $g, g':C\to D$ in **L**, if $f\sqsubseteq f'$ and $g\sqsubseteq g'$, then $T(f,g)\sqsubseteq T(f',g')$.

LEMMA 4. *If T is locally monotonic, a covariant functor $T^E:\mathbf{K}^E\times\mathbf{L}^E\to\mathbf{M}^E$ can be defined by putting, for objects A, $B: T^E(A,B)=T(A,B)$ and for morphisms $f,g: T^E(f,g)=T((f^R)^{\mathrm{op}},g)$.*

Proof. First if $f:A\to B$ in $\mathbf{K}^E$ and $g:C\to D$ in $\mathbf{L}^E$, then $T(f^R,g)$ is an embedding with right adjoint $T(f,g^R)$ as: $T(f,g)^R\circ T(f^R,g)=T(f^R\circ f, g^R\circ g)=T(\mathrm{id}_A,\mathrm{id}_C)=\mathrm{id}_{T(A,C)}$ and also: $T(f^R,g)\circ T(f,g^R)=T(f\circ f^R, g\circ g^R)\sqsubseteq T(\mathrm{id}_B,\mathrm{id}_D)$ (by local monotonicity) $=\mathrm{id}_{T(B,D)}$.

Secondly, $T^E(\mathrm{id}_A,\mathrm{id}_C)=T(\mathrm{id}_A^R,\mathrm{id}_C)=T(\mathrm{id}_A,\mathrm{id}_C)=\mathrm{id}_{T(A,C)}$.

Thirdly if $A\to^f A'\to^{f'} A'$ in $\mathbf{K}^E$ and $B\to^g B'\to^{g'} B''$ in $\mathbf{L}^E$ then

$$T^E(f',g')\circ T^E(f,g)=T(f'^R,g')\circ T(f^R,g)=T(f^R\circ f'^R, g'\circ g)$$
$$=T((f'\circ f)^R, g'\circ g)=T^E(f'\circ f, g'\circ g). \qquad \square$$

Under some assumptions on **K** and **L**, we can transfer a local continuity property of T to the ω-continuity of T^E.

DEFINITION 10. The functor T is *locally continuous* (equivalently, is an **O**-functor) if and only if it is ω-continuous on the hom-sets—that is, if $f_n:A\to B$ is an increasing ω-sequence in $\mathbf{K}^{\mathrm{op}}$ and $g_n:C\to D$ is one in **L**, then $T(\bigsqcup_{n\in\omega}f_n,\bigsqcup_{n\in\omega}g_n)=\bigsqcup_{n\in\omega}T(f_n,g_n)$.

Note that the constant and projection functors are locally continuous, and that the locally continuous functors are closed under composition, tupling and taking opposite functors.

THEOREM 3. *Suppose T is locally continuous and both* **K** *and* **L** *have locally determined* **ω***-colimits of embeddings. Then T^E is ω-continuous.*

Proof. Let $\Delta=\langle\langle A_n,B_n\rangle,\langle f_n,g_n\rangle\rangle$ be an ω-chain in $\mathbf{K}^E\times\mathbf{L}^E$ and let $\mu:\Delta\to\langle A,B\rangle$ be colimiting, where $\mu=\langle\sigma_n,\tau_n\rangle_{n\in\omega}$. Then $\langle\sigma_n\rangle:\langle A_n,f_n\rangle\to A$ is colimiting in $\mathbf{K}^E$ and $\langle\tau_n\rangle:\langle B_n,g_n\rangle\to B$ is colimiting in $\mathbf{L}^E$. It follows by the assumptions on **K** and **L** that $\mathrm{id}_A=\bigsqcup\sigma_n\circ\sigma_n^R$ and $\mathrm{id}_B=\bigsqcup\tau_n\circ\tau_n^R$ with the right-hand sides increasing.

We have to show that $T^E(\mu):T^E(\Delta)\to T^E(A,B)$ is a colimiting cone in $\mathbf{M}^E$, which we do by showing that it is an **O**-limit (and then applying Theorem 2). First

$$\langle T^E(\mu_n)\circ T^E(\mu_n)^R\rangle_{n\in\omega}=\langle T(\sigma_n^R,\tau_n)\circ T(\sigma_n^R,\tau_n)^R\rangle_{n\in\omega}$$
$$=\langle T(\sigma_n^R,\tau_n)\circ T(\sigma_n,\tau_n^R)\rangle_{n\in\omega}$$
$$=\langle T(\sigma_n\circ\sigma_n^R,\tau_n\circ\tau_n^R)\rangle_{n\in\omega},$$

which is increasing as $\langle \sigma_n \circ \sigma_n^R \rangle_{n\in\omega}$ and $\langle \tau_n \circ \tau_n^R \rangle_{n\in\omega}$ are, and as T is locally monotonic. Next,

$$
\begin{aligned}
\bigsqcup_{n\in\omega} T^E(\mu_n) \circ T^E(\mu_n)^R &= \bigsqcup_{n\in\omega} T(\sigma_n \circ \sigma_n^R, \tau_n \circ \tau_n^R) \qquad \text{(by the above)} \\
&= T\Big(\bigsqcup_{n\in\omega} \sigma_n \circ \sigma_n^R, \bigsqcup_{n\in\omega} \tau_n \circ \tau_n^R \Big) \qquad \text{(by local continuity)} \\
&= T(\mathrm{id}_A, \mathrm{id}_B) \qquad \text{(by the above)} \\
&= \mathrm{id}_{T(A,B)}. \qquad \square
\end{aligned}
$$

4. Examples. In this section we present several useful **O**-categories where our general theory can be applied. In general we only sketch proofs and even omit them when they are either evident or not directly relevant to the main line of the argument (but for Example 2, see [22]). The first example is elementary, being little more than an illustration of the ideas. The second example is the category of cpos where all the needed domains for denotational semantics can be constructed. This is an approach where as few axioms as possible are imposed. That makes the axioms very easy to understand but admits domains of little computational interest. The third example illustrates various completeness conditions that are weaker than Scott's original requirement of complete lattices, which rules out some natural and useful domains. The fourth example considers axioms of algebraicity and continuity which attempt to force the domains to be computationally realistic. One use of the completeness axioms (cf. Example 3) is that when combined with algebraicity (or continuity), function spaces exist although they need not otherwise do so. Example 5 turns in a different direction, suggesting a certain category of continuous algebras as an appropriate place for the semantics of programming languages with nondeterministic constructs. Example 6 considers a category of relations over cpos where it is possible to construct a wide variety of recursively specified relations; these are useful when relating different semantics.

Example 1. *Partial functions.* We consider the category **Pfn** of sets and the partial functions between them. The partial order relation between partial functions is just set inclusion and can also be defined for any $f, g: A \to B$ by

$$f \sqsubseteq g \equiv \forall a \in A . f(a)\downarrow \supset f(a) = g(a)$$

(where $f(a)\downarrow$ means that $f(a)$ is defined). Clearly limits of increasing ω-sequences $\langle f_n \rangle_{n\in\omega}$ exist being just the set-union $\bigcup f_n$ so that

$$(\bigsqcup f_n)(a) = \begin{cases} b & (\exists n . f_n(a) = b), \\ \text{undefined} & (\forall n . f_n(a) \text{ is undefined}). \end{cases}$$

It is easy to see that $f: A \to B$ is an embedding if and only if it is total and one-to-one; in that case $f^R = f^{-1}$. Thus to within isomorphism, embeddings are just inclusions. Note that the totally undefined function $\varnothing: A \to B$ is the least element of hom (A, B) and that composition is left strict in the sense of Theorem 1. The empty set is the terminal object, the unique mapping being $\varnothing: A \to \varnothing$ and so the conditions of Theorem 1 apply, and we see that $\varnothing$ is the initial object in $\mathbf{Pfn}^E$ (and of course that is trivial anyway).

Turning to $\boldsymbol{\omega}$-colimits in $\mathbf{Pfn}^E$, it is obvious that they exist, as $\boldsymbol{\omega}$-chains $\Delta = \langle A_n, f_n \rangle$ are, to within isomorphism, just increasing sequences $A_0 \subseteq A_1 \subseteq \cdots$, and so $A = \bigcup A_n$

is the colimiting object with the colimiting cone of inclusions $\mu_n : A_n \subseteq A$. This also follows from Theorem 2, since **Pfn** has $\boldsymbol{\omega}^{\mathrm{op}}$-limits. These are constructed as in **Set**: let $\Delta = \langle A_n, f_n \rangle$ be an $\boldsymbol{\omega}^{\mathrm{op}}$-chain; put $A = \{a \in \Pi A_n | \forall n.\, a_n = f_n(a_{n+1})\}$ and define $\nu_n : A \to A_n$ to be the "nth projection" $a \mapsto a_n$. Then A is the limit of Δ and ν is the colimiting cone.

Turning to functors, we define the Cartesian product on morphisms $f: A \to A'$ and $g: B \to B'$ by:

$$(f \times g)(a, b) = \begin{cases} \langle fa, gb \rangle & (\text{if } f(a) \text{ and } g(b)\downarrow), \\ \text{undefined} & (\text{otherwise}). \end{cases}$$

This makes the Cartesian product locally continuous and so, by Theorem 3, ω-continuous. Amusingly, the categorical product exists but is different from Cartesian product and is not even locally monotonic. On the other hand, the categorical sum exists and is locally continuous. On objects it is disjoint union:

$$A + B = (\{0\} \times A) \cup (\{1\} \times B)$$

and for morphisms $f: A \to A'$ and $g: B \to B'$,

$$(f+g)(c) = \begin{cases} \langle 0, f(a) \rangle & (\exists a \in A.\, c = \langle 0, a \rangle \text{ and } f(a)\downarrow), \\ \langle 1, g(b) \rangle & (\exists b \in B.\, c = \langle 1, b \rangle \text{ and } g(b)\downarrow), \\ \text{undefined} & (\text{otherwise}). \end{cases}$$

Finally, there is a natural function-space construction defined by

$$A \to B = \hom(A, B),$$

and for $f: A' \to A$ and $g: B \to B'$

$$(f \to g)(h) = g \circ h \circ f.$$

This is not even locally monotonic, and so Theorem 3 does not apply. This is as expected, as the recursive domain equation (1) *cannot* have solutions in **Pfn** (for nontrivial At) by evident cardinality considerations. For another example of an elementary **O**-category, the reader can consider the category **Rel** of binary relations between sets, with the subset ordering on relations.

Example 2. *Complete partial orders.* We consider the category **CPO** (essentially introduced in § 2) of complete partial orders and ω-continuous functions. It is an **O**-category when we define the order between morphisms $f, g: A \to B$ in the natural pointwise fashion

$$f \sqsubseteq g \equiv \forall a \in A.\, f(a) \sqsubseteq g(a).$$

Limits of increasing ω-sequences of morphisms are defined pointwise. The conditions of Theorem 1 are satisfied, as the trivial one-point partial order is the terminal object and any given hom (A, B) has least element $a \mapsto \bot_B$ and composition is left-strict.

Turning to $\boldsymbol{\omega}$-limits (to apply Theorem 3), let $\Delta = \langle A_n, f_n \rangle$ be an $\boldsymbol{\omega}$-chain and construct $\nu: A \to \Delta$ as in the case of **Pfn** taking the partial order on A componentwise so that for any a, a' in A.

$$a \sqsubseteq a' \equiv \forall n.\, a_n \sqsubseteq a'_n.$$

This makes A a cpo with least upper bounds of increasing ω-sequences, $\langle a^{(n)} \rangle_{n \in \omega}$ taken componentwise

$$\bigsqcup_n a^{(n)} = \left\langle \bigsqcup_m a_m^{(n)} \right\rangle_{m \in \omega}$$

and with least element $\langle \bigsqcup_{n \geq m} f_{nm}(\bot_{A_n}) \rangle$. Further ν is a cone of continuous functions and it is limiting, as if $\nu' : A' \to \Delta$ is any other, then if θ is a mediating morphism we have, for all n, that $\theta(a')_n = \nu'_n(a')$, determining θ as a continuous function. Thus we have sketched the proof that **CPO** has all $\boldsymbol{\omega}^{\mathrm{op}}$-limits.

Turning to functors, we have categorical product and function space functors. The product of two cpos A and B is their Cartesian product with the componentwise ordering; it is easily verified to be the categorical product. Its action on morphisms $f: A \to A'$ and $g: B \to B'$ is also defined as usual:

$$(f \times g)(a, b) = \langle fa, gb \rangle.$$

Clearly, product is locally continuous. The function-space functor has the same (formal) definition as in **Pfn**; however, this time it is easily seen to be locally monotonic and, indeed, continuous. The function-space functor is the categorical one and **CPO** is Cartesian closed.

Unfortunately, **CPO** does not have categorical sums. It is therefore better to consider the category $\mathbf{CPO}_\bot$ of cpos and strict continuous functions (where for any cpos A and B a function $f: A \to B$ is *strict* if $f(\bot_A) = \bot_B$). This has a categorical sum which is defined on cpos A and B by putting

$$A + B = [\{0\} \times (A \backslash \{\bot\})] \cup [\{1\} \times (B \backslash \{\bot\})] \cup \{\bot\},$$

with the partial order defined by

$$\begin{aligned} c \sqsubseteq c' \equiv & [\exists a, a' \in A \,.\, a \sqsubseteq a' \wedge c = \langle 0, a \rangle \wedge c' = \langle 0, a' \rangle] \\ & \vee [\exists b, b' \in B \,.\, b \sqsubseteq b' \wedge c = \langle 1, b \rangle \wedge c' = \langle 1, b' \rangle] \\ & \vee c = \bot \end{aligned}$$

In other words, $A + B$ is the *coalesced* sum, that is, it is the disjoint union of A and B, but with least elements identified. The action of sum on morphisms turns out to be given by putting for $f: A \to A'$ and $g: B \to B'$

$$(f+g)(c) = \begin{cases} \langle 0, f(a) \rangle & (\exists a \in A \,.\, f(a) \neq \bot \wedge c = \langle 0, a \rangle), \\ \langle 1, g(b) \rangle & (\exists b \in B \,.\, g(b) \neq \bot \wedge c = \langle 1, b \rangle), \\ \bot & (\text{otherwise}), \end{cases}$$

and this shows that the sum is a locally continuous functor. Now we know that $+^E$ and, for example $\times^E$ are covariant $\boldsymbol{\omega}$-continuous bifunctors on $\mathbf{CPO}_\bot^E$ and $\mathbf{CPO}^E$ respectively. Luckily however these latter categories are the same, as both embeddings and projections in **CPO** are strict. (To see this let $f: A \to B$ be an embedding in **CPO**. Then $f(\bot) \sqsubseteq f(f^R(\bot)) \sqsubseteq \bot$ and so $f(\bot) = \bot$; also $f^R(\bot) = f^R(f(\bot)) = \bot$.).

In the same vein we can consider the *smash product* $A \otimes B$ in $\mathbf{CPO}_\bot$ defined as $\{\langle a, b \rangle \in A \times B \mid a \neq \bot \equiv b \neq \bot\}$ with the componentwise ordering inherited from the product $A \times B$ (which happens also to be the categorical product in $\mathbf{CPO}_\bot$). On morphisms $f: A \to A'$ and $f: B \to B'$ the functor acts as follows:

$$(f \otimes g)(a, b) = \begin{cases} \langle f(a), g(b) \rangle & (\text{if } f(a) \neq \bot \text{ and } g(b) \neq \bot), \\ \langle \bot, \bot \rangle & (\text{otherwise}). \end{cases}$$

This definition shows that the smash product is locally continuous. It can be characterized categorically. Say that a function $f: A \times B \to C$ of cpos is *bistrict* if and only if for any b in B, we have $f(\bot, b) = \bot$ (*left-strictness*) and also for any a in A we have $f(a, \bot) = \bot$ (*right-strictness*). Then the evident *bistrict* function $\otimes: A \times B \to A \otimes B$ is the *universal* bistrict continuous function from $A \times B$ in $\mathbf{CPO}_\bot$.

The *strict function-space* functor is defined as before (formally) but this time in $\mathbf{CPO}_\perp$ and we denote it by $\rightarrow_\perp$. It also is easily seen to be locally continuous. From a categorical point of view (see [21]), smash product makes $\mathbf{CPO}_\perp$ a symmetric monoidal category. Further, as we have a natural bijection

$$\hom(A \otimes B, C) \cong \hom(A, B \rightarrow_\perp C),$$

$\mathbf{CPO}_\perp$ is even a closed category. This to some extent repairs the fact that it is not Cartesian closed and explains the appearance of the smash product.

Finally we note a useful functor $(\cdot)_\perp : \mathbf{CPO} \rightarrow \mathbf{CPO}_\perp$ called *lifting*. For any cpo D,

$$(D)_\perp = (\{0\} \times D) \cup \perp,$$

with the partial order defined for any d, d' in $(D)_\perp$ by:

$$d \sqsubseteq d' \equiv (\exists c, c' \in D . c \sqsubseteq c' \wedge d = \langle 0, c\rangle \wedge d' = \langle 0, c'\rangle) \vee d = \perp.$$

On morphisms $f : D \rightarrow E$ we have for any d in $(D)_\perp$

$$(f)_\perp(d) = \begin{cases} \langle 0, f(c)\rangle & (\text{if } d = \langle 0, c\rangle), \\ \perp & (\text{if } d = \perp). \end{cases}$$

From a categorical point of view, lifting is the left adjoint to the forgetful functor from $\mathbf{CPO}_\perp$ to $\mathbf{CPO}$. We now have a wide variety of covariant continuous functors over $\mathbf{CPO}^E = \mathbf{CPO}_\perp^E$; Lehmann and Smyth discuss many of their uses in [20]. What is more, all of them arise naturally from a categorical point of view.

It does not appear to be useful to use $\mathbf{O}$ itself (a variant of Reynolds' predomains [32]) as $\mathbf{O}^E$ has no initial object. (To see this, suppose to the contrary that D is initial. Then there is an embedding $f : D \rightarrow \varnothing$, and we must therefore have $D = \varnothing$; but clearly $\varnothing$ is not initial, as there is no embedding $g : \varnothing \rightarrow X$ for any nonempty X.) On the other hand, one often sees an alternative definition of cpo where it is assumed that all *directed* sets have l.u.b.s rather than just increasing ω-sequences. (A subset X of a partial order P is directed if it is nonempty and any two elements of X have an upper bound in X.) Let us call these partial orders dcpos (*directed complete partial orders*). One easily adapts the above discussion to this case. Which definition to take is not a choice of great significance. On the one hand, the restriction to ω-sequences gives a larger category and is also computationally natural, as they arise when taking least fixed points; on the other hand the directed sets are natural mathematically. The following fact shows the difference is essentially one of cardinality.

FACT 1.a. *A partial order with a least element is a* cpo *if and only if it has all l.u.b.s of countable directed sets.*

b. *A function* $f : A \rightarrow B$ *between* cpos *is* ω*-continuous if and only if it preserves all l.u.b.s of countable directed sets.*

Proof. First note that for any countable directed set X there is an increasing sequence $\langle x_n \rangle_{n \in \omega}$ of elements of X such that any element of X is less than some x_n. Then we have $\bigsqcup X = \bigsqcup x_n$ and part a easily follows. For part b, suppose $f : A \rightarrow B$ is ω-continuous and let $X \subseteq A$ be directed. With $\langle x_n \rangle_{n \in \omega}$ as above, we calculate

$$f(\bigsqcup X) = f(\bigsqcup x_n) = \bigsqcup f(x_n) = \bigsqcup_{x \in X} f(x),$$

and this finishes the proof, as the other direction is immediate. □

Another way to look at these matters was discussed by Markowsky [22], who noted that a partial order is a dcpo if and only if it has l.u.b.s of all (well-ordered)

chains, and a function between dcpos preserves all l.u.b.s of directed sets if and only if it preserves all l.u.b.s of (well-ordered) chains.

Example 3. *Completeness.* We consider some full subcategories of **CPO** defined by imposing various completeness conditions.

DEFINITION 11. Let D be a partial order. A subset, X, of D is κ-consistent if and only if whenever $Y \subseteq X$ and $\|Y\| < \kappa$, Y has an upper bound in D.

Any subset, X, of a nonempty partial order is 0-, 1- and 2-consistent; it is 3-consistent if and only if it is *pairwise consistent* in the sense that any pair of its elements has an upper bound in D; it is ω-consistent if and only if any finite subset of its elements has an upper bound in D. Clearly if $\kappa \leqq \kappa'$ then every κ'-consistent subset is also ω-consistent; clearly too, any directed subset is ω-consistent.

DEFINITION 12. A partial order, D, is κ-complete if and only if it is nonempty and every κ-consistent subset has a least upper bound.

It follows from the above remarks that if $\kappa \leqq \kappa'$, then every κ-complete partial order is κ'-complete; also every ω-complete partial order is a cpo (and even a dcpo). Clearly for $0 \leqq n < 3$ the n-complete partial orders are the complete lattices and the 3-complete partial orders are the *coherent* cpos, in the sense of [24], [29] and, essentially, [11]. We now see that a partial order is ω-complete if and only if it is consistently complete in the sense of [29], [36] and, essentially, [11], [24].

FACT 2. *Let D be a partial order. It is ω-complete if and only if it is a dcpo with l.u.b.s of all subsets with upper bounds in D.*

Proof. Let D be ω-complete. We have already noted that it is a dcpo. Also any subset with an upper bound in D is ω-consistent and so has a least upper bound.

With the converse hypotheses, let X be an ω-consistent subset. Then every finite subset has an upper bound in X and so has a least upper bound. The set of such l.u.b.s is then directed and so must itself have a l.u.b. which is also the l.u.b. of X. $\square$

Turning to the properties of the full subcategory of κ-complete partial orders, we see that Theorem 1 may be applied, as the one-point cpo is a complete lattice. To see that $\boldsymbol{\omega}^{\mathrm{op}}$-limits exist, let $\Delta = \langle A_n, f_n \rangle$ be an $\boldsymbol{\omega}$-chain and define $\nu : A \to \Delta$ as before. As this defines a limiting cone in **CPO**, it only remains to show that A is κ-complete. The proof employs an idea of Scott, for the case of complete lattices (we have already employed it to show that A has a least element).

FACT 3. *A is κ-complete.*

Proof. Suppose $X \subseteq A$ is κ-consistent. Then for every m, so is $\{x_m | x \in X\}$, and then the least upper bound of X is $\langle \bigsqcup_{m \geqq n} f_{mn} (\bigsqcup \{x_m | x \in X\}) \rangle_{n \in \omega}$. $\square$

So the category of embeddings is a full subcategory of $\mathbf{CPO}^E$ with the same colimiting cones of $\boldsymbol{\omega}$-chains. It follows that any ω-continuous functors over $\mathbf{CPO}^E$ which preserve ω-completeness cut down to ω-continuous functors on the subcategory. This remark applies to all the functors discussed in Example 2 except the sum functor, which only preserves κ-completeness for $\kappa \geqq 3$. Sums of lattices can be defined by adding a new top element or by equating top elements as in [34], [30], and can be dealt with by local continuity. General completeness concepts have been considered in [3]; it would be interesting to see how they fit into the present considerations. One approach to handling nondeterminism and concurrency is to use one of several available powerdomain functors. These are available over **CPO** (see [15]), and the Smyth powerdomain [37] is available over the ω-complete cpos. However, the Plotkin powerdomain [27] does not preserve ω-completeness; a very weak notion of completeness was needed, leading to the so-called **SFP** objects (briefly considered in § 5).

Example 4. *Continuity and algebraicity.* Now we consider the ω-continuous and the ω-algebraic cpos. Our main definitions (13 and 15) are formulated entirely in

"countable" terms, but we pause to show that one could just as well start from definitions (14 and 16) formulated without any countability restrictions.

DEFINITION 13. Let D be a cpo. The *countable way-below* (=*relative compactness*) relation is defined by: $x \ll_\omega y$ if and only if for every countable directed subset Z of D, if $y \sqsubseteq \bigsqcup Z$ then $x \sqsubseteq z$ for some z in Z.

A countable subset, B, of D is an ω-basis of D if and only if for every element x of D the set $B_\omega(x) =_{\text{def}} \{b \in B \mid b \ll_\omega x\}$ is directed with l.u.b. x.

The cpo D is *ω-continuous* if and only if it has an ω-basis.

This definition is not quite the very similar one considered in the case of complete lattices by Scott in [33] and more generally for dcpos in [36], [23] and several papers in [7].

DEFINITION 14. Let D be a dcpo. The *way-below* (=*relative compactness*) relation is defined by: $x \ll y$ if and only if for every directed subset Z of D, if $y \sqsubseteq \bigsqcup Z$ then $y \sqsubseteq z$ for some z in Z.

A subset B of D is a *basis* of D if and only if for every element x of D the set $B(x) =_{\text{def}} \{b \in B \mid b \ll x\}$ is directed with l.u.b. x.

The dcpo D is *continuous* if and only if it has a basis.

To relate these two notions, we first note a few useful and easily proved facts. In a cpo we have that $x \ll_\omega y$ implies $x \sqsubseteq y$ and $x \ll_\omega y \sqsubseteq z$ implies $x \ll_\omega z$; analogous facts with $\ll$ replacing $\ll_\omega$ hold in a dcpo; for any elements x, y of a dcpo, if $x \ll y$ then $x \ll_\omega y$.

FACT. 4. A partial order is an ω-continuous cpo if and only if it is a continuous dcpo with a countable basis.

Proof. Let D be an ω-continuous cpo with ω-basis B. First, we see it is a dcpo. For if X is any directed set then $A = \bigcup_{x \in X} B_\omega(x)$ is countable and directed (by the above facts), and so its l.u.b. exists and is the l.u.b. of X. Next we show for any elements x and y of D that $x \ll y$ if and only if $x \ll_\omega y$. Suppose $x \ll_\omega y$ and $y \sqsubseteq \bigsqcup Z$ where Z is directed. Then $y \sqsubseteq \bigsqcup A$ as before, and so for some z in Z and a in $B_\omega(z)$ we have $x \sqsubseteq a \ll_\omega z$, and so $x \sqsubseteq z$. This establishes $x \ll y$. We have already noted the converse, that $x \ll y$ implies $x \ll_\omega y$. Therefore $B_\omega(x) = B(x)$ for any x, and so B is a countable basis. The converse assertion—that any continuous dcpo with a countable basis is ω-continuous—is proved along the same lines. □

The ω-algebraic cpos are a subclass of the ω-continuous ones and can also be presented in two ways.

DEFINITION 15. Let D be a cpo. An element x is ω-finite (=ω-compact) if and only if $x \ll_\omega x$. The cpo is ω-algebraic if and only if there is an ω-basis of finite elements.

DEFINITION 16. Let D be a dcpo. An element x is *finite* (=*compact*) if and only if $x \ll x$. The cpo is *algebraic* if and only if there is a basis of finite elements.

One then sees that D is an ω-algebraic cpo if and only if it is an algebraic dcpo with a countable basis of finite elements. Also in any (ω-) algebraic dcpo (cpo), there is only one (ω-) basis, namely the set of all (ω-) finite elements.

Turning to the full subcategory of the ω-continuous cpos, we note that it contains the one-point cpo, so Theorem 1 applies; however, it does not have all $\boldsymbol{\omega}^{\text{op}}$-limits, and we conjecture it does not have $\boldsymbol{\omega}$-colimits. The same remarks apply to the ω-algebraic cpos. Fortunately, however, the embedding subcategories inherit $\boldsymbol{\omega}$-colimits from $\mathbf{CPO}_\perp$. We need a preliminary lemma.

LEMMA 5a. *Embeddings (in* **CPO***) preserve the countable relative compactness relation.*

b. *Let E be a* cpo *and B be a countable subset of E. If x is an element of E and there is a directed subset C of $B_\omega(x)$ with l.u.b. x, then $B_\omega(x)$ is directed, with l.u.b. x.*

c. *Let E be a* cpo *and B and C be subsets of E, with B countable. Suppose that for every element y of C, $B_\omega(y)$ is directed with l.u.b. y and suppose too that for every element x of E there is a countable directed subset, C_x, of C such that $x = \bigsqcup C_x$. Then B is a basis for E.*

Proof. a. Let $f: D \to E$ be an embedding in **CPO** and let x, y be elements of D where $x \ll_\omega y$. If $f(y) \sqsubseteq \bigsqcup Z$ where Z is a countable directed subset of E, then $y = f^L(f(y)) \sqsubseteq \bigsqcup f^L(Z)$. So for some z in Z we have $x \sqsubseteq f^L(z)$, and so $f(x) \sqsubseteq f(f^L(z)) \sqsubseteq z$, showing that $f(x) \ll_\omega f(y)$.

b. If $u \ll_\omega x$ and $v \ll_\omega x$, then there are u', v' in C such that $u \sqsubseteq u'$, $v \sqsubseteq v'$. But as C is directed, this shows that $B_\omega(x)$ is directed too.

c. Take x in E and consider $\{B_\omega(y) | \in C_X\}$. This is a directed set, with respect to $\subseteq$, of directed sets as C_x is directed; its union is therefore directed and is clearly a subset of $B_\omega(x)$ with l.u.b. x. So by part b, $B_\omega(x)$ is directed with l.u.b. x. □

FACT 5. *Let $\Delta = \langle D_n, f_n \rangle$ be an* **ω**-*chain in* $\mathbf{CPO}^E$ *of* ω-*continuous* (ω-*algebraic*) cpos. *Suppose $\mu: \Delta \to D$ is colimiting. Then D is ω-continuous (ω-algebraic).*

Proof. We use Lemma 5c to show that D is ω-continuous when the D_n are. Let $B^{(n)}$ be an ω-basis for D_n $(n \in \omega)$; we claim $B \overset{\text{def}}{=} \bigcup_n \mu_n(B^{(n)})$ is an ω-basis for D. Let C be $\bigcup_n \mu_n(D_n)$. By Theorem 2 applied to **CPO**, we can take $C_x = \{\mu_n(\mu_n^R(x)) | n \in \omega\}$. Now (Lemma 5a) for each y in D_n, $\mu_n(B_\omega^{(n)}(y))$ is a directed subset of $B_\omega(\mu_n(y))$ with l.u.b. y. So by Lemma 5b $B_\omega(\mu_n(y))$ is directed with l.u.b. $\mu_n(y)$. Thus Lemma 5c applies. In the case where the D_n are all ω-algebraic, we take $B^{(n)}$ to be the ω-finite elements of D_n and find a basis of ω-finite elements of D. □

So the full subcategory of $\mathbf{CPO}^E$ of the ω-continuous (ω-algebraic) cpos is an ω-category that inherits **ω**-colimits, from $\mathbf{CPO}^E$. It follows that any ω-continuous functor over $\mathbf{CPO}^E$ that preserves ω-continuity (ω-algebraicity) cuts down to an ω-continuous functor over the subcategory. This enables all the functors discussed above for $\mathbf{CPO}^E$ to be handled except the function space functors which preserve neither ω-continuity nor ω-algebraicity (see [24], [23] for a counterexample). Here completeness considerations help. The full subcategory of $\mathbf{CPO}^E$ of the κ-complete and ω-continuous (ω-algebraic) cpos is clearly an ω-category that inherits **ω**-colimits from $\mathbf{CPO}^E$ (for $\kappa \leqq \omega$).

Now all the functors discussed above preserve the property of being both κ-complete and ω-continuous (ω-algebraic). We have already noted this for all except the function-space functors. For these, one notes that the proofs in [24], [23] for ω-complete and continuous (algebraic) dcpos adapt easily to ω-complete and ω-continuous (algebraic) cpos whether we consider all continuous functions or only the strict ones. The general case then follows from the facts that κ-complete implies ω-complete for $\kappa \leqq \omega$ and that κ-completeness is preserved.

Example 5. *Nondeterministic domains.* The category **NDO** was found useful for the semantics of nondeterministic and parallel programs in [15]. Its objects are the *nondeterministic* cpos $\langle D, \sqsubseteq, \cup \rangle$ where $\langle D, \sqsubseteq \rangle$ is a cpo and $\cup : D^2 \to D$ is an associative, commutative absorptive ω-continuous binary function (called *union*); the morphisms $f: D \to E$ are those ω-continuous functions which preserve union.

The trivial one-point object is terminal in $\mathbf{NDO}_\perp$ and the conditions of Theorem 1 are satisfied. Further, **NDO** has $\boldsymbol{\omega}^{\text{op}}$-limits. Indeed, the forgetful functor $U: \mathbf{NDO} \to \mathbf{CPO}$ creates them. Let $\Delta = \langle D_n, f_n \rangle$ be an ω-chain in **NDO** and suppose $\nu: E \to U\Delta$ is universal in **CPO**, being constructed as shown above. Then if we want a union on E so that the ν_n are **NDO** morphisms, we have for elements x, y, of E:

$$(x \cup y)_n = \nu_n(x \cup y) = \nu_n(x) \cup \nu_n(y) = x_n \cup y_n.$$

So this determines union, and it is easily seen that with this definition we obtain a universal cone in **NDO**. One interesting locally continuous function is $\rightarrow_{\subseteq}$, where on objects D, E:

$$D \underset{\subseteq}{\rightarrow} E = \{f : D \rightarrow E \mid f \text{ is } \omega\text{-continuous and preserves } \subseteq\},$$

(where $x \subseteq y \equiv_{\text{def}} x \cup y = y$), with the pointwise order and union, and defined as usual on morphisms. Other examples can be found in [15].

Of course, there are many interesting varieties (or pseudovarieties of one kind or another) subject to similar considerations [9], [25]. However, we have no clear idea of the possible applications.

Example 6. *ω-complete relations.* This category (or rather a slight variation of it) has been found to be useful for relating different semantics by Reynolds [31] (see also [12], [26]). It has as objects structures $\langle D, E, R\rangle$ where D and E are cpos and $R \subseteq D \times E$ is a binary relation which is *ω-complete* in the sense that if $\langle d_n\rangle$, $\langle e_n\rangle$ are increasing sequences in D and E, respectively, such that $R(d_n, e_n)$ holds for all integers n, then $R(\bigsqcup d_n, \bigsqcup e_n)$ holds too; the morphisms are pairs $\langle f, g\rangle : \langle D, E, R\rangle \rightarrow \langle D', E', R'\rangle$ where $f : D \rightarrow D'$, $g : E \rightarrow E'$ are morphisms in **CPO**, and for all x in D and y in E if $R(x, y)$ holds then so does $R'(fx, gy)$.

The terminal object is $\langle \bot, \bot, R_\bot\rangle$ where $\bot$ is the one-point cpo and $R_\bot$ is the complete binary relation over $\bot$; clearly, too, all the other conditions of Theorem 1 apply. Next $\boldsymbol{\omega}$-limits exist. To see this, let $\Delta = \langle\langle D_n, E_n, R_n\rangle, \langle f_n, g_n\rangle\rangle$ be an $\boldsymbol{\omega}$-chain. Let $\nu' : D \rightarrow \langle D_n, f_n\rangle$ and $\nu'' : E \rightarrow \langle E_n, g_n\rangle$ be limiting cones in **CPO**, constructed as above. Then $\nu : \langle D, E, R\rangle \rightarrow \Delta$ is limiting where $\nu_n = \langle \nu'_n, \nu''_n\rangle$ and $R(d, e)$ holds if and only if for all n the relation $R_n(d_n, e_n)$ holds.

A useful function space functor is given on objects by putting $\langle D, E, R\rangle \rightarrow \langle D', E', R'\rangle =^{\text{def}} \langle D \rightarrow D', E \rightarrow E', R \rightarrow R'\rangle$ where $D \rightarrow D'$ and $E \rightarrow E'$ are the cpos of all ω-continuous functions, and where

$$(R \rightarrow R')(f, g) \equiv \forall x \in D, y, \in E . R(x, y) \supset R'(f(x), g(y))$$

The action of the functor on morphisms is defined analogously to the case of **CPO**. Other examples can be found in [31]. This idea can be extended to several relations and to relations of any denumerable degree. It can also be combined with the ideas of Example 4 to consider continuous structures of various kinds. However, we must point out that the scope and usefulness of these mathematical possibilities is not known. We do not have a nice language for functors which permits a uniform treatment of the examples in [31], [12], [26]; we do not know why these relations seem to be needed only when function-spaces arise, for in other cases structural induction [20] seems to be sufficient; we wonder if the continuous structures should be accompanied by suitable logics along the lines of LCF [14] but possibly intuitionistic [35].

5. Computability. The approach to domain equation theory presented above may be seen as an abstraction from Scott's "D_∞" method [33]. As we have said, the "universal domain" method (Scott [34]), and its relation to the above theory, are to be treated in a separate paper. However, there is an aspect of universal domain theory, stressed by Scott, which must be mentioned here: computability. Starting from a suitable universal domain, it is possible to provide a smooth treatment of effective computability for all the constructs of interest, generalizing the relevant parts of classical recursion theory (Scott [34]). Lacking anything like this, the theory presented above has to be considered as seriously defective.

Fortunately, however, the deficiency can be remedied. Effectiveness can be built into **O**-categories in a satisfactory way. Here we will simply indicate some of the main points; for a fuller and more accurate treatment, see Smyth [38]. We again work by lifting suitable properties from domains to categories. *Algebroidal* categories are introduced as a generalization of algebraic cpos; they are categories with a (countable) "basis of finite objects". Then we get a handle on computability by requiring that bases be effectively presented.

Approach A. *Algebroidal categories.* These are the same as what Smyth previously called "algebraic categories" [37]. It has been brought to our attention that closely related notions have been discussed quite extensively in the literature of category theory, and this is what has prompted the change in nomenclature. Our algebroidal categories are essentially the "strongly ω-algebroidal categories" in (a slight extension of) the terminology of Banaschewski and Herrlich [6].

DEFINITION 17. An object A of a category $\mathbf{K}$ is *finite* in $\mathbf{K}$ provided that, for any $\boldsymbol{\omega}$-chain $\Delta = \langle V_n, f_n \rangle_{n \in \omega}$ in $\mathbf{K}$ with colimit $\mu : \Delta \to V$, the following holds: for any morphism $v : A \to V$, and for any sufficiently large n, there is a unique morphism $u : A \to V_n$ such that $v = \mu_n \circ u$. We say that $\mathbf{K}$ is *algebroidal* provided (i) $\mathbf{K}$ has an initial object and at most countably many finite objects, (ii) every object of $\mathbf{K}$ is a colimit of an $\boldsymbol{\omega}$-chain of finite objects, and (iii) every $\boldsymbol{\omega}$-chain of finite objects has a colimit in $\mathbf{K}$.

Notation. If $\mathbf{K}$ is algebroidal, we denote by $\mathbf{K}_0$ the full subcategory of $\mathbf{K}$ with objects the finite objects of $\mathbf{K}$.

The principal examples of interest to us are $\mathbf{SFP}^E$ (the category of **SFP** objects and embeddings [27]) and various of its subcategories, for example the category of bounded complete ω-algebraic ω-cpos and embeddings. The finite objects are in each case the finite domains.

THEOREM 4. *Every algebroidal category has all ω-colimits.*

Proof. See Smyth [37]. □

THEOREM 5. *Let $\mathbf{K}$ be an algebroidal category, and let $\mathbf{L}$ be an ω-category. Any functor F_0 from $\mathbf{K}_0$ into $\mathbf{L}$ extends uniquely* (*up to natural isomorphism of functors*) *to an ω-functor from $\mathbf{K}$ into $\mathbf{L}$.*

Outline of proof. For each nonfinite object D of $\mathbf{K}$, choose a particular colimiting cone $\mu D : \Delta_D \to D$, with Δ_D an $\boldsymbol{\omega}$-chain in $\mathbf{K}_0$; and for each $\boldsymbol{\omega}$-chain Δ in $\mathbf{L}$ choose a particular colimiting cone $\mu_\Delta : \Delta \to D_\Delta$ in $\mathbf{L}$. The extension of F_0 to all objects of $\mathbf{K}$ is immediate (via the chosen colimiting cones in $\mathbf{K}$, $\mathbf{L}$). To define the extension F of F_0 to morphisms, consider first morphisms $v : A \to V$ where A is finite and V nonfinite. Since A is finite, v factorizes as $v = (\mu_V)_n \circ u$. Then we put $Fv = (\mu_{F_0(\Delta_V)})_n \circ F_0 u$. Next, for morphisms $h : V \to W$, where V is nonfinite, define Fh as the mediating morphism from the colimiting cone $F(\mu_V)(= \mu_{F_0(\Delta_V)})$ to $F(h \circ \mu_V)$. Of course, it has to be checked that F so defined preserves composition of morphisms, and so is a functor (this is nontrivial).

Now suppose that $F, F' : \mathbf{K} \to \mathbf{L}$ are two ω-continuous functors which extend F_0. For each object V of $\mathbf{K}$ we have a canonical isomorphism $\tau_V : FV \to F'V$, namely the mediating morphism from $F(\mu_V)$ to $F'(\mu_V)$. Naturality of τ means that for $h : V \to W$, $F'h \circ \tau_V = \tau_W \circ Fh$; this is established by showing that $\tau_W \circ Fh$ mediates between the colimiting cone $F(\mu_V)$ and $F'h \circ \tau_V \circ F(\mu_V)$. □

Theorem 4 yields at once that $\mathbf{SFP}^E$ (for example) is an ω-category. Theorem 5 can be useful, at least heuristically, in setting up the definitions of appropriate ω-functors. Under these circumstances, the solution of typical domain equations, via the basic lemma, is unproblematic. More interesting is the question of effectiveness. The following definition seems natural:

DEFINITION 18. Let $\langle A_n \rangle_{n \in \omega}$, $\langle f_n \rangle_{n \in \omega}$ be enumerations of the objects and morphisms, respectively, of $\mathbf{K}_0$, where $\mathbf{K}$ is an algebroidal category. We say that $\mathbf{K}$ is *effectively given*, relative to these enumerations, provided that the following predicates are recursive in the indices:

i) $A_i = A_j$; $f_i = f_j$
ii) $\mathrm{dom}\,(f_k) = A_i$; $\mathrm{cod}\,(f_k) = A_i$
iii) f_k is an identity
iv) $f_i \circ f_j = f_k$.

This enables us to define an *effectively given object* (of $\mathbf{K}$) as an object that is given as the colimit of an effective ω-chain of finite objects, that is, as the colimit of a chain of the form

$$A_{r(0)} \xrightarrow{f_{s(0)}} A_{r(1)} \xrightarrow{f_{s(1)}} \cdots$$

(r, s recursive). One will naturally try to define a *computable morphism*, similarly, as the colimit of an effective ω-chain of finite morphisms (that is, morphisms of $\mathbf{K}_0$). Actually, such a characterization would be inadequate. The definitions given so far are, strictly speaking, appropriate only for categories of the form $\mathbf{K}^E$, whereas we are certainly interested in computability of morphisms other than embeddings. For an adequate treatment, we have to reformulate the definitions so as to apply to $\mathbf{O}$-categories; this is done in Smyth [38] where, for example, we find that an "admissible" $\mathbf{O}$-category $\mathbf{K}$ is, roughly speaking, one for which $\mathbf{K}^E$ is algebroidal. We can then define a *computable* functor, roughly, as a continuous functor F for which we can effectively assign to each finite object (morphism) $A(f)$ an effective ω-chain having $F(A)$ $(F(f))$ as colimit. A basic result, in terms of these definitions, will be that the initial fixpoint of a computable functor is computable.

Approach B. *Effective domains/categories.* Kanda [17] proposes that only computable items should be admitted to the domains and categories which we study—in contrast to the usual practice of first building all the continuous/countably-based items and then picking out the computable items from among these. This entails a modification of the closure properties required of the domains and categories: we now demand closure of domains with respect to sups only of *effective* ω-chains, and of categories with respect to effective colimits of effective ω-cochains. This approach works quite smoothly, and indeed yields a theory which is formally very close to Smyth [36] as far as concerns effective domains. In regard to the theory of effective categories (as developed by Kanda), perhaps the most striking feature of this theory is the very simple definition of *computable functor* (Kanda has "effective functor") in terms of indexings of hom-sets, which it permits.

Unlike Approach A, however, Kanda's theory does not pretend to give a general account of effectiveness in domains. In his theory, the definitions of an effective domain and of an effective category are quite independent. In order to apply the theory, we first define a particular category of "effective domains", and then show that this category satisfies the axioms for an "effective category". The definition is ad hoc, in the sense that no general or uniform notion of effective domains is proposed: we cannot, for example, define an effective domain to be an object of an "effective category of domains" (in contrast with our Approach A).

We incline to the view that these problems can best be attacked by means of the ideas mentioned in Approach A (finite objects in categories, etc.); but that it may be worthwhile to develop the argument in accordance also with the main ideas of Approach B, namely, that only computable items should be admitted to the field of discourse.

REFERENCES

[1] J. ADEMEK, *Free algebras and automata realizations in the language of categories*, Comment. Math. Univ. Carolina., 15 (1974), pp. 589–602.

[2] J. ADEMEK AND V. KOUBEK, *Least fixed point of a functor*, J. Comput. System Sci., 19 (1979), pp. 163–178.

[3] ADJ (J. A. GOGUEN, J. W. THATCHER, E. G. WAGNER AND J. B. WRIGHT) *A uniform approach to inductive p.o. sets and inductive closure*, Theoret. Comput. Sci., 7 (1978), pp. 57–77.

[4] M. ARBIB AND E. MANES, *Structures and Functors: The Categorical Imperative*, Academic Press, New York, 1975.

[5] M. ARBIB, *Free dynamics and algebraic semantics*, in FCT Proceedings, M. Karpinski, ed., Lecture Notes in Computer Science, 56 (1977), Springer-Verlag, Berlin, pp. 212–227.

[6] B. BANASCHEWSKI AND H. HERRLICH, *Subcategories defined by implications*, Houston J. Math., 2 (1976), pp. 149–171.

[7] B. BANASCHEWSKI AND R.-E. HOFFMAN, *Continuous lattices* in Proceedings, Bremen 1979. Lecture Notes in Mathematics 871, Springer-Verlag, New York, 1981.

[8] M. BARR *Coequalisers and free triples*, Math. Z. 116 (1970), pp. 307–322.

[9] S. L. BLOOM, *Varieties of ordered algebras*, J. Comput. System Sci. 13 (1976), pp. 200–212.

[10] W. H. BURGE, *Reccursive Programming Techniques*, Addison-Wesley, Reading, MA., 1975.

[11] H. EGLI AND R. CONSTABLE, *Computability concepts for programming language semantics*, Theoret. Comput. Sci., 2 (1976), pp. 143–145.

[12] M. GORDON, *Towards a semantic theory of dynamic binding*, Memo AIM-265, Computer Science Department, Stanford Univ. Stanford, CA, 1975.

[13] ———, *The Denotational Description of Programming Languages*, Springer-Verlag, New York, 1979.

[14] M. GORDON, R. MILNER AND C. WADSWORTH, *Edinburgh LCF*, Lecture notes in Computer Science 78, Springer-Verlag, Berlin, 1979.

[15] M. C. B. HENNESSY AND G. D. PLOTKIN, *Full abstraction for a simple parallel programming language*, in MFCS Proceedings, J. Bečvář, ed., Lecture Notes in Computer Science 74, Springer-Verlag, Berlin, pp. 108–120.

[16] H. HERRLICH AND G. STRECKER, *Category Theory*, Allyn and Bacon, Boston, 1974.

[17] A. KANDA, *Fully effective solutions of recursive domain equations*, in MFCS 1979, J. Bečvář, ed., Lecture Notes in Computer Science, 74, Springer-Verlag, Berlin, 1979.

[18] D. LEHMANN, *Categories for mathematical semantics*, in Proc. 17th IEEE Symposium on Foundations of Computer Science, 1976.

[19] D. LEHMANN AND M. B. SMYTH, *Data types*, in Proc. 18th Annual IEEE Symposium on Foundations of Computer Science, 1977.

[20] ———, *Algebraic specification of data types: A synthetic approach*, Math. Systems Theory, 14 (1981), pp. 97–139.

[21] S. MACLANE, *Categories for the Working Mathematician*, Springer-Verlag, Berlin, 1971.

[22] G. MARKOWSKY, *Chain-complete posets and directed sets with applications*, Algebra Universalis, 6 (1976), pp. 53–68.

[23] ———, *A motivation and generalisation of Scott's notion of a continuous lattice*, in Continuous Lattices, B. Banaschewski and R.-E. Hoffman, eds., Lecture Notes in Mathematics, 871, Springer-Verlag, Berlin, 1981, pp. 298–307.

[24] G. MARKOWSKY AND B. ROSEN, *Bases for chain-complete posets*, IBM J. Res. Develop., 20 (1976), pp. 138–147.

[25] J. MESEGUER, *Varieties of chain-complete algebras*, J. Pure Appl. Algebra, 19 (1980), pp. 347–383.

[26] R. MILNE AND C. STRACHEY, *A Theory of Programming Language Semantics*, Chapman and Hall, London 1976.

[27] G. D. PLOTKIN, *A powerdomain construction*, this Journal, 5 (1976), pp. 452–487.

[28] G. D. PLOTKIN AND M. B. SMYTH, *Category-theoretic solution of recursive domain equations* (extended abstract), in Proc. of 18th IEEE Symposium on Foundations of Computer Science, (1977), pp. 13–17.

[29] G. D. PLOTKIN, $\mathbb{T}^{\omega}$ *as a universal domain*, J. Comput. System. Sci., 16, (1978), pp. 207–236.

[30] J. C. REYNOLDS, *Notes on a lattice-theoretic approach to the theory of computation*, Systems and Information Science Dept., Syracuse Univ., Syracuse, NY, 1972.

[31] ———, *On the relation between direct and continuation semantics*, in Proc. 2nd Colloq. on Automata, Languages and Programming, Saarbrucken, J. Loeckx, ed., Lecture Notes in Computer Science 14, Springer-Verlag, Berlin, 1974, pp. 141–156.

[32] ———, *Semantics of the domain of flow diagrams*, 24 (1977), pp. 484–503.

[33] D. S. SCOTT, *Continuous lattices. toposes, algebraic geometry and logic*, in Proc. 1971 Dalhousie Conference, F. W. Lawvere, ed., Lecture Notes in Mathematics, 274, Springer-Verlag, Berlin, 1972, pp. 97–136.

[34] ———, *Data types as lattices*, this Journal, 5 (1976), pp. 522–587.

[35] ———, *Relating theories of the λ-calculus*, in To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, J. P. Seldin and J. R. Hindley, eds., Academic Press, New York, 1980.

[36] M. B. SMYTH, *Effectively given domains*, Theoret. Comput. Sci., 5 (1978), pp. 257–274.

[37] ———, *Powerdomains*, J. Comput. System Sci. 16 (1978), pp. 23–26.

[38] ———, *Computability in categories*, in Proc. 7th ICALP, J. De Bakker and J. van Leeuwen, eds., Lecture Notes in Computer Science, 85, Springer-Verlag, Berlin, 1980, pp. 609–620.

[39] J. E. STOY, *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, MA, 1978.

[40] R. D. TENNENT, *The denotational semantics of programming languages*, Comm. ACM, 19 (1976), pp. 437–453.

[41] ———, *Principles of Programming Languages*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[42] C. P. WADSWORTH, *The relation between computational and denotational properties for Scott's D_∞-models of the lambda-calculus*, this Journal, 5 (1976), pp. 488–521.

[43] M. WAND, *On the recursive specification of data types*, in Category Theory Applied to Computation and Control, E. G. Manes, ed., Lecture Notes in Computer Science 25, Springer-Verlag, Berlin, 1974.

[44] ———, *Fixed-point constructions in order-enriched categories*, Tech. Report 23, Computer Science Department, Indiana University, Bloomington, Indiana, 1977.

[45] ———, *fixed-point constructions in order-enriched categories*, Theoret. Comput. Sci. 8 (1979), pp. 13–30.

SIAM J. COMPUT.
Vol. 11, No. 4, November 1982

© 1982 Society for Industrial and Applied Mathematics

0097-5397/82/1104-0012 $01.00/0

OPTIMIZATION OF COST AND DELAY IN CELLULAR PERMUTATION NETWORKS*

C. RONSE†

Abstract. Among all known cellular permutation networks built from 2-cells, the network of Waksman and Green on n bits, built with $n\lceil\log_2(n)\rceil-2^{\lceil\log_2(n)\rceil}+1$ binary cells, has the lowest cost and delay. We show here that among a new class of cellular permutation networks it also has the lowest cost and delay.

Key words. cellular permutation networks, cost, delay

1. Introduction. A *permutation network* (or *connector*) on n bits is a switching circuit with n data inputs, n outputs and a certain number of control inputs, such that under an appropriate setting of the control inputs, any one-to-one connection between the data inputs and the outputs can be realized.

Generally a permutation network is built from small standard prefabricated permutation networks called *cells*. A cell on 2 bits is a *binary cell*, a cell on 3 bits is a *ternary cell*, etc. In VLSI a cell is built from logical gates (in particular, multiplexers). In telephony a cell is a crossbar (built with one switch between every input and every output; thus an n-bit crossbar has n^2 switches).

A *cellular* permutation network is a network built from cells. One can speak of binary, ternary (etc.) cellular networks.

The *cost* and *delay* of a network are the mathematical equivalents of the material size of that network and the maximal time that a signal takes when going from an input to an output. In VLSI the cost is measured by the chip area taken by the network; here connections count more than gates. In telephony and other domains where discrete components are used, connections do not account for the cost; here the cost is measured as the number of switches or cells.

The delay is measured as the maximal number of switches (or gates) that a signal goes through between an input and an output.

Clos showed how to build a permutation network on mn bits with two stages of m networks on n bits and one (middle) stage of n networks on m bits (see [5]). Using this construction recursively, Benes constructed a permutation network on 2^m bits using $2m-1$ stages of 2^{m-1} binary cells, thus using $2^m\cdot m-2^{m-1}$ binary cells. This construction was improved by Waksman ($2^m\cdot m-2^m+1$ binary cells) [4].

Green [3] generalized Waksman's network to permutation networks on any number of bits; Green's network on n bits is built with $n\lceil\log_2(n)\rceil-2^{\lceil\log_2(n)\rceil}+1$ bits (for any a, we write $\lceil a\rceil$ for the smallest integer not smaller than a).

It is easily seen that these three networks all have costs of the form $n\log_2(n)+O(n)$ in terms of 2-cells.

Now similar constructions can be made using k-cells ($k>2$) instead of binary cells. A permutation network on n bits can be built by recursion with $2\lceil\log_k(n)\rceil-1$ stages of $\lceil n/k\rceil$ cells of size k or less. This makes a cost of the form $(2n/k)(\log(n)/\log(k))+O(n)$ in terms of cells of size k or less.

Now if one uses crossbars as cells (a k-crossbar has cost k^2), the cost becomes $2kn\log(n)/\log(k)+O(n)$ in terms of switches. This is proportional to $k/\log(k)$. For all integers $k\geqq 2$, it takes its minimum for $k=3$, followed by $k=2$ or 4:

$$3/\ln(3)'=2.73\cdots,\qquad 2/\ln(2)=2.88\cdots.$$

* Received by the editors May 29, 1981, and in revised form February 9, 1982.

† Philips Research Laboratory, Av. Van Becelaere 2-Box 8, B-1170 Brussels, Belgium.

Thus $k=3$ is often chosen. However, for $k=2$ the increase in cost is small (about 5%). On the other hand the choice $k=2$ has two advantages:

1) The network of Waksman and Green has a relatively easy control algorithm, called "looping" (see [4]), which is not the case for other networks derived from Clos' network.

2) A 2-cell has two states and can be controlled with one binary control input. A 3-cell has $3!=6$ states and requires 3 binary control inputs ($3=\lceil \log_2(6) \rceil$). Thus the number of binary control inputs required for a binary and a ternary cellular permutation network of this type is, respectively, $n \log_2(n)+O(n)$ and $(2/\log_2(3))n \log_2(n)+O(n)$. Thus with ternary cells one needs 26% more control inputs than with binary cells.

This is why we will consider binary cellular permutation networks only.

2. The problem. Let a, b and k be integers such that $a \geqq 2$, $b \geqq 2$ and $0 \leqq k \leqq a-1$. Let A, D, B and E be permutation networks on a, $a-k$, b and $b-1$ bits respectively. Now take

- $2(b-1)$ copies of A: $A_0, \cdots, A_{b-2}, A'_0, \cdots, A'_{b-2}$,
- 1 copy of D,
- $a-k$ copies of B: $B_0, \cdots, B_{a-k-1}$,
- k copies of E: $E_{a-k}, \cdots, E_{a-1}$.

Then we can construct a permutation network (A, D, B, E) on $ab-k$ bits in the following way:

- The inputs of (A, D, B, E) are the inputs of $A_0, \cdots, A_{b-2}$ and D.
- The outputs of (A, D, B, E) are the outputs of $A'_0, \cdots, A'_{b-2}$ and C, where C is an "empty" network consisting of $a-k$ simple connections.
- The output j of A_i is connected to the input i of B_j (if $j<a-k$) or of E_j (if $j \geqq a-k$) ($j=0, \cdots, a-1$; $i=0, \cdots, b-2$).
- The output j of D is connected to the input $b-1$ of B_j ($j=0, \cdots, a-k-1$);
- The input j of A'_i is connected to the output i of B_j (if $j<a-k$) or of E_j (if $j \geqq a-k$) ($j=0, \cdots, a-1$; $i=0, \cdots, b-2$).
- The input j of C is connected to the output $b-1$ of B_j ($j=0, \cdots, a-k-1$).

The construction is illustrated in Fig. 1 for $a=4$, $b=3$ and $k=2$. The proof that

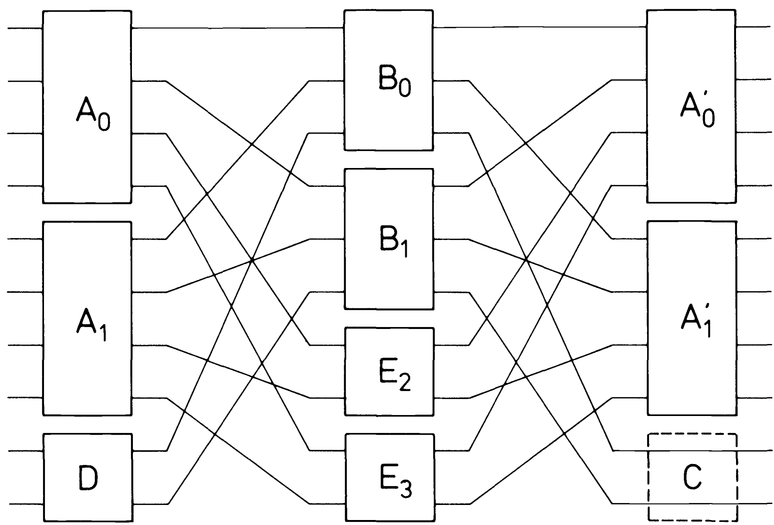


FIG. 1. (A, D, B, E).

(A, D, B, E) is a permutation network is elementary and can be found in [2]. It is based on a truncation of the construction of Goldstein and Leibholz [1].

Now the network G_n of Green and Waksman on n bits is defined inductively as follows [3], [4]:

- G_1 is a simple connection.
- G_2 is a binary cell.
- For $n > 2$,

$$G_n = \begin{cases} (G_2, G_2, G_a, G_{a-1}) & \text{if } n = 2a, \\ (G_2, G_1, G_a, G_{a-1}) & \text{if } n = 2a - 1. \end{cases}$$

This network has the following cost and delay in terms of 2-cells [2], [3], [4]:

$$\gamma_n = \sum_{j=2}^{n} \lceil \log_2 (j) \rceil = n \lceil \log_2 (n) \rceil - 2^{\lceil \log_2 (n) \rceil} + 1, \tag{1}$$

$$\delta_n = \begin{cases} 0 & \text{if } n = 1, \\ 2 \lceil \log_2 (n) \rceil - 1 & \text{if } n > 1. \end{cases} \tag{2}$$

(Here $\lceil a \rceil$ designates the smallest integer m such that $m \geqq a$.)

Now we define a family Π of permutation networks by recursion as follows:

- G_1 and G_2 belong to Π.
- If $a, b \geqq 2$ and $0 \leqq k \leqq a - 1$, if P_a, P_{a-k}, P_b and P_{b-1} are permutation networks on a, $a - k$, b and $b - 1$ bits respectively, which all belong to Π, then $(P_a, P_{a-k}, P_b, P_{b-1})$ belongs to Π.

Clearly any element of Π is a cellular permutation network.

We will show that the networks G_n are optimal for cost and delay among all networks of Π.

3. The result. We show the following:

THEOREM. *If P is a permutation network on n bits and if P belongs to Π, then*

(i) *Either $P = G_n$ or the cost of P in terms of binary cells is larger than γ_n.*

(ii) *The delay of P in terms of binary cells is not lower than δ_n.*

We first prove the following:

LEMMA. *Let a, b and k be integers such that $b \geqq 2$, $a \geqq 3$ and $0 \leqq k \leqq a - 1$. Then:*

$$\gamma_{ab-k} < 2(b-1)\gamma_a + \gamma_{a-k} + k\gamma_{b-1} + (a-k)\gamma_b.$$

Proof. Let $\phi(a, b, k)$ be the right-hand side of this inequality. The proof consists of 3 steps:

Step 1. For any $a \geqq 3$, $2\gamma_a \geqq a \lceil \log_2 (a) \rceil$, and the equality holds only for $a = 3$.

Proof. The result is true for $a \leqq 8$, as can easily be checked. Now for $a \geqq 9$ we have $\lceil \log_2 (a) \rceil \geqq 4$ and so:

$$a \lceil \log_2 (a) \rceil \geqq 4a > 2(2(a-1) - 1) \geqq 2(2^{\lceil \log_2 (a) \rceil} - 1). \tag{3}$$

Thus:

$$\begin{aligned} 2\gamma_a &= 2(a \lceil \log_2 (a) \rceil - 2^{\lceil \log_2 (a) \rceil} + 1) \qquad \text{by (1))} \\ &= a \lceil \log_2 (a) \rceil + (a \lceil \log_2 (a) \rceil - 2(2^{\lceil \log_2 (a) \rceil} - 1)) \\ &> a \lceil \log_2 (a) \rceil \qquad \text{(by 3)).} \end{aligned}$$

Step 2. The result is true for $b = 2$.

Proof. $\phi(a, 2, k) = 2\gamma_a + \gamma_{a-k} + (a - k)$.

If $a=3$, then $\gamma_{2a-k}=\gamma_{6-k}=11-3k$, while $\phi(a,2,k)=2\gamma_3+\gamma_{3-k}+3-k=9-k+\gamma_{3-k}$, and so $\phi(a,2,k)>\gamma_{2a-k}$ since $\gamma_{3-k}>2-2k$. Thus the result holds for $a=3$. Suppose now that $a\geqq 4$. Then we have

$$\gamma_{2a-k}=\gamma_a+\sum_{x=a+1}^{2a-k}\lceil\log_2(x)\rceil \qquad \text{(by (1))}$$

$$\text{(4)} \qquad \leqq \gamma_a+(a-k)\lceil\log_2(2a)\rceil$$

$$\leqq \gamma_a+(a-k)\lceil\log_2(a)\rceil+(a-k),$$

since $\lceil\log_2(2a)\rceil=\lceil\log_2(a)\rceil+1$. Now

$$(a-k)\lceil\log_2(a)\rceil=a\lceil\log_2(a)\rceil-k\lceil\log_2(a)\rceil$$

$$<2\gamma_a-k\lceil\log_2(a)\rceil \qquad \text{(by Step 1)}$$

$$\text{(5)} \qquad \leqq 2\gamma_a-\left(\sum_{x=a-k+1}^{a}\lceil\log_2(x)\rceil\right)$$

$$=2\gamma_a-(\gamma_a-\gamma_{a-k}) \qquad \text{(by (1))}$$

$$=\gamma_a+\gamma_{a-k}.$$

Combining (4) and (5) we get

$$\gamma_{2a-k}<\gamma_a+(\gamma_a+\gamma_{a-k})+(a-k)=\phi(a,2,k).$$

Step 3. The result is true for any b.

Proof. We use induction on b. The result is true for $b=2$. Suppose that $b>2$ and that the result is true for $b-1$. Then we get

$$\gamma_{ab-k}-\gamma_{a(b-1)-k}=\sum_{x=ab-a-k+1}^{ab-k}\lceil\log_2(x)\rceil \qquad \text{(by (1))}$$

$$=\sum_{x=ab-a-k+1}^{ab-a}\lceil\log_2(x)\rceil+\sum_{x=ab-a+1}^{ab-k}\lceil\log_2(x)\rceil$$

$$\leqq k\lceil\log_2(ab-a)\rceil+(a-k)\lceil\log_2(ab)\rceil$$

$$\leqq k(\lceil\log_2(a)\rceil+\lceil\log_2(b-1)\rceil)+(a-k)(\lceil\log_2(a)\rceil+\lceil\log_2(b)\rceil)$$

$$\leqq a\lceil\log_2(a)\rceil+\mathrm{k}\lceil log_2(b-1)\rceil+(a-k)\lceil\log_2(b)\rceil$$

$$\leqq 2\gamma_a+k\lceil\log_2(b-1)\rceil+(a-k)\lceil\log_2(b)\rceil \qquad \text{(by Step 1)}$$

$$\leqq 2\gamma_a+k(\gamma_{b-1}-\gamma_{b-2})+(a-k)(\gamma_b-\gamma_{b-1}) \qquad \text{(by (1))}$$

$$\leqq \phi(a,b,k)-\phi(a,b-1,k) \qquad \text{(by definition of } \phi(a,b,k)).$$

Thus we have

$$\phi(a,b,k)\geq\phi(a,b-1,k)+\gamma_{ab-k}-\gamma_{a(b-1)-k}$$

$$\geqq \gamma_{ab-k}+(\phi(a,b-1,k)-\gamma_{a(b-1)-k})$$

$$>\gamma_{ab-k} \qquad \text{(by induction hypothesis).}$$

Therefore the result holds for any b. □

Proof of the theorem. We use induction on n. The result is obvious for $n\leqq 2$. Suppose that $n>2$ and that the result is true for any $m<n$. Take P_n a permutation

network on n bits belonging to Π. We can write $P_n = (P_a, P_{a-k}, P_b, P_{b-1})$, where each P_j $(j = a, a-k, b, b-1)$ is a permutation network on j bits and belongs to Π. It is easy to see that $a, b < n$.

For any network N, write $\gamma(N)$ and $\delta(N)$ for the cost and delay of N. We have:

$$\gamma(P_n) = 2(b-1)\gamma(P_a) + \gamma(P_{a-k}) + k\gamma(P_{b-1}) + (a-k)\gamma(P_b)$$

$$\geqq 2(b-1)\gamma_a - \gamma_{a-k} + k\gamma_{b-1} + (a-k)\gamma_b \qquad \text{(by induction hypothesis)} \tag{6}$$

$$\geqq \gamma_{ab-k} = \gamma_n \qquad \text{(by the lemma for } a \geqq 3\text{, by definition for } a = 2\text{)}. \tag{7}$$

Thus $\gamma(P_n) > \gamma_n$, except if the equality holds in both (6) and (7). If it holds in (6), then we must have $P_j = G_j$ for $j = a, a-k, b, b-1$ (by induction hypothesis). If it holds in (7), then we must have $a = 2$ by the lemma. Thus either $\gamma(P_n) > \gamma_n$ or $P_n = (G_a, G_{a-k}, G_b, G_{b-1}) = G_n$.

Now we have

$$\begin{aligned}
\delta(P_n) &= \max\{\delta(P_a), \delta(P_{a-k})\} + \max\{\delta(P_b), \delta(P_{b-1})\} + \delta(P_a) \\
&\geqq \max\{\delta_a, \delta_{a-k}\} + \delta_b + \delta_a \qquad \text{(by induction hypothesis)} \\
&\geqq 2\delta_a + \delta_b \\
&\geqq 4\lceil \log_2(a) \rceil + 2\lceil \log_2(b) \rceil - 3 \qquad \text{(by (2))} \\
&\geqq 2(\lceil \log_2(a) \rceil + \lceil \log_2(b) \rceil) - 1 + 2(\lceil \log_2(a) \rceil - 1) \\
&\geqq 2(\lceil \log_2(ab) \rceil) - 1 + 2(\lceil \log_2(2) \rceil - 1) \\
&\geqq 2(\lceil \log_2(ab) \rceil) - 1 \\
&\geqq 2(\lceil \log_2(ab-k) \rceil) - 1 = \delta_n.
\end{aligned}$$

Therefore the result holds. Note that if $a > 2$, then $\lceil \log_2(a) \rceil - 1 > 0$ and so $\delta(P_n) > \delta_n$. However, we can have $\delta(P_n) = \delta_n$ with $P_n \neq G_n$. An example is given in [2].

REFERENCES

[1] L. J. GOLDSTEIN AND S. W. LEIBHOLZ, *On the synthesis of signal switching networks with transient blocking*, IEEE Trans. Elec. Comp., C-16 (1967), pp. 637–641.

[2] C. RONSE, *Cellular permutation networks: A survey*, MBLE Research Lab. Report R415, December 1979.

[3] W. H. KAUTZ, K. N. LEVITT AND A. WAKSMAN, *Cellular interconnection arrays*, IEEE Trans. Elec. Comp., C-17 (1968), pp. 443–451.

[4] A. WAKSMAN, *A permutation network*, J. Assoc. Comput. Mach., 15 (1968), pp. 159–163.

[5] V. E. BENES, *Mathematical Theory of Connecting Networks and Telephone Traffic*, Academic Press, New York, 1965.

SIAM J. COMPUT.
Vol. 11, No. 4, November 1982

© 1982 Society for Industrial and Applied Mathematics
0097-5397/82/1104-0013 $01.00/0

ON A PRIMALITY TEST OF SOLOVAY AND STRASSEN*

A. O. L. ATKIN† AND R. G. LARSON‡

Abstract. Solovay and Strassen [SIAM J. Comput., 6 (1977), pp. 84–85] propose a primality test based on the fact that for primes $p>2$ we have $(a/p)\equiv a^{(p-1)/2} (\bmod p)$, where (a/p) is the Jacobi symbol. We prove here that the strong pseudoprime test is better, in the sense that it never takes more time nor is less effective, and sometimes is quicker or more effective. We also discuss the probability of error in the strong pseudoprime test, and show that it is never greater than $\frac{1}{4}$.

Key words. Monte-Carlo tests, primality, pseudoprime

1. Description of the tests. We consider testing whether an odd integer >1 is prime or composite. In what follows we shall use "(time w)" to mean "this step can be done by $w \log_2 n$ multiprecise operations," and "compute z modulo n" to mean the reduction of z modulo n to a residue r in the range $-1 \leqq r \leqq n-2$. All of the tests we discuss have the property that they demonstrate a number to be composite under certain conditions, and assume without proof that the number is prime if the conditions are not satisfied. We denote by "DECIDE prime" or "DECIDE composite" the points in the program where the decision is taken. The test given in [1] is the

Solovay–Strassen test (SST).

A. Take a random number a from the interval $1 \leqq a \leqq n-1$.
B. Compute $(a, n)=\gcd(a, n)$. (time 1.5). If $(a, n)>1$ DECIDE composite.
C. Compute $a^{(n-1)/2}$ modulo n and the Jacobi symbol (a/n). (time 2.5) and (time 1.5) respectively. If not equal, DECIDE composite.
D. DECIDE prime.

Our description above follows that in [1]. Note that step B is superfluous if $a^{(n-1)/2}$ modulo n is computed, and thus is only justified if $(a, n)>1$ occurs very frequently, which for a fixed n and random a would imply that n has many small factors. We suggest that a better version of this test would be

Solovay–Strassen test 2 (SST2).

AA. Take a random number a from the interval $1< a \leqq n-1$.
BB. Compute $a^{(n-1)/2}$ modulo n. Call this b. If $b \neq \pm 1$, DECIDE composite.
CC. Compute (a/n). If $(a/n)\neq b$, DECIDE composite.
DD. DECIDE prime.

In this version the steps AA and BB are essentially the regular pseudoprime test, and the interesting new idea of Solovay and Strassen is to obtain further information from the Jacobi symbol. Unfortunately, it turns out that the strong pseudoprime test will always decide correctly when SST2 does, and its time is at most the time of step BB.

---

* Received by the editors July 31, 1978.

† Department of Mathematics, University of Illinois at Chicago Circle, Chicago, Illinois 60680. The research of this author was supported in part by the National Science Foundation under grant MCS75-07478 AO2.

‡ Department of Mathematics, University of Illinois at Chicago Circle, Chicago, Illinois 60680. The research of this author was supported in part by the National Science Foundation under grant MCS76-06638 AO1.

Strong pseudoprime test (SPPT).

E. Take a random number a from the interval $1<a<n-1$.
F. With $n-1=m2^\alpha$, where m is odd, compute a^m modulo n. Call this b. If $b=\pm 1$, DECIDE prime.
G. Set $\beta=\alpha$.
H. If $\beta=1$, DECIDE composite.
I. Decrease β by 1, and replace b by b^2 modulo n.
J. If $b=1$, DECIDE composite. If $b=-1$, DECIDE prime.
K. Go to step H.

It is clear that step F, together with all the repeats of step I, take time at most the time of step BB of SST2 (we neglect the trivial time needed for testing $b=\pm 1$). The basis of this test is merely the fact that 1 and -1 are the only square roots of 1 modulo a prime. This test was brought to the attention of one of us by John Selfridge.

2. We now prove our main result.

THEOREM. SPPT *is better than* SST2.

Proof. It is clear that SPPT is no longer than SST2, and usually shorter. We need therefore only prove that it is more effective, that is, whenever SST2 decides correctly, so does SPPT, but not conversely.

First, as in step F above, let us write $n-1=2^\alpha m$, with m odd and $\alpha \geqq 1$, and let also $n=\prod_p p^\sigma$ be the canonical factorization of n into prime powers. Both tests are correct in their decisions that n is composite. Let us assume that SPPT has decided that n is prime; we shall prove that SST2 necessarily decides that n is prime. Now SPPT decides prime in two cases, which we consider separately.

Case 1. *There exists* β, *with* $1\leqq\beta\leqq\alpha$, *such that* $a^{m2^{\beta-1}}\equiv -1 \pmod n$. For each p dividing n, let a have order $d=2^\delta m'$ modulo p, where $\delta\geqq 0$ and m' is odd. Then $m'2^\delta | m2^\beta$ and $m'2^\delta \nmid m2^{\beta-1}$ imply that $m'|m$ and $\beta=\delta$. Also $d|p-1$ implies that $p=1+xm'2^\beta$ (here x may be odd or even). Now $(a/p)\equiv a^{(p-1)/2} \pmod p$, and since both are congruent to ± 1 we may raise the right-hand side of this congruence to any odd power, in particular to the power m/m', obtaining

$$(a/p)\equiv a^{m(p-1)/(2m')}=(a^{m2^{\beta-1}})^x\equiv(-1)^x \pmod p.$$

But now the first and last terms are only ± 1, and $p>2$, so that $(a/p)=(-1)^x$, and hence $(a/n)=(-1)^s$, where $s=\sum_p x\sigma$. On the other hand

$$n=\prod_p(1+xm'2^\beta)^\sigma\equiv 1+2^\beta\sum_p xm'\sigma\equiv 1+2^\beta\sum_p x\sigma=1+2^\beta s \pmod{2^{\beta+1}}.$$

Thus $(n-1)/2\equiv 2^{\beta-1}s \pmod{2^\beta}$. Now $a^{(n-1)/2}=a^{m2^{\alpha-1}}$ is congruent to 1 or $-1 \pmod n$ according as $\beta<\alpha$ or $\beta=\alpha$. But we have just seen that the power of 2 contained in $(n-1)/2$ is exactly $\beta-1$ or $\geqq\beta$ according as s is odd or even, that is, according as (a/n) is -1 or 1. Thus we always have $(a/n)\equiv a^{(n-1)/2} \pmod n$, and the step CC in SST2 will necessarily decide prime.

Case 2. $a^m\equiv 1 \pmod n$. In this case we have also $a^m\equiv 1 \pmod p$ for each p, so that the order of a modulo p is odd and hence $(a/p)=1$. Thus $(a/n)=1$ also, and clearly $a^{(n-1)/2}=a^{m2^{\alpha-1}}\equiv 1 \pmod n$. Hence again step CC in SST2 will decide prime.

We have therefore shown in both cases that SPPT is no less effective than SST2; in Case 2 it is no more so (but note that in Case 2 SPPT exits at step F, and so is certainly shorter). To show that SPPT can be more effective than SST2 one may analyze Case 1 more closely, or more simply observe that with $a=14$ and $n=65$

SPPT proves n composite in one step while SST2 decides that n is prime. This completes the proof of the theorem.

3. It is of interest to observe that the proof above implies the following

COROLLARY. *Let* $n>1$ *be odd, and let* a *be any integer such that* $a^{(n-1)/2} \equiv -1 \pmod{n}$. *Then the Jacobi symbol* $(a/n) = -1$.

In turn, this implies that one could save time in SST2 by deciding that n is prime when $a^{(n-1)/2} \equiv -1 \pmod{n}$ without computing (a/n).

4. It is fairly easy to show that the probability of an incorrect decision for a single trial of SPPT is less than $\frac{1}{4}$. In the other direction, suppose that p and q are primes with $q = 2p-1$, $p \equiv 3 \pmod{4}$. Let $n = pq$. Note that $(n-1)/2 = (2p+1)(p-1)/2$ is odd. Now $a^{(n-1)/2} \equiv \varepsilon \pmod{n}$, where $\varepsilon = \pm 1$, if and only if the congruence holds both modulo p and modulo q. Since

$$(n-1)/2 = (2p+1)((p-1)/2) = (2p+1)((q-1)/4),$$

and $2p+1$ is odd, $a^{(n-1)/2} \equiv \varepsilon \pmod{n}$ if and only if $a^{(p-1)/2} \equiv \varepsilon \pmod{p}$ and $a^{(q-1)/4} \equiv \varepsilon \pmod{q}$. Therefore SPPT will incorrectly decide that such an n is prime in the cases where a is a quadratic residue modulo p, and a biquadratic residue modulo q, or where it is a quadratic nonresidue modulo p and a quadratic residue but a biquadratic nonresidue modulo q. These cases together comprise $(p-1)(q-1)/4-1$ values of a (recall that $a = 1$ is not tried). Thus the probability of failure is $(p^2-2p-1)/(4p^2-2p-6) \to \frac{1}{4}$ as $p \to \infty$. Of course, p cannot tend to infinity if the number of such pairs (p, q) is finite, but this is unlikely to be proved or disproved in the near future.

REFERENCE

[1] R. SOLOVAY AND V. STRASSEN, *A fast Monte-Carlo test for primality*, this Journal, 6 (1977), pp. 84–85.